UNIVERSITY OF CALIFORNIA,
IRVINE

# Near Memory Processing in Hybrid Memory System
# 3D-DRAM vs. 3D-NVM

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Electrical and Computer Engineering

by

Maryam S. Hosseini

2021

# DEDICATION

To
My dearest loving deeply missed Mom,
forever you remain in my heart and my soul

My loving husband Pooria,
for his true love and support

My dear Dad,
for always believing in me

# TABLE OF CONTENTS

# LIST OF FIGURES

Page

# LIST OF TABLES

# ACKNOWLEDGMENTS

My journey towards Ph.D. at UCIrvine has been incredibly amazing and memorable. I am thankful for all of the people who have helped me to go through all the challenges.

First and foremost, I am sincerely grateful to my Ph.D. advisor, Professor Nader Bagherzadeh, for his guidance and patience over these years. His support and invaluable inspiration encouraged me throughout this research. He is also a mentor of life.

I would also like to specially thank my co-advisor, Professor Masoumeh (Azin) Ebrahimi, for providing me insightful guidance, feedback, and advice on my research. She is the most caring motivator and a dear friend on mine.

I would like to express my gratitude to Professor Jean-Luc Gaudiot and Professor Philip Sheu for serving on my dissertation committee.

Many thanks to my colleagues at Advanced Computer Architecture Group, especially Dr. Ahmad Albaqsami and my dear friend Zahraa Marafie. Cooperation and discussion with them have been very helpful for my research. Also thanks to my other colleague Nezam Rohbani for the collaboration we had in part of my research.

Finally, I would like to give my greatest gratitude to the most important person in my life, my husband, Pooria. I am extremely grateful to his support during my Ph.D. study. He has always been my source of happiness. My deepest appreciation goes to my family, especially my dear parents for their endless love and support.

# VITA

## Maryam S. Hosseini

### EDUCATION

| | |
|---|---|
| **Ph.D. in Computer Systems and Software** | **July 2021** |
| University of California, Irvine | *Irvine, California* |
| **M.S. in Computer Systems and Software** | **May 2017** |
| University of California, Irvine | *Irvine, California* |
| **B.S. in Computer Engineering** | **Sep. 2009** |
| University of Mashhad | *Mashhad, Iran* |

### RESEARCH EXPERIENCE

| | |
|---|---|
| **Graduate Student Researcher** | **Oct. 2016–June 2021** |
| University of California, Irvine | *Irvine, California* |
| **Research Intern** | **May 2018–Sep. 2018** |
| Western Digital Company | *Irvine, California* |

### TEACHING EXPERIENCE

| | |
|---|---|
| **Teaching Assistant** | **2017–2021** |
| University of California, Irvine | *Irvine, California* |
| **Instructor** | **Summer 2017** |
| University of California, Irvine | *Irvine, California* |
| **Lecturer** | **2018–Present** |
| California State University, Long Beach | *Long Beach, California* |

### RESEARCH INTERESTS

Computer architecture, Near memory processing, High performance computing, Hybrid memory systems, Heterogeneous computing systems, Performance evaluation.

## AWARDS

**PhD Bridge Fellowship** **Oct. 2016**
University of California, Irvine *Irvine, California*

**Graduate Student Fellowship** **March. 2018**
University of California, Irvine *Irvine, California*

**Division of Teaching Excellence and Innovation Fellowship** **June. 2020**
University of California, Irvine *Irvine, California*

## PUBLICATIONS AND PRESENTATIONS

1. **Maryam S. Hosseini**, Masoumeh Ebrahimi, Pooria Yaghini and Nader Bagherzadeh, "Application Characterization for Near Memory Processing", in Parallel, Distributed, and Network-based Processing (PDP) Conference, 2021.

2. **Maryam S. Hosseini**, Masoumeh Ebrahimi, Pooria Yaghini and Nader Bagherzadeh, "Near Volatile and Non-Volatile Memory Processing in 3D Systems", in IEEE Transactions on Emerging Topics in Computing (TETC) Journal, 2021.

3. **Maryam S. Hosseini**, Masoumeh Ebrahimi, Nezam Rohbani, Pooria Yaghini, Nader Bagherzadeh, "Near Memory Processing in Hybrid Systesm: 3D-DRAM vs. 3D-NVM", **under review** in Journal of Systems Architecture (JSA), 2021.

4. Ahmad Albaqsami, **Maryam S. Hosseini**, Masoomeh Jasemi and Nader Bagherzadeh, "Adaptive HTF-MPR: An Adaptive Heterogeneous TensorFlow Mapper Utilizing Bayesian Optimization and Genetic Algorithms", in Transactions on Intelligent Systems and Technology (TIST) Journal, 2020.

5. Ahmad Albaqsami, **Maryam S. Hosseini** and Nader Bagherzadeh, "HTF-MPR: A heterogeneous TensorFlow mapper targeting performance using genetic algorithms and gradient boosting regressors", in Design, Automation, and Test in Europe Conference (DATE), 2018.

6. **Maryam S. Hosseini**, "Reliability Enhancement of Many-core Processors", Master Thesis, University of California, Irvine, 2017.

# ABSTRACT OF THE DISSERTATION

Near Memory Processing in Hybrid Memory System
3D-DRAM vs. 3D-NVM

By

Maryam S. Hosseini

Doctor of Philosophy in Electrical and Computer Engineering

University of California, Irvine, 2021

Professor Nader Bagherzadeh, Chair

The cost of transferring data between the off-chip memory system and compute unit is the fundamental energy and performance bottleneck in conventional multi-core computing systems. Furthermore, in the era of big data and with the advent of emerging data-intensive applications, such as graph processing, machine learning, deep learning, media processing, data mining, computer vision, computational biology, and speech recognition, this bottleneck has continuously increased. For such applications, the expensive data movement between memory and compute unit dominates both execution time and energy/power consumption which results in impeding future performance scaling. Moreover, the technology scaling (the end of Moore's law and failure of Dennard scaling) has made all compute units energy and power constrained. In order to satisfy the energy and power constraints, researchers are forced to stop further increasing the frequency and to reduce the chip utilization. Thus, to continue scaling the performance, energy overhead must be minimized for every operation. To overcome these difficulties, different approaches either *algorithmic-level* or *architectural-level* can be applied. The later promising approach commonly referred to as *Near Memory Processing (NMP)* has become a potential and practical technology to transform the *computation-centric* systems towards *memory-centric* systems. The introduction of 3D die stacking technology and more importantly hybrid memory systems have revolutionized the concept of NMP. 3D die stacking, built using Through-Silicon Via (TSV), offers higher bandwidth, shorter wire lengths, lower power (due to short-length low-capacitance wires), and better performance compared to traditional 2D planner memories. This memory technology allows architects to implement practical NMP systems by vertically stacking multiple memory layers on top of a logic die in the same package. The logic layer is typically the most bottom layer which provides an area for adding a wide range of processing logic (general-purpose cores, FPGAs, ASICs, or a combination of all types). It enables higher density many-core architectures to happen and helps for improving the power-performance characteristics to increase capabilities of modern integrated circuits.

The focus of this dissertation is to explore and evaluate the feasibility and efficacy of NMP architecture constructed based on an emerging Non-Volatile Memory (NVM) technology in a 3D structure. And to compare it with the conventional NMP architecture built based on 3D-DRAM in terms of performance and power consumption. To this purpose, first, a set of NMP-centric performance metrics are redefined in order to analyze the efficacy of mapping a given processing unit to a specific application. Leveraging the proposed metrics, a comprehensive characterization is conducted on a wide range of multi-threaded applications (various computation and memory patterns) from different domains as a case study to reveal their performance bottleneck. Then, two different NMP architectures are explored and the impact of constructing NMP architecture based on an emerging non-volatile memory technology (3D-NVM) is analyzed. Also the feasibility of having an NMP subsystem on a hybrid 3D memory system is motivated in this dissertation. Finally, the experimental results demonstrate that executing certain data-intensive (memory-intensive) applications on the evaluated NMP architectures (3D-PCM and 3D-DRAM) improve the performance by **1.3x to 5x** and reduce memory power/energy consumption by an average of **47%** compared to executing them on conventional multi-core Host CPU system. These improvements make the hybrid NMP system a great design technique for acceleration in performance and power across a wide range of data-intensive applications.

# Chapter 1

# Introduction

Over the years, memory technology has not been able to keep up with the improvements in processor technology in terms of latency and energy consumption, which is referred as *memory wall*. The term "memory wall" goes back to 1994, when Dr. Wulf and Dr. McKee talked about it in their short paper [152]. The whole idea was that the main memory will become the bottleneck of the whole computing system, since there would be a diverging exponential increase in performance of the processor and main memory. Based on Figure 1.1 which visualizes the memory wall problem, multi-core CPU performance has been improving 60% per year, while the improvement for the memory performance is less than 10% per year. Therefore, the performance gap between memory and processor increases exponentially. This has become a real challenge for today's multi-core processors. Multi-level data and instruction intelligent caching, utilizing large register files, and increasing off-chip DRAM row-buffer size are as the main techniques to mitigate memory wall effect. However, the latency of off-chip memory access due to cache misses, still limits the performance of processors.

Also, the end of Moore's law [103] (the doubling of transistors on chip every 18 months) and failure of Dennard scaling [41] are causing the computer performance to reach a plateau [47].

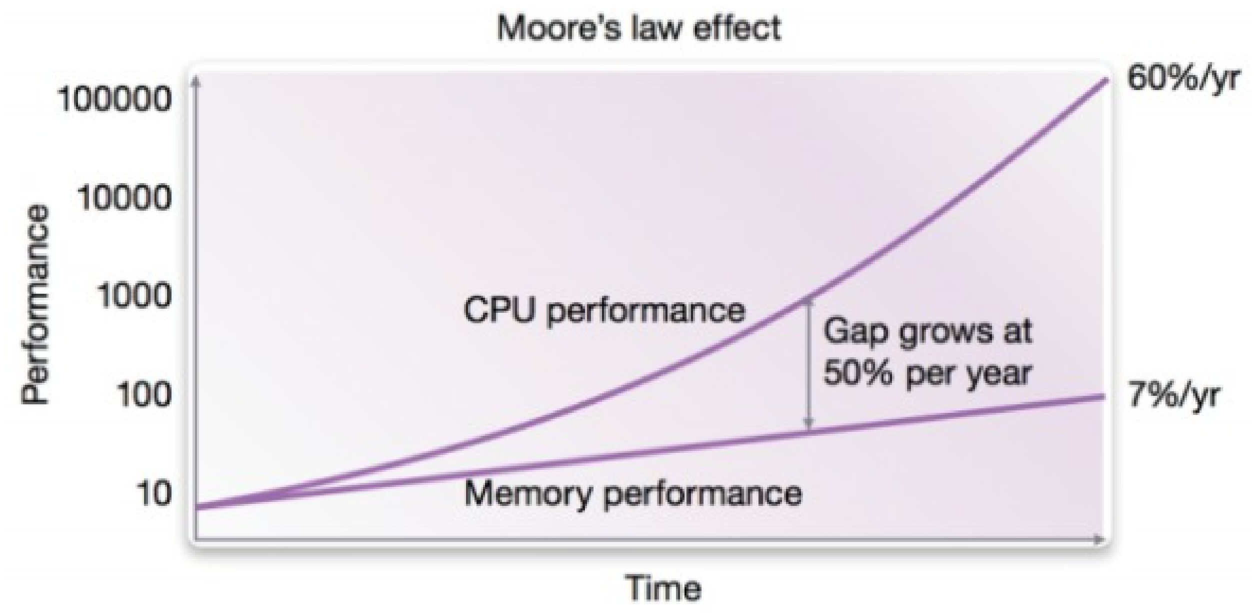


Figure 1.1: Memory Wall problem and Moore's Law. Figure is taken from [5].

Based on Dennard scaling, system performance can improve with a constant power density while maintaining the same cost in terms of power consumption and area. At the same time, we are witnessing another challenge in today's conventional computing systems. In the era of big data and with the advent of emerging data-intensive applications, an enormous amount of data is being generated across multiple areas such as health sciences, chemistry, physics, IoT, etc. Processing this huge amount of data results in frequent data movement between the memory subsystem and the processor unit, which incurs a heavy penalty in terms of both performance and energy consumption on conventional CPU-centric processing systems. Data-intensive applications also increase power and bandwidth pressures to the memory system. Based on [89], the off-chip memory which includes last-level cache, DRAM, memory controller and their interfaces can consume up to 41% of the total energy of a computer system. This energy waste which is a huge burden, limits the performance and efficiency of all modern computing systems. Tackling these challenges, novel alternative approaches (either algorithmic-level or architectural-level) can be applied. Two algorithmic-level approaches are also discussed in this dissertation (see appendix A), to demonstrate the efficacy of these methods in increasing the performance of emerging applications in the field of machine learning and deep learning [14] [15]. In the architectural-level approach, researchers have proposed *Near Memory Processing (NMP)* based on 3D-stacked memory technology that integrates processing units within memory package to offer higher memory bandwidth with lower data access latency to the processing units. NMP architecture exhibits a significant potential for performance and energy efficiency, since it reduces the aggregate need for transferring data within large memory hierarchy.

3D-stacked memory technology is one of the most promising solutions to address the *memory wall* problem in modern computing systems [151] [152]. 3D die stacking or vertical integration is an exciting path to boost the performance and extend the capabilities of modern integrated circuits. These capabilities are inherent to 3D Integrated Circuits (3D ICs). The former enhancement is due to the considerably shorter interconnecting wires in the vertical direction. It is also worth noting that vertical integration is particularly compatible with the integrated circuit design process that has been developed over the past several decades. These distinctive characteristics make 3D die stacking highly attractive as compared to other radical technological solutions that have been proposed to resolve the increasingly difficult issue of on-chip interconnect [156] [157].

Micron's Hybrid Memory Cube (HMC) [65] [4], JEDEC's High Bandwidth Memory (HBM) [2], and Samsung's Wide I/O [72] are examples of 3D integration memory technology. Figure 1.2 shows a high-level view of Micron's HMC chip architecture. This technology enables stacking multiple high capacity memory layers vertically on top of a logic tier using short and fast Through-Silicon Vias (TSVs) bus within one package and provides a massive internal memory bandwidth with lower power consumption and latency [46] [88] [40]. The logic layer can embed the processing elements. In particular, 3D die stacking technology is endorsed as the true enabler of processing near to the memory (data). It supports new opportunities by providing feasible and cost effective approaches for integrating heterogeneous cores to real-

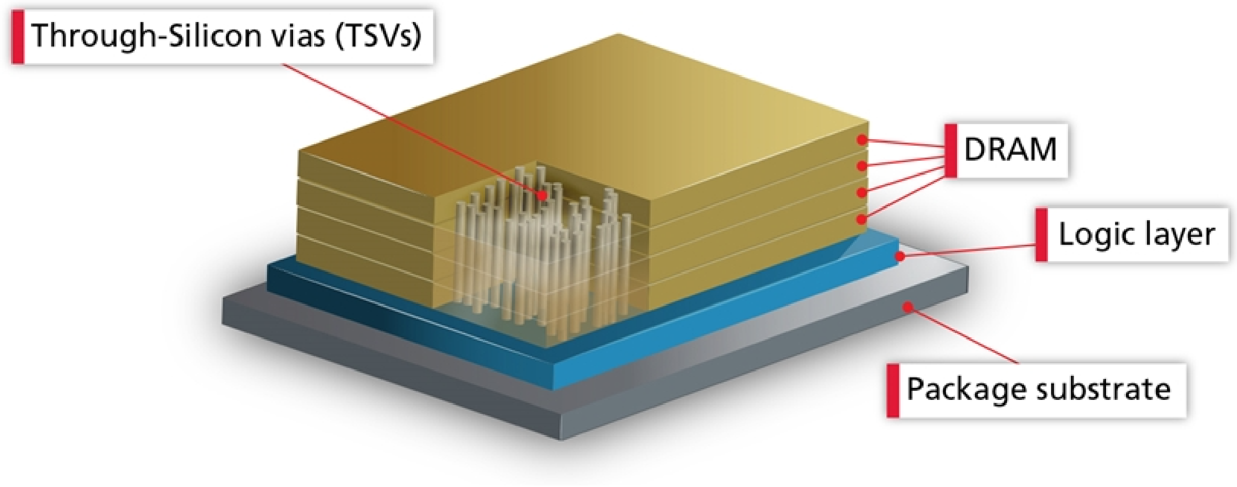


Figure 1.2: Micron's Hybrid Memory Cube (HMC) chip architecture. Figure is taken from [3].

ize future computer systems. It supports heterogeneous stacking because different types of components can be fabricated separately, and silicon layers can be implemented with different technologies. This technology can be applied to both volatile and emerging non-volatile memory technologies, which makes it more practical and beneficial to exploit the advancements of emerging memory technologies. 3D-stacked memory system provides significantly more internal and external bandwidth than conventional DDR modules while providing a high-level vault (vertical partition composed of multiple memory banks) parallelism [65]. For instance, the memory bandwidth of HBM can reach up to 450 GB/s comparing with 19.2 GB/s per channel in DDR4 (2400 MHz) [147]. This value is 160-320 GB/s for HMC [4]. This is achieved by utilizing far larger number of connections between processing elements and memory, since TSVs can be much more abundant than I/O pins.

The next promising innovation for the next generation memory systems is the use of byte-addressable cutting edge Non-Volatile Memories (NVMs). Phase Change Memory (PCM) [87], Spin-Torque Transfer Random Access Memory (STTRAM) [83], Resistive RAM (ReRAM or memristor) [149], and Ferroelectric RAM (FeRAM) [28] [102] [99] are examples of emerging

NVMs with different characteristics which are explored by researchers and manufactures for replacing DRAM at the main memory layer. These NVM technologies are attracting huge attention as the promising candidates to be used together with DRAM to architect heterogeneous memory systems (next-generation memory systems) [161] [117]. The expectation from NVM types is to provide larger capacity per chip, memory access latency and energy consumption (low-power) competitive to the DRAM technology, and better technology scaling. Across emerging NVM technologies, PCM is considered as the most mature one that can benefit from more reduction in the switching power and can scale better than DRAM technology [87] [32]. It has been reported that the PCM is expected to scale to 9nm in the near future which introduces memories with higher density that can meet the capacity requirements of many-core computing systems [42].

## 1.1 Dissertation Contributions

The focus of this dissertation is to motivate, explore, and analyze near memory processing architecture based on 3D die stacking technology (NMP) in a hybrid memory system to accelerate data-intensive problem caused by data movement (memory wall) bottleneck. This work motivates the efficiency of NMP subsystems when 3D-NVM technology is employed.

With this goal in mind, the contributions of this dissertation can be summarized as follows:

- To redefine and investigate a set of NMP-centric performance metrics with the focus of application characterization on conventional Host CPU system and NMP architecture.
- To perform a systematic and comprehensive characterization based on Roofline analysis, data locality (temporal and spatial) analysis, and memory access behavior analysis for various sets of multi-threaded applications (mixture of compute-bound and memory-bound) as a case study in order to analyze the efficacy of mapping a processing

unit to a specific application and evaluate the potential benefits of NMP architecture over conventional Host CPU to efficiently accelerate data-intensive problems.

- To explore two NMP subsystems based on different 3D-stacked memory technologies (3D-DRAM and 3D-PCM) and to analyse the impact of constructing NMP architecture based on an emerging NVM technology.

- To demonstrate that executing certain data-intensive (memory-intensive) applications on NMP architecture based on 3D-NVM can improve performance and reduce memory power consumption compared to 3D-DRAM based NMP and conventional Host CPU executions which explains the benefits of processing near hybrid memory system.

## 1.2 Dissertation Organization

The rest of this dissertation consists of five Chapters and one Appendix section. This dissertation is organized as follows:

- After the introduction to the research background of this work in Chapter 1, Chapter 2 discusses application and technology trends that motivate new computing devices (3D-based NMP architectures) for data-intensive applications. Chapter 3 provides relevant background and surveys related work.

- Chapter 4 describes the methodology used for application characterization with the focus on NMP and how to leverage the proposed metrics to evaluate three studied processing platforms. Chapter 5 presents an envisioned NMP architecture in hybrid memory systems used to process data-intensive applications in an efficient way. The evaluation methodology and experimental platforms are explained in Chapter 5. Data analysis and results are presented at the end of this Chapter. Having reviewed all of the Chapters, the conclusions and future work are given in Chapter 6. After this, there are the references of all the Chapters including Appendix section.

- Finally, the last portion of this dissertation (Appendix A) is based on research work which I previously published with my colleagues. This appendix discusses two algorithmic-level approaches to accelerate emerging applications in the field of machine learning and deep learning.

# Chapter 2

# Motivation

Almost all computing systems are built to efficiently support processing various software applications by leveraging modern technology innovations. To motivate research on near memory processing architectures for data-intensive applications, this Chapter explores emerging data-intensive applications and technology trends, as well as the conventional processing system architectures and their inefficiencies.

## 2.1 Emerging Data-Intensive Applications

As we go deeper into the “big data” age, we witness a huge growth in the volume, and variety of data available around us. It has been reported that data is growing faster than before and at the end of 2020, about 1.7 megabytes of new information was created every second for every human being on the planet. Also, “with 75 billion IoT-connected devices (all generating data) expected by 2025, there will be no shortage of data to analyze”. Such “data explosions” which is growing significantly faster than Moore’s law has led to emergence of data-intensive applications. These emerging applications including graph analytics, data mining, machine learning, augmented reality, and deep learning scan through a massive datasets within careful time limitations in order to extract meaningful and compact knowledge. For example, an object classification algorithm in an augmented reality application typically uses millions of example images and video clips for training the network and performs classification on real-time high-definition video streams [61].

The common characteristics of data-intensive applications are summarized as follows:

- Highly parallel applications with opportunities to exploit data-level and thread-level parallelism.
- Poor performance on conventional multi-level and large cache hierarchies because of limited data locality (temporal and/or spatial locality) with a huge diversity in computation patterns.
- Variety of challenging memory access pattern. Irregular memory access pattern (frequent random memory accesses) for graph processing applications. Sequential and stride memory accesses for deep neural networks compute on multidimensional arrays [12].

To accelerate data-intensive problems, some architectural requirements are needed which are summarized as follows:

- Support for high parallelism considering various memory access patterns (random, sequential and stride data accesses).
- High performance computing (low latency) with low power/energy consumption, and low area overhead.
- Support for high bandwidth, low latency, low energy, and low area overhead from memory system.

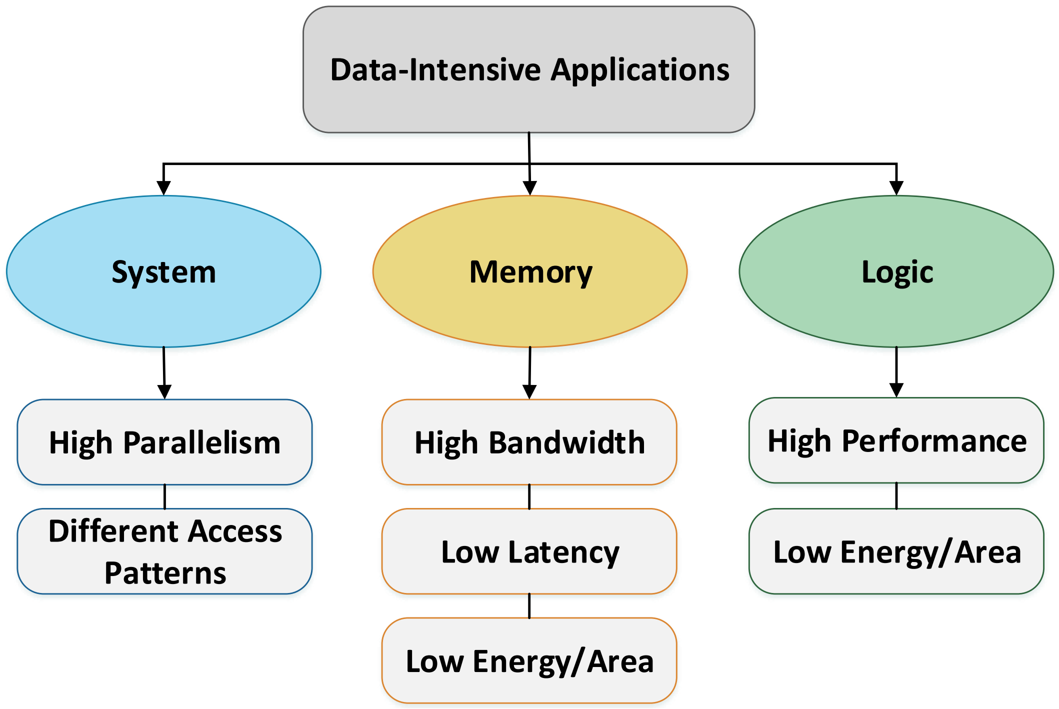


Figure 2.1: Architectural requirements for data-intensive applications.

Figure 2.1 summaries the discussed architectural requirements for data-intensive processing.

## 2.2 An Overview on Memory Technologies

In this section, most popular memory technologies are shortly illustrated which are (or can be) utilized as main memory in computing systems. These technologies are categorized in volatile and non-volatile memories.

### 2.2.1 Volatile Memory Technology

In volatile memories, the stored data is erased when the power supply is disconnected from the memory which makes the system susceptible to data loss due to power outage. Some

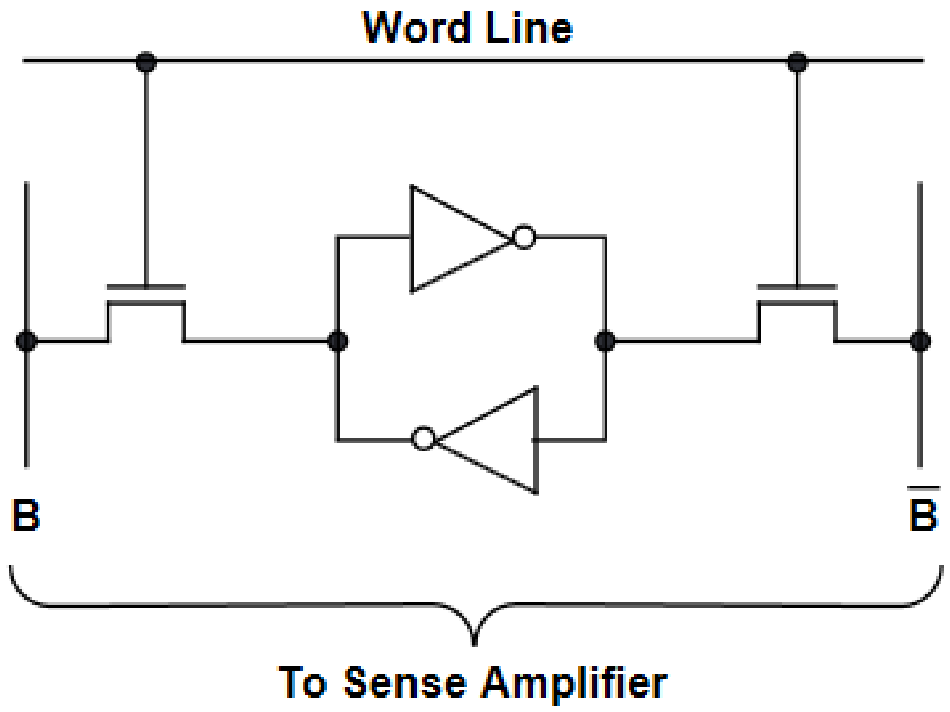


Figure 2.2: The 6T SRAM cell structure: Access to a cell is enabled by the word line (also called row) which controls the two access transistors in the cell. The bitlines (also called columns), $B$ and $\overline{B}$, are used to transfer data for both read and write operations.

modern memory technologies, like Dynamic RAM (DRAM) that store data for few milliseconds (in room temperature) after power disconnection, and Static RAM (SRAM) which is typically used for the cache and internal registers of a processor, are categorized in volatile memories.

**Static Random Access Memory (SRAM)**

SRAM is a major component of many digital systems. It is designed to provide a direct interface with the processor at a fast speed. Fast access times and design for high density are the most important features of this memory technology for many years. This technology uses bi-stable latching circuitry to store data. Each SRAM cell consists of a bi-stable flip-flop

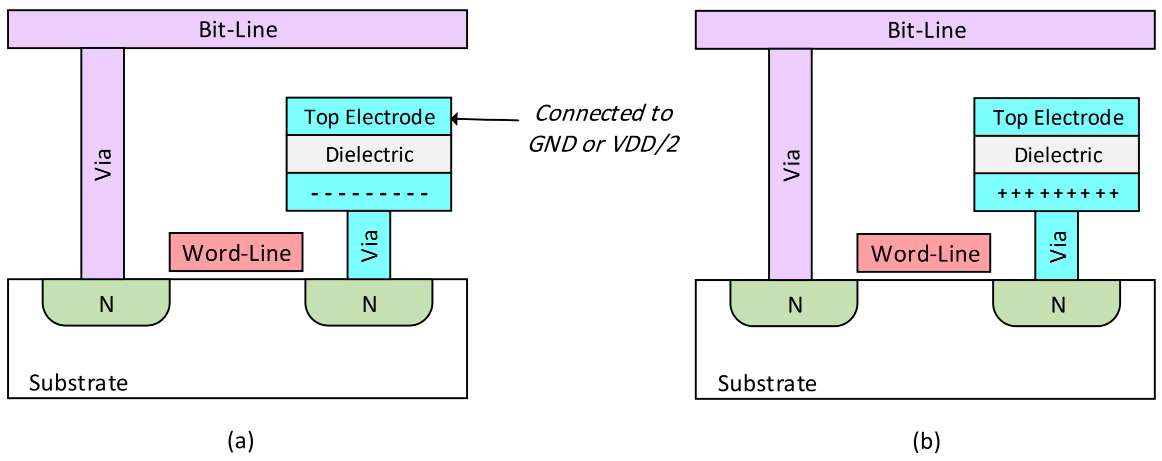


Figure 2.3: The DRAM cell structure: Stored charge in the tiny cell capacitor is used to store values. (a) Negative and (b) positive stored charge in the cell represents '0' and '1', respectively.

which is connected to the internal circuitry using two access transistors. The structure of a 6T SRAM cell is shown in Figure 2.2. Access to the SRAM cell is enabled by the word line. To select a SRAM cell, the access transistors should be "on", so the flip-flop can be connected to the internal circuitry. When the cell is not addressed, the access transistors are closed (off) and the data is latched within the flip-flop. The flip-flop needs power supply to hold the information. There is no refresh cycle in SRAM, since data does not leak away.

**Dynamic Random Access Memory (DRAM)**

DRAM is the mainstream memory technology used as main memory in computing systems for decades which plays an essential role regarding the efficient data access. This is because of low fabrication cost, low latency, maturity of technology, acceptable density and power consumption, and very high endurance ($> 10^{15}$). The structure of a DRAM cell is shown in Figure 2.3. An access transistor connects corresponding bitline to the cell capacitor during memory access. The stored charge in the cell capacitor represents the stored value. During cell access, the stored charge, negative or positive (Figure 2.3a and Figure 2.3b, respectively), is shared with the charge in the bitline and makes a voltage perturbation in the bitline which can be detected by sense amplifier [64].

The main drawbacks of DRAM technology are physical limitations in scaling, low density (since only one bit can be stored in a cell), need for refresh, and leakage power. DRAM scaling affects all of its major characteristics such as capacity, latency, bandwidth, and cost. Furthermore, access to a cell in DRAM is destructive, thus a power-hungry recovery phase should be performed right after a cell access. The mechanism of periodic refresh in DRAM technology incurs power consumption even if there is no activity in the memory. Also, the background power which is related to the peripheral components is another concern for this memory [49].

### 2.2.2 Non-Volatile Memory (NVM) Technology

Regarding the limitations of DRAM, i.e., scalability, density, and static power dissipation, many research work are conducted to propose a proper replacement for DRAM as main memory and last-level caches in computing systems. Between different storage technologies, Non-Volatile Memories (NVMs) are expected to be the most promising ones for replacing DRAM. These memories are named as storage class memories (SCM) and are trying to fill in the latency gap between main memory and disk. They can scale better than DRAM at smaller feature size that can satisfy the memory capacity and high density requirements of multi-core systems. As an example, PCM is expected to scale to 9nm around 2022 [105]. Unlike DRAM that stores data in the form of charge, an NVM cell uses resistance to store the data. Endurance and density have been conventional limitations in NVM technologies, but recent technology trends encourages that these constraints can be addressed [118]. The main advantages of NVMs are very low static power and smaller cell size than DRAM cells.

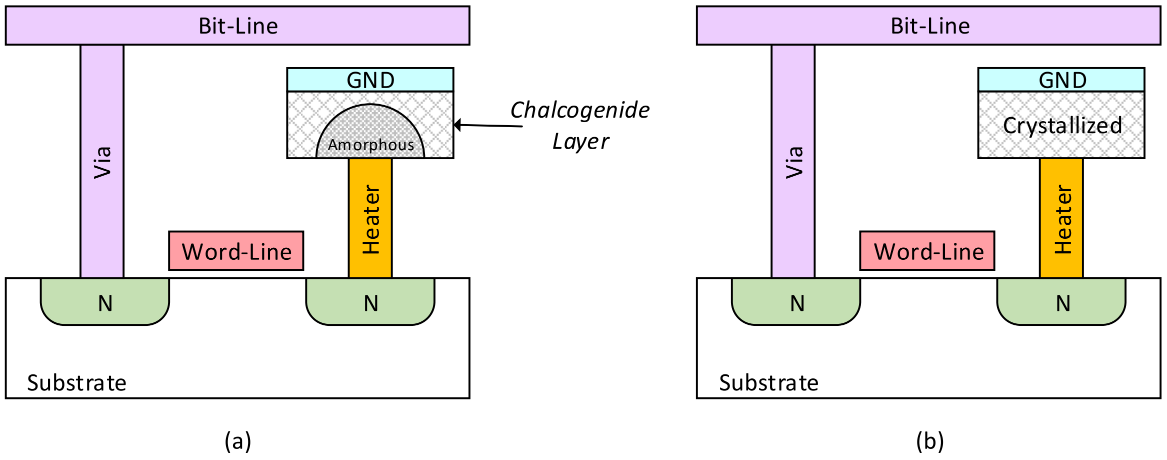


Figure 2.4: The PCM cell structure: A heater (resistor) and chalcogenide material are the main components of the PCM cell. (a) High resistance of the chalcogenide layer in the amorphous state, (b) Low resistance of the chalcogenide layer in the crystallized state.

Since each NVM technology has superiority and weaknesses in comparison with the others, there is still no definite winner. Between the NVM technologies, PCM, STT-RAM, ReRAM, and FeRAM are the most promising memory technologies which are shortly investigated in this section.

**Phase Change Memory (PCM)**

PCM, also called PRAM, was first proposed by Gordon Moore in 1970 [109]. This technology is constructed of a chalcogenide alloy layer and a heater that can be accessed through an access transistor in a two-dimensional bitlines and wordlines array. By applying an electric signal to the heater, the generated heat changes the alloy layer to an amorphous/crystallized state based on the pulse shape. A short high-temperature (600 C) pulse resets the chalcogenide layer to the amorphous state and a long low-temperature (300 C) pulse sets this layer to the crystallized state [82]. Figure 2.4 shows a PCM cell in amorphous (a) and crystallized (b) states. The resistance of the chalcogenide layer increases/decreases when it is in the amorphous/crystallized state and these physical states can be used to store values. Since the resistance of the alloy in the amorphous state is three to four orders of magnitude higher than its resistance in the crystallized state, multiple bits can be stored in one PCM cell, which is called Multi-Level Cell (MLC) PCM [163] [102].

The main challenges with PCM are alloy resistance drift, endurance, write disturbance error, read disturbance, and high write latency. The crystallized state of the alloy returns to the amorphous state over time which may lead to a read error. A PCM cell may only tolerate about $10^8$ to $10^{15}$ writes, making it essential to apply wear-leveling techniques to use it as main memory. The generated heat to write on one cell can propagate to the adjacent idle cells and change their states over time. This is while generated heat due to applying a higher voltage to a cell for turbo read access can lead to bit-flip over time in that cell. Unbalanced read and write latencies (about 20 ns and more than 150 ns, respectively) need a more complex memory management unit. Besides, the memory performance drops for write-intensive applications. However, the controllable data retention, high density, robust performance, and fast read access are the outstanding advantages of this memory [119] [102] [73].

### Spin-Transfer Torque RAM (STT-RAM)

In STT-RAM, Magnetic Tunneling Junction (MTJ) phenomenon is utilized to store data. An STT-RAM cell is composed of two ferroelectric layers stacked on top of each other which can be accessed through an access transistor. The structure of the STT-RAM cell is shown in Figure 2.5. The polarization of one of these layers is fixed, while the polarization of the other one can be changed by a higher current passing through the access transistor. When both ferroelectric layers are in the same direction (Figure 2.5 a), the resistance of stacked layers is minimum, and its resistance increases when the free layer orientation is opposite of the fixed layer (Figure 2.5 b) [102] [101] [81].

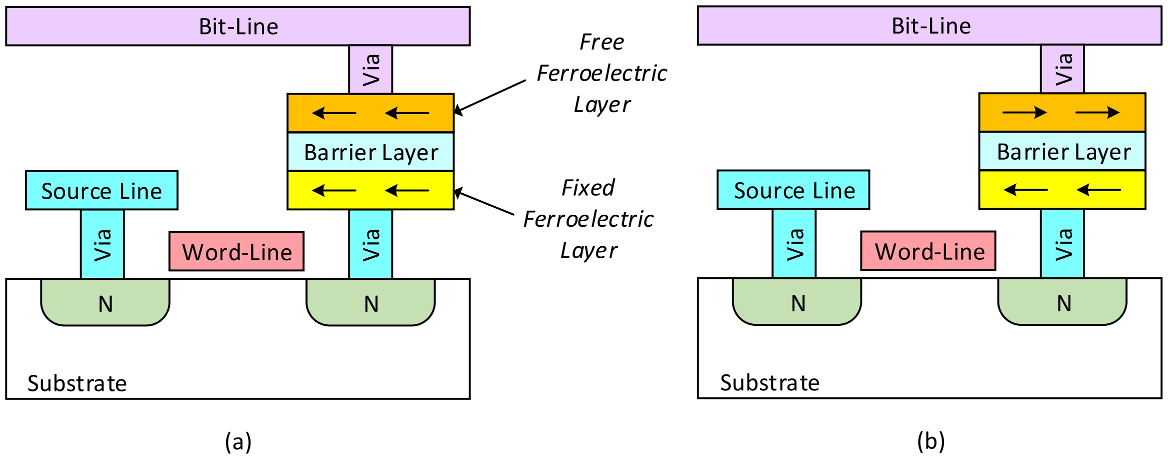


Figure 2.5: The STT-RAM cell structure: (a) The same polarization direction on free and fixed ferroelectric layers leads to the low-resistance state of stacked ferroelectric layers, (b) Different polarization directions leads to the high-resistance state.

Read disturbance, write errors, and lower density are the main limitations of STT-RAM. By shrinking the STT-RAM cell size, read current approaches the write current which can lead to a bit-flip in a cell in the case of consequent read accesses. The stochastic characteristic of MTJ during the free layer polarization change is another source for the faulty write operation. The main advantage of STT-RAMs over other NVMs is a higher endurance.

**Resistive RAM (ReRAM)**

ReRAM technology goes back to 2003 and it is composed of a stacked metal-insulator-metal that its resistance can be controlled by applying voltage through access transistor. This structure stores data by changing the trapped ions in the insulator. The higher aggregation of ions in the insulator (metal oxide layer) leads to lower Ohmic resistance between top and bottom electrodes stacked on top of each other. The ions are generated by applying sufficient high voltage to the stack. Excellent scalability, long endurance, and CMOS compatibility are the main advantages of ReRAM. However, many companies are reluctant to fabricate

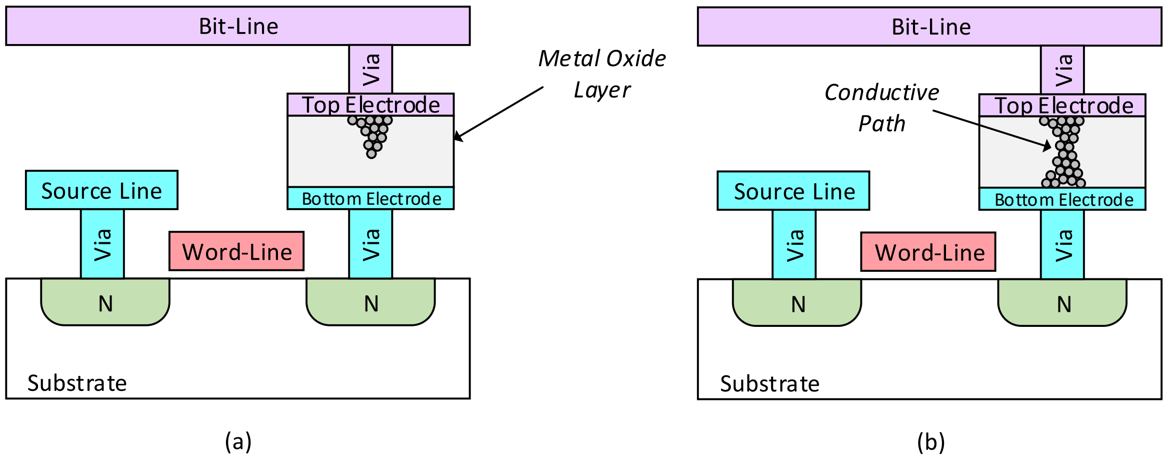


Figure 2.6: The ReRAM cell structure: (a) The conductive path inside metal oxide layer, generated by applying voltage to the top and bottom electrodes, determines the set state of ReRAM, and (b) the reset state.

ReRAM chips due to high-cost and complex etching process as its main drawbacks of this memory technology [34] [98] [154]. Figure 2.6a and Figure 2.6b illustrate the set state and reset state of a ReRAM cell, respectively.

### Ferroelectric RAM (FeRAM)

In FeRAM, the hysteresis characteristic of the ferroelectric material as the dielectric in the cell capacitor is used to store data. The structure of FeRAM is very close to DRAM and is presented in Figure 2.7. For a write operation, a higher voltage pulse (than a read access) is applied through the access transistor to the cell capacitor to modify the crystal orientation of the ferroelectric material for program/erase operations. The direction of applied voltage, to Bit-Line and Plate-Line, determines the stored value in the cell [139] [70]. Figure 2.7a and Figure 2.7b show positive and negative polarization directions, corresponding to '1' and '0', respectively.

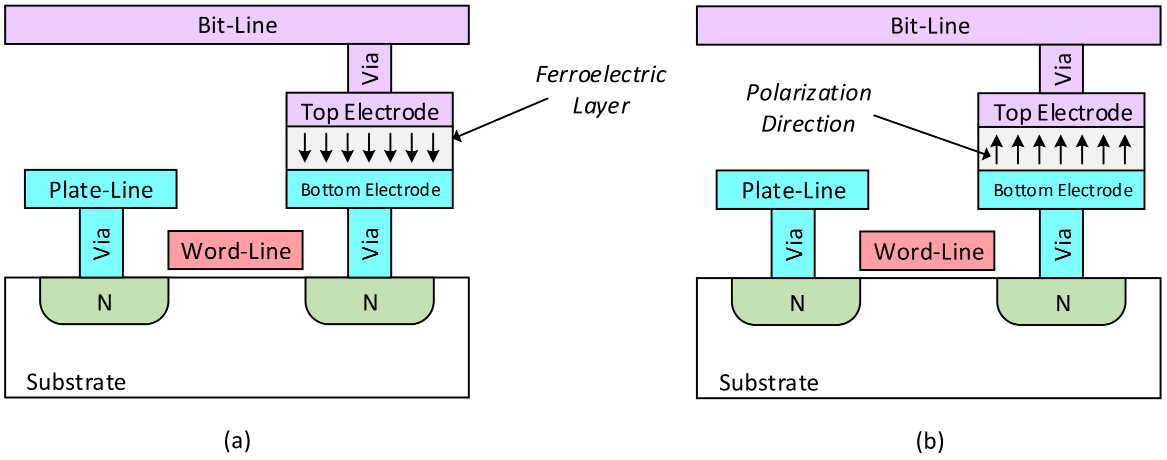


Figure 2.7: The FeRAM cell structure: The applied voltage to top electrode and bottom electrode can change the polarization of ferroelectric layer to positive (a) and negative (b) polarization.

Beside the advantages of FeRAM such as low power consumption, high endurance (up to $10^{13}$), and high bandwidth read/write operations, the main disadvantage of FeRAM is that the read memory access is destructive. Thus, read access should be accompanied by a restoration phase.

Table 2.1 compares traditional volatile SRAM and DRAM technologies with a few popular emerging NVM technologies, including PCM, STTRAM, ReRAM, and FeRAM. Across all NVM types, their low leakage power is noted and it is because of not requiring any data refresh mechanism to keep written values in memory. Among all NVMs, STTRAM is the fastest in terms of access speed, while its cell area is larger. PCM, ReRAM, and FeRAM can store multiple bits in a memory cell which indicates their superior density with technology scaling. Furthermore, they inherently support parallel processing of data which is useful for data-intensive applications. In this work, DRAM and PCM as the representatives of volatile and non-volatile memory technologies are selected to enable near memory processing in a hybrid 3D memory system.

Table 2.1: Different memory material comparison [100] [91] [27]

| Metrics | SRAM | DRAM | PCM | STTRAM | ReRAM | FeRAM |
|---|---|---|---|---|---|---|
| Leakage Power | High | Medium | Low | Low | Low | Low |
| Cell Size ($F^2$) | $> 100$ | $6 \sim 10$ | $4 \sim 12$ | $6 \sim 50$ | $4 \sim 10$ | $6 \sim 40$ |
| Multibit | No | No | Yes | No | Yes | Yes |
| Access Granularity | Bit-level | 64 Byte | 64 Byte | 64 Byte | 64 Byte | 64 Byte |
| Read Latency (ns) | $0.2 \sim 2$ | $20 \sim 50$ | $20 \sim 50$ | $2 \sim 35$ | $20 \sim 50$ | $20 \sim 80$ |
| Write Latency (ns) | $0.2 \sim 2$ | $20 \sim 50$ | $50 \sim 150$ | $3 \sim 50$ | $\sim 50$ | $50 \sim 75$ |
| Write Energy (J/bit) | $\sim > 10^{-15}$ | $\sim > 10^{-14}$ | $\sim > 10^{-11}$ | $\sim > 10^{-13}$ | $\sim > 10^{-13}$ | $\sim > 10^{-13}$ |
| Write Endurance | $10^{16}$ | $> 10^{15}$ | $10^{8} - 10^{15}$ | $> 10^{15}$ | $10^{8} - 10^{12}$ | $10^{14} - 10^{15}$ |
| Maturity | Mature | Mature | Test chips | Test chips | Test chips | Manufactured |

## 2.3 Conventional Computing Architectures

In conventional computing systems based on Von Neumann architecture (shown in Figure 2.8) usually memory hierarchy consists of multiple levels of caches, the main memory and the storage. In this computing system, the processor is interfaced with a memory that holds both instructions and data. In this organization, data shuttles back and forth between the off-chip memory and the processing unit. This data movement has been revealed as a crippling performance and energy bottleneck for rising data-intensive applications such as media processing, data mining, computer vision, graph analytics, machine learning, deep learning, and etc. It is noticeable that these bottlenecks will become more significant in the future, since technology scaling will not help. From the processor side, various architectural techniques such as Pipelining, Superscalar, and VLIW (very long instruction word) are employed to increase the parallelism and hide the off-chip data access latency as much as possible. Pipelined processors provide parallelism with smaller clock cycle time. However, there is overhead in pipelining, both in terms of performance (extra delay interfacing with pipeline latches) and area. Both Superscalar and VLIW processors exploit instruction-level parallelism (ILP) by issuing more than one instruction at each clock cycle with a different method for instruction scheduling. In VLIW model, the complexity is moved to the compiler level with static scheduling of instructions and superscalar processor uses dynamic scheduling at run-time which makes the hardware more complex compared to VLIW model. All of these techniques may not be sufficient to provide continued demand in performance from emerging applications.

On-chip SRAM caches, which are typically used in conventional processors, have high leakage power and moving data across the large cache hierarchies consumes significant dynamic energy. Therefore, novel alternative approaches are required to address these inefficiencies. Based on [60], the energy overhead of accessing data from memory systems and moving data across memory hierarchy dominates the cost of arithmetic operations. The huge cost of

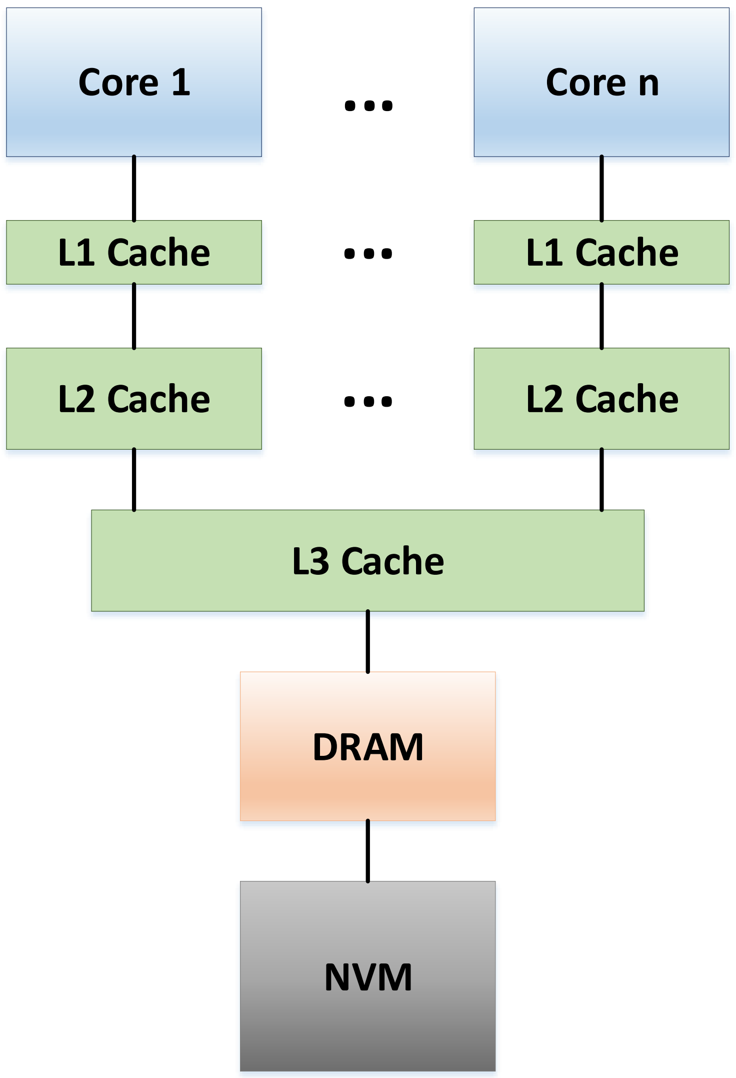


Figure 2.8: Conventional multi-core computing system architecture based on a CPU-centric approach where data is moved to the core for processing. In this architecture, DRAM as the predominant data storage technology is used to build main memory.

data access and data movement which dominate the total cost of computation (in terms of performance and energy) forces architects to reevaluate the fundamental design of computing systems. In particular, this huge cost is a major problem for data-intensive applications that have poor data locality to payoff the high overhead.

As data-intensive applications become more widespread, conventional computing architec-

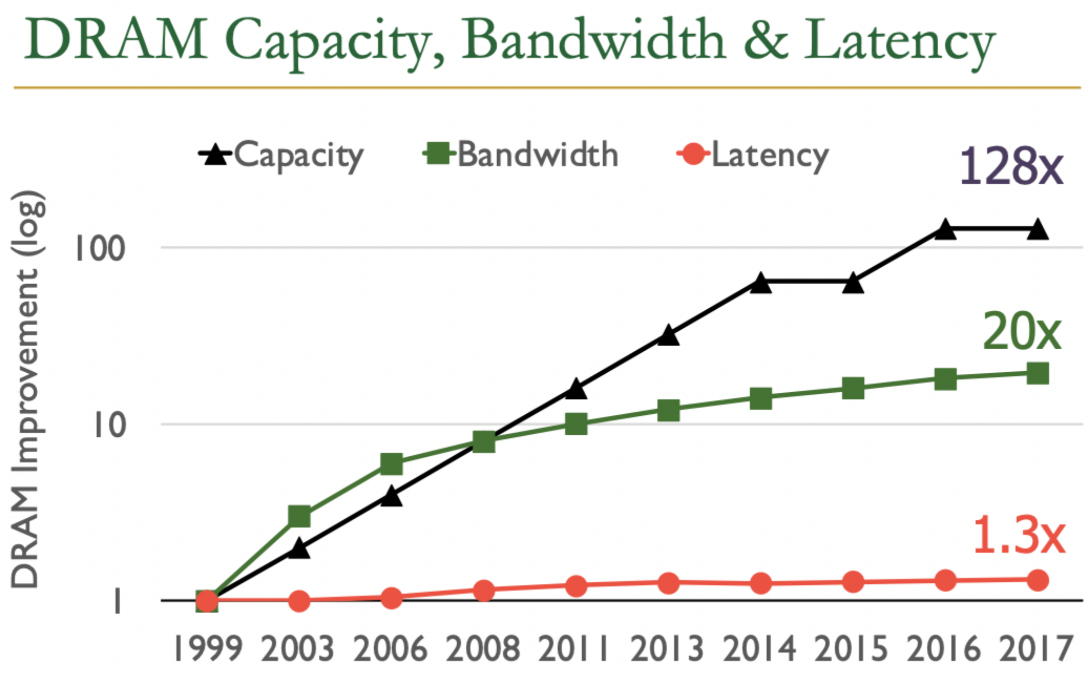


Figure 2.9: DRAM improvements in terms of capacity, bandwidth, and latency over two decades (from 1999 to 2017). Figure is taken from [106].

tures are not able to satisfy systems requirements of these applications. Thus, the need to bring processing closer to the data (memory) will arise.

Following architectural needs for data-intensive applications in Section 2.1 (see Figure 2.1), here I discuss the memory, logic and system inefficiencies of conventional computing systems when dealing with data-intensive applications:

- Conventional computing system architectures with multi-core processors use off-chip DRAM modules in their memory system. Huge data movement between computing unit and off-chip memory in a long-distance results in high access latency with a significant energy consumption. Further, it is also difficult to improve the bandwidth between the processor and memory chip. Based on [106], between 1999 and 2017 (as shown in Figure 2.9), while DRAM capacity and bandwidth has improved by 128x and 20x, respectively, its latency has remained almost the same which makes it a major performance bottleneck for many emerging applications. Moreover, today's bandwidth demands of multi-core processor cannot be satisfied by the memory package. It is worth mentioning that large cache hierarchies cannot reduce the discussed memory inefficiencies. Large multi-level cache hierarchies reduce the off-chip data access latency and energy consumption by leveraging the data locality in the application access patterns. Unfortunately, most data-intensive applications do not exhibit data locality to exploit large cache hierarchies exist in conventional system architectures. Therefore, the off-chip data accesses cannot be reduced and as the result, main memory system must be optimized.

- Multi-core general-purpose processors in conventional computing systems support processing different types of operations with significant energy and area overhead. So, they may not be adequate to provide increase in performance continuously. It has been reported that for a single in-order RISC-V processor, 43% of the total energy consumption is because of programmability support. It has been also noted that this overhead is much higher in out-of-order processors [50] [57]. Furthermore, scaling these systems above hundreds of cores to support extensive data processing with coherent cache hierarchies is costly and limits the system performance.

# Chapter 3

# Background and Related Work

This Chapter reviews the background and related work in 3D die stacking memory technology, processing using memory, processing in memory and near memory processing as three different architectural-level techniques proposed to accelerate data-intensive problems (latency and energy) caused by *memory wall* bottleneck in conventional computing system architectures. The presented research contributions in the following chapters is built upon the key insights of previous work in these areas.

## 3.1 3D-Stacked Memory Technology

Silicon wafer or die stacking is a promising solution to reduce interconnections signals length and chip area. The stacked dies/wafers are interconnected to each other using short-length and low-parasitic capacitance Through-Silicon Vias (TSVs) [104]. Beside numerous benefits of stacking dies such as increase in capacity and bandwidth, increasing power density in the stacked layers limits the number of possible stacked layers. However, since memory dies generally have low power density and occupy a large area, 3D stacking of memory dies has

Table 3.1: Approximate Device Level Characteristics of DRAM and PCM [161] [100] [33] [91]

| **Characteristics** | **DRAM** | **PCM** |
|---|---|---|
| Standby Power | Refresh Power | Very Low ($\sim 0$) |
| Leakage Power | Medium | Low |
| Cell Size ($F^2$) | $6 \sim 10$ | $4 \sim 12$ |
| Access Granularity | 64 Byte | 64 Byte |
| Read Latency | 20ns $\sim$ 50ns | 20ns $\sim$ 50ns |
| Write Latency | 20ns $\sim$ 50ns | 50ns $\sim$ 150ns |
| Read Energy | Medium | Medium |
| Write Energy | Medium | High |
| Write Endurance | $> 10^{15}$ | $10^8 - 10^{15}$ |
| Power Consumption | Very High | Medium |
| 3D die stacking Capability | Yes | Yes |
| TSV Power Saving | Less | More |
| Maturity | Mature | Test chips |

considerable benefits and is less challenging.

Fortunately die staking is applicable to almost all of the mentioned NVM technologies in Section 2.2, as well as DRAM technology. In this dissertation, *3D-DRAM* and *3D-PCM* are considered for main memory, as the representatives of volatile and non-volatile memory types in the evaluations. It is obvious that the other memory types (such as ReRAM, FeRAM, and etc) can be considered as main memory as well, but they are left for the future work.

Table 3.1 introduces some of the device-level characteristics of DRAM and PCM technologies according to literature [27] [159] [162]. The reported numbers are representative (not the best or the worst cases). It should be noted that in general at system-level, the interconnect used for memory access has a considerable effect on the memory metrics. Based on Table 3.1, PCM as an emerging NVM technology suffers from a very high write latency/energy comparing to DRAM technology. It provides desirable properties such as satisfactory read latency (comparable to DRAM), very low (close to zero) standby power, no refresh power (it does not require any refresh mechanism to keep written values in memory), superior scalability, CMOS process compatibility, higher memory capacity for the same chip area, 3D die-stacking capability, and more benefit from TSVs in terms of power saving in the 3D structure [95] [161]. In die-stacked PCM design with TSVs, the resistance is reduced due to the short-length wires which results in saving programming power. Based on [162], PCM can achieve less than $4F^2$ through 3D integration. In the context of trends such as 3D die stacking, multi-core, and improved networking, PCM technology can inspire more crucial architectural change for data-intensive processing than conventional approaches that use such memory technology as storage in the memory hierarchy [118].

## 3.2 Processing Using Memory (PUM)

The reason behind large amount of data movement is due to the heavily processor-centric design approaches. Eliminating or reducing this massive data movement is crucial to make computing systems high performance and energy-efficient [106]. *Processing Using Memory (PUM)* which is an architectural-level approach utilizes the intrinsic properties and existing peripheral circuits in memory cells to perform widely-used operations. There are many research work in this area which indicate possibility of this approach using different memory technologies such as Static-RAM, DRAM, PCM, and ReRAM [44] [92] [67] [127] [90]. PUM architectures enable a wide range of operations, such as bulk bit-wise operations and simple arithmetic operations, within memory cells with minimal changes [30] [107] [130] [19] [93] [128].

## 3.3 Processing In Memory (PIM)

The cost of moving data in an application continues to increase significantly as applications process larger amount of data. *Processing In Memory (PIM)* chip that integrates processing logic into memory devices provides an opportunity to eliminate unnecessary data movement by bringing part of the computation into the memory, specially for applications with high memory bandwidth demands. Start of PIM architecture proposals goes back to 1960s. Logic-in-Memory computer is one of the earliest PIM architectures [137]. In this project, small processing elements are combined with small amount of RAM to perform computation within memory array. Approximately two decades ago (late 1990s and early 2000s), several research studies continued investigating the integration of processing logic, which ranges from simple cores to accelerators and FPGAs, and DRAM (or embedded DRAM) modules on a single chip [53] [56] [68] [76] [112] [22]. In this type of architecture, a host processor was connected to the PIM chip with a custom interconnect. Xi et al. in JAFAR project [153] includes an accelerator in a DRAM module to implement the select operator. This implementation only allows qualifying data to travel up to the host CPU. Alien et al. in MCN project [16] integrates a lightweight processor with a buffer device on DRAM DIMM to enable processing in memory for data-intensive applications. The integrated processor runs a simple operation system with network software layers for running a distributed computing framework. Although it was reported that there was potential for a significant speedup in some classes of applications (e.g, image processing, machine learning, and graph processing). There was a limited success on the past PIM projects and the major reason comes from additional cost (integrating logic and DRAM module) and density shortcoming of 2D chips.

## 3.4 Near Memory Processing Based on 3D Stacking (NMP)

The most recent and promising innovation that can provide continued scaling of performance is the ability to stack multiple memory layers on a multi-core processor die. In 3D-stacked memory (e.g., HMC and HBM), a logic layer and multiple memory layers are stacked vertically on top of each other using short and high bandwidth TSVs. TSV-based interconnection provides a low latency and energy efficient data transfer between logic layer and memory layers. Currently, this memory technology provides an opportunity to architects to embed a wide range of computational logic in the logic layer considering the area, energy, and thermal dissipation constraints. These benefits can potentially improve system performance and energy efficiency in a practical manner, but only with careful design of NMP architectures.

It is reported that the 3D-stacked package can communicate with a maximum bandwidth up to 320GB/s with internal memory layers through TSVs and external units through high bandwidth links [4]. Unfortunately, today's processors are not capable of taking full advantage of the improvements offered by the 3D memory technology. NMP systems enabled by 3D-stacking can address one of the major reasons for the limited success of previous PIM projects. This technique avoids additional cost of integrating processing cores with DRAM on the same chip.

NMP systems are the biggest opportunity for emerging data-intensive applications. Such applications scan through massive datasets with a very low temporal locality. As a result, they cannot benefit from large and multi-level cache hierarchies and thus waste memory bandwidth and energy.

Figure 3.1 depicts an abstract view of a system that is capable of processing close to memory in which the NMP subsystem is connected to the Host CPU through high-speed links.

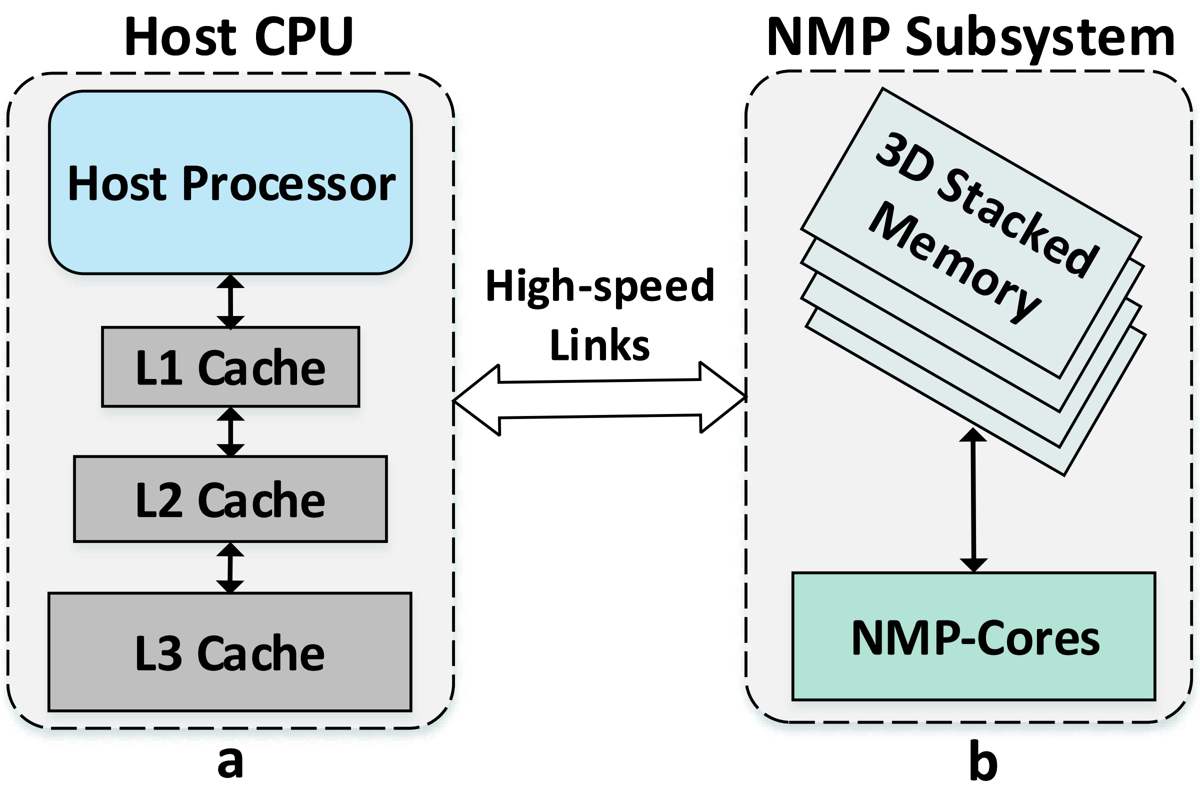


Figure 3.1: The overall architecture of a system with NMP capability. An application can run on the Host CPU system as in the conventional manner, or it can be offloaded to the NMP subsystem in which data can be accessed more efficiently.

Host CPU can offload kernel to the NMP subsystem. NMP transfers data through high-bandwidth, low-latency, and low-energy 3D interconnects between memory layers and processing cores in the logic layer. The NMP subsystem (Figure 3.1.b) consists of a 3D processor-memory architecture, in which processing cores are embedded in the logic layer and memory layers are stacked vertically on top of it. Figure 3.2 illustrates a conceptual view of a NMP architecture which is based on 3D stacking. In this architecture, the logic layer composed of multiple vault logic which are connected to each other through an interconnect network such as Network-on-Chip (NoC). NoC is the dominant communication infrastructure which provides a scalable efficiency in hardware area and power [129] [156]. The memory layer is divided into multiple vertical partitions called vaults in which each vault has its own memory

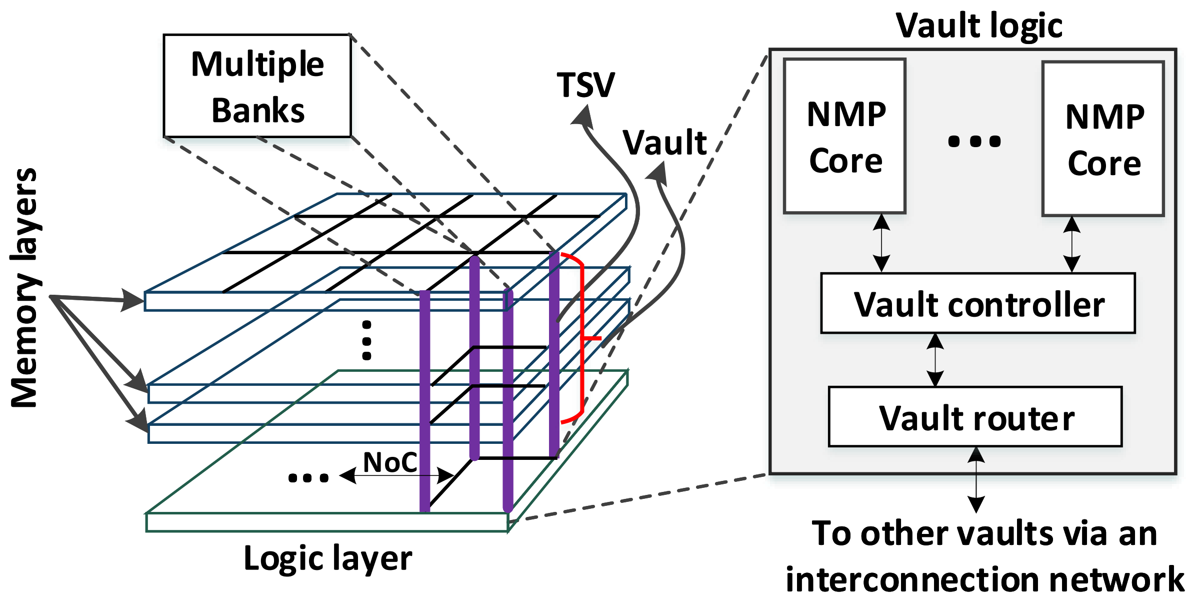


Figure 3.2: Conceptual view of a NMP architecture based on 3D die stacking. The most bottom layer which is called logic layer can embed processing cores. Each processing unit can utilize high-bandwidth, low-latency, and low-power TSV connection to access data in memory with higher internal bandwidth.

controller in the logic layer. Each of these vertical vaults can be accessed in parallel as they have independent processing cores and memory controllers in the logic layer.

There are several research work on integration of the computation unit to the logic layer of 3D-stacked DRAM. Zhang et al. [160] proposed to integrate programmable GPUs to the logic die of 3D-DRAM to offer high throughput. Pugslet et al. [55] created a near data computing architecture for MapReduce workloads. In this work, a host processor is connected to many daisy-chained 3D-stacked DRAM devices with energy-efficient processor cores in their logic layer. Gao et al. [51] proposed a practical near-data processing architecture for in-memory analytics frameworks where a high-end host processor with out-of-order cores is attached to multiple 3D-stacked memory devices (e.g., HMC). In this work, near-data processing cores are responsible for executing the portions of applications with a very low temporal locality, and host processor is responsible for executing the portions of applications with a significant temporal locality. Taeho et al. [71] proposed PicoServer which employs 3D memory technology to build energy efficient servers. This work targeted server workloads and key-value store which are not considered as memory-intensive. The Active Memory Cube which is a processing near memory architecture embeds a set of processing units in the logic layer of a 3D-DRAM. In this project, the tuned instruction set architecture and microarchitecture of the processing units support vector processing in common scientific applications and low power requirements of exascale computing systems [108]. However, to the best of the our knowledge, this dissertation is the first to study a 3D-stacked NMP architecture based on an emerging non-volatile memory technology (PCM).

## 3.5 Summary

This Chapter reviewed and organized the literature related to the 3D die stacking technology and novel area of processing close to where data resides. Conceptually, this memory-centric approach which includes processing using memory, processing in memory, and processing near memory can be applied to any type and level of memory to improve the overall system performance. Figure 3.3 concludes this Chapter by illustrating a high-level view of classification for processing options which is based on the level in the memory hierarchy.

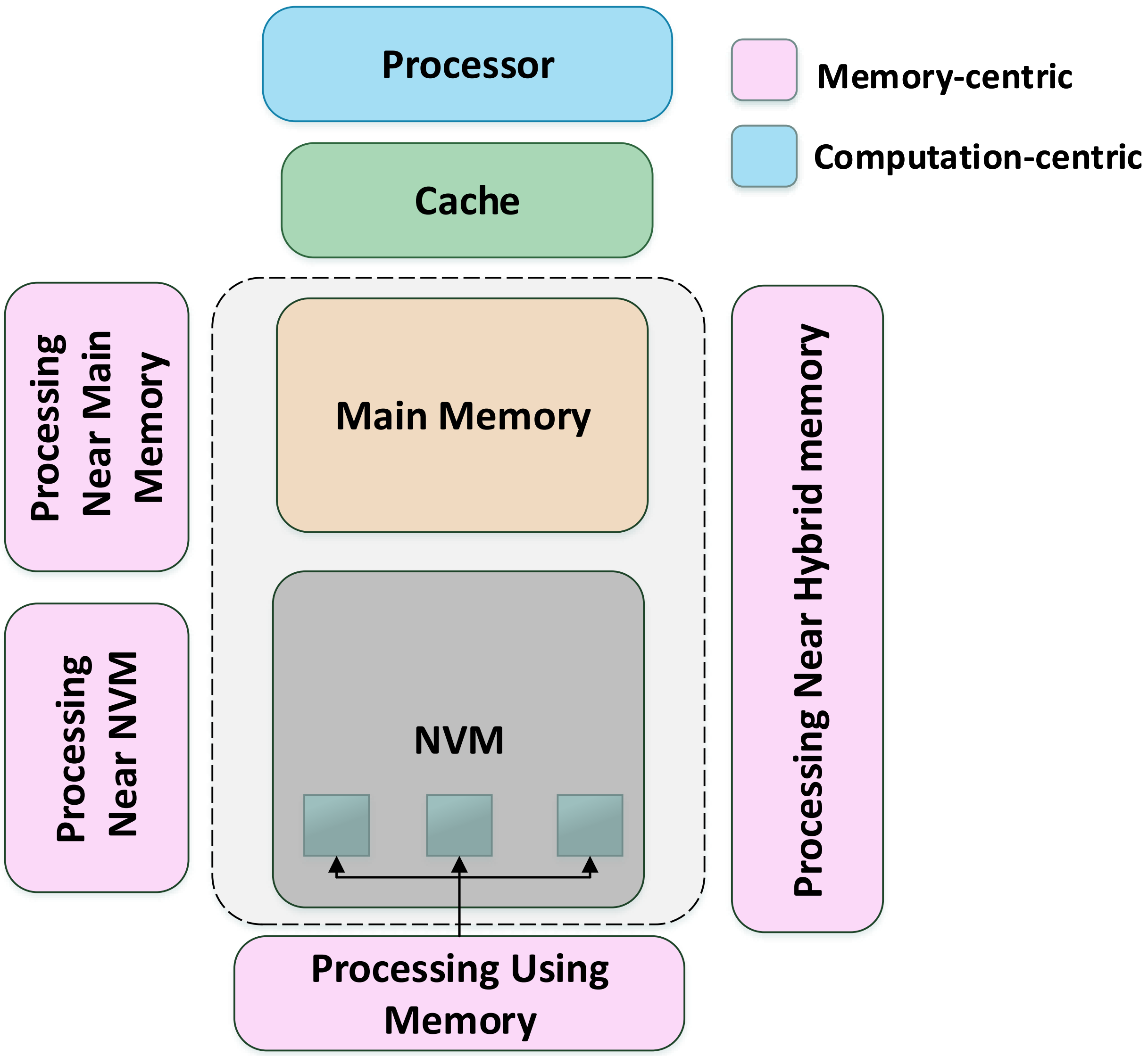


Figure 3.3: Processing options (memory-centric versus computation-centric) in the memory hierarchy. Memory-centric approach can be applied to any level (main memory or storage memory) and type (volatile or non-volatile, 2D planner memory or 3D-stacked memory) of memory in the memory hierarchy.

# Chapter 4

# Application Characterization for Near Memory Processing

Due to the increasing number of data-intensive applications in the era of big data, application characterization has taken an important role in system design. Application characterization is used to extract meaningful information by using specific metrics to decide which architecture could have the best performance and energy efficiency for a certain set of applications. This Chapter characterizes various multi-threaded applications from different benchmark suites for a set of performance and NMP-centric metrics to extract useful information. The goal is to exploit the redefined metrics to evaluate the amenability of various sets of applications to conventional Host CPU processing and two different NMP architectures.

## 4.1 Application Set

Several multi-threaded applications from different benchmark suites (Rodinia [31], Parboil [7], PARSEC [25], and Starbench [9][18]) are selected to cover a wide range of com-

Table 4.1: Evaluated applications and their description.

| Application | Suite | Name | Description |
|---|---|---|---|
| Back Propagation | Rodinia | BP | Pattern Recognition, Machine Learning |
| Breadth-First Search | Rodinia | BFS | Graph Analysis |
| HotSpot 3D | Rodinia | HS-3D | Physics Simulation |
| Sparse Matrix Vector Mult. | Parboil | SpMV | Graph Analysis, Machine Learning |
| Myocyte | Rodinia | MO | Biological Simulation |
| HeartWall Tracking | Rodinia | HW | Medical Imaging |
| Stream Cluster | PARSEC | SC | Data Mining |
| VASARI Image Processing Sys. | PARSEC | VIPS | Media Processing |
| Kmeans Clustering | Starbench | Kmeans | Artificial Intelligence, Data Mining |
| Ray Tracing | Starbench | C-ray | Computer Graphics |
| Image Rotation | Starbench | Rotate | Image Processing |
| Stencil | Starbench | Stencil | Physics Simulation, Machine Learning |

putation and memory patterns. Table 4.1 summarizes all the evaluated applications and their description.

Here is a short description for each of the applications characterized in this Chapter:

- *Back Propagation (BP)* is a commonly used algorithm in neural networks which are a widely used machine learning techniques. It is a training algorithm which takes differences between output of the untrained data and the desired output (supervised learning). Then it pushes the differences in the backward path through the network and updates weight of the nodes proportionally as it goes.

- *Breath-First Search (BFS*) is a fundamental building block found in many graph algorithms (path findings, network flow, and etc). Very large graphs which have millions of vertices are common in scientific and engineering applications. BFS is known for being memory-intensive with irregularly memory access pattern (poor data locality).

- *HostSpot 3D (HS-3D)* is a simulation tool which is used for estimating processor temperature. It works based on an architectural floor plan and simulated power measurements.

- *Sparse Matrix Vector Multiplication (SpMV)* is an important kernel found in many high performance computing applications such as scientific computing, economic modeling, and information retrieval. The computation is $y = A \times x$, where $A$ is a sparse matrix and $x$ and $y$ are dense vectors. It is considered as a memory-intensive application that solve large-scale linear systems and eigenvalue problems.

- *Myocyte (MO)* application models heart muscle cell (cardiac Myocyte) and simulates its behavior based on work by [138]. The model integrates electrical activity of heart muscle cell with the calcineurin pathway.

- *HeartWall Tracking (HW)* application tracks movements of a mouse heart over sequence of more than 100 ultrasound images. Image processing is performed in initial stage of the program to detect partial shapes of inner and outer heart walls.

- *Stream Cluster (SC)* algorithm solves the online clustering problem. For a stream of input points, the kernel finds a number of medians to assign each data point to its nearest cluster.

- *VASARI Image Processing System (VIPS)* is an image processing application which includes fundamental image operations such as convolution and transformation.

- *Kmeans Clustering (Kmeans)* kernel which is used extensively in artificial intelligence and data mining domains executes K-means clustering algorithm. This clustering application divides the cluster on sub-cluster and calculates the mean values of each sub-cluster.

- *Ray Tracing (C-ray)* is a brute force ray tracer algorithm which exhibits data-level parallelism. It renders an image in the PPM binary format from a scene description file. Regardless of being a simple algorithm, C-ray considered to be a very compute-intensive applications with a high computation to communication ratio.

- *Image Rotation (Rotate)* is an application that rotates an RGB image in binary representation by some degrees (0, 90, 180 or 270). Similar to Ray Tracing application, Rotate exhibits data-level parallelism. Comparing Rotate with Ray Tracing (C-ray), it features lower computations with more stress on the memory subsystem.

- *Stencil* kernel represents an iterative Jacobi solver of the heat equation on a multidimensional grid. Stencil computations are core components of many emerging applications. They are used in a wide range of domains from physical simulations to machine learning.

## 4.2 Simulation Setup

The modeled conventional Host CPU system is evaluated using gem5-NVMain hybrid simulator [39]. This hybrid simulator integrates the full system gem5 simulator [26] with NVMain 2.0 [116]. Full system gem5 simulator accurately evaluates the system performance. It runs unmodified operating system and produces comprehensive execution statistics. NVMain is an architectural-level memory system simulator for both DRAM and emerging non-volatile memory technologies. The Host CPU is modeled with an 8-core ALPHA processor running at 2 GHz frequency with two levels of private caches (L1 and L2) and a shared L3 cache. DRAM memory is modeled using Micron DDR4 timing parameters [6] which includes four DDR4-2666 MHz memory channels with four banks per rank and four ranks per channel. Each memory channel has a theoretical bandwidth of 21.3 GB/s. The

Table 4.2: The key parameters of the simulated Host CPU system

| **Host CPU System** | |
|---|---|
| Processor | 8 ALPHA cores @ 2 GHz frequency |
| Caches | per-core L1 (I): 32 KB, 2-way set associative |
| | per-core L1 (D): 32 KB, 2-way set associative |
| | per-core L2: 256 KB, 4-way set associative |
| | shared L3: 16 MB, 8-way set associative |
| | cache-line size: 64 B |
| **DRAM Memory** | |
| DDR4-2666 MHz | 16 GB: 16 Gb $\times 8$ |
| | 4 channels $\times$ 4 ranks $\times$ 4 banks |
| | Row buffer size: 8 KB |
| | Bandwidth: 21.3 GB/s per channel (theoretical) |
| | 15 GB/s per channel (empirical) |
| Timing Parameters | $t_{ck} = 1.25ns$ |
| | $t_{RAS} = 42$, $t_{RCD} = 19$, $t_{CAS} = 10$ |
| | $t_{CCD} = 4$, $t_{RP} = 19$, $t_{WR} = 210$ |

application characterization is conducted on this system. The architectural details for the simulated Host CPU system are summarized in Table 4.2.

## 4.3 Characterization Methodology

This section describes the general behavior of the studied applications and their performance bottleneck by running them on the modeled Host CPU system. As a case study, a thorough characterization (Roofline analysis, temporal and spatial data locality analysis, and memory access behavior analysis) is conducted to illustrate the unique behaviour (memory requirement and access behavior) of the studied applications and to justify the use of NMP architectures in terms of performance and energy consumption.

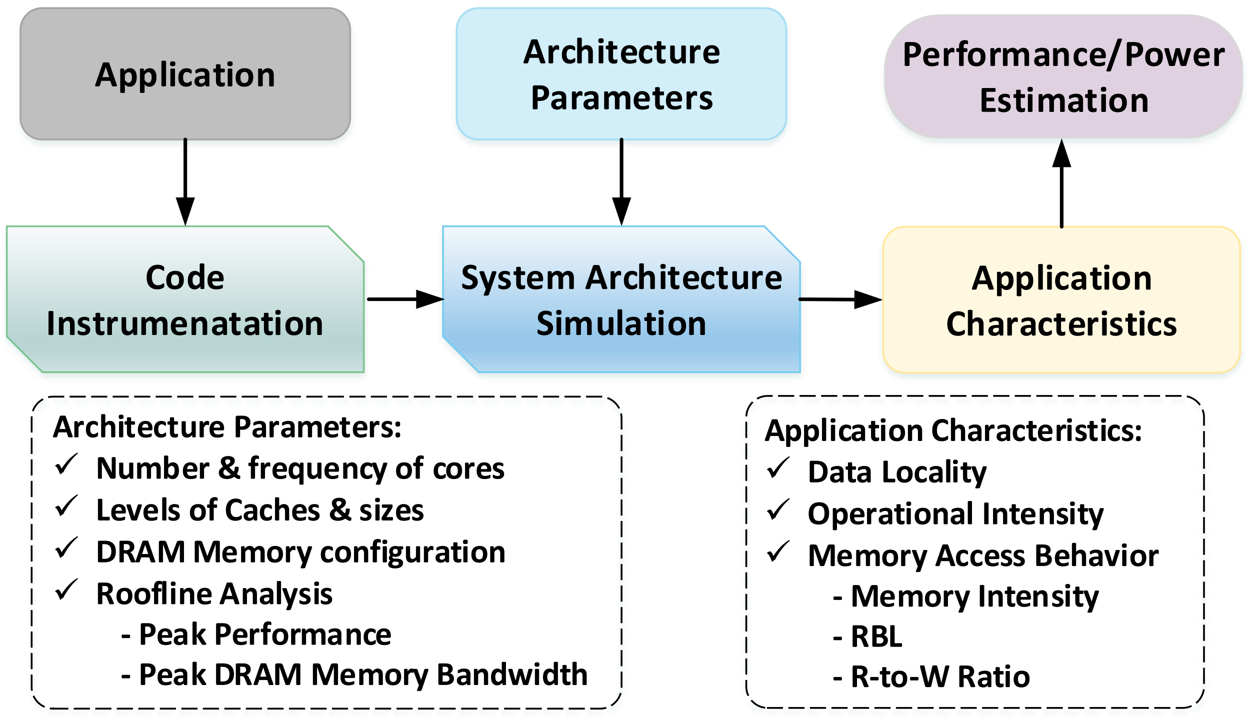

Application
Architecture Parameters
Performance/Power Estimation
Code Instrumenatation
System Architecture Simulation
Application Characteristics
Architecture Parameters:
✓ Number & frequency of cores
✓ Levels of Caches & sizes
✓ DRAM Memory configuration
✓ Roofline Analysis
- Peak Performance
- Peak DRAM Memory Bandwidth
Application Characteristics:
✓ Data Locality
✓ Operational Intensity
✓ Memory Access Behavior
- Memory Intensity
- RBL
- R-to-W Ratio

Figure 4.1: Application characterization methodology with system architecture simulation as a performance/power evaluation technique.

Various evaluation techniques are used by architects to explore the design space of an architecture. Based on the required details, architects usually use analytical models or cycle-accurate simulators for performance and power evaluation [135]. Figure 4.1 illustrates application characterization methodology with system architecture simulation as a performance and power estimation technique proposed in this dissertation.

### 4.3.1 Roofline Analysis

By applying the *Roofline model* which is a throughput-oriented performance model for floating point programs and multi-core CPU architectures, it can be found if an application lies in the *Memory-bandwidth-bound* region or *Performance-bound* region of the underlying hardware. This model combines floating point performance, operational intensity, and memory performance all together in a 2-dimensional graph [148].

A Roofline model is constructed in this section which describes the theoretical limits (peak theoretical performance and peak memory bandwidth) of the modeled Host CPU system. Peak theoretical performance can be found from hardware specification or by running micro benchmark such as STREAM benchmark [94]. Using the hardware specification method, the peak theoretical performance (in GFlops) of the modeled Host CPU system is defined as:

$$(CPU\ speed) \times (number\ of\ CPU\ cores) \times (CPU\ IPC) \times (number\ of\ CPUs\ per\ node) \tag{4.1}$$

Figure 4.2 presents the constructed Roofline model along with the Roofline data points for all evaluated applications.

In this model, Host CPU system has the theoretical performance limit of 240 GFlops/sec and peak memory bandwidth of 85.3 GB/s (21.3 GB/s per channel). The y-axis in this graph shows the attainable performance for each application which is defined as:

$$Attainable\ Performance = min \begin{cases} \text{Peak GFlops} \\ \text{Operational Intensity} \times \text{DRAM GB/s} \end{cases} \tag{4.2}$$

While an architecture has a fixed peak bandwidth and peak performance, Operational Intensity (OI) varies from one kernel to another. Table 4.3 shows the performance attained (in unit of GFlops per second) by each application executing on Host CPU system.

The data points in the graph represent the OI of each application. OI is used by Roofline to model the memory bandwidth an application uses. It provides a general overview of an application by determining the number of floating point operations per byte of memory

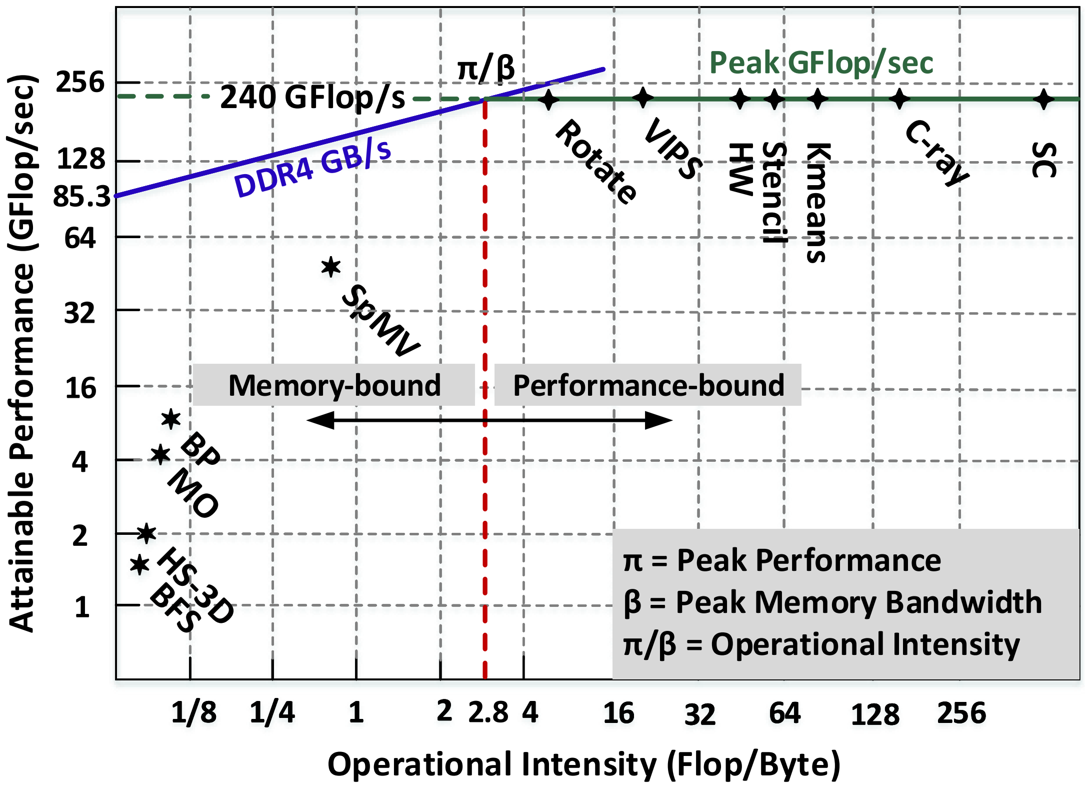


Figure 4.2: Constructed Roofline model for the modeled multi-core Host CPU system with 8-core ALPHA processor running at 2 GHz frequency, peak floating point performance of 240 GFlops/sec and peak memory bandwidth of 85.3 GB/s (theoretical). For each application, Roofline data point is shown on the graph based on its operational intensity and attainable performance. The minimum operational intensity to get the maximum performance is $\pi/\beta = 2.81$ Flops/Byte. As it is shown, applications with operational intensity less (more) than 2.8 are categorized as Memory-bound (Performance-bound).

Table 4.3: Attainable performance (GFlops/sec) of each studied application running on the modeled Host CPU system with peak performance of 240 GFlops/s. Applications with attainable performance less than 240 GFlops/s (BFS, HS-3D, MO, BP, and SpMV) cannot exploit Host CPU processing power.

| **Applications** | **BFS** | **HS-3D** | **MO** | **BP** | **SpMV** | **Rotate** | **VIPS** | **HW** | **Stencil** | **Kmeans** | **C-ray** | **SC** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Attainable Performance (GFlops/s)** | 1.84 | 2.00 | 4.26 | 7.46 | 60.85 | 240 | 240 | 240 | 240 | 240 | 240 | 240 |

traffic (Flops per Byte ratio). Using this metric, applications can be characterized into *Performance-bound* (i.e., suitable for Host CPU processing) and *Memory-bound* (i.e., suitable for NMP).

OI of an applications is defined as:

$$\frac{Number\ of\ Floating\ Point\ Operations}{Total\ Bytes\ Transferred\ Between\ DRAM\ and\ LLC} \tag{4.3}$$

In this formula, LLC refers to Last-Level Cache in the memory hierarchy. Applications with a very low operational intensity (low FLOPs/Byte) cannot exploit host's processing power.

Figure 4.3 shows the OI for all the evaluated applications. Application such as BFS, HS-3D, MO, BP, and SpMV are characterized by extremely low OI (the byte ratio is less than 1) which makes them inherently memory-bound based on the Roofline model. For such applications, data movement is the major performance bottleneck which causes excessive cache misses. Rotate, VIPS, HW, Stencil, Kmeans, C-ray, and SC have OI greater than 2.8 Flops/Byte. These applications are categorized into Performance-bound that can fully utilize the host processing power and large cache hierarchies (in case of having high data

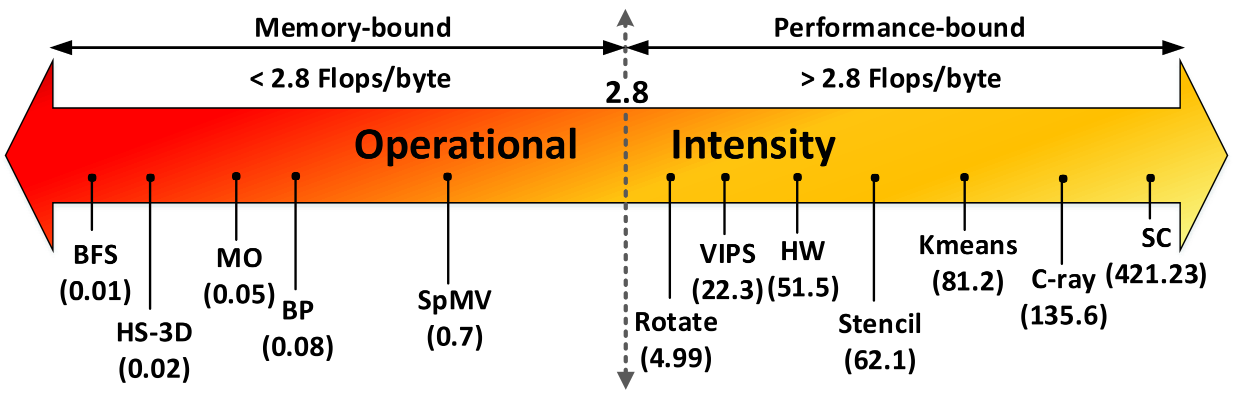


Figure 4.3: Application categorization based on OI and Roofline model of the simulated Host processor analyzed in this dissertation. As it is depicted, applications with OI less than 2.8 are categorized as memory-bound and applications with OI greater than 2.8 are bounded by performance (Performance-bound).

locality).

Based on the Roofline analysis and applications' OI (see Figure 4.2 and Figure 4.3), it can be concluded that:

- The attainable performance of applications such as *Rotate, VIPS, HW, Stencil, Kmeans, C-ray, and SC* is approaching the theoretical performance bound of the Host CPU (240 GFlops/sec) which categories them into applications with high compute bound. These applications have a high computation to communication ratio.

- *BFS, HS-3D, MO, BP, and SpMV* applications have a very low compute bound. These applications are bounded by memory bandwidth, and they cannot fully utilize the Host CPU processing power.

### 4.3.2 Temporal and Spatial Data Locality

The principal of locality of references justifies the use of large cache hierarchies in the Host processing unit. Temporal and spatial data locality are intrinsic to the reference stream which explains their in-dependency on cache parameters. In order to confirm the results obtained by Roofline analysis (see Section 4.3.1), this section estimates the amount of data locality (temporal and spatial) in compute-bound applications (Rotate, VIPS, HW, Stencil, Kmeans, C-ray, and SC). Applications with high Flops/Byte ratio (high OI) and high data locality can exploit benefits offered by Host processing power and large cache hierarchies.

*Temporal data locality (locality in time)* is the measure of how likely a data is to appear again in a sequence of requests after being requested within a time span. This metric helps to categorize applications in two classes:

- Applications with *poor or no temporal data locality* which cannot benefit from large and multi-layer Host caches.
- Applications with *high temporal data locality* which can exploit large cache hierarchies and are more suitable for Host CPU processing.

One way to estimate the temporal data locality of an application is to analyze how the cache hit rate of a processor changes as we increase the last-level cache capacity with a fixed cache-line size. Figure 4.4 shows temporal data locality sweeping cache size from 8MB to 64MB with fixed cache-line size of 64B for applications with high compute-bound (see Figure 4.2). Figure 4.4 illustrates that HW, VIPS, C-ray, and Stencil have enough temporal data locality to leverage from cache hierarchies in Host CPU system. Kmeans and Rotate exhibit a very small improvement in their cache hit rate which implies a poor temporal locality. SC application with no improvement in the cache hit rate shows no temporal locality.

*Spatial data locality (locality in space)* is the phenomenon that if a program references a

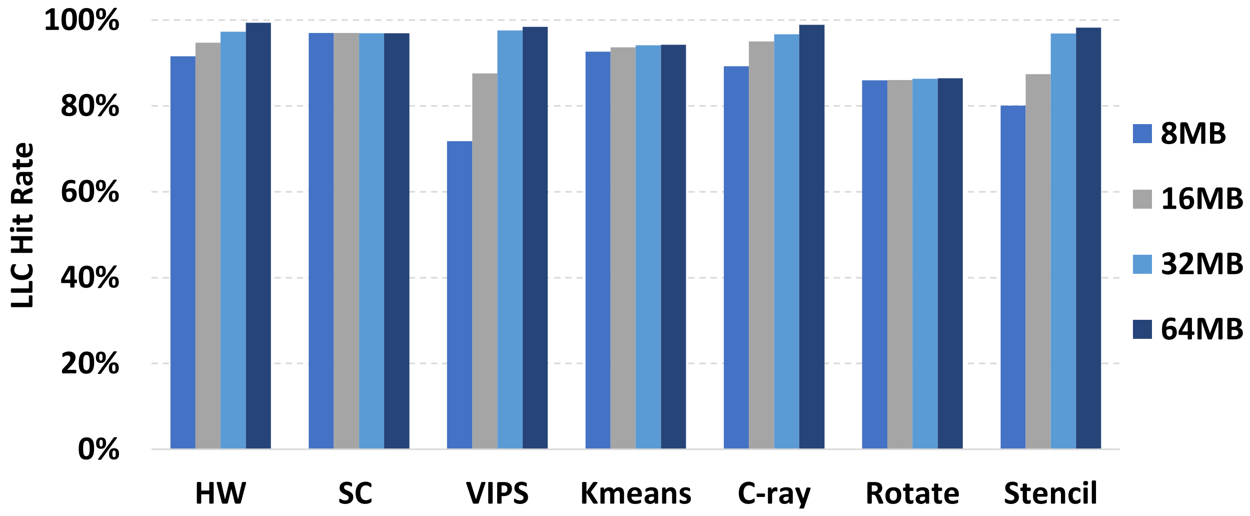


Figure 4.4: Temporal data locality sweeping LLC capacity 8-64MB with fixed cache-line size of 64B across all compute-bound applications.

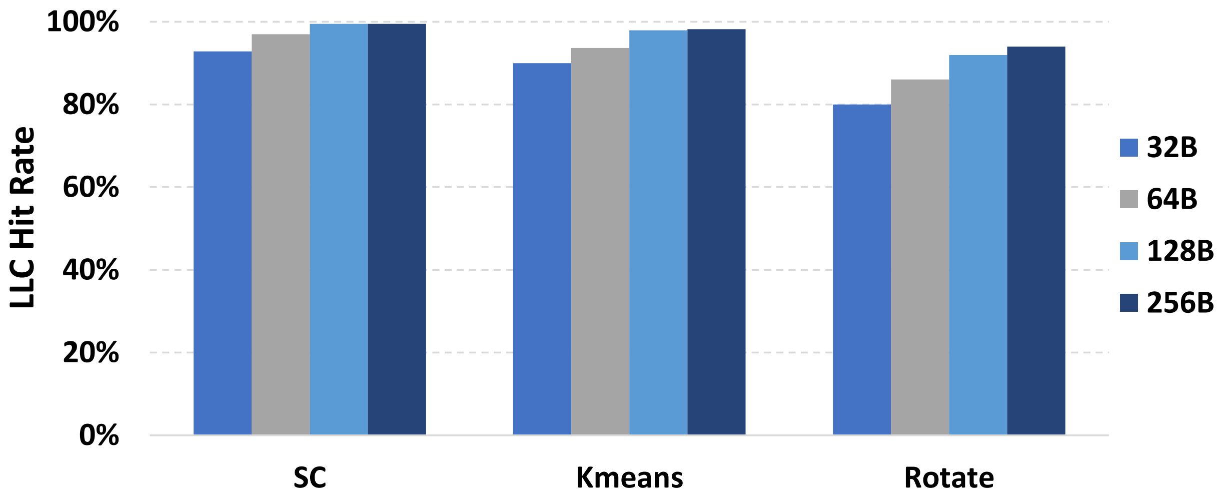


Figure 4.5: Spatial data locality sweeping cache-line (LLC) size 32-256B with fixed cache capacity of 16MB across compute-bound applications with poor/no temporal data locality.

particular data, then it is extremely likely that the program will also reference other data that are nearby to the referenced data. This data locality determines sensitivity to the cache-line size and can be estimated by sweeping the cache-line size with a fixed cache capacity [150]. Figure 4.5 shows spatial data locality for three compute-bound applications with poor/no temporal locality (see Figure 4.4) by sweeping cache-line size from 32B to 256B with a fixed cache size of 16MB. All of three compute-bound applications with poor/no temporal locality (SC, Kmeans, and Rotate) have enough spatial locality. These applications can utilize the benefits of large cache hierarchies provided by the Host CPU system.

### 4.3.3 Memory Access Behavior

*Memory Intensity, Row Buffer Locality (RBL), and Read-to-Write (R-to-W) Ratio* are three components which are used to estimate the memory access behavior of each application. Table 4.4 lists all applications used in this study (both memory-intensive (highlighted in gray in the table) and compute-intensive) and their memory characteristics. The focus of this section is on memory-intensive applications which have high and middle memory intensity (memory intensity > 2).

#### Memory Intensity

*Memory intensity* is the frequency at which a request misses in the last-level cache which is determined in the unit of misses per kilo instructions (MPKI) of LLC. LLC MPKI can be defined as:

$$\frac{Number\ of\ Miss\ Memory\ Accesses}{(Total\ Number\ of\ Committed\ Instructions\ /\ 1000)} \tag{4.4}$$

Applications with memory intensity greater than two (LLC MPKI > 2) are classified into

Table 4.4: List of all evaluated applications and their memory access behavior. The reported numbers are measured from Host CPU execution. Applications with "High" and "Middle" memory intensity are classified into memory-intensive (highlighted in gray in the table) and other applications are labeled as memory-non-intensive (compute-intensive).

| Application | Memory Access Behavior | | |
|---|---|---|---|
| | Memory Intensity | RBL | R-to-W Ratio |
| BP | 21.5 (High) | 24.51% (Low) | 1.63 (< 3) |
| MO | 7.2 (High) | 14.82% (Low) | 1.33 (< 3) |
| BFS | 2.4 (Middle) | 76.5% (High) | 3.93 (> 3) |
| HS-3D | 2.3 (Middle) | 75.61% (High) | 4.26 (> 3) |
| SpMV | 2.1 (Middle) | 16.24% (Low) | 4.48 (> 3) |
| Rotate | 0.4 (Low) | 28.80% | 1.72 |
| VIPS | 0.21 (Low) | 45.48% | 1.86 |
| HW | 0.13 (Low) | 5.94% | 1.08 |
| Kmeans | 0.085 (Low) | 75.53% | 2.17 |
| C-ray | 0.07 (Low) | 21.10% | 1.41 |
| Stencil | 0.03 (Low) | 58.91% | 1.63 |
| SC | 0.006 (Low) | 58.47% | 3.47 |

memory-intensive and other applications (with LLC MPKI $< 2$) are labeled as non-memory intensive (compute-intensive). Data shown in Table 4.4 confirms the results obtained by Roofline analysis and data locality analysis. Rotate, VIPS, HW, Kmeans, C-ray, Stencil, and SC have a very low memory intensity (LLC MPKI $< 1$) which lies them in the non-memory-intensive category. BP, MO, BFS, HS-3D, and SpMV applications (highlighted in gray in Table 4.4) with memory intensity greater than two are categorized them into memory-intensive class.

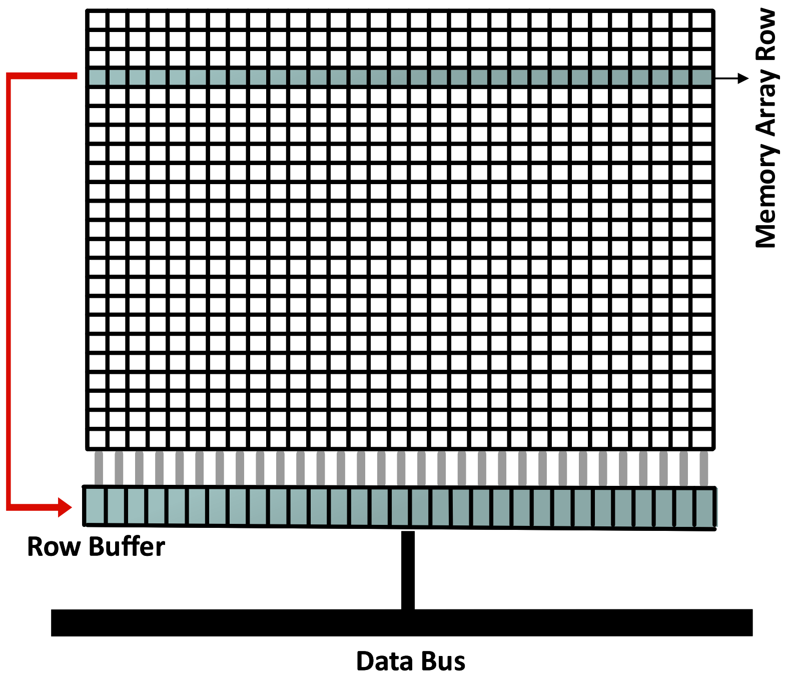


Figure 4.6: Memory bank organization. Each bank in memory has a row buffer that caches the last accessed row. A row buffer hit is much cheaper than a row buffer miss. Figure is adopted from [106].

### RBL

Memory device organization includes a peripheral storage known as *Row Buffer (RB)* which acts as a cache for memory array rows and is independent from the memory technology. Figure 4.6 shows memory bank organization. As it is shown in this figure, each bank has a row buffer that caches the last accessed row. This memory component is present in both DRAM and PCM technologies. When content of a memory array's row is placed in the row buffer, successive memory requests to the same row are served immediately from the row buffer. These memory accesses are called row buffer hits. If a memory request refers to a row which is different from the latched one in the row buffer, then this request causes row buffer miss. Based on [95], row buffer hits produce same latency and cost in both DRAM and PCM technologies, while row buffer misses incur larger latency and cost in PCM than DRAM. Examining this metric, a same style buffering and size is assumed for row buffer

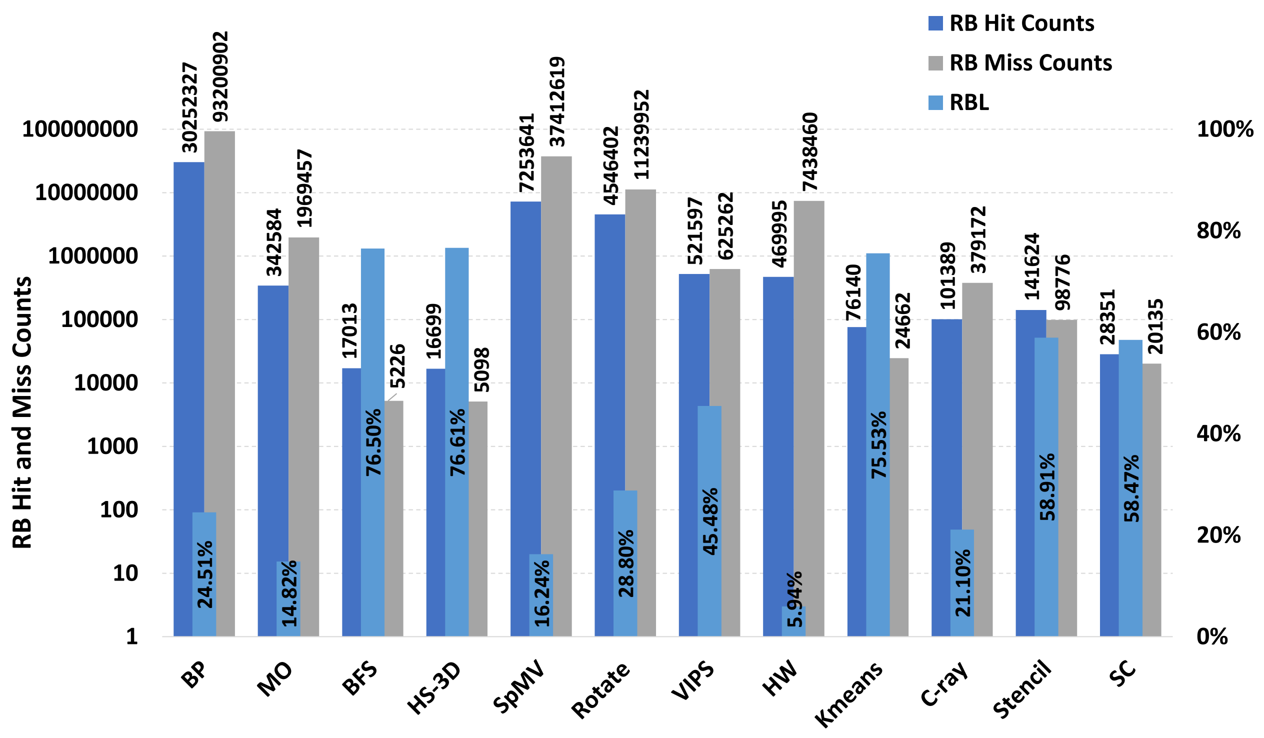


Figure 4.7: Application characterization based on row buffer locality (RBL). Left y-axis shows row buffer hit and miss counts, and right y-axis indicates RBL (RB hit rate) for all the evaluated applications across all memory channels. Along x-axis, applications are sorted based on their memory intensity, from highest to least.

in both DRAM and PCM technologies. The RBL of an application is the average hit rate of the row buffer across all memory channels and is considered as an important measure of data locality in memory. Figure 4.7 shows row buffer hit and miss counts along with RBL for all the studied applications across all memory channels.

Based on result shown in Table 4.4 and Figure 4.7 for memory-intensive applications, BFS and HS-3D have high RBL (RB hit rate $> 75\%$). BP, MO, and SpMV with RB hit rate less than 25% are considered as applications with low RBL.

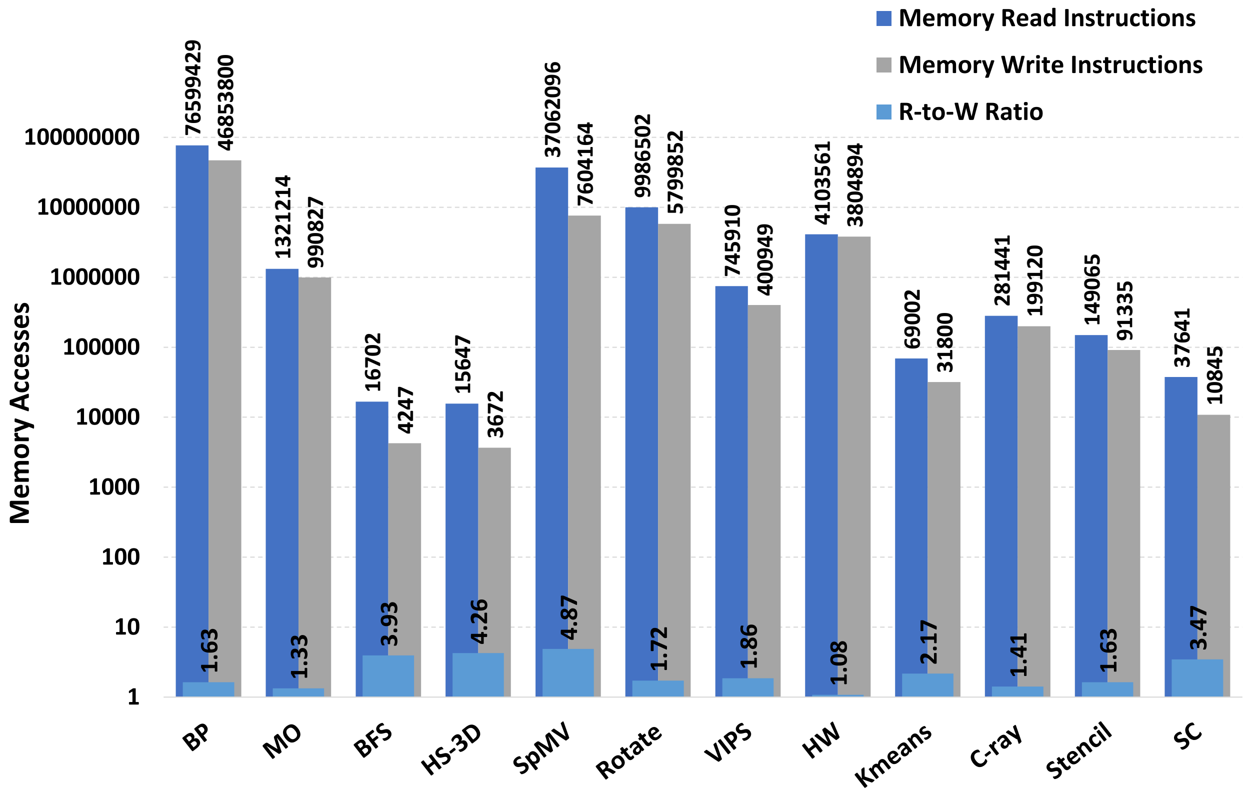


Figure 4.8: Application characterization based on average R-to-W ratio. Left y-axis shows memory accesses (read and write accesses) across all evaluated applications. Along x-axis, applications are sorted based on their memory intensity, from highest to least.

### R-to-W Ratio

*R-to-W* ratio metric is used to categorize Memory-bound applications into read-intensive and write-intensive. Although PCM offers the read latency/energy close to DRAM technology, it suffers from high write latency/energy which has an adverse impact on the system performance. Based on different characteristics of DRAM and PCM such as read latency/energy, write latency/energy, and power consumption, it can be decided which NMP subsystem (3D-DRAM or 3D-PCM) would perform better in terms of memory power saving and potentially performance for Memory-bound applications.

R-to-W ratio of an application is defined as:

$$\frac{Number\ of\ Memory\ Read\ Instructions}{Number\ of\ Memory\ Write\ Instructions} \tag{4.5}$$

Figure 4.8 shows memory accesses (read and write) across all studied applications in this dissertation (memory-intensive and compute-intensive). Along the x-axis, applications are sorted based on their LLC MPKI (memory intensity), from highest to least (see Table 4.4). The focus of this section is on applications with high and middle memory intensity (BP, MO, BFS, HS-3D, and SpMV).

Based on multiple research work, while DRAM write latency is from 20ns to 50ns, PCM write latency varies 50ns to 150ns (shown in Table 3.1). Average R-to-W ratio higher than 3 (i.e., PCM to DRAM write latency ratio) is used to characterize Memory-bound applications into read-intensive (R-to-W ratio $> 3$) and write-intensive (R-to-W ratio $< 3$). This approach helps to differentiate between NMP systems based on average R-to-W ratio of each application. For memory-read-intensive applications, read energy/latency is the dominant factor in determining the memory power consumption and performance. Thus, such applications may benefit from 3D-PCM NMP.

### 4.3.4 Read Disturbance

Devoting the 3D-PCM to memory read-intensive applications with high RBL has major benefits such as improving the PCM endurance and reducing memory access power dissipation and delay. However, consecutive read accesses to NVMs, with no write operation in between, may lead to a bit-flip in the accessed cells which is called *read disturbance.*

Among NVMs, STT-RAM is the most susceptible NVM technology to read disturbance due to the close read and write currents in this technology. Although PCM is much more robust against bit-flip due to read disturbance, turbo read accesses with high current on bit-lines, to improve PCM performance, can lead to read disturbance fault and data loss.

There are various techniques to minimize or mask read disturbance effect which are applicable in the proposed memory architecture:

- Error Correcting Codes (ECCs) are widely utilized to detect and correct bit-flip in memories. Thanks to the process variation, aging, and susceptibility to transient faults, the ECC unit is a popular component almost in all of memory structures. This unit can be utilized together with the proposed technique to prevent read disturbance errors.
- In this study, 3D-PCM technology is selected which is the most robust NVM against read disturbance faults. By avoiding turbo read accesses to PCM, this memory technology can well resist against read disturbance.
- Utilizing multi-level cache hierarchies and larger memory row buffers help to reduce the read disturbance effect. Due to locality of read references, caches mask a large number of memory accesses which reduces read disturbance error in NVMs.

In memory-read-intensive applications, the expectation of consecutive read accesses increases. Thus, the average number of read accesses with no write access in between increases by only storing memory-read-intensive applications' data in 3D-PCM. However, this does not exacerbate worst-case consecutive memory read accesses. It is worth mentioning that for the guaranteed operation of memory system, the worst-case stress condition should be considered. Thus, the proposed technique does not impose extra overheads in respect to the read disturbance fault model, and the same considerations must be taken into account for any NVM-based memory system.

## 4.4 Insights and Discussions

In order to assess computational demand and memory footprint and to suggest the most suitable processing unit (Host CPU processing or NMP (3D-DRAM or 3D-PCM)) in terms of performance and energy efficiency, a systematic characterization has been conducted on a wide range of multi-threaded applications (mixture of compute-bound and memory-bound). The characterization revealed the performance bottleneck of the studied applications. The first three metrics (Roofline analysis, data locality (temporal and spatial) analysis, and memory intensity) classified applications into Performance-bound and Memory-bound. Then, by estimating applications' memory access behavior (R-to-W ratio and RBL), Memory-bound applications are classified into write-intensive and read-intensive with high/low RBL to suggest the most suitable NMP system based on 3D-DRAM and 3D-PCM technologies, respectively. The read disturbance (as a memory metric) is also discussed to consider the case when 3D-PCM, or in general non-volatile memory, is regularly chosen for read-intensive applications.

Table 4.5 summarizes application characterization based on high-level (Data locality, Roofline analysis with OI, memory intensity) and low-level (R-to-W ratio, RBL, and read disturbance) metrics to suggest a suitable processing unit for each class of applications.

Application characterizations are summarized as follows:

1. **CPU-Intensive (Performance-bound) class with high OI (high Flops/Byte ratio), high data locality (temporal and/or spatial), and low memory intensity (LLC MPKI $< 2$):**

   This class includes `CPU-friendly` applications that can fully utilize Host processing power and large multi-level cache hierarchies. The attainable performance of these applications approaches to the peak performance of the modeled Host CPU system.

Table 4.5: List of all evaluated applications and their characteristics. Based on the redefined metrics (data locality, operational intensity based on Roofline analysis, memory intensity, R-to-W ratio, and row buffer locality), the best processing unit (Host CPU, 3D-DRAM NMP or 3D-PCM NMP) is suggested for each class of applications.

| **Application's Class** | | **Name** | **Best Processing Unit** |
|---|---|---|---|
| **Performance-bound:** | | Rotate | Host CPU |
| $\star$ High Data Locality | | VIPS | Host CPU |
| $\star$ High OI | | HW | Host CPU |
| † (Roofline Analysis) | | Stencil | Host CPU |
| $\star$ Low Memory Intensity | | Kmeans | Host CPU |
| † (LLC MPKI) | | C-ray | Host CPU |
| | | SC | Host CPU |
| **Memory-bound:** | **Write-Intensive:** | BP | NMP (3D-DRAM) |
| $\star$ Low/No Data Locality | $\star$ R-to-W ratio $< 3$ | MO | NMP (3D-DRAM) |
| $\star$ Low OI | $\star$ Low RBL | | |
| † (Roofline Analysis) | | | |
| $\star$ High Memory Intensity | **Read-Intensive:** | BFS | NMP (3D-PCM) |
| † (LLC MPKI) | $\star$ R-to-W ratio $> 3$ | HS-3D | NMP (3D-PCM) |
| | $\star$ High/Low RBL | SpMV | NMP (3D-PCM) |
| | (*Read Disturbance Consideration) | | |

Rotate, VIPS, HW, Stencil, Kmeans, C-ray, and SC are `CPU-friendly` applications that can benefit more from Host CPU processing comparing to NMP systems.

2. **Memory-Intensive (Memory-bound) class with low OI, poor/no data locality (poor/no temporal and spatial locality), and high/middle memory intensity (LLC MPKI $> 2$):**

   This class of applications show a significant memory footprint (with high and middle LCC MPKI) with Host CPU under-utilization. It includes `NMP-friendly` applications that can be categorized into memory-write and memory-read intensive with high/low RBL. The characterization is based on R-to-W ratio and RBL (RB hit rate) metrics. BP and MO are memory-write-intensive applications with R-to-W ratio $< 3$ and low RBL (see Table 4.4) that can be processed efficiently on 3D-DRAM NMP. BFS and HS-3D are memory-read-intensive (R-to-W ratio $> 3$) applications with high RBL that may benefit from 3D-PCM NMP architecture in terms of performance and power/energy consumption. For SpMV application with R-to-W ratio $> 3$ (memory-read-intensive) and low RBL, 3D-PCM NMP may outperform 3D-DRAM NMP in terms of power consumption due to its read-intensive property. It should be noted that regular processing of memory-read-intensive applications on a NMP subsystem based on 3D-PCM leads to the read disturbance effect, which should be carefully taken into consideration.

## 4.5 Summary

Through detailed and comprehensive characterization conducted on each application, we are ready to accelerate data-intensive problem caused by memory wall bottleneck of the conventional processing architectures using architectural-level technique. NMP systems are proven to be practical and efficient against memory-intensive applications and are specialized towards this type of applications. In Chapter 5, a hybrid memory system composed of two different NMP subsystems (3D-DRAM and 3D-PCM) is explored and evaluated in order to process memory-intensive applications efficiently in terms of performance and memory power consumption.

# Chapter 5

# Near Memory Processing in Hybrid Memory Systems

Chapter 3 discussed near memory processing based on 3D stacking (NMP) as an effective architectural-level approach to avoid expensive data movement between off-chip memory and processing unit. With the advent of 3D die stacking technology and more importantly hybrid memory systems, the long-wished NMP capability is enabled.

In Chapter 4, first, a set of NMP-centric performance metrics are redefined and investigated in order to analyze the efficacy of mapping a processing unit (Host CPU system, 3D-DRAM based NMP, and 3D-PCM based NMP) to a specific application. Then, leveraging the redefined metrics, various set of application domains (memory-intensive and compute-intensive) are characterized to estimate the efficiency of a processing unit in terms of performance and power/energy consumption in order to suggest the most suitable architecture.

This Chapter introduces a hybrid processing architecture in which a multi-core Host CPU system is connected to a hybrid NMP subsystem composed of two different 3D memory technologies (3D-DRAM NMP and 3D-PCM NMP). Based on results obtained from application characterization in Chapter 4, data-intensive (memory-intensive) applications can be processed using the proposed NMP architectures to evaluate the efficacy of constructing NMP architecture when 3D-PCM or in general 3D-NVM technology is employed. Leveraging the benefits offered by emerging non-volatile memory technology encourages architects to design more efficient NMP systems in a hybrid 3D structure.

## 5.1 NMP Hardware Architecture in Hybrid System

NMP systems can be designed with different technology techniques such as processing on buffer-on-board (BoB) devices [38], edge-bonding small processor dies on DRAM chips, and 3D stacking with TSVs [141][104]. In this dissertation, 3D die stacking technology with TSVs is used to design efficient NMP subsystem.

Figure 5.1 shows an abstract view of the proposed architecture in which a multi-core Host CPU (with large and multi-level cache hierarchies) communicates with the hybrid NMP subsystem using high-speed links. The proposed architecture is similar to a conventional system where the Host processor uses multiple DDR memory channels to connect to multiple memory modules, but instead of DDR interface, high-speed links are used. In the NMP subsystem, two different memory technologies (3D-DRAM and 3D-PCM) are selected to enable the NMP capability. An application can run on the multi-core Host CPU or it can be transferred to the NMP subsystem (3D-DRAM NMP or 3D-PCM NMP) where the data can be accessed more efficiently.

An NMP stack (see Figure 5.1 (b)) is composed of several vertical vaults in which each vault is equivalent to a channel in the DDR memory module. Each vertical vault includes multiple partitions which are located in different story of memory layers and consists of many memory banks. In this architecture, NMP cores are located in the logic layer (one or more per vault)

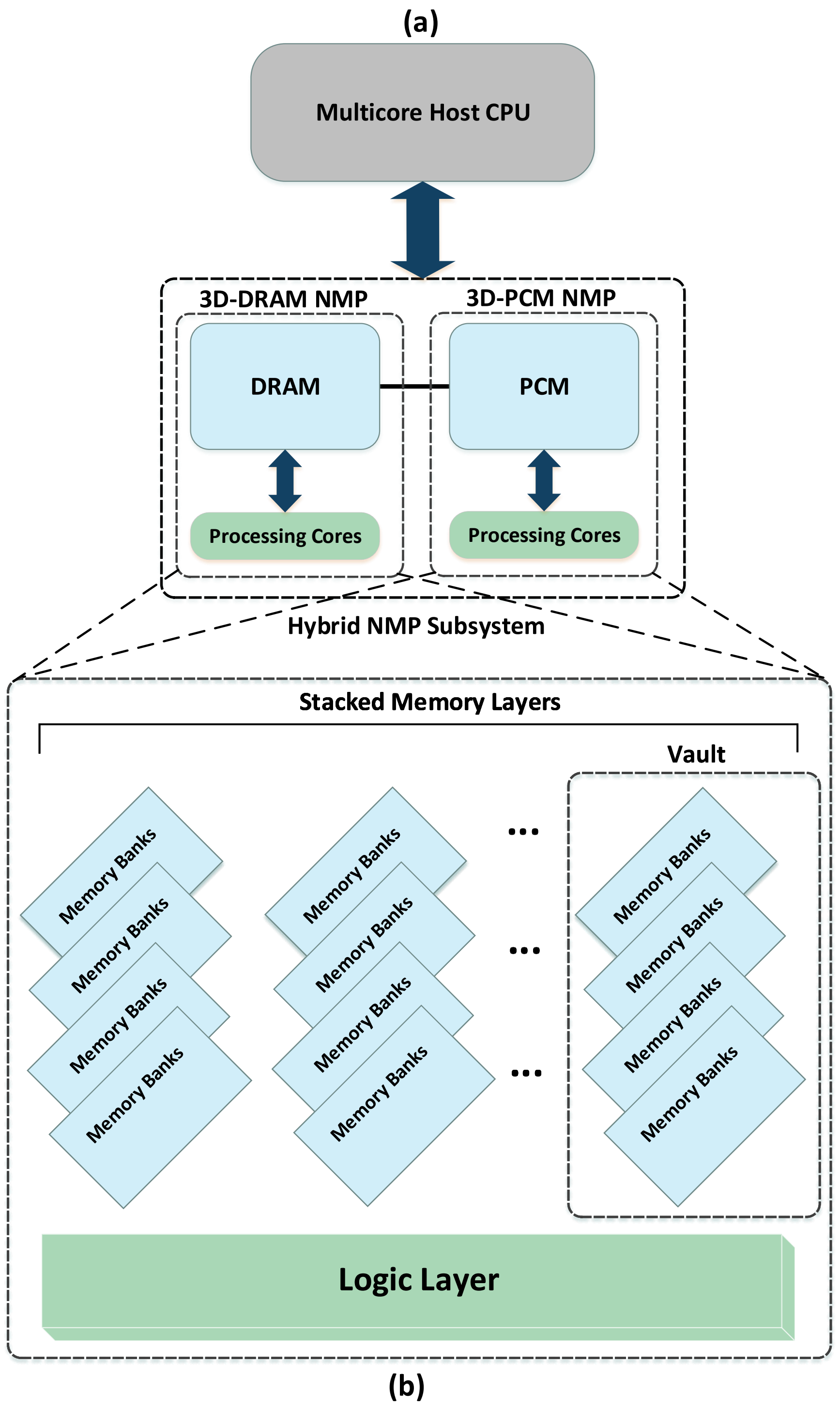


Figure 5.1: An envisioned hybrid processing system where a multi-core Host CPU with large cache hierarchies is connected to a hybrid NMP subsystem. In each NMP subsystem, multiple memory layers (composed of many memory banks) are stacked on top of a logic layer that provides the computation ability with high internal parallelism. In this architecture, an application can run on the Host CPU system as in the conventional manner, or it can be offloaded to one of the NMP subsystems in which data can be accessed more efficiently.

and a total of $\{$*number of vaults* $\times$ *number of cores per each vault*$\}$ cores constitute a NMP processor.

Different evaluation techniques can be used to explore the design-space of an architecture. Based on the level of information required, architects leverage "analytical model" or "cycle-accurate simulators". Analytical technique uses low-level system details and provides fast estimations for performance and power at the cost of accuracy. Authors at [155] use machine learning techniques for designing an analytical model to estimate final architecture performance. Cycle-accurate simulation-based techniques provide better performance and power numbers in terms of accuracy. In this technique, the entire micro-architecture is modeled precisely and compared to analytical mode, it is quite slow. There are many academic efforts to build NMP simulators which are open-source. This dissertation uses simulation-based modeling to evaluate the studied architectures more precisely in terms of performance and power.

## 5.2 Evaluation Methodology

This section discusses the evaluation methodology and simulation configurations for three different processing units (conventional multi-core Host CPU, 3D-DRAM based NMP, and 3D-PCM based NMP systems).

The study of identifying the potential of NMP subsystem to boost the performance and power consumption of the memory-intensive applications (discussed in Chapter 4) is based on matching the characteristics of these applications to NMP systems (3D-DRAM NMP and 3D-PCM NMP). Twelve real-world applications (presented in Section 5.2.1) are simulated for the evaluation. The application characterization is conducted on conventional Host CPU system (see Table 5.1 for Host CPU configuration) to define the unique behavior (e.g. computational demand and memory footprint) of each application. Roofline analysis along with data locality (temporal locality and spatial locality) analysis and memory intensity metrics classify applications into Performance-bound and Memory-bound. Then by defining memory access behavior of Memory-intensive applications (applications with high and middle memory intensity highlighted in gray in Table 4.4), their memory characteristics (RBL and R-to-W ratio) can be matched to the two existing NMP systems.

### 5.2.1 Simulation Models

Conventional multi-core Host CPU system is evaluated using full-system (FS) gem5-NVMain hybrid simulator [39]. The gem5 simulator [26] is a tool for computer system architecture research which is widely used in industry and academia. Because of its high configurability for a fine-grained architecture modeling, it evaluates system performance in an accurate way. This simulator supports cycle-accurate emulation for most instruction set architectures and a wide range of CPU models. Providing two system modes (full-system and system-call emulation), it allows configurations for trade off between speed and accuracy. Even at the micro-architecture level, this simulator provides thorough execution statistics for power, processor, and memory [49]. The NVMain simulator [116] is an architectural-level and cycle-accurate main memory simulator which models both DRAM and emerging non-volatile memory technologies. It can be built as an executable to run trace-based simulations, or it can be patched into a CPU simulator (such as gem5). The later implementation is used in this dissertation.

Ramulator-Pim [8] [136], a processing-in-memory simulation framework, is used to evaluate the NMP subsystems discussed in this dissertation. The framework is based on two simulators, ZSim [124], a fast and accurate simulator for thousand core systems, and Ramulator [74] which is a fast and cycle-accurate DRAM simulator. Dynamic execution traces of the instrumented code are collected from ZSim. Then, the acquired traces are fed to Ramulator. Ramulator simulates the memory accesses of the Host cores and NMP cores using the traces generated by ZSim.

Table 5.1 summarizes the key parameters and configurations of the simulated systems (Host CPU system and two NMP systems). The Host CPU system includes a 8-core ALPHA processor running at 2 GHz frequency with two levels of private caches (L1 and L2) per core and a shared L3 cache. The memory subsystem (DRAM) is modeled using Micron DDR4 timing parameters [6] which includes four DDR4-2666 MHz memory channels with four banks per rank and four ranks per channel. Each memory channel has a theoretical bandwidth of 21.3 GB/s. The application characterization (see Chapter 4) is conducted on this system.

The evaluated NMP systems are based on two different memory technologies, 3D-DRAM and 3D-PCM, as the representatives of volatile and non-volatile types to construct a hybrid NMP subsystem. DRAM has already adopted 3D staking technology and PCM also proved to be 3D stackable [69]. Micron and Intel revealed a 3D-stacked PCM-like memory product called 3D XPoint which is designed for memory hungry applications [66] [45]. The simulated NMP systems extend the 3D memory systems by introducing a number of simple in-order cores with caches into the logic layer. Out-of-order or wide-issue cores are not necessary because of poor/no data locality and instruction-level parallelism in applications that execute near memory. Table 5.1 includes more details regarding the simulated NMP systems. Table 5.2 and Table 5.3 show the values assigned to memory configuration parameters for modeling DDR4 and PCM memory models.

Table 5.1: The key parameters of the simulated systems

| **Host CPU System** | |
|---|---|
| Processor | 8 cores @ 2 GHz frequency |
| Caches | per-core L1 (I): 32 KB, 2-way |
| | per-core L1 (D): 32 KB, 2-way |
| | per-core L2: 256 KB, 4-way |
| | shared L3: 16 MB, 8-way |
| | cache-line size: 64 B |
| **DRAM Memory** | |
| DDR4-2666 MHz | 16 GB: 16 Gb $\times 8$ |
| | 4 channels $\times$ 4 ranks $\times$ 4 banks |
| | Row buffer size: 8 KB |
| | Bandwidth: 21.3 GB/s per channel (theoretical) |
| | 15 GB/s per channel (empirical) |
| Timing Parameters | $t_{ck} = 1.25ns$ |
| | $t_{RAS} = 42$, $t_{RCD} = 19$, $t_{CAS} = 10$ |
| | $t_{CCD} = 4$, $t_{RP} = 19$, $t_{WR} = 210$ |
| **NMP System** | |
| Cores | 8 cores @ 1.8 GHz frequency |
| Caches | per-core L1 (I): 32 KB, 4-way |
| | per-core L1 (D): 32 KB, 4-way |
| | cache-line size: 64 B |
| **3D-stacked Memory** | |
| 3D-DRAM | 16 GB: 2 layers $\times$ 4 vaults $\times$ 4 rank |
| (HMC) | 4 ranks/vault |
| | 2 cores per vault logic |
| | Row buffer size: 256 B |
| 3D-PCM | 16 GB: 2 layers $\times$ 4 vaults $\times$ 4 rank |
| | 4 ranks/vault |
| | 2 cores per vault logic |
| | Row buffer size: 256 B |

### 5.2.2 Memory Model Parameters in NVMain for DRAM and PCM Technologies

This section provides the different memory configuration parameters (interface properties, memory system, memory timing, energy/power, memory controller, and memory endurance) of NVMain used to model the memory technologies considered in this dissertation. Table 5.2 and Table 5.3 shows values assigned to those parameters for modeling the DDR4-2666MHz (Micron) and PCM memory models [6] [35]. All results obtained in Chapter 4 and Chapter 5 are based on these memory configurations.

### 5.2.3 Applications

The applications used in this study are mixture of different benchmark suites, Rodinia [31], Parboil [7], PARSEC [25], and Starbench [9][18], to cover a wide range of application domain. Table 5.4 lists all of the evaluated applications with their domain. A short description for each of the studied applications is provided in Chapter 4, Section 4.1. The focus of this Chapter is on memory-intensive applications.

## 5.3 Evaluation Results

This section presents the experimental results from executing memory-intensive applications (BP, MO, BFS, HS-3D, and SpMV) under three different processing architectures (Host CPU system, 3D-DRAM NMP and 3D-PCM NMP). A summary of the experimental setup for the conventional Host CPU and NMP systems is shown in Table 5.1. Unless otherwise stated, all results are normalized to the Host CPU system.

Table 5.2: Configuration of DDR4_Micron and PCM memory models: interface properties, memory system and memory timing [6] [35].

| | **Parameters** | **DRAM** | **PCM** |
|---|---|---|---|
| Interface | CLK | 2666 MT/s DDR | 800 MT/s |
| | MULT | 4 | 8 |
| | Rate | 2 | 2 |
| | BusWidth | 64 | 64 |
| | DeviceWidth | 8 | 8 |
| | BPC | 8 | 8 |
| | CPUFreq | 2000 | 2000 |
| Memory system | Banks | 4 | 1 |
| | Ranks | 4 | 4 |
| | Channels | 4 | 4 |
| | Row | 131072 | 16384 |
| | COLS | 32 | 1024 |
| | MatHeight | 131072 | 16384 |
| | UseRefresh | true | false |
| | BanksPerRefresh | 4 | 1 |
| | Delayed refresh threshold | 8 | 8 |
| Memory timing | tBURST | 4 | 4 |
| | tCMD | 1 | 1 |
| | tRAS | 42 | 0 |
| | tRCD | 19 | 48 |
| | tAL | 0 | 0 |
| | tCCD | 4 | 2 |
| | tCWD | 7 | 4 |
| | tWTR | 5 | 3 |
| | tWR | 210 | 0 |
| | tRP | 19 | 1 |
| | tCAS | 10 | 1 |
| | tRTRS | 1 | 1 |
| | tRTP | 5 | 3 |
| | tRFC | 107 | 100 |
| | tOST | 1 | 0 |
| | tRRDR | 4 | 4 |
| | tRRDW | 4 | 4 |
| | RAW | 4 | 4 |
| | tRAW | 21 | 20 |
| | tRDPDEN | 14 | 5 |
| | tWRPDEN | 19 | 68 |
| | tWRAPDEN | 20 | 68 |
| | tPD | 4 | 1 |
| | tXP | 5 | 3 |
| | tXPDLL | 16 | 200000 |
| | tXS | 5 | - |
| | tXSDLL | 854 | - |
| | tREFW | 42666667 | 42666667 |

Table 5.3: Configuration of DDR4_Micron and PCM memory models: energy/power parameters, memory controller and memory endurance model [6] [35].

| | **Parameters** | **DRAM** | **PCM** |
|---|---|---|---|
| Energy/power | UselowPower | true | - |
| | PowerDownMode | FASTEXIT | - |
| | EnergyModel | current | energy |
| | Ewrpb | 0.000202 | 0.000202 |
| | Erd | 3.405401 | 7.1513421 |
| | Ewr | 1.023750 | 44.123625 |
| | Eref | 38.558533 | 0 |
| | Eleak | - | 3120.202 |
| | Eactstdby | 0.090090 | - |
| | Eprestdby | 0.083333 | - |
| | Epda | 0.000000 | 0.000000 |
| | Epdpf | 0.078829 | 0 |
| | Epdps | 0.000000 | 0.000000 |
| | Voltage | 1.2 | 1.5 |
| | EIDD0 | 59 | - |
| | EIDD1 | 76 | - |
| | EIDD2P0 | 22 | - |
| | EIDD2P1 | 22 | - |
| | EIDD2N | 42 | - |
| | EIDD2NT | 54 | - |
| | EIDD3P | 33 | - |
| | EIDD3N | 58 | - |
| | EIDD4R | 145 | - |
| | EIDD4W | 140 | - |
| | EIDD5B | 66 | - |
| | EIDD6 | 25 | - |
| Memory controller | MEM_CTL | FRFCFS | FRFCFS-WQF |
| | CTL_DUMP | false | - |
| | ClosePage | 0 | 0 |
| | ScheduleScheme | 2 | 2 |
| | Address Mapping Scheme | SA:R:RK:BK:CH:C | R:RK:BK:CH:C |
| | INTERCONNECT | OffChipBus | OffChipBus |
| | ReadQueueSize | 32 | 32 |
| | WriteQueueSize | 32 | 32 |
| | HighWaterMark | 32 | 32 |
| | LowWaterMark | 16 | 16 |
| Endurance model | EnduranceModel | NullModel | NullModel |
| | EnduranceDist | Normal | Normal |
| | EnduranceDist Mean | 1000000 | 1000000 |
| | EnduranceDist Variance | 100000 | 100000 |
| | FlipNWrite Granularity | 32 | - |

Table 5.4: Studied applications and their description.

| **Application** | **Suite** | **Name** | **Description** |
|---|---|---|---|
| Back Propagation | Rodinia | BP | Pattern Recognition, machine Learning |
| Breadth-First Search | Rodinia | BFS | Graph Analysis |
| HotSpot 3D | Rodinia | HS-3D | Physics Simulation |
| Sparse Matrix Vector Mult. | Parboil | SpMV | Graph Analysis, Machine Learning |
| Myocyte | Rodinia | MO | Biological Simulation |
| HeartWall Tracking | Rodinia | HW | Medical Imaging |
| Stream Cluster | PARSEC | SC | Data Mining |
| VASARI Image Processing Sys. | PARSEC | VIPS | Media Processing |
| Kmeans Clustering | Starbench | Kmeans | Artificial Intelligence, Data Mining |
| Ray Tracing | Starbench | C-ray | Computer Graphics |
| Image Rotation | Starbench | Rotate | Image Processing |
| Stencil | Starbench | Stencil | Physics Simulation, Machine Learning |

Table 5.5: List of memory-intensive applications and their memory access behavior. The reported numbers are measured from Host CPU execution.

| Memory-Intensive Applications | Memory Access Behavior | | |
|---|---|---|---|
| | Memory Intensity | RBL | R-to-W Ratio |
| BP | 21.5 (High) | 24.51% (Low) | 1.63 ($< 3$) |
| MO | 7.2 (High) | 14.82% (Low) | 1.33 ($< 3$) |
| BFS | 2.4 (Middle) | 76.5% (High) | 3.93 ($> 3$) |
| HS-3D | 2.3 (Middle) | 75.61% (High) | 4.26 ($> 3$) |
| SpMV | 2.1 (Middle) | 16.24% (Low) | 4.48 ($> 3$) |

### 5.3.1 Performance Comparison

To study performance and perform speed up comparison between different processing units, the execution stage average Instruction Per Cycle (IPC) of memory-intensive applications is used as a performance metric. Ramulator [8] [74] is used in trace-driven mode with a CPU model to estimate the average IPC when NMP systems are employed. To generate the traces, Zsim [124] is used to identify region of interest (ROI) of one billion instructions for each of memory-intensive applications.

Figure 5.2 shows a performance (speedup) comparison of memory-intensive applications (BP, MO, BFS, HS-3D, and SpMV) under three different computing platforms: Host CPU system with the conventional DDR4 memory and NMP systems with processing units embedded in logic layer of 3D-DRAM and 3D-PCM. Along the x-axis, the applications are sorted by memory-intensity (LLC MPKI), from highest to least (see Table 5.5) and average IPC results are normalized to the Host CPU system.

Figure 5.2 illustrates that in both NMP systems (3D-DRAM and 3D-PCM) the average IPC of target applications has improved by 1.31x to 5x compared to the Host CPU system. Comparing two NMP systems, 3D-PCM NMP has a very negligible difference with 3D-DRAM NMP in IPC improvement, which makes it a practical and efficient NMP architecture to accelerate executing memory-intensive applications. Figure 5.3 provides further insights into the performance comparison using memory access latency metric. In this figure, Along the y-axis, represented values are normalized to the Host CPU system. Two findings can be drawn out from this figure:

1. It depicts a significant performance benefit (memory access latency reduction) for BP, MO, and SpMV applications when running them on two NMP systems (3D-PCM NMP and 3D-DRAM NMP). As it is shown in Table 5.5 (which is also discussed in Section 4.3.3), all three applications exhibit a low RBL (RB hit rate of 24.51%, 14.82%,

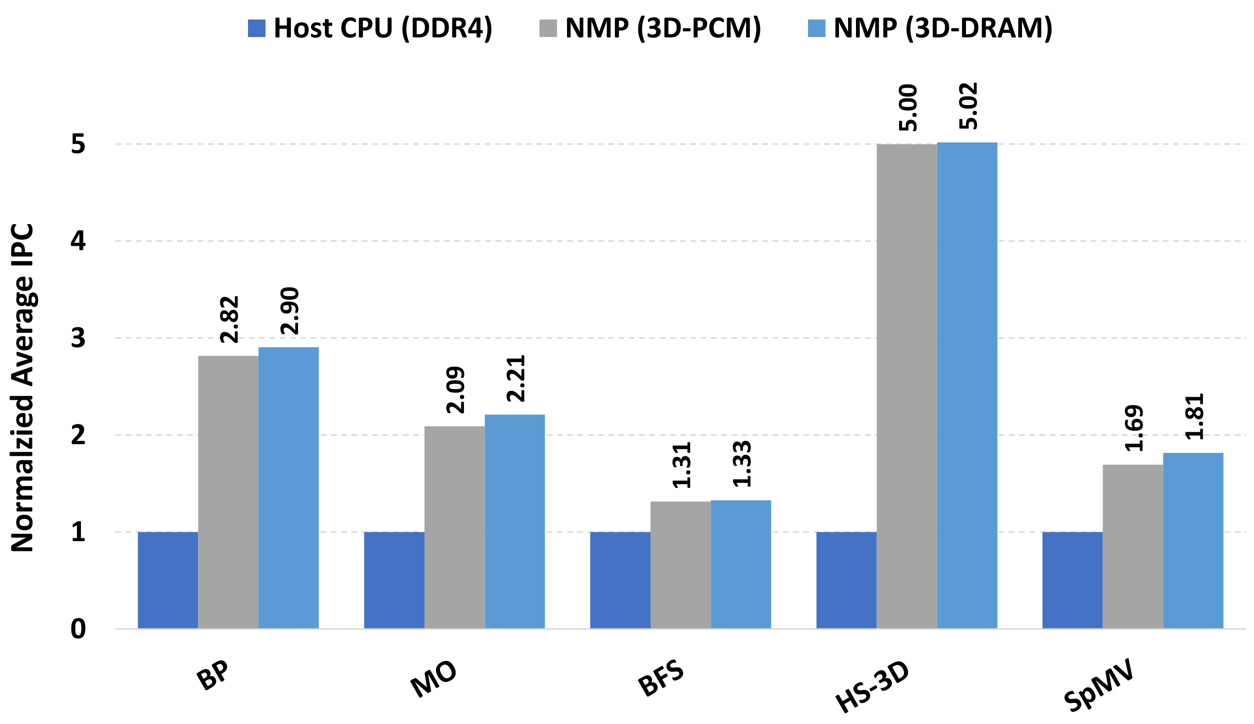


Figure 5.2: Performance (speedup) comparison based on average IPC between Host CPU, 3D-PCM NMP, and 3D-DRAM NMP systems across all memory-intensive applications. Along the x-axis, applications are sorted by memory-intensity (LLC MPKI), from highest to least. IPC results are normalized to the Host CPU system.

and 16.24%, respectively) on Host CPU system with DDR4 memory. Having poor data locality at the memory array level, these applications can benefit from 3D-stacked memories (3D-PCM and 3D-DRAM) which deliver higher bandwidth and memory-level parallelism compared to DDR4 memory. Furthermore, NMP systems eliminate data movement bottleneck which significantly improves the memory access latency for such applications.

2. Though there is an improvement in average IPC of BFS and HS-3D applications (see Figure 5.2), an increase can be observed in memory access latency of these applications when NMP systems are employed (see Figure 5.3). To understand the reason, we look at RBL locality of these applications when running them on Host CPU system with DDR4 memory. As it is shown in Table 5.5, BFS and HS-3D have a high RBL (RB

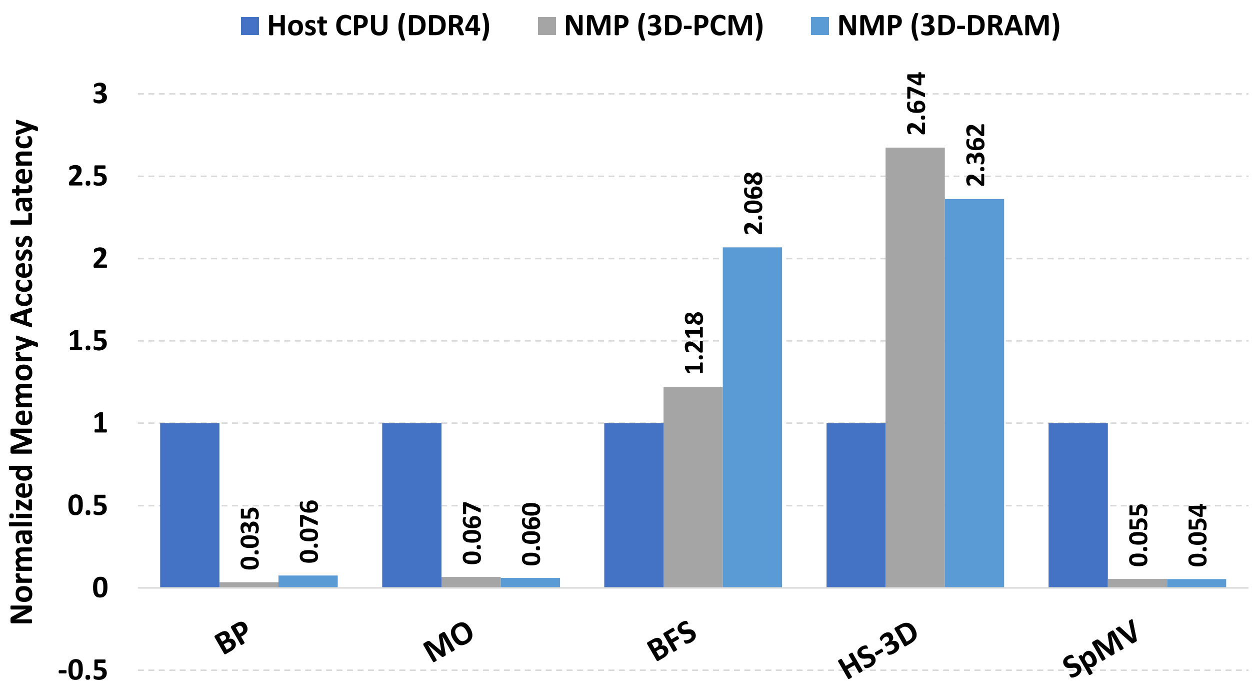


Figure 5.3: Memory access latency comparison between Host CPU, 3D-PCM NMP, and 3D-DRAM NMP systems across all memory-intensive applications, normalized to the Host CPU system. Along the x-axis, applications are sorted based on memory intensity (LLC MPKI) from highest to least.

hit rate > 75%) which is exploited by DRR4 memory because of its large row buffer size (8KB). Lower memory access latency (see Figure 5.3 for BFS and HS-3D) on Host CPU with DDR4 is the result of exploiting high data locality at memory array row. Running these applications on NMP system which is enabled by memory with very small row size (256B) increases the memory access latency, since memory row misses occur more frequently. High internal parallelism (bank-level parallelism) offered by 3D-stacked memories (which have processing cores in their logic layer) is exploited by such applications. Thus, due to the high bank-level parallelism, BFS and HS-3D exhibit a significant improvement in their IPC with NMP execution cases.

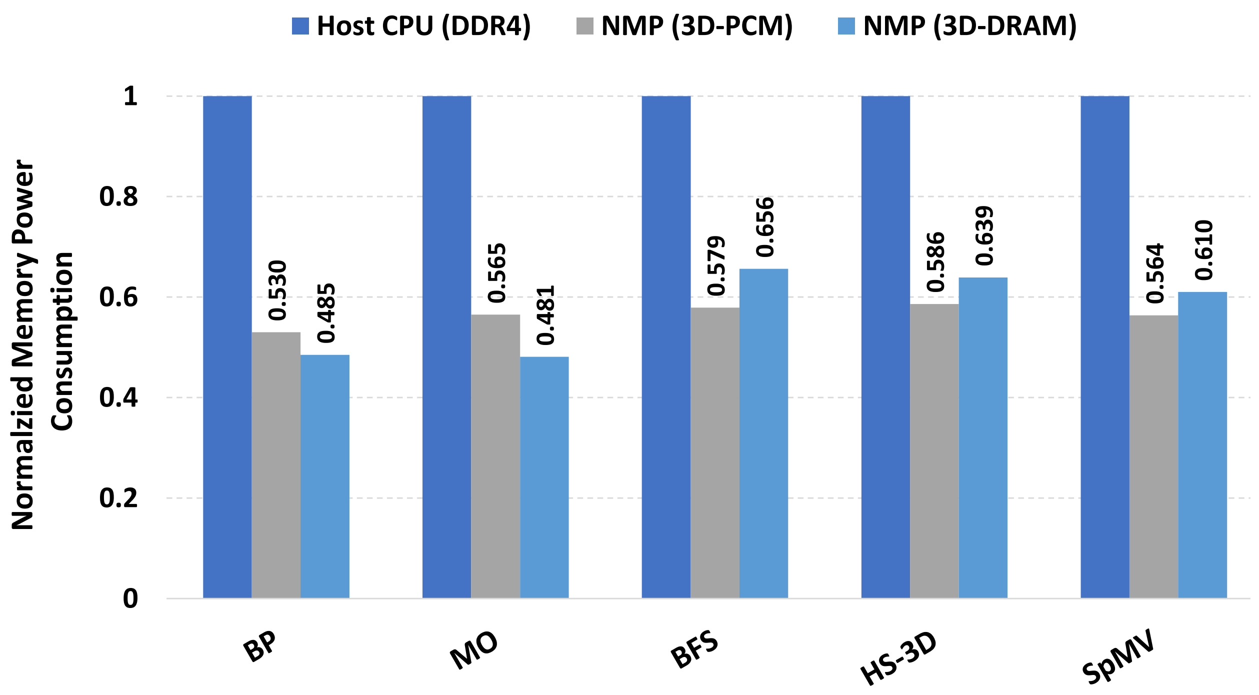


Figure 5.4: Memory power consumption across all memory-intensive applications for Host CPU and two different NMP execution cases, normalized to the Host CPU system.

### 5.3.2 Memory Power Consumption Comparison

Power analysis for DDR4 memory in Host CPU system has been done using gem5-NVMain simulator [26]. DRAMPower [1] [29] simulator is used for evaluating power consumption of memory devices (3D-DRAM and 3D-PCM) in two NMP systems.

Figure 5.4 shows the memory power consumption of memory-intensive applications for Host CPU system with DDR4 and two NMP systems with 3D-PCM and 3D-DRAM memories. Memory power consumption values are normalized to the Host CPU system. The power savings are realized across all memory-intensive applications with NMP execution which are obtained having shallow cache hierarchy in NMP systems that avoids excess memory access latency. Based on Figure 5.4, two interesting findings can be observed:

1. For BP and MO applications that have high LLC MPKI (see Table 5.5), NMP systems

(3D-PCM and 3D-DRAM) outperform the Host CPU execution by an average of 47% and 51% in memory power saving, respectively. Other applications (BFS, HS-3D, and SpMV) with middle MPKI (see Table 5.5) also experience power savings with NMP execution, still significant but lower than BP and MO.

2. While 3D-DRAM NMP outperforms 3D-PCM NMP for BP and MO applications, 3D-PCM NMP exhibits more power saving compared to 3D-DRAM NMP for other applications (BFS, HS-3D, and SpMV). This is due to the difference in applications' average R-to-W ratio (Table 5.5). Considering that PCM technology suffers from a high write energy/power, it can be inferred that for BFS, HS-3D, and SpMV, read operations and for BP and MO, write operations are the dominant factor in determining the memory power consumption. This explains the reason as to why some memory-intensive applications see more power saving than others.

## 5.4 Summary

In this Chapter, two NMP computing devices which are constructed based on 3D stacking technology (3D-DRAM and 3D-PCM) are studied to accelerate data-intensive problems caused by *memory wall* bottleneck of the conventional processing architectures. In order to reveal the performance and power bottlenecks, a systematic characterization is conducted on a wide range of multi-threaded applications (compute-intensive and memory-intensive) from different benchmark suites (Rodinia, Parboil, PARSEC, and Starbench). Overall, the system-level evaluation demonstrates that the evaluated NMP systems (3D-DRAM NMP and 3D-PCM NMP) improve the performance of memory-intensive applications by **1.31x to 5x** and reduce their total memory energy/power consumption by an average of **47%**. These improvements make the hybrid NMP system a great design technique for acceleration in performance and power across a wide range of data-intensive applications.

# Chapter 6

# Conclusion and Future Work

## 6.1 Put it All Together

With the emergence of applications that work with very large datasets (data-intensive), conventional computing systems are not efficient in handling such large-scale data. The performance and energy cost of moving this large amounts of data between off-chip memory and processing unit dominate the total cost of computation which is known as data movement bottleneck. To mitigate this bottleneck, different architectural-level techniques (processing using memory (PUM), processing-in-memory (PIM), and near memory processing based on 3D stacking (NMP)) are proposed, where excess data movement is reduced or avoided by performing computation within memory or bringing computation close to memory (data).

3D die stacking is considered to be a practical solution for embedding processing cores on the same package as memory dies which imposes a small foot print and better timing performance comparing to 2D planar architectures. In a 3D package, multiple memory layers (homogeneous or heterogeneous layers) are stacked vertically on top of a logic layer using short-length, low-power and low-latency TSV bus. By placing memory dies on the same substrate as the logic die with processing unit embedded in it, each part of the system can do its job much more optimally than any previous technology. This technology is one the most promising solutions to address the memory wall problem by allowing architects to enable ability of processing near memory. The next major new approach is the development of byte-addressable non-volatile memory that can be exploited with 3D die stacking technology to conquer previous barriers to implementing practical and efficient NMP architectures.

## 6.2 Summary of Contributions

This dissertation presented practical and efficient NMP architecture in a hybrid memory system. The goal was to identify potential reasons of data movement over a set of applications and to compare conventional computation-centric processing unit to a memory-centric technique (NMP). In particular, the contributions of this dissertation are summarized in the following sections:

- A set of NMP-centric performance metrics are redefined and investigated with the focus of application characterization on conventional multi-core Host CPU system and NMP architecture.
- As a case study, a comprehensive characterization is performed on a wide range of application domains (mixture of compute-bound and memory-bound) from different benchmark suites to reveal their performance bottleneck. The characterization is based on Roofline analysis, data locality (temporal and spatial) analysis, and memory access behavior analysis. Further, the efficacy of mapping a given processing unit to a specific application is analyzed and the potential benefits of NMP architecture over conventional Host CPU system is evaluated to efficiently accelerate data-intensive processing.
- Considering a hybrid NMP system, two NMP subsystems based on volatile and non-

volatile 3D-stacked memory technologies (3D-DRAM and 3D-PCM) are explored and the impact of constructing NMP architecture based on an emerging NVM technology is analyzed.

- Finally, it is demonstrated that executing certain data-intensive (memory-intensive) applications on NMP architecture which is constructed based on 3D-NVM (3D-PCM) can improve performance and reduce memory power consumption compared to 3D-DRAM based NMP and conventional Host CPU executions.

## 6.3 Future Work

The research contributions in this dissertation introduce some promising directions for future research work.

- On the memory side, two memory technologies (DRAM and PCM) can be combined to have a heterogeneous 3D memory architecture for NMP system. This memory architecture will have attractive properties. Performance of the memory system can efficiently improve by stacking DRAM layers with PCM layers together on top of a logic die in a 3D structure. This hybrid NMP architecture can provide excellent speedup with smaller area, since it takes advantages of both DRAM and PCM technology in one package. This architecture can be used for custom data-intensive applications such as graph processing. Also, other non-volatile memory technologies (such as ReRAM and MRAM) can be explored to architect hybrid NMP systems.
- Absence of coherency and virtual memory support makes programming difficult which prevents advocacy of this computing model. Supporting both coherency and virtual memory in this architecture can be considered as a future work. This type of challenges requires organized work across both hardware and software.

- Concurrent execution of both host processor and NMP cores for processing hybrid applications can be considered as an interesting future work.

## 6.4 Concluding Remarks

In conclusion, this dissertation provides an insight to computer architects to leverage from emerging non-volatile memory technologies for improving performance and efficiency of data-intensive applications. It helps the near memory processing researchers to rethink the design of computing systems in a way to benefit from hybrid memory architecture in the NMP context.

# Bibliography

[1] Drampower: Open-source dram power & energy estimation tool. [online]. available:. `http://www.drampower.info`.

[2] HBM. JEDEC Standard, High Bandwidth Memory JESD235B, 2018.

[3] HMC chip architecture:. `https://community.cadence.com/cadence_blogs_8/b/fv/posts/what-s-new-with-hybrid-memory-cube-hmc`.

[4] HMC. Consortium, Hybrid Memory Cube Specification 2.0, 2014.

[5] Memory wall problem and moore's law, figure is from:. Forget Moore's law: Hot and slow DRAM is a major roadblock to exascale and beyond.

[6] Micron ddr4 sdram, 16gb:. `https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr4/16gb_ddr4_sdram.pdf`.

[7] Parboil benchmarks:. `http://impact.crhc.illinois.edu/parboil/parboil.aspx`.

[8] Safari research group: Ramulator for processing in memory. `https://github.com/CMU-SAFARI/ramulator-pim`.

[9] Starbench benchmark suite:. `https://www.aes.tu-berlin.de/menue/forschung/projekte/abgeschlossene_projekte/starbench_parallel_benchmark_suite/`.

[10] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, and et. al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems, 2015.

[11] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, 2016. USENIX Association.

[12] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, and et. al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[13] M. Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[14] A. Albaqsami, M. S. Hosseini, and N. Bagherzadeh. Htf-mpr: A heterogeneous tensorflow mapper targeting performance using genetic algorithms and gradient boosting regressors. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 331–336, March 2018.

[15] A. Albaqsami, M. S. Hosseini, M. Jasemi, and N. Bagherzadeh. Adaptive htf-mpr: An adaptive heterogeneous tensorflow mapper utilizing bayesian optimization and genetic algorithms. *ACM Trans. Intell. Syst. Technol.*, 11(5), Aug. 2020.

[16] M. Alian, S. W. Min, H. Asgharimoghaddam, A. Dhar, D. K. Wang, T. Roewer, A. McPadden, O. O'Halloran, D. Chen, J. Xiong, D. Kim, W.-m. Hwu, and N. S. Kim. Application-transparent near-memory processing architecture with memory channel network. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, page 802–814. IEEE Press, 2018.

[17] S. Aminikhanghahi and D. J. Cook. A survey of methods for time series change point detection. 51(2):339–367, 2016.

[18] M. Andersch, B. Juurlink, and C. C. Chi. A benchmark suite for evaluating parallel programming models. *PARS: Parallel-Algorithmen, -Rechnerstrukturen und -Systemsoftware*, 28:7–17, 10 2014.

[19] S. Angizi and D. Fan. Graphide: A graph processing accelerator leveraging in-dram-computing. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, GLSVLSI '19, page 45–50, New York, NY, USA, 2019. Association for Computing Machinery.

[20] T. T. Authors. mnist classifier using softmax in tensorflow. `https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/tutorials/mnist/`, 2017.

[21] T. T. Authors. tensorflow device factory. `https://github.com/tensorflow/tensorflow/blob/master/tensorflow`, 2017.

[22] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson. Near-data processing: Insights from a micro-46 workshop. *IEEE Micro*, 34(4):36–42, 2014.

[23] J. Bergstra, D. Yamins, and D. D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages I–115–I–123. JMLR.org, 2013.

[24] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 2546–2554. Curran Associates, Inc., 2011.

[25] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *PACT 2008*, pages 72–81, New York, NY, USA, 2008. ACM.

[26] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[27] J. Boukhobza, S. Rubini, R. Chen, and Z. Shao. Emerging nvm: A survey on architectural integration and research challenges. 23(2), Nov. 2017.

[28] I. Chakraborty, A. Jaiswal, A. Saha, S. Gupta, and K. Roy. Pathways to efficient neuromorphic computing with non-volatile memory technologies. *Applied Physics Reviews*, 7(2):021308, 2020.

[29] K. Chandrasekar, C. Weis, B. Akesson, N. Wehn, and K. Goossens. System and circuit level power modeling of energy-efficient 3d-stacked wide i/o drams. In *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 236–241, 2013.

[30] K. K. Chang, A. Kashyap, H. Hassan, S. Ghose, K. Hsieh, D. Lee, T. Li, G. Pekhimenko, S. Khan, and O. Mutlu. Understanding latency variation in modern dram chips: Experimental characterization, analysis, and optimization. *SIGMETRICS Perform. Eval. Rev.*, 44(1):323–336, June 2016.

[31] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, Oct 2009.

[32] A. Chen. A review of emerging non-volatile memory (nvm) technologies and applications. *Solid-State Electronics*, 125:25–38, 2016. Extended papers selected from ESSDERC 2015.

[33] S. Chen, P. B. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *CIDR*, January 2011.

[34] Y. Chen. Reram: History, status, and future. *IEEE Transactions on Electron Devices*, 67(4):1420–1433, 2020.

[35] Y. Choi, I. Song, M.-H. Park, H. Chung, S. Chang, B. Cho, J. Kim, Y. Oh, D. Kwon, J. Sunwoo, J. Shin, Y. Rho, C. Lee, M. G. Kang, J. Lee, Y. Kwon, S. Kim, J. Kim, Y.-J. Lee, Q. Wang, S. Cha, S. Ahn, H. Horii, J. Lee, K. Kim, H. Joo, K. Lee, Y.-T. Lee, J. Yoo, and G. Jeong. A 20nm 1.8v 8gb pram with 40mb/s program bandwidth. In *2012 IEEE International Solid-State Circuits Conference*, pages 46–48, 2012.

[36] F. Chollet. keras. `https://github.com/charlespwd/project-title`, 2015.

[37] R. Collobert and J. Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 160–167, New York, NY, USA, 2008. ACM.

[38] E. Cooper-Balis, P. Rosenfeld, and B. Jacob. Buffer-on-board memory systems. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 392–403, 2012.

[39] Cyjseagull. gem5-NVMain hybrid simulator. `https://github.com/cyjseagull/gem5-nvmain-hybrid-simulator`, Apr. 2016.

[40] W. Davis, J. Wilson, S. Mick, J. Xu, H. Hua, C. Mineo, A. Sule, M. Steer, and P. Franzon. Demystifying 3d ics: the pros and cons of going vertical. *IEEE Design Test of Computers*, 22(6):498–510, 2005.

[41] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, Oct. 1974.

[42] B. DeSalvo, V. Sousa, L. Perniola, C. Jahan, S. Maitrejean, J. Nodin, C. Cagli, V. Jousseaume, G. Molas, E. Vianello, C. Charpin, and E. Jalaguier. Emerging memory technologies: Challenges and opportunities. In *Proceedings of Technical Program of 2012 VLSI Technology, System and Application*, pages 1–2, 2012.

[43] P. A. Diaz-Gomez and D. F. Hougen. Initial population for genetic algorithms: A metric approach. In *GEM*, pages 43–49, 2007.

[44] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaaauw, and R. Das. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 383–396, 2018.

[45] D. Eggleston. 3D XP: What the hell?!! `https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2015/20150813_S301C_Eggleston.pdf`.

[46] A. Eghbal, P. M. Yaghini, and N. Bagherzadeh. Capacitive coupling mitigation for tsv-based 3d ics. In *2015 IEEE 33rd VLSI Test Symposium (VTS)*, pages 1–6, 2015.

[47] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *ISCA*, pages 365–376, June 2011.

[48] J. H. Friedman. Stochastic gradient boosting. *Computational Statistics and Data Analysis*, 38(4):367 – 378, 2002. Nonlinear Methods and Data Mining.

[49] A. Gamatie, A. Nocua, G. Sassatelli, D. Novo, M. Robert, and L. Torres. D3.7 - final report on memory hierarchy investigations. `https://www.montblanc-project.eu/wp-content/uploads/2019/02/MB3_D3.7-Final-Report-on-Memory-Hierarchy-Investigation-1.pdf`.

[50] M. Gao. *Scalable Near-Data Processing Systems for Data-Intensive Applications*. PhD thesis, 2018. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2021-05-18.

[51] M. Gao, G. Ayers, and C. Kozyrakis. Practical near-data processing for in-memory analytics frameworks. In *PACT 2015*, pages 113–124. IEEE, 2015.

[52] E. C. Garrido-Merchán and D. Hernández-Lobato. Dealing with categorical and integer-valued variables in bayesian optimization with gaussian processes, 05 2018.

[53] M. Gokhale, B. Holmes, and K. Iobst. Processing in memory: the terasys massively parallel pim array. *Computer*, 28(4):23–31, 1995.

[54] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[55] S. H Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li. Ndc: Analyzing the impact of 3d-stacked memory+logic devices on mapreduce workloads. pages 190–200, 03 2014.

[56] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park. Mapping irregular applications to diva, a pim-based data-intensive architecture. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, SC '99, page 57–es, New York, NY, USA, 1999. Association for Computing Machinery.

[57] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, page 37–47, New York, NY, USA, 2010. Association for Computing Machinery.

[58] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[59] J. H. Holland. *Adaptation in Natural and Artificial Systems.* MIT Press, Cambridge, MA, USA, 1992.

[60] M. Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, 2014.

[61] K. Hsieh, G. Ananthanarayanan, P. Bodik, S. Venkataraman, P. Bahl, M. Philipose, P. B. Gibbons, and O. Mutlu. Focus: Querying large video datasets with low latency and low cost. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 269–286, Carlsbad, CA, Oct. 2018. USENIX Association.

[62] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[63] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[64] K. Itoh. *VLSI memory chip design*, volume 5. Springer Science & Business Media, 2013.

[65] J. Jeddeloh and B. Keeth. Hybrid memory cube new dram architecture increases density and performance. In *2012 Symposium on VLSI Technology (VLSIT)*, pages 87–88, 2012.

[66] L. Jiang, S. Mittal, and W. Wen. Building a fast and power efficient inductive charge pump system for 3d stacked phase change memories. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, GLSVLSI '17, page 275–280, New York, NY, USA, 2017. Association for Computing Machinery.

[67] M. Kang, M.-S. Keel, N. R. Shanbhag, S. Eilert, and K. Curewitz. An energy-efficient vlsi architecture for pattern recognition via deep embedding of computation in sram. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 8326–8330, 2014.

[68] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas. Flexram: toward an advanced intelligent memory system. In *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors (Cat. No.99CB37040)*, pages 192–201, 1999.

[69] D. Kau, S. Tang, I. V. Karpov, R. Dodge, B. Klehn, J. A. Kalb, J. Strand, A. Diaz, N. Leung, J. Wu, S. Lee, T. Langtry, K. wei Chang, C. Papagianni, J. Lee, J. Hirst, S. Erra, E. Flores, N. Righos, H. Castro, and G. Spadini. A stackable cross point phase change memory. In *2009 IEEE International Electron Devices Meeting (IEDM)*, pages 1–4, 2009.

[70] S. Kawashima and J. S. Cross. Feram. In *Embedded Memories for Nano-Scale VLSIs*, pages 279–328. Springer, 2009.

[71] T. Kgil, A. Saidi, N. Binkert, S. Reinhardt, K. Flautner, and T. Mudge. Picoserver: Using 3d stacking technology to build energy efficient servers. *J. Emerg. Technol. Comput. Syst.*, 4(4), Nov. 2008.

[72] J.-S. Kim, C. S. Oh, H. Lee, D. Lee, H. R. Hwang, S. Hwang, B. Na, J. Moon, J.-G. Kim, H. Park, J.-W. Ryu, K. Park, S. K. Kang, S.-Y. Kim, H. Kim, J.-M. Bang, H. Cho, M. Jang, C. Han, J.-B. LeeLee, J. S. Choi, and Y.-H. Jun. A 1.2 v 12.8 gb/s 2 gb mobile wide-i/o dram with 4 $\times$ 128 i/os using tsv based stacking. *IEEE Journal of Solid-State Circuits*, 47(1):107–116, 2012.

[73] M. Kim, H.-J. Lee, and H. Kim. An on-demand scrubbing solution for read disturbance error in phase-change memory. In *2020 International Conference on Electronics, Information, and Communication (ICEIC)*, pages 1–2. IEEE, 2020.

[74] Y. Kim, W. Yang, and O. Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Computer Architecture Letters*, 15(1):45–49, 2016.

[75] D. E. Knuth. Postscript about np-hard problems. *SIGACT News*, 6(2):15–16, Apr. 1974.

[76] P. M. Kogge. Execube-a new architecture for scaleable mpps. In *1994 International Conference on Parallel Processing Vol. 1*, volume 1, pages 77–84, 1994.

[77] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, pages 1137–1143, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[78] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, pages 1137–1143, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[79] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[80] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[81] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. Evaluating stt-ram as an energy-efficient main memory alternative. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 256–267. IEEE, 2013.

[82] T. Kwon, M. Imran, and J.-S. Yang. Cost-effective reliable mlc pcm architecture using virtual data based error correction. *IEEE Access*, 8:44006–44018, 2020.

[83] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. Evaluating stt-ram as an energy-efficient main memory alternative. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 256–267, 2013.

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

[85] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521:436–44, 05 2015.

[86] Y. LeCun and C. Cortes. MNIST handwritten digit database. 2010.

[87] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger. Phase-change technology and the future of main memory. *IEEE Micro*, 30(1):143–143, Jan 2010.

[88] D. Lee, S. Ghose, G. Pekhimenko, S. Khan, and O. Mutlu. Simultaneous multi-layer access: Improving 3d-stacked memory bandwidth at low cost. *ACM Trans. Archit. Code Optim.*, 12(4), Jan. 2016.

[89] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. Keller. Energy management for commercial servers. *Computer*, 36(12):39–48, 2003.

[90] Y. Levy, J. Bruck, Y. Cassuto, E. G. Friedman, A. Kolodny, E. Yaakobi, and S. Kvatinsky. Logic operations in memory using a memristive akers array. *Microelectron. J.*, 45(11):1429–1437, Nov. 2014.

[91] B. Li, B. Yan, and H. Li. An overview of in-memory processing with emerging non-volatile memory for data-intensive applications. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, GLSVLSI '19, page 381–386, New York, NY, USA, 2019. Association for Computing Machinery.

[92] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2016.

[93] J. A. Mandelman, R. H. Dennard, G. B. Bronner, J. K. DeBrosse, R. Divakaruni, Y. Li, and C. J. Radens. Challenges and future directions for the scaling of dynamic random-access memory (dram). *IBM Journal of Research and Development*, 46(2.3):187–212, 2002.

[94] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, Dec. 1995.

[95] J. Meza, J. Li, and O. Mutlu. Evaluating row buffer locality in future non-volatile main memories. 12 2018.

[96] M. Mitchell. *An introduction to genetic algorithms.* MIT press, 1998.

[97] T. Mitchell. *Machine Learning.* McGraw-Hill Education, 1997.

[98] S. Mittal. A survey of reram-based architectures for processing-in-memory and neural networks. *Machine learning and knowledge extraction*, 1(1):75–114, 2019.

[99] S. Mittal and J. S. Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1537–1550, 2015.

[100] S. Mittal and J. S. Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE TPDS*, 27(5):1537–1550, May 2016.

[101] S. Mittal, J. S. Vetter, and L. Jiang. Addressing read-disturbance issue in stt-ram by data compression and selective duplication. *IEEE Computer Architecture Letters*, 16(2):94–98, 2016.

[102] S. Mittal, J. S. Vetter, and D. Li. A survey of architectural approaches for managing embedded dram and non-volatile on-chip caches. *IEEE Transactions on Parallel and Distributed Systems*, 26(6):1524–1537, 2014.

[103] G. E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, 2006.

[104] M. Motoyoshi. Through-silicon via (tsv). *Proceedings of the IEEE*, 97(1):43–48, 2009.

[105] O. Mutlu. Opportunities and challenges of emerging memory technologies 2017. ARM research summit. [online] Available.

[106] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun. A Modern Primer on Processing in Memory. *arXiv e-prints*, page arXiv:2012.03112, Dec. 2020.

[107] O. Mutlu and J. S. Kim. Rowhammer: A retrospective. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 39(8):1555–1571, Aug. 2020.

[108] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C.-Y. Cher, C. H. A. Costa, J. Doi, C. Evangelinos, B. M. Fleischer, T. W. Fox, D. S. Gallo, L. Grinberg, J. A. Gunnels, A. C. Jacob, P. Jacob, H. M. Jacobson, T. Karkhanis, C. Kim, J. H. Moreno, J. K. O'Brien, M. Ohmacht, Y. Park, D. A. Prener, B. S. Rosenburg, K. D. Ryu, O. Sallenave, M. J. Serrano, P. D. M. Siegl, K. Sugavanam, and Z. Sura. Active memory cube: A processing-in-memory architecture for exascale systems. *IBM Journal of Research and Development*, 59(2/3):17:1–17:14, 2015.

[109] R. Neale, D. L. Nelson, and G. Moore. Non - volatile and reprogrammable, the read mostly memory is here. 1970.

[110] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. Chung. Accelerating deep convolutional neural networks using specialized hardware, February 2015.

[111] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper*, 2(11), 2015.

[112] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent ram. *IEEE Micro*, 17(2):34–44, 1997.

[113] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[114] M. Pelikan, D. E. Goldberg, and E. Cantú-Paz. Boa: The bayesian optimization algorithm. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 1*, GECCO'99, pages 525–532, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[115] M. Pinedo and K. Hadavi. Scheduling: Theory, algorithms and systems development. In *Operations Research Proceedings 1991*, pages 35–42. Springer, 1992.

[116] M. Poremba, T. Zhang, and Y. Xie. Nvmain 2.0: A user-friendly memory simulator to model (non-)volatile memory systems. *IEEE Computer Architecture Letters*, 14(2):140–143, July 2015.

[117] L. E. Ramos, E. Gorbatov, and R. Bianchini. Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing*, ICS '11, page 85–95, New York, NY, USA, 2011. Association for Computing Machinery.

[118] P. Ranganathan. From microprocessors to nanostores: Rethinking data-centric systems. *Computer*, 44(01):39–48, jan 2011.

[119] S. Rashidi, M. Jalili, and H. Sarbazi-Azad. Improving mlc pcm performance through relaxed write and read for intermediate resistance levels. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(1):1–31, 2018.

[120] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, Oct. 1986.

[121] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Neurocomputing: Foundations of research. chapter Learning Representations by Back-propagating Errors, pages 696–699. MIT Press, Cambridge, MA, USA, 1988.

[122] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, and et. al. Imagenet large scale visual recognition challenge. *arXiv preprint arXiv:1409.0575*, 2014.

[123] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, and et. al. Imagenet large scale visual recognition challenge. *arXiv preprint arXiv:1409.0575*, 2014.

[124] D. Sanchez and C. Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. *SIGARCH Comput. Archit. News*, 41(3):475–486, June 2013.

[125] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, Jan 2015.

[126] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

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

[128] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry. Buddy-ram: Improving the performance and efficiency of bulk bitwise operations using dram, 2016.

[129] M. SeyyedHosseini. *Reliability Enhancement of Many-core Processors*. PhD thesis, 2017. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2021-05-21.

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

[131] S. Shahhosseini, A. Albaqsami, M. Jasemi, and N. Bagherzadeh. Partition pruning: Parallelization-aware pruning for deep neural networks, 2019.

[132] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, Jan 2016.

[133] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[134] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[135] G. Singh, L. Chelini, S. Corda, A. J. Awan, S. Stuijk, R. Jordans, H. Corporaal, and A.-J. Boonstra. Near-memory computing: Past, present, and future, 2019.

[136] G. Singh, J. Gómez-Luna, G. Mariani, G. F. Oliveira, S. Corda, S. Stuijk, O. Mutlu, and H. Corporaal. Napel: Near-memory computing application performance prediction via ensemble learning. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019.

[137] H. S. Stone. A logic-in-memory computer. *IEEE Transactions on Computers*, C-19(1):73–78, 1970.

[138] L. G. Szafaryn, K. Skadron, and J. J. Saucerman. Experiences accelerating matlab systems biology applications.

[139] D. Takashima. Overview of ferams: Trends and perspectives. In *2011 11th Annual Non-Volatile Memory Technology Symposium Proceeding*, pages 1–6. IEEE, 2011.

[140] U. S. B. tool. SuperPosition Software. `https://benchmark.unigine.com/superposition/`, 2017.

[141] A. W. Topol, D. C. L. Tulipe, L. Shi, D. J. Frank, K. Bernstein, S. E. Steen, A. Kumar, G. U. Singco, A. M. Young, K. W. Guarini, and M. Ieong. Three-dimensional integrated circuits. *IBM Journal of Research and Development*, 50(4.5):491–506, 2006.

[142] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science (Vol. A): Algorithms and Complexity*. MIT Press, Cambridge, MA, USA, 1990.

[143] V. Vanhoucke, A. Senior, and M. Z. Mao. Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

[144] V. Vanhoucke, A. Senior, and M. Z. Mao. Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

[145] M. Vogt and H. Dette. Detecting gradual changes in locally stationary processes. *Ann. Statist.*, 43(2):713–740, 04 2015.

[146] L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski. Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. *J. Parallel Distrib. Comput.*, 47(1):8–22, Nov. 1997.

[147] Z. Wang, H. Huang, J. Zhang, and G. Alonso. Shuhai: Benchmarking high bandwidth memory on fpgas. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 111–119. IEEE, 2020.

[148] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *ACM 2009.*

[149] H.-S. P. Wong, H.-Y. Lee, S. Yu, Y.-S. Chen, Y. Wu, P.-S. Chen, B. Lee, F. T. Chen, and M.-J. Tsai. Metal–oxide rram. *Proceedings of the IEEE*, 100(6):1951–1970, 2012.

[150] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *Proceedings 22nd Annual International Symposium on Computer Architecture*, pages 24–36, 1995.

[151] Q. Wu, K. Rose, J.-Q. Lu, and T. Zhang. Impacts of though-dram vias in 3d processor-dram integrated systems. In *2009 IEEE International Conference on 3D System Integration*, pages 1–6, 2009.

[152] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, Mar. 1995.

[153] S. L. Xi, A. Augusta, M. Athanassoulis, and S. Idreos. Beyond the wall: Near-data processing for databases. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*, DaMoN'15, New York, NY, USA, 2015. Association for Computing Machinery.

[154] C. Xu, D. Niu, N. Muralimanohar, N. P. Jouppi, and Y. Xie. Understanding the trade-offs in multi-level cell reram memory design. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2013.

[155] L. Xu, D. P. Zhang, and N. Jayasena. Scaling deep learning on multiple in-memory processors. 2015.

[156] P. M. Yaghini. *Resilient 3D Network-on-Chip Design and Analysis*. PhD thesis, 2016. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2021-05-20.

[157] P. M. Yaghini, A. Eghbal, S. S. Yazdi, N. Bagherzadeh, and M. M. Green. Capacitive and inductive tsv-to-tsv resilient approaches for 3d ics. *IEEE Transactions on Computers*, 65(3):693–705, 2016.

[158] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer. Imagenet training in minutes. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, pages 1:1–1:10, New York, NY, USA, 2018. ACM.

[159] S. Yu and P.-Y. Chen. Emerging memory technologies: Recent trends and prospects. *IEEE Solid-State Circuits Magazine*, 8(2):43–56, 2016.

[160] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski. Top-pim: Throughput-oriented programmable processing in memory. In *HPDC '14*, pages 85–98, New York, NY, USA, 2014. ACM.

[161] W. Zhang and T. Li. Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures. In *PACT*, pages 101–112, Sep. 2009.

[162] W. Zhang and T. Li. Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 101–112, 2009.

[163] M. Zhao, L. Jiang, Y. Zhang, and C. J. Xue. Slc-enabled wear leveling for mlc pcm considering process variation. In *Proceedings of the 51st Annual Design Automation Conference*, pages 1–6, 2014.

# Appendix A

# Algorithmic Approaches to Accelerate Emerging Applications

## A.1 Approach One: Heterogeneous TensorFlow Mapper

TensorFlow [10] is a library developed by Google to implement Artificial Neural Networks using computational dataflow graphs. The neural network has many iterations during training. A distributed, parallel environment is ideal to speed-up learning. Parallelism requires proper mapping of devices to TensorFlow operations. We developed HTF-MPR framework for that reason. HTF-MPR utilizes a genetic algorithm approach to search for the best mapping that outperforms the default Tensorflow mapper. By using Gradient Boosting Regressors to create the fitness predictive model, the search space is expanded which increases the chances of finding a solution mapping. Our results on well-known neural network benchmarks, such as ALEXNET, MNIST softmax classifier, and VGG-16, show an overall speedup in the training stage by 1.18, 3.33, and 1.13, respectively.

### A.1.1 Introduction

Machine Learning (ML) algorithms [97] have found a large number of applications in computer vision, data tracking, recommender systems, search engines and Artificial Intelligence (AI) in games. One notable advancement in ML algorithms is the use of Artificial Neural Networks (ANNs) [84]. ANNs are constructs that mimic how the brain works. In their most basic form, they consist of synapses and neurons, where the synapses are the weights and neurons are the functions (see Fig A.1). Companies invest in improving and utilizing ANNs for different tasks [37], leading to many applications applied to ANNs, creating deep and complex ANN architectures, finding techniques and accelerating the training and the inference of ANNs [143, 62, 111]. A number of software libraries have been developed to ease the construction of ANNs for end-users. One such library is TensorFlow [12]; a computational graph and numerical models library developed by Google. The application programming interface (API) makes it possible for data scientists to work with large models and many data samples in a distributed system without prior knowledge of the hardware architecture.

The current state-of-the-art ANNs consist of hundreds of thousands of parameters, and require large data sets to train. The number of layers, features (inputs) and interconnections, result in a large number of parameters that require training, which prolongs the training process.

Training in ANNs are iterative [120]; in each iteration, the process would require a feed-forward step through the ANN, and a back-propagation that flows backwards. With each iteration, a set of data-samples (batch, or mini-batch) are fed to the ANN. This modifies the parameters (weights and biases) which reduces a given loss-function.

In TensorFlow, parameters, functions, and inputs are represented by computational graphs [10]. Computational graphs consist of edges and vertices in a Directed A-cyclic Graph (DAG). Edges carry multi-dimensional arrays known as tensors, and vertices are the functions, known

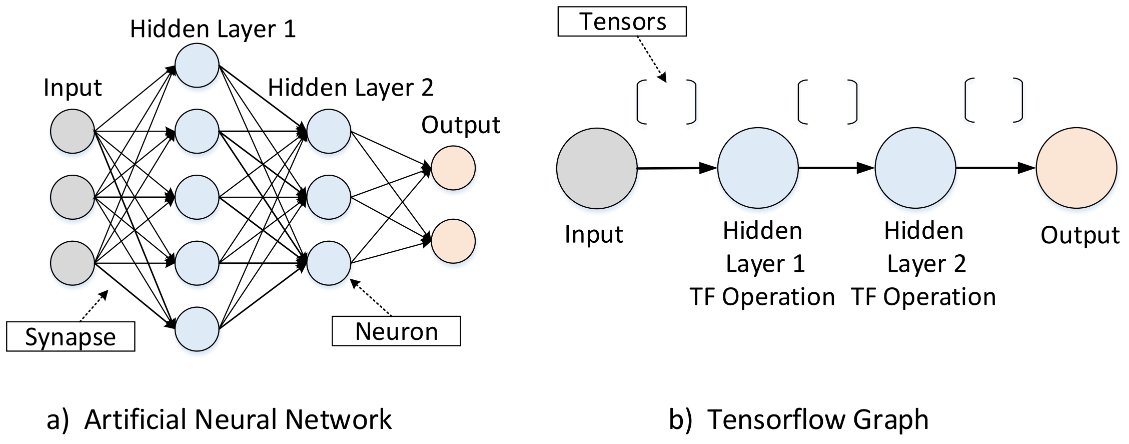


Figure A.1: Artificial Neural Network and its TensorFlow depiction

as operations, applied to tensors. A simple translation from ANN to a TensorFlow computational graph is shown in Fig A.1b.

Speedup of these computational graphs is of importance. One such approach is to reduce the number of parameters in a ANN [58]. In [58], the authors compressed the ANN by pruning the number of neurons and synapses, which reduces the number of computations. However, this would slightly change the accuracy of the prediction model [58]. In regards to TensorFlow, pruning would require a lot of invasive changes such as changing tensors or using sparse tensors. We intend to keep the prediction accuracy of the ANN intact.

Another approach is to allocate resources to operations efficiently in the TensorFlow computational graph, i.e., splitting up the ANN so that different processors may work in parallel. In TensorFlow, the graph is constructed at design-time and then run [12]. After constructed and runs for the first time, no modification is allowed to the structure of the computational graph. A work around of such limitation to reconstruct the computational graph. Currently, TensorFlow carries out the mapping in a simplified way as specified in *device_factory.cc* [21]. Luckily, the TensorFlow API allows the programmer to override the default mapping in a per operation manner. Note that scheduling is taken care of by the TensorFlow engine and the API has no access to manipulate the scheduling.

With the above approach, there are two ways to get a better (faster runtime) mapping. One is to use static-list-based mapping algorithms such as Heterogeneous Earliest Finish Time (HEFT); a fast Heuristic greedy approach with a well proven record. To utilize HEFT, three pieces of information are required; the operation dependencies (represented by the DAG), the execution time of said operations on every device, and the communication cost between each device given each operation. Unfortunately, the later two can not be obtained; both are a limitation to the API, while communication cost would require a very large number of mappings to be tested.

With the absence of the above mentioned pieces of information, another way would be to use a meta-heuristic approach. In HTF-MPR, A genetic-algorithms-based [96] approach is used. Genetic algorithms (GA) have been used in many combinatorial problems and have provided good solutions in heterogeneous computing mapping problems [146].

In GA-based approaches, the following steps are taken: 1. *initial population* of mappings are generated. 2. The *fitness* of each mapping is obtained. 3. The breeding of new mappings according to crossovers.

steps *2* and *3* are repeated for a prescribed number of times, until a solution is found (or stop due to time constraints).

To be able to generate many mappings and obtain their fitness, while bypassing the overhead of actually running the TensorFlow computational graph of said benchmark, a predictive model of the fitness is to be used. We use an ML ensemble algorithm, called *Gradient Boosting Regressors* (GBR) [48], to construct the mapping-to-fitness predictive model. The initial population of mappings and their actual fitness are used to construct the said predictive model, and the accuracy of the model is tested using Kendall tau rank distance as a metric. The metric takes into affect the pair-wise agreement between two lists (in this case between

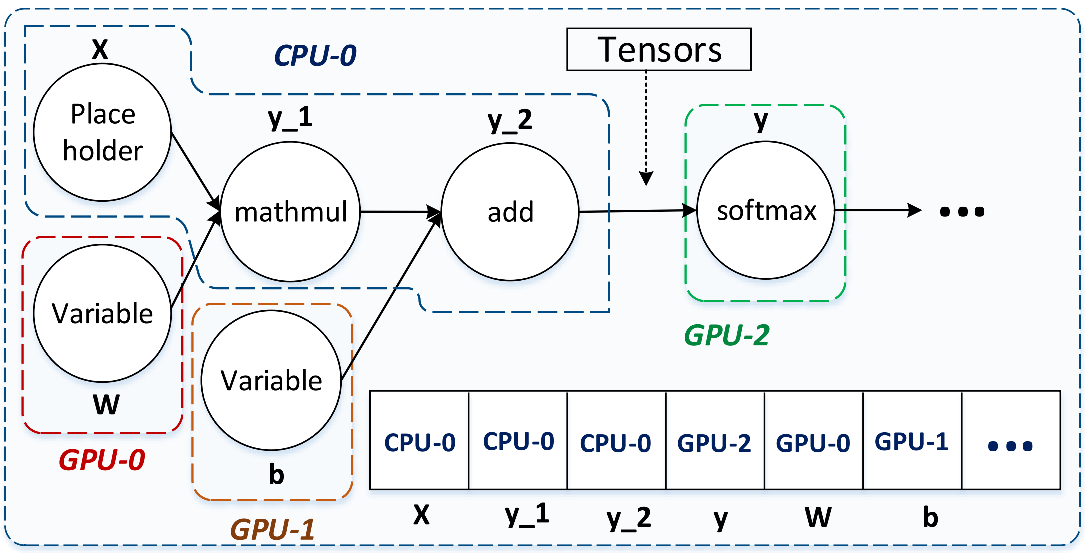


Figure A.2: Example of a model in TensorFlow, and of device-operation mapping

the order of the actual fitness of the mappings versus the order of the predicted fitness of the mappings). We used k-fold cross validation method [77] to validate the accuracy of the Kendall tau rank distance of the mappings.

## A.1.2 Background

### TensorFlow

TensorFlow is a library for constructing machine learning algorithms [12]. One of the upsides of TensorFlow is th ease of use and seamless integration into heterogeneous systems. Models in TensorFlow are described with Directed Graphs, where the input/output to/from each TensorFlow operation is zero or more tensors(see Fig A.2). The following is an example of a simple code snippet that describes a TensorFlow dataflow graph in Python (taken from [20] with slight modification): Code A.1 is represented in Fig A.2 (without the device mapping).

Code A.1: Model in TensorFlow

```
import tensorflow as tf

x = tf.placeholder(tf.float32, [None, 784])

W = tf.Variable(tf.zeros([784, 10]))

b = tf.Variable(tf.zeros([10]))

y_1 = tf.matmul(x, W)

y_2 = tf.add(y_1,b)

y = tf.nn.softmax(y_2)

...
```

The graph is then described but not yet constructed. To create and run the graph the following is done (see Code A.2):

Code A.2: Run model in Session

```
...

sess = tf.Session()

sess.run(y,feeddict=...)

...
```

In Code A.2, a *Session* is created and the operation for which an output is desired is given as the input argument to the session object's run method (in this case $y$). A single run would provide the actual *makespan* of the graph. The makespan is the time it takes to complete one *iteration* (i.e run) of the graph. TensorFlow would take care of the rest; the **Distributed Master** would take in the graph and evaluate the nodes and distributes the tasks. The **worker services** would take in requests from the master and schedule the execution of the tasks.

### Task Mapping

TensorFlow uses dataflow graphs for the computation, in addition the design of most ANNs is done in a DAG approach. In this work, we assume that the dataflow graphs are DAG, and that no changes to the mapping may occur during runtime. Note that **edges** carry **tensors**, and **vertices** are the **operations**. During design-time, the API user has the option of assigning a **device** (CPU-0, GPU-0, GPU-1, etc) to each TensorFlow operation rather than TensorFlow default mapping. No changes may occur while the TensorFlow graph is running. The structure of a single mapping $M_i$ is shown in the array in Fig A.2. Note that $M_i \in \boldsymbol{M}_P$ where $\boldsymbol{M}_P$ is the set of all mappings in a particular population, and $\boldsymbol{M}_P \subset \boldsymbol{M}_U$ where $\boldsymbol{M}_U$ is the set of all possible mappings within a given DAG, also described as the universal set of mappings.

### Fitness and Makespan

In order to evaluate the mapping's performance a metric is required. In this case the *makespan* is the metric of choice. The makespan is the total time it takes for a single run of the TensorFlow graph, i.e. a single session run. The fitness is inversely proportional to the makespan; The higher the fitness the shorter the makespan. Our target is to find a mapping that results in a smaller makespan than the one provided by TensorFlow's default mapping. Note that the makespan is dependent on several factors; operation-device mapping,communication costs, and scheduling of operations.

Given the restrictions on the availability of communication cost and the scheduling, the makespan value can only be found by a session run.

**Meta-Heuristic Approach and Prediction Model**

A session run is costly when considering the search space of finding a solution. An alternative would be to use a predictive model to obtain the makespan (fitness), or rather, the relative rank of a mapping in relation to other mappings. Such a model would then be used in a meta-heuristic approach to navigate the search space and find the best possible mapping. The choice of mappings used to train the fitness predictive model and used for the initial search, in the meta-heuristic approach, is of importance [43]. In addition, the *features* that are used for the training and the ML algorithm will determine the success of the predictive model. We have found that the simplest feature selection (i.e the tasks) and the values (i.e the devices) is sufficient. Feature extraction was used but did not result in a better prediction model. We thus reverted to simple features. In terms of the initial population of mappings (also used in the training) a restriction was made which will be elaborated in A.1.4.

### A.1.3 HTF-MPR

**System Model**

Our framework targets TensorFlow dataflow models, which are numerical computations that use dataflow graphs [10][12]. The Graph $T = (V, E)$ consists of vertices $V = \{\tau_i, \tau_{i+1}, ...\}$ where $\tau_x$ is a TensorFlow operation and $E = \{(\tau_i, \tau_j), (\tau_m, \tau_n), ..\}$ where $(\tau_i, \tau_j)$ is a directed edge from operation $\tau_i$ to $\tau_j$ . The directed edges in a TensorFlow graph carry tensors.

The list of devices is $D = \{d_x, d_y, ..\}$, where $d_x$ is a device (CPU-0, GPU-0, GPU-1 etc). Each mapping $M_i$ is a unique mapping of operations-to-devices ($M_i = \{(\tau_0, d_x), (\tau_1, d_x), (\tau_2, d_y)..\}$). Every $M_i$ has a runtime (makespan) of $t_i$ and a fitness $f_i = 1/t_i$.

### HTF-MPR Overview

The objective of the HTF-MPR is to create a modified Python file that contains the *tf.with(..)* directive. This directive would allow the user, via API, to allocate a device to a particular operation or set of operations. In the HTF-MPR case, the Python file would have the sub-optimal device allocated to each operation. An example of an output, a modified file, of HTF-MPR:

Code A.3: Addition of tf.device

```
import tensorflow as tf
with tf.device('/cpu:0'):
       x = tf.placeholder(tf.float32, [None, 784])
with tf.device('/gpu:0'):
       W = tf.Variable(tf.zeros([784, 10]))
with tf.device('/gpu:1'):
       b = tf.Variable(tf.zeros([10]))
with tf.device('/cpu:0'):
       y_1 = tf.matmul(x, W)
with tf.device('/cpu:0'):
       y_2 = tf.add(y_1,b)
with tf.device('/gpu:2'):
       y = tf.nn.softmax(y_2)
...
```

Code A.3 is reflected in Fig A.2. An overview of HTF-MPR is shown in Fig A.3. First, the TensorFlow operations from a *Model.py* file are identified. Then, to account for all the hidden operations that are created by TensorFlow when the Task-graph is constructed, we create and run a *Session*. The directed graph is then pruned to remove hidden operations that cannot be assigned via API. This dependency graph structure will determine the initial mappings that will be generated according to certain criteria which will be discussed in section A.1.4. From the initial population of mappings, a predictive model of the mappings-to-fitness is created using the modified TensorFlow file. This predictive model is then used in the GA to search the for a better mappings. Finally once a mapping has been found a mapped model file is created where the file will be run for the remaining number of iterations while starting with the updated ANN parameters that were saved during the initial mappings sessions run stage.

### A.1.4 Extract Operations

First step in HTF-MPR framework is to identify the essential TensorFlow operations. Each operation is assigned a name that coincides with its variable name.

Once those operations have been identified, a single *Session* is run in order to construct the graph. Note that TensorFlow creates other operations not specified in the Python model file.

#### Extract Task-Graph

Once operations are extracted, and one of the operations (the *final* operation in the DAG) had been run on the *Session*, all the hidden operations are pruned because they cannot be mapped. Hidden operations are mapped automatically by TensorFlow in accordance to whatever the *Master* operation is mapped to.

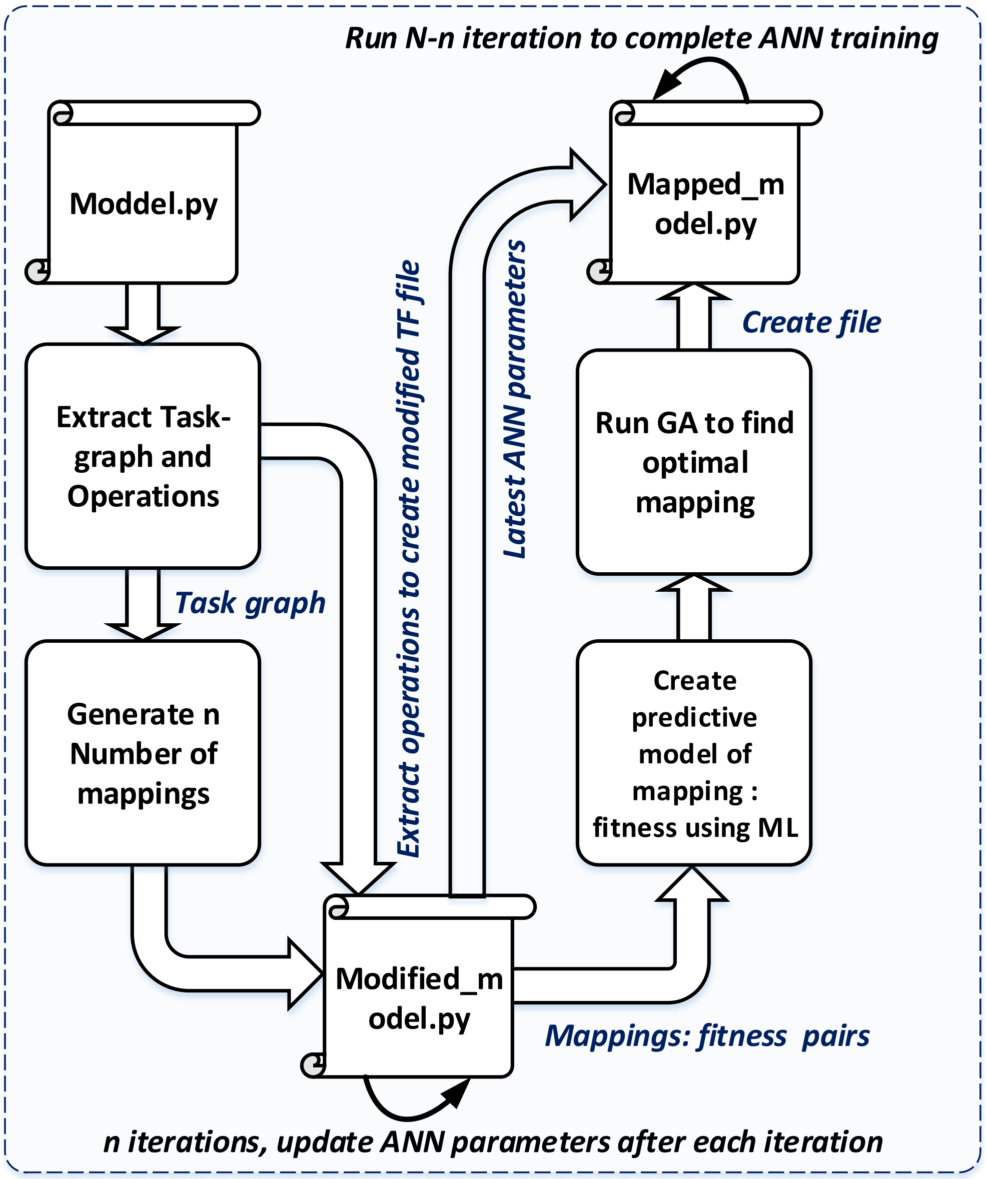

Run N-n iteration to complete ANN training
Moddel.py
Mapped_model.py
Create file
Extract Task-graph and Operations
Extract operations to create modified TF file
Latest ANN parameters
Run GA to find optimal mapping
Task graph
Generate n Number of mappings
Create predictive model of mapping : fitness using ML
Modified_model.py
Mappings: fitness pairs
n iterations, update ANN parameters after each iteration

Figure A.3: HTF-MPR workflow.

### Initial Mappings Generation

The initial mappings will serve two purposes; Training the fitness predictive model and initial population in the GA stage.

The types of generated mappings are;

- **Homogeneous mapping**: single device for all operations. The number of mappings will be $N_D$.
- **Longest Path mapping**: single device on the longest path. #mappings=$N_D!$.
- **Random Homogeneous mapping**: single device to a non-longest single path.
- **Color-mapping**:whenever possible, no two connected operations should be mapped to the same device. #mappings=$N_D!$.

An illustrative example of the mappings types is shown in Fig A.4. The initial mapping types provide variety, useful in training the fitness predictive model, and are good initial starting points for the GA. One of the mappings is the default mapping of heterogeneous (GPU support) TensorFlow (all GPU-0). The other mapping is the default mapping of for non-GPU supported TensorFlow (all CPU-0).

## Prediction Model for Fitness

We use a fitness predictive model to obtain the proper ranking as compared to other mappings, rather than obtaining a highly accurate makespan (and therefore fitness). To evaluate the accuracy of the model (which is done off-line, and is used as a justification for our choice of GBR) we use the Kendall tau rank distance as a metric and the k-fold cross validation method [77] .

**Kendall tau rank distance:** The Kendall tau rank distance is calculated by obtaining the actual fitness $f_i$ and the predicted fitness $\hat{f}_i$ of $M_i$ for all initial $\boldsymbol{M_P}$. if the size of $\boldsymbol{M_P}$ is $n$,

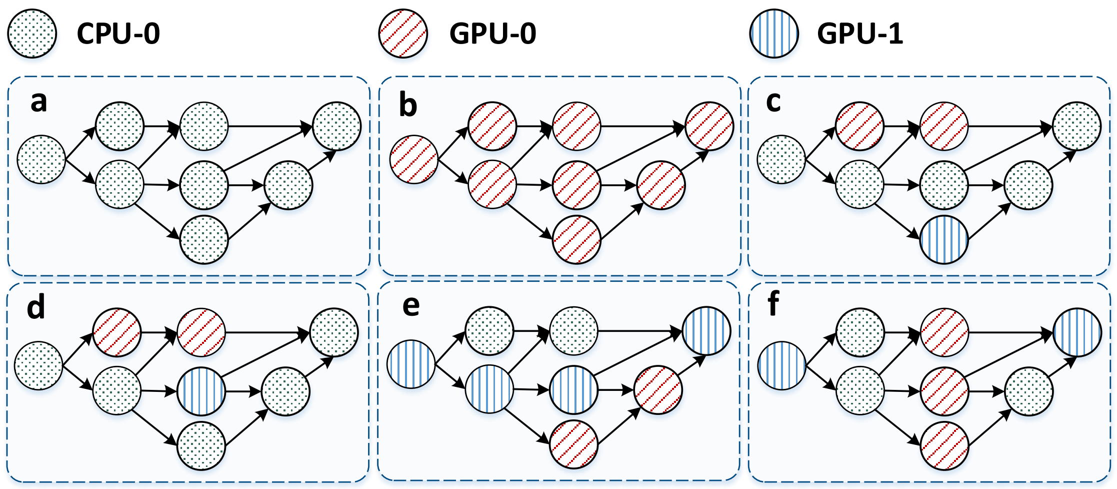


Figure A.4: Examples of some initial mappings; **a** and **b** are homogeneous (single device), **c**,and **d** are longest paths, **e** is non-longest path, and **f** is color-mapped.

then there are $n(n-1)/2$ ranking comparisons. The Kendall number between $f_i$ and $f_j$ is:

$$k(f_i, \hat{f}_i, f_j, \hat{f}_j) = \begin{cases} 1, & \text{if } f_i < f_j \text{ and} \hat{f}_i > \hat{f}_j. \\ 1, & \text{if } f_i > f_j \text{ and} \hat{f}_i < \hat{f}_j. \\ 0, & \text{otherwise.} \end{cases} \tag{A.1}$$

where $i \neq j$. The normalized Kendall tau ranking distance;

$$K_{norm}(F_P, \hat{F}_P) = \sum_i \sum_j \frac{2 \cdot k(f_i, \hat{f}_i, f_j, \hat{f}_j)}{n(n-1)} \tag{A.2}$$

Where $F_P$ and $\hat{F}_P$ are the actual and predicted fitnesses of the mappings in the population, respectively.

**K-fold Cross Validation:** The mappings $\boldsymbol{M}_P$ and actual fitnesses $F_P$ are split into a number of partitions (i.e. K-partitions). The samples are randomly partitioned, where *cross-validation* is done $K$ times. Each partition is set as the validation set and the others are used in the training (see Fig A.5).

Given the training set $\{((M_1, f_1), (M_2, f_2), ..., (M_n, f_n)\}$ where $M_i$ is the mapping and $f_i$ is the actual fitness, a predictive model $\mathbf{F}(M)$ is to be found which minimizes the loss function $\mathbf{L}(f, \mathbf{F}(M))$. Note $\mathbf{F}(M_1) = \hat{f}_1$. The loss function in use is the least square, i.e. $\mathbf{L}(f_i, \mathbf{F}(M_i)) = (f_i - \mathbf{F}(M_i))^2$ . After investigating several machine learning algorithms and using different feature extraction methods, we settled on the Gradient Boosting Regression (GBR) [48] algorithm. GBR consists of weak learners, in the form of decision trees, that are added together (ensemble) to make a stronger prediction model. This is done by iterative means. At each iteration, a weak learner is introduced that compensates for the shortcomings of the previous iteration's weak learner. The overall prediction model is updated by the gradient descent method. The residual, also known as the negative gradient $g(M_i)$ is calculated as:

$$-g(M_i) = \frac{\delta L(f_i, F(M_i))}{\delta F(M_i)} \tag{A.3}$$

At each iteration, $-g(M_i)$ is calculated and a regression tree $h_j$ is fit to the $-g(M_i)$ updating the overall predictive model:

$$F(M) \leftarrow F(M) + h \tag{A.4}$$

**Search via Genetic Algorithm**

When searching, the following factors are taken into consideration:The search space, the method search, the crossover operation, and the fitness function.

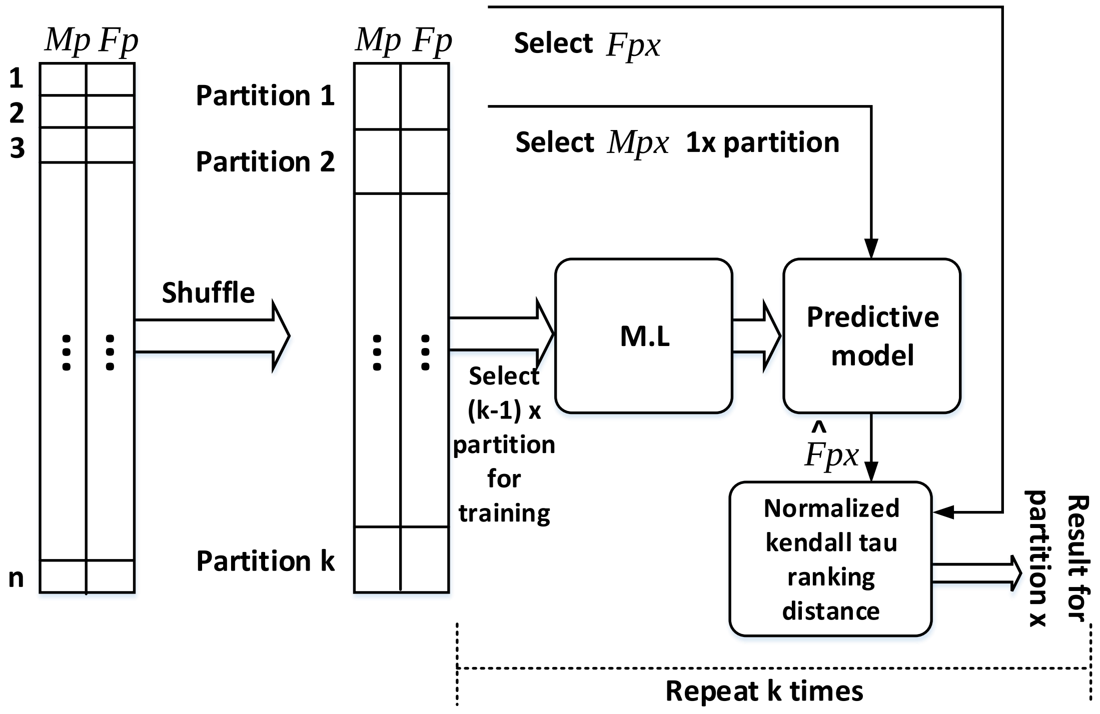


Figure A.5: Cross-Validation using k-fold.

Selection of mappings is proportional to the fitness; the higher the fitness the higher the probability of selecting the mapping for breeding.

Once two mappings are selected for breeding, crossover takes place. We implemented two types of breeding. The first looks at each individual $\tau$ mapped and stochastically decides which parents $\tau$ is chosen(see Fig A.6). The second uses crossover points where the number (between 1 and 3) and position of the crossover points are determined randomly within each mappings pair. The number of crossover points determines the number of new mappings generated from the parent pair (i.e. $(N_C + 1)^2 - 2$ where $N_C$ is the number of crossover points). See Fig A.7. both breeding methods are used so that one provides randomness during exploration (Fig A.6) while the other provides improved performance (Fig A.7).

The top $x$ mappings (corresponding to the top $\hat{f}$) are then run on TensorFlow to get the actual fitness $f$. This is done to compensate for the error of the fitness predictive model.

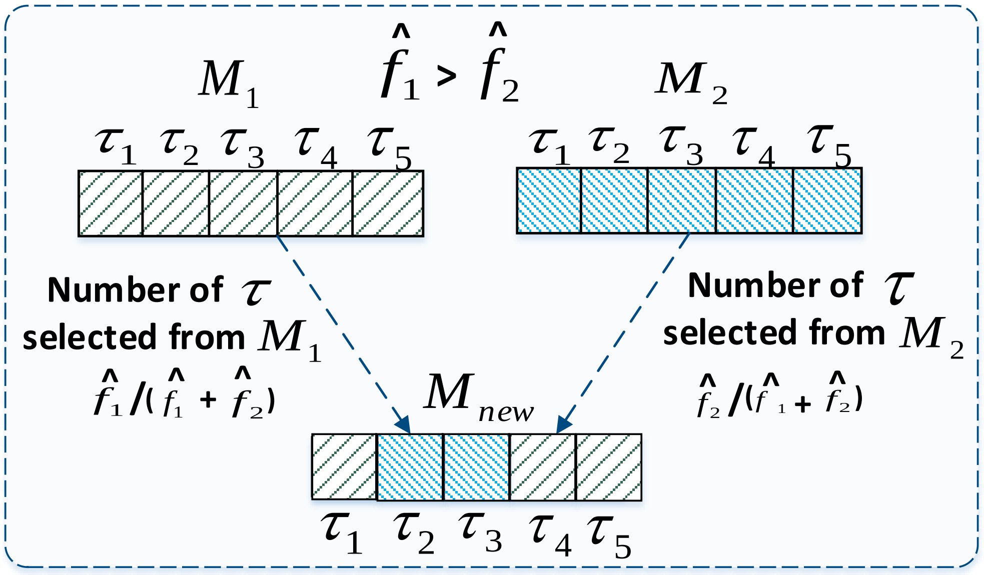

$M_1$
$\hat{f}_1 > \hat{f}_2$
$M_2$
$\tau_1$ $\tau_2$ $\tau_3$ $\tau_4$ $\tau_5$
$\tau_1$ $\tau_2$ $\tau_3$ $\tau_4$ $\tau_5$
Number of $\tau$ selected from $M_1$
$\hat{f}_1/(\hat{f}_1 + \hat{f}_2)$
Number of $\tau$ selected from $M_2$
$\hat{f}_2/(\hat{f}_1 + \hat{f}_2)$
$M_{new}$
$\tau_1$ $\tau_2$ $\tau_3$ $\tau_4$ $\tau_5$

Figure A.6: Breeding using a stochastic method. Newly generated mapping takes more from the fitter mapping parent

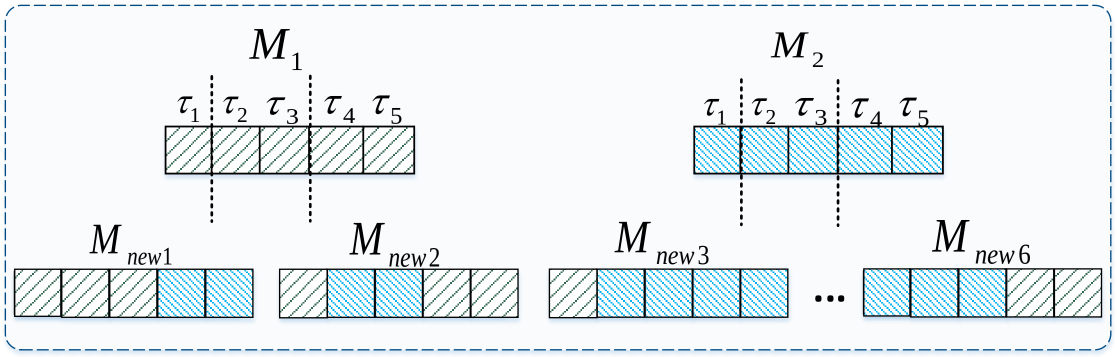

$M_1$
$M_2$
$\tau_1$ $\tau_2$ $\tau_3$ $\tau_4$ $\tau_5$
$\tau_1$ $\tau_2$ $\tau_3$ $\tau_4$ $\tau_5$
$M_{new1}$
$M_{new2}$
$M_{new3}$
...
$M_{new6}$

Figure A.7: Breeding using Crossover points. In this case 2 crossover points resulting in 6 new mappings

Once the top mapping corresponding to the highest $f$ is found, the ANN training continues with said optimal mapping.

### A.1.5 Experimental Results

#### Experimental setup

To test our framework, we used a system that consists of a multi-core CPU (Intel(R) Core(TM) i7-7700 CPU 3.60Ghz), and 2x GPUs (Nvidia GeForce GTX 1050 Ti). The TensorFlow version used is 1.1 with GPU capability (using Nvidia CUDA 8.0 and cuDNN v5), and Python version is 2.7.12. For constructing the predictive model of the makespan, we used the Python-based library scikit-learn version 0.19.0. The following benchmarks were tested: ALEXNET [79] and VGG-16 [133], are convolutional neural network used to classify images from ImageNet [122]. MNIST softmax classifier [20], a very simple image classifier used for characters.

The benchmarks are run on TensorFlow without explicit mapping where the total execution time of the benchmark is observed. The same benchmarks are then run through HTF-MPR.

Bellow is a summary of the benchmarks. Note that the learning process is an iterative process. Thus, the DAG is run several times.

| Benchmark | Mapped Operations | Total Operations | Iterations |
|---|---|---|---|
| ALEXNET | 55 | 295 | 12800 |
| MNIST softmax | 10 | 99 | 60000 |
| VGG-16 | 69 | 376 | 12800 |

The majority of operations are not dealt with directly in HTF-MPR; this is because these operations are generated by TensorFlow, thus, the user may not explicitly assign a mapping via a *tf.device*. With these hidden operations, TensorFlow handles the mapping via its default mechanism, which is to assign these operation to the device of their master operation.

**Predictive Model Analysis Results**

To validate the effectiveness of our predictive model, we used k-fold cross validation (see Fig A.5) across a number of ML algorithms. The number of mappings used is different for each benchmark while the value of $k = 5$ is used for all benchmarks:

| Benchmark | $M_p$ Size | M Size | Training Size | Testing Size |
|---|---|---|---|---|
| ALEXNET | 105 | 55 | 84 | 21 |
| MNIST softmax | 135 | 10 | 108 | 27 |
| VGG-16 | 105 | 69 | 84 | 21 |

The average results are shown in Fig A.8. GBR outperforms all other ML algorithms across the board. In MNIST benchmark, GBR error is considerably higher than in the other benchmarks. So, for the GA stage, we took the top 50 $\hat{f}$ mappings to be run while in the other two benchmarks we only took the top 10 $\hat{f}$ mappings.

**Results and Discussion**

Given the small overhead of running HTF-MPR, we were still able to accomplish an average speedup of **1.18** with ALEXNET, **3.33** with MNIST softmax classifier, and **1.13** with VGG-16. The overhead of identifying the operations, pruning the DAG, generating the initial mappings, performing ML to obtain the fitness predictive model, searching using GA, and construction of the TF graphs for the various mappings (to obtain the actual fitness) is less than **3%** for ALEXNET, **10%** for MNIST softmax, **2%** for VGG-16. The increased overhead for MNIST softmax is due to the increase in TF graph constructions (135+50+1) in addition to the GAs stage of generating mappings (5400). The choice for increasing is due to the performance of the fitness predictive model with MNIST softmax. The speed up

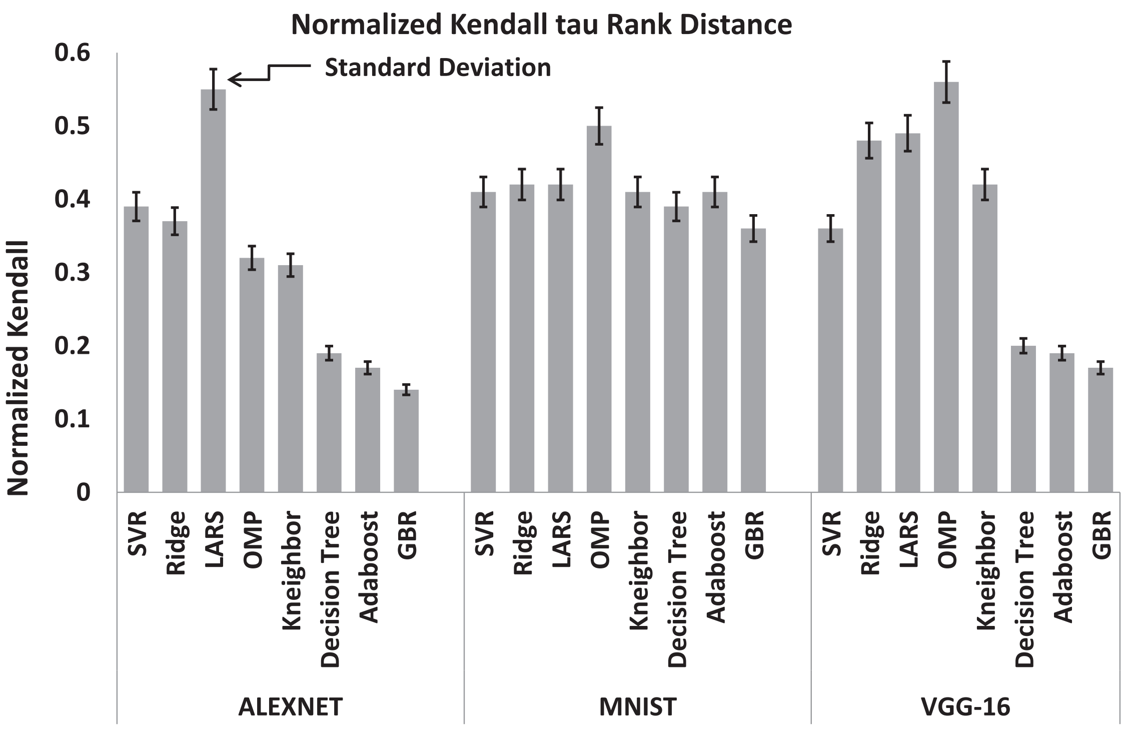


Figure A.8: Predictive Model performance using k-fold (k=5) and different ML algorithms. The chart shows the average from 5 runs and includes the standard deviation of the 5 runs. **SVR**: Support Vector Regression, **Ridge**: Ridge Regression,**LARS**: Least Angle Regression,**OMP**:Orthogonal Matching Pursuit,**Kneighbor**:Regression-based on k-nearest neighbors.

was higher due to the fact that the search space is much smaller (size of $\mathbf{M}_U$ for MNIST softmax is $N_D^{N_V} = 3^{10}$). As a side, we used brute force to find the percentage of mappings in MNIST softmax that outperform the TF default mapping (all GPU-0). 13% of all mappings are better than the TF default. HTF-MPR did not favor GPU for every operation as can be seen by the device distribution of the mappings (see Fig A.10). With such mapping, the performance was improved (see Fig A.9).

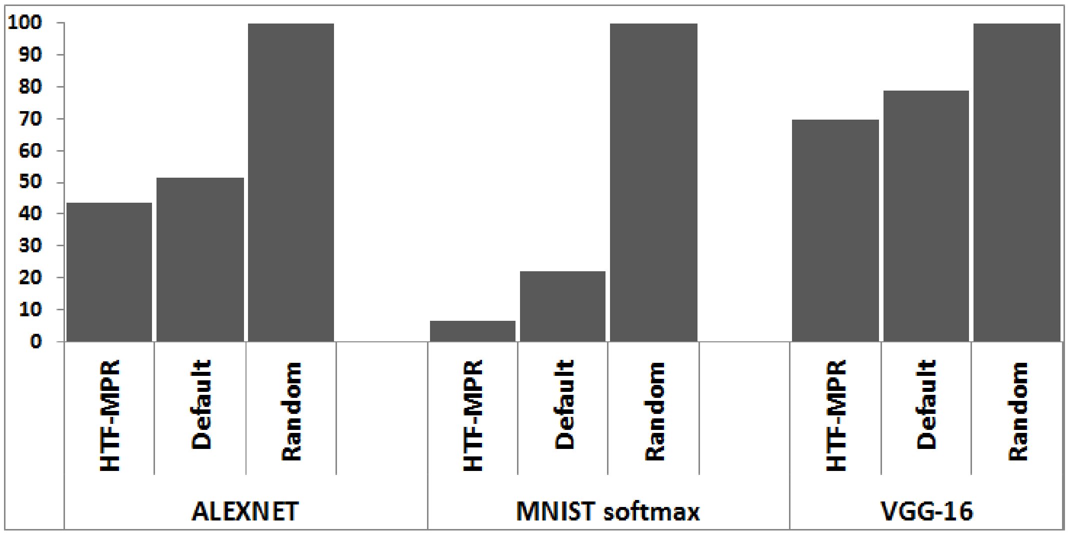


Figure A.9: Relative training time

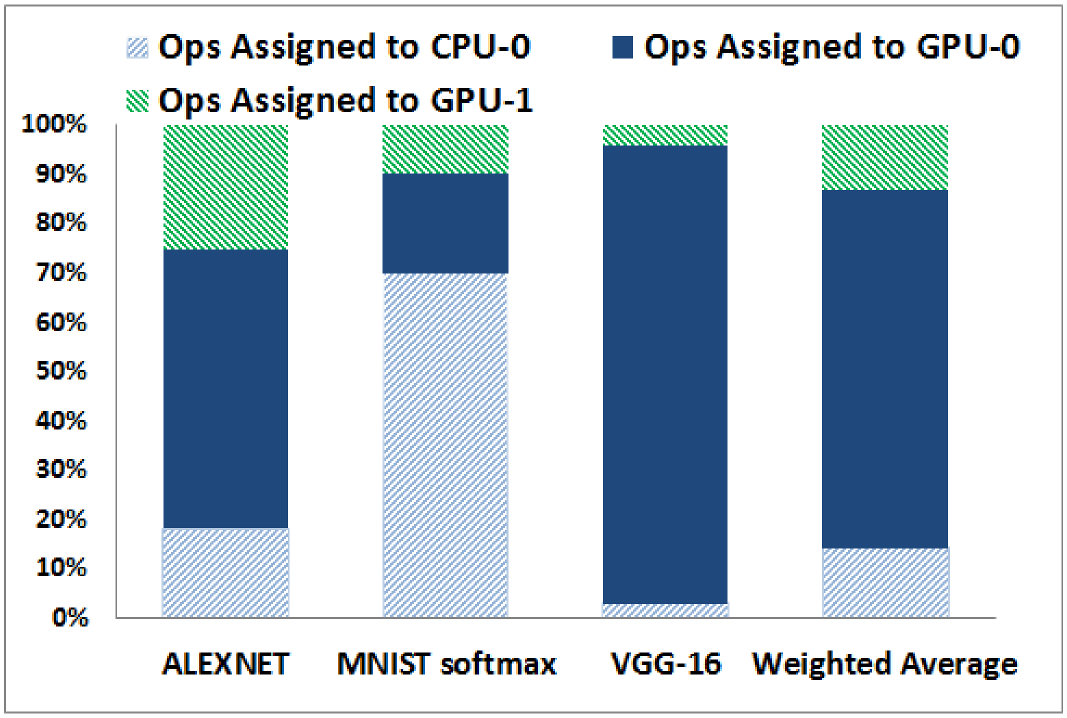


Figure A.10: Device distribution per benchmark. Weighted Average indicates all operations across all benchmarks.

### A.1.6 Summary of Approach One

In this work, we presented our HTF-MPR framework to optimize the mapping of devices to TensorFlow operations. The HTF-MPR uses a genetic algorithm approach to search the mappings space, utilizing a fitness prediction model to evaluate each searched mapping. The fitness prediction model is trained by using an initial population of mappings that are generated in a directed manner. Compared to the default TensorFlow mapper, our results show an overall speedup in the benchmarks, where ALEXNET, MNIST softmax classifier, and VGG-16 show a speedup of **1.18**, **3.33**, and **1.13** respectively.

## A.2 Approach Two: Adaptive Heterogeneous TensorFlow Mapper

Deep Neural Networks (DNNs) are widely used in many Artificial Intelligence (AI) applications. They have demonstrated state-of-the-art accuracy on many AI tasks. For this high accuracy to occur, DNNs require to have the right parameter values. This is achieved by a process known as *training*. The training of large amounts of data via many iterations comes at a high cost in regards to computation time and energy. Optimal resource allocation would therefore reduce the training time. TensorFlow, a computational graph library developed by Google, alleviates the development of Neural Network models as well as providing the means to train these networks. In this article, we propose *Adaptive HTF-MPR* to carry out the resource allocation, or mapping, on TensorFlow. Adaptive HTF-MPR searches for the best mapping in a hybrid approach. We applied the proposed methodology on two well known Image Classifiers; VGG-16 and Alexnet. We also performed a full analysis of the solution-space of MNIST Softmax. Our results demonstrate that Adaptive HTF-MPR outperforms the default homogeneous TensorFlow mapping. In addition to the speed up, Adaptive HTF-MPR can react to changes in the state of the system and adjust to an improved mapping.

### A.2.1 Introduction

*Machine Learning* (ML), and more recently *Deep Learning* (DL), has been utilized as a powerful tool in many fields including computer vision, finance, recommender systems, search engines, and games. ML is a rather dominant branch of Artificial Intelligence (AI) based on the idea that a system uses algorithms to *learn* from data, identify patterns, and make decisions rather than being explicitly programmed via a *Rule-based* approach. This paradigm shift in AI has required systems to be engaged in learning.

Deep Neural Networks (DNNs) [125], are the latest iteration in ML tool-sets. In their simplest form, they were inspired by the human brain and thus were developed to mimic the human brain's synapses and neurons; the neurons in this case are the functions and the synapses are the connections that carry the information from one neuron to the next. The strength of the connections, also known as the synaptic *weights*, dictates how the neural network would perform and function. The values of these weights in turn need to be learned via training. This happens with *experience*, which would also be described as *data*. The amount of time it takes to train a neural network model is very much correlated to the amount of data as well as the number of weights that need to be adjusted.

Previous efforts were made to accelerate the execution times of both training and *inference* [144, 63, 110, 131]. Additionally, works have been done in speeding up the modeling, design, and prototyping of *complex* neural networks. State-of-the-art neural networks have million of weights that need to be adjusted through training. DNNs are structured in *layers*, where the layers have differing types. Convolutional neural networks (CNNs) for example, known for their use on images data, have convolutional layers that contain weights bundled up to function as filters on images. Due to structure and number of parameters, training would take a considerable amount of time.

In this work, we target reducing the time it takes to train a neural network via resource management approach, i.e. finding a better mapping for devices to operations. The training time is highly dependent on what device is used on what operation, in addition to the inter-communication between said devices. Our framework provides a strategy to find a better mapping that would outperform the current TensorFlow mapping. Note that this optimization approach will also work on deployed neural networks, i.e. a trained neural network ready for inference. Our results will focus on some well known CNN benchmarks, but the method may be applied to any type of neural network.

The work is organized as follows: Section A.2.2 covers the background, where the concept of *deep learning* is explained followed by an introduction to *TensorFlow's computational graph.* We then describe *mapping* followed by a description of the target *optimization* problem, i.e. improving *performance* via heterogeneous mapping. In Section A.2.3, we describe in detail the original *HTF-MPR* Framework. In Section A.2.4, we define *adaptivity* in the context of the framework. In Section A.2.4, we introduce the *Adaptive HTF-MPR* Framework, where we explain the added features to the original Framework as well as the changes to further improve performance. Our experimental setup is described in Section A.2.5. In Section A.2.6,we present the results of the evaluation of our approach compared to the default TensorFlow device mapper and the original HTF-MPR. Finally, the work is concluded in Section A.2.7.

## A.2.2 Background

### Deep Learning

Neural networks contain parameters that need to be *tuned* which are referred to as *weights* of the model. In contrast, other parameters such as the structure of the neural network model, the number of layers, the *activation functions*, the size of the layers, and the connections between layers are *fixed* and are not tuned during the learning process. These are referred as *hyperparameters* (see Figure A.11a). The values of the tunable parameters are changed during training according to the input data. This training takes place insofar to be later used for *inference.* Note that an untrained neural network may be used for inference but is not optimized for its intended purpose and will perform poorly.

Prior to *deploying* any machine learning model, to be used for inference, the model should be trained. In the case of *supervised learning*, the input $x$ and the intended output $y$ are provided which subsequently produce a trained model $f_{NN}$. It is intended that for any given input data $x_i$, $f_{NN}(x_i) \longrightarrow \hat{y}_i$ where the *predicted* $\hat{y}_i \approx y_i$. The untrained model would most likely produce a $\hat{y}$ that is not close to the intended $y$. To evaluate this discrepancy a loss

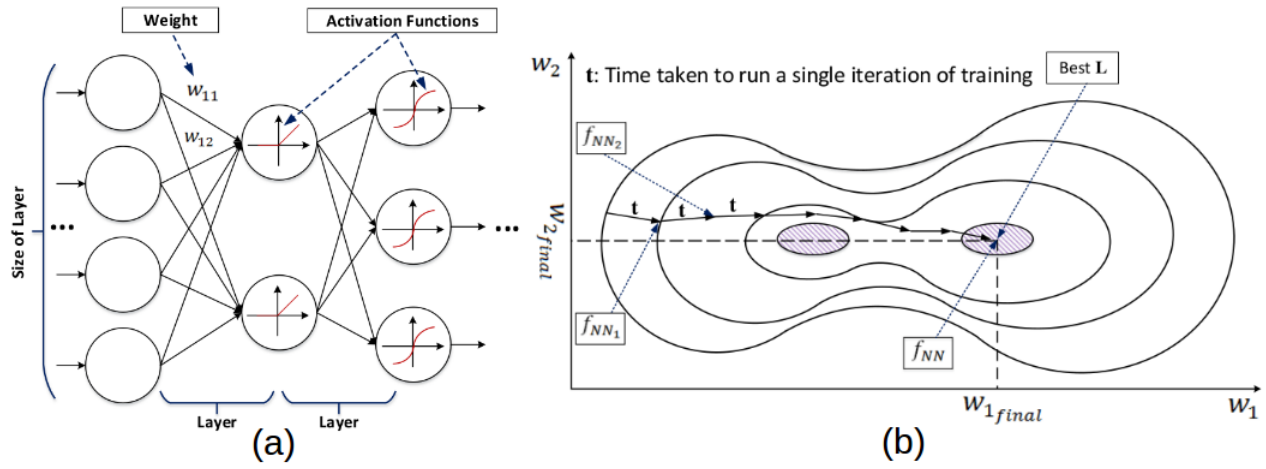


Figure A.11: **a)** A simple two layer neural network. The value of each $w_{ij}$ is tunable and may change during training. While the hyperparameters stay intact and are static. **b)** The Gradient Path of the model. $t$ does not change in each iteration (as long as device-mapping does not change) while $w_1$ and $w_2$ change.

function $L$ is used. One such metric in deep neural networks is the *L2 norm* loss function;

$$L(y, \hat{y}) = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{A.5}$$

In the case of *classification* problems, where the output is a category rather than a specific number (as is the case of *regression* problems), the *cross-entropy* function is therefore used;

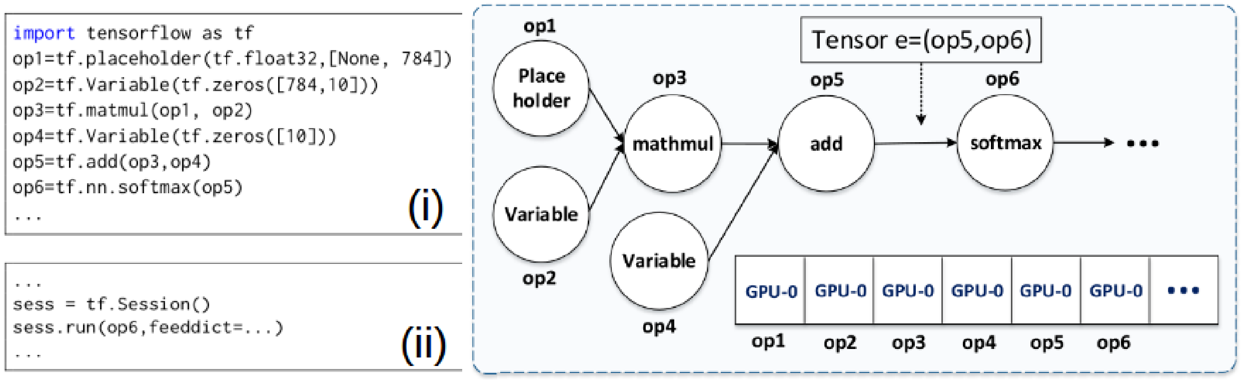
$$L(y, \hat{y}) = -\frac{1}{n} \sum_i ln\left(\frac{e^{y_i}}{\sum_j e^{\hat{y}_j}}\right) \tag{A.6}$$

The objective of training is to find a model $f_{NN}$ where the finally tuned weights would ideally result in a $f_{NN}$ that is $L \approx 0$, i.e. the objective is to minimize the loss function $L$. A method that is employed to train deep neural networks is known as *stochastic gradient descent* (SGD). SGD provides the direction in which to change $\hat{y}$ to be closer to $y$. $\hat{y}$ is not changed directly, but rather, the weights are changed which affects $\hat{y}$. This change happens using a technique known as *backpropagation* [121]. Simply put, backpropagation adjusts the weights resulting in a $\hat{y}$ closer to $y$. Therefore, in each training *iteration*, using a subset or *mini-batch* of input data $x$, $\hat{y}$ is the resultant and $L$ is the assessment. Backpropagation updates the weights, and the process is repeated. This whole process in deep neural networks is referred to as *deep learning* [85, 126]. The number of times, or training iterations, is related to many factors including the dataset size, the number of *features* per data-point, the desired accuracy, etc.

Figure A.11b illustrates a two-weight model and the loss $L$ as an example of the training path and the updating of the model's parameters until a satisfactory $L$ is reached. The training process can take a long time given the number of parameters, size of dataset used for training and the number of *epochs*, which is the number of times we run the same dataset. Each training iteration takes $t$ time. In this work, our target is to reduce $t$, while keeping the path intact, i.e. the target of this work is neither changing the path taken, nor increasing the accuracy nor modifying the structure of the model. We change the device-mapping to reduce $t$, i.e. improving the hardware utilization.

### Computational Graphs and TensorFlow

TensorFlow [13, 11], is a library that is heavily used in industry and academia for building and training machine learning models. Keras [36], a high-level *application programming interface* (API) runs TensorFlow as it's defacto library. TensorFlow uses the computational graph approach: Each *node* in the TensorFlow graph $G$, is an operation $op_i$, and the *vertices* are *tensors*, i.e. multi-dimensional matrices. Figure A.12 illustrates a *directed* graph in TensorFlow. The *dataflow* graph in Figure A.12 shows the dependencies; meaning certain operations will not execute unless all data dependencies are executed. Let $G = (Op, E)$, where $Op = \{op_1, op_2, ..op_{N_{op}}\}$ are the operations, and $E = \{(op_a, op_b), (op_c, op_d)...\}$ are the *directed edges* where $e_i = (op_a, op_b)$ is a tensor from $op_a \longrightarrow op_b$ and $a \neq b$. Note that the TensorFlow graph $G$ is assumed to be an *a-cyclic* dataflow graph.



```
import tensorflow as tf
op1=tf.placeholder(tf.float32,[None, 784])
op2=tf.Variable(tf.zeros([784,10]))
op3=tf.matmul(op1, op2)
op4=tf.Variable(tf.zeros([10]))
op5=tf.add(op3,op4)
op6=tf.nn.softmax(op5)
...
```

(i)

```
...
sess = tf.Session()
sess.run(op6,feeddict=...)
...
```

(ii)

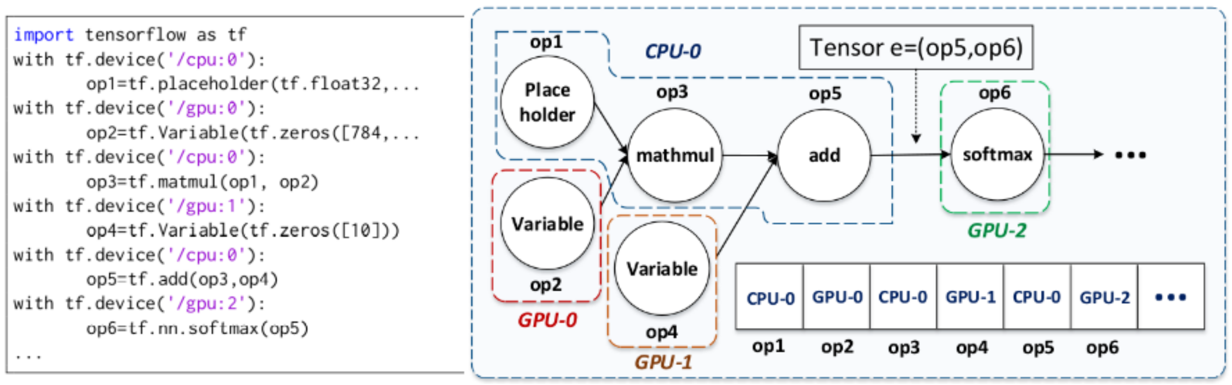
Figure A.12: Homogeneous Mapping: All the operations, by default, are mapped to GPU-0. i) The model in TensorFlow. ii) The code to run the model in a Session.

The operation execution order is handled by the TensorFlow scheduler; The *Distributed Master* would evaluate the TensorFlow Graph $G$'s nodes, i.e. the *Worker Services* schedules the operations according to the Distributed Master's request.

In TensorFlow, the programming paradigm requires a construction of a model (Graph) where the hyperparameters are set before the model is run. A model run could either be for training or inference. Figure A.12i shows the code of the model, while Figure A.12ii shows the script required to run the model: A *Session* is created and the last operation in $G$, in this case $op_6$, is passed on as a parameter to the Session.

### Mapping

TensorFlow, by default, maps all the operations to a single device. If GPU-enabled TensorFlow is installed and the hardware is supported, then all the operations in a TensorFlow graph are mapped to a single GPU. Otherwise, all the operations are mapped to a CPU. A workaround to defining your own mapping is to use the *tf.device* directive. An illustration of a mapped TensorFlow graph and its accompanying code is shown in Figure A.13.



```
import tensorflow as tf
with tf.device('/cpu:0'):
        op1=tf.placeholder(tf.float32,...
with tf.device('/gpu:0'):
        op2=tf.Variable(tf.zeros([784,...
with tf.device('/cpu:0'):
        op3=tf.matmul(op1, op2)
with tf.device('/gpu:1'):
        op4=tf.Variable(tf.zeros([10]))
with tf.device('/cpu:0'):
        op5=tf.add(op3,op4)
with tf.device('/gpu:2'):
        op6=tf.nn.softmax(op5)
...
```

Figure A.13: Heterogeneous Mapping. The code shows the addition of tf.device to enable heterogeneous mapping.

Given a set of devices $D = \{d_1, d_2, ..d_{N_D}\}$, and a set of operations $Op = \{op_1, op_2, ...op_{N_{op}}\}$, a particular mapping is defined as $m_i = \{(op_1, d_x), (op_2, d_y)...(op_{N_op}, d_z)\}$. Note that the size of $m_i$,$|m_i| = |Op| = N_{op}$. The mapping of devices to operations is one-to-many, meaning that several operations could be mapped to a single device in any particular mapping, while the opposite is not true (see Figure A.13).

In our notation $m_{TF} = m_{gpu-0} = m_2$ is the GPU-0 mapping which is the default TensorFlow mapping (as shown in Figure A.12), while $m_{cpu} = m_1$ is the *homogeneous* CPU mapping and $m_{gpu-1} = m_3$ is the homogeneous GPU-1 mapping. The rest of the mappings, $m_i|i > 3$, are different *heterogeneous* mappings (an example of a heterogeneous mapping shown in Figure A.13). No two mappings are the same i.e $m_i \neq m_j|i \neq j$ where $m_i, m_j \in M$.

## Optimization

The objective is to find a mapping $m$ that results in a faster execution time than the default TensorFlow mapping. This would thus speed up the whole training time. The optimization problem is therefore;

$$m^\star =_{m \in M} f_t(m) \tag{A.7}$$

where $f_t(m)$ is the *makespan* (execution time) of a single training iteration of the TensorFlow graph using mapping $m$. $m^*$ is any mapping that outperforms TensorFlow's mapping. Note that possible mappings of $G$ is represented by $M$, which has a size of $|M| = N_D^{N_{op}}$, where $N_D$ and $N_{op}$ is the number of devices and operations, respectively. The search for an optimal mapping in the search space is thus considered an *NP-hard* [115, 75, 142] problem. The mapping problem could be reduced to *NP-Complete* by relaxing the condition, i.e. by finding a mapping that outperforms the default homogeneous TensorFlow mapping rather than finding the global optimal mapping. i.e, $M^* \subset M$ and $m^* \in M^* | f_t(m^*) < f_t(m_{TF})$ . Some of the characteristics of the makespan $f_t(m)$;

- It is a *continuous* function, i.e. the execution time is a real number.
- The input data $m$ is a *tuple* of *categorical* data, i.e. the values of the operations are device labels which are discrete.
- A single evaluation is *expensive*, meaning that an actual run of the graph has to occur to find the makespan value.
- It is a *black-box* function, i.e. its structure is unknown (not convex nor linear etc).
- Given it is a black-box function, therefore it is *non-differentiable*. Neither the first nor the second-order derivative may be utilized.

For this to be worth-while, the whole training time needs to be less than the training time of the default homogeneous TensorFlow mapping;

$$F_t(\pi, oh_\pi) < F_t(\pi_{m_{TF}}, 0) \tag{A.8}$$

Where $F_t(\pi, oh_\pi)$ is the sum of execution times plus overhead given a *policy* of mappings $\pi = m_a, m_b, ....$ Note that generating such policy is an overhead, represented by $oh_\pi$. $F_\tau(\pi_{m_{TF}}, 0)$ is the TensorFlow total training time with a policy of using a single type of mapping which is a homogeneous mapping, $m_{TF}$ and no search overhead.

$$F_t(\pi_{m_{TF}}, 0) = \sum_{i=1}^{I} f_t(m_{TF}) \tag{A.9}$$

Where $I$ is the number of training iterations it takes to reach the final desired model $f_{NN}$. Regardless of policy used, and as long as the number of iterations is $I$, the desired $f_{NN}$ is unchanged, i.e. the *path* as described in section A.2.2, remains the same regardless of the mappings policy (See Figure A.11b).

### A.2.3 HTF-MPR

*HTF-MPR* [14] is a framework that finds a better *device-to-operations* mapping in-order to speedup execution times of TensorFlow computational graphs. Mappings are evaluated by measuring the execution's runtime using a particular mapping, i.e. $f_t(m) \longrightarrow t_m$. Once a reasonable sized sample of mappings and their speeds are collected, a *predictive model* $f'_t(m)$ is produced. The reason for the use of a predictive model, rather than the actual run, is that the amount of time it takes to return the makespan of a certain mapping is almost *750 times faster*. In other words, 750 mappings would be analyzed using the makespan predictive model compared to one mapping using the actual run. Note that there are accuracy issues with the predictive model, as is the case with any model, therefore the best mappings, according to $f'_t(m)$, are run again, i.e. we evaluate their $f_t(m)$. An overview of HTF-MPR is shown in Figure A.14. The following Subsections will go into further detail of the HTF-MPR framework.

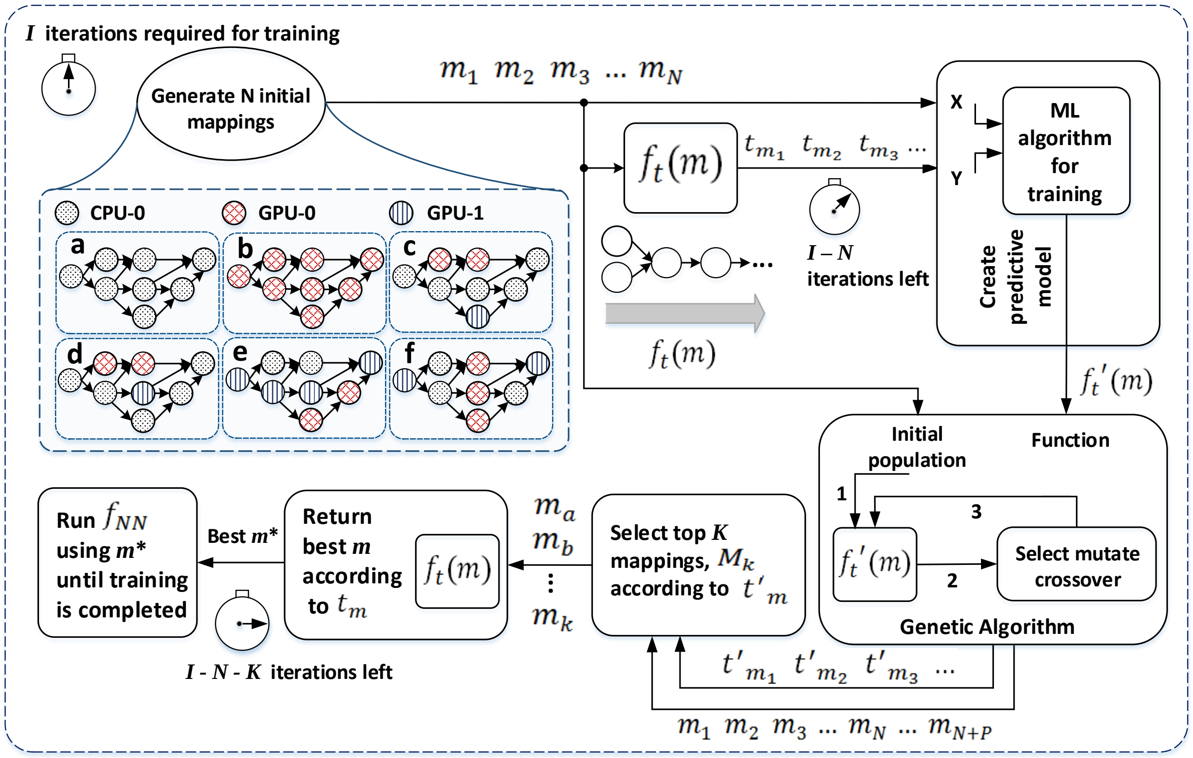

*I* iterations required for training
Generate N initial mappings
$m_1$ $m_2$ $m_3$ ... $m_N$
CPU-0
GPU-0
GPU-1
a
b
c
d
e
f
$f_t(m)$
$t_{m_1}$ $t_{m_2}$ $t_{m_3}$ ...
*I* – *N* iterations left
X
Y
ML algorithm for training
Create predictive model
$f_t'(m)$
Initial population
Function
1
2
3
$f_t'(m)$
Select mutate crossover
Genetic Algorithm
Select top *K* mappings, $M_k$ according to $t'_m$
$t'_{m_1}$ $t'_{m_2}$ $t'_{m_3}$ ...
$m_1$ $m_2$ $m_3$ ... $m_N$ ... $m_{N+P}$
$m_a$
$m_b$
⋮
$m_k$
Return best *m* according to $t_m$
$f_t(m)$
Best *m**
Run $f_{NN}$ using *m** until training is completed
*I* - *N* - *K* iterations left

Figure A.14: HTF-MPR Overview: **1.** $N$ initial mappings are generated (Subsection A.2.3). **2.** These mappings are then run on the TensorFlow graph where their makespans, $f_t(m) \longrightarrow t_m$, are recorded. The number of iterations left to train the model (and therefore get it closer to the final model $f_{NN}$) is $I - N$. **3.** The input data $X$ and output data $Y$ are used to construct the predictive model(Subsection A.2.3). **4.** The predictive model as well as the mappings are provided to the Genetic Algorithm (Subsection A.2.3). **5.** Top mappings are selected according to the predicted makespans. **6.** The top mappings are then run on the TensorFlow graph to obtain actual makespans $f_t(m)$. The number of training iterations is advanced by $K$ (the number of top mappings), thus reducing the required runs to $I-N-K$. **6.** Finally, the top mapping, $m^*$ is found and used for the rest of the training i.e. for $I-N-K$ iterations.

### Initial Mappings

The initial mappings are used for two purposes; to create the predictive model and as an initial population to the genetic algorithm which is shown in Figure A.14. Briefly, the initial mappings are generated with the following characteristics:

- **Homogeneous mapping**: A single device for all operations. Figure A.12 shows an example of homogeneous mapping. The number of mappings is proportional to the number of devices, i.e. $N_D$. (Figure A.14, in the *initial mappings*, shows two examples in **a** and **b**).

- **Longest Path mapping**: A single device mapped to the operations making up the longest path in the graph. While the other operations are mapped to different devices than the one on the longest path. The number of mappings in this case is $N_D$. (Figure A.14, in the *initial mappings*, shows two examples in **c** and **d**).

- **Random Path Homogeneous mapping**: A single device mapped to a non-longest single path. While the rest of the operations are mapped to different devices than the one on the designated path. (Figure A.14, in the *initial mappings*, shows an example in **e**).

- **Color mapping**: No two neighboring operations (operations that share a tensor) should be mapped to the same device. The number of mappings is $N_D$. Note that in terms of makespan these would result in the worst possible performance. (Figure A.14, in the *initial mappings*, shows an example in **f**).

The initial mappings are the *training dataset* of the predictive model. A varied dataset would make the predictive model more robust. In addition, more points in the search space will be evaluated, using the predictive model, due to the varied initial mappings.

Note that a large variety in the initial mappings leads to a more versatile generalized predictive model, but a less accurate model given a search on a concentrated region.

**Makespan Prediction**

The purpose of a makespan predictor $f_t'(m)$ is to speed up the overall training time $F_t$. By having a reliable makespan predictor, it is possible to perform a search on $M$ in a fraction of the time that is required when using the results of $f_t(m)$, i.e. the actual run. The training set for building the predictor is therefore $\{(m_i, t_{m_i})\}_{i=1}^{N}$. After investigating several machine learning algorithms, the *Gradient Boosting Regression* (GBR) [48] algorithm outperformed those that were tested when it came to the *Kendall tau rank distance* [78] metric. GBR consists of weak learners that are assembled together and made into an ensemble of a strong prediction model. This ensambling of weak learners happens after each iteration where the new weak learner improves upon the whole predictive model. Therefore, the weights, as well as the hyperparameters, are adjusted during training:

$$f'_{t,k+1}(m) = f'_{t,k}(m) + h(m) = t_m \tag{A.10}$$

Where $f'_{t,k+1}(m)$ is the makespan predictor at step $k+1$ of it's training, which is made up of the previous predictor $f'_{t,k}(m)$ and an estimator $h(m)$. Therefore, the final makespan predictor $f'_t(m)$ is made up of many weak predictors:

$$f'_t(m) = \sum_{j=1}^{n} h_j(m)\gamma_j + const. \tag{A.11}$$

Where $n$ is the total number of training iterations to construct the predictive model. Training happens in an incremental manner, initially set as:

$$f'_{t,0} =_{\gamma} \sum_{i=1}^{N} L(t_i, \gamma) \tag{A.12}$$

Where $f_t(m) \longrightarrow t_m$, and during the training of the predictive model. $N$ mappings are used as input $X$, and $N$ timings are used as the output $Y$ (see Figure A.14, *ML algorithm for training*). Subsequently, the makespan predictive model is updated by computing the *residual*:

$$r_j(m_i) = -\left[\frac{\delta L(t_{m_i}, f'_t(m_i))}{\delta f'_t(m_i)}\right], \text{for } i = 1, ..N \tag{A.13}$$

Then, the *base learner*, i.e. estimator $h_j(m)$, is constructed using the residual $r_j(m)$ and input $m$. Therefore the training set for $h_j(m)$ is $\{(m_i, r_j(m_i)\}_{i=1}^{N}$. Afterwards the $\gamma_j$ is updated:

$$\gamma_j =_{\gamma} \sum_{i=1}^{N} L(t_i, f'_{t,j-1}(m_i) + \gamma h_j(m_i)) \tag{A.14}$$

The model is then updated as referred to in Equation A.10:

$$f'_{t,j}(m) = f'_{t,j-1}(m) + \gamma_j h_j(m) \tag{A.15}$$

This whole process is repeated $n$ times resulting in a final predictive model $f'_t(m)$ which is used by the Genetic Algorithm.

## Search with Genetic Algorithm

Genetic Algorithms (GAs) [96] are an optimization technique that are *metaheuristic*, meaning they are designed to work on non-deferential and non-linear search spaces [54]. They are known for solving *task-mapping* [59] problems. In HTF-MPR, the GA uses $f'_t(m)$ as the inverse *fitness* of a particular *solution*, i.e. mapping. The fitness of a solution is proportional to how probable it would be chosen as one of the parents to generate a new solution. This new solution is assessed using $f'_t(m)$ and added to the *population*. The search process is shown in the Genetic Algorithm's part of Figure A.14. Initially, the algorithmically generated mappings are provided to the GA, where the $t'_m$ of each mapping is calculated. Then, *selecting* two parents with a probability proportional to the fitness, i.e. inverse of $t'_m$, whereby these two parents generate a new mapping via *crossover*. In our approach we have used two methods for crossover (Figure A.15 and Figure A.16). Figure A.15 shows a *stochastic* approach of how a new mapping is generated. The fitness of the parent will dictate the percentage of operations-mapping the new mapping will inherit from that parent. Figure A.16 shows

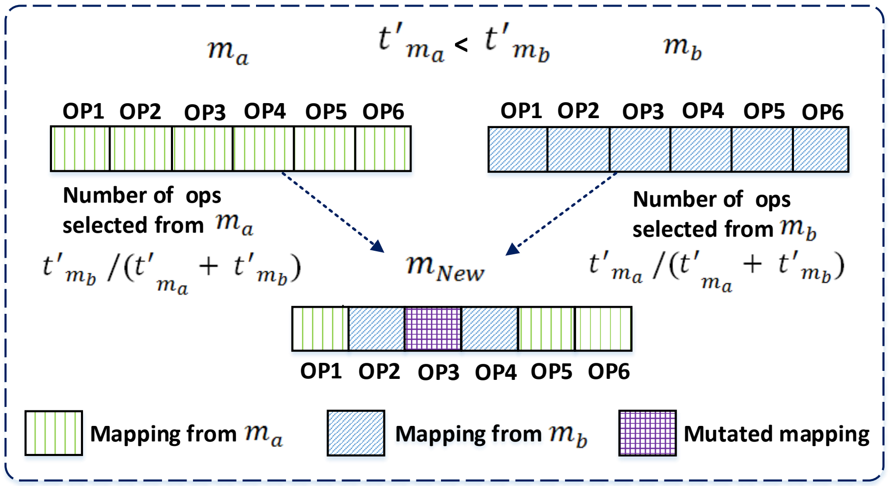


Figure A.15: Crossover using a stochastic method whereby the number of mappings taken from a particular parent is relative to how fit the parent is. In this case $m_a$ is more fit than $m_b$ given the lower predicted makespan, i.e. $t'_{m_a} < t'_{m_b}$. Therefore, more operation mappings are copied from $m_a$ than $m_b$. Some operations' mappings also go through *mutation*, meaning it does not copy from either parent. In this example $op3$ got mutated.

another approach, where the crossover points dictate the number of new generated mappings. For example, if there were 2 crossover points, then *at most* 6 new mappings would be generated from the parents (see Figure A.16). If there were three crossover points, then *at most* 14 new generated mappings would occur, i.e. at most $2^{N_c+1} - 2$ generated mappings, where $N_c$ is the number of crossover points.

### Final Selection

$P$ new mappings, and their predicted makespans $t'_m$, are generated by the GA as shown in Figure A.14. These mappings are then sorted in $t'_m$ ascending order. The Top $K$ mappings are then chosen and run on the TensorFlow graph to get the actual makespans $t_m$. The top mapping $m^*$, according to $t_m$, is then run until the training of the TensorFlow graph is

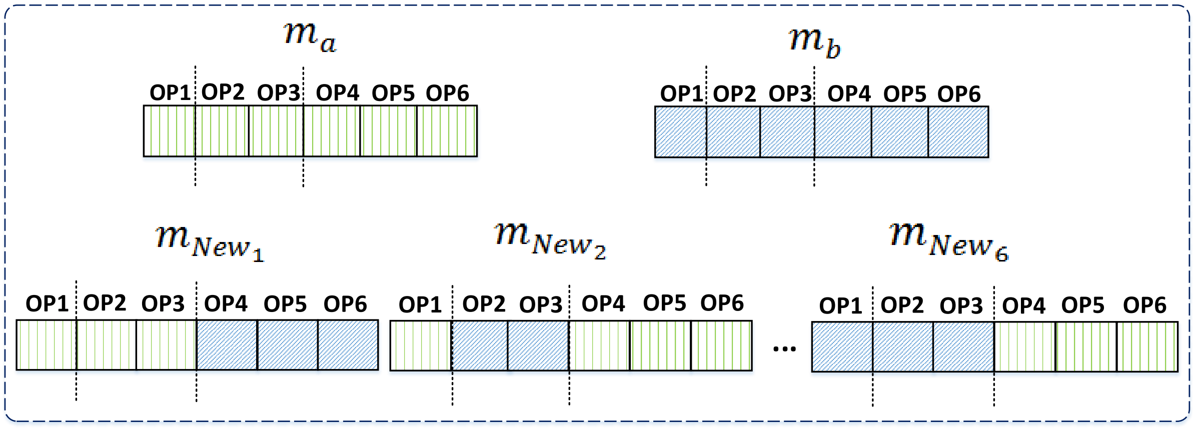


Figure A.16: Crossover using a crossover-points. In this example, 6 new mappings are generated from the parents $m_a$ and $m_b$.

completed, thus reaching the final state of the model $f_{NN}$. Note that the training advances when the model is run. During a run, regardless of the mapping used, we get the added benefit of acquiring $f_t(m)$ while not affecting the training's path. This indicates that the final destination of $f_{NN}$ is the same, the only difference is how fast we get there.

### A.2.4 Adaptive HTF-MPR

#### Adaptivity

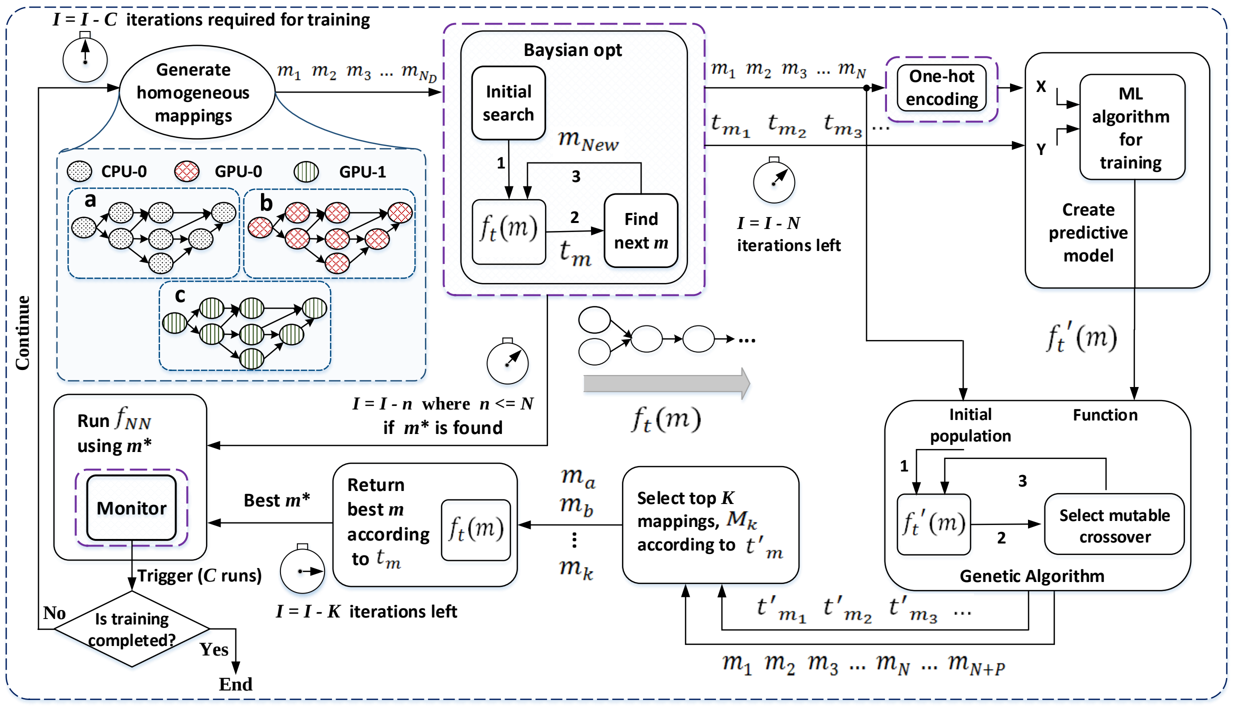
The training time for some state-of-the-art neural networks could take up to hundreds of thousands of iterations [158], each iteration would take some time depending on the employed hardware and the batch-size of the input data. There is no guarantee that the state, or performance, of the system remain consistent throughout the training, i.e. parts of the system's hardware, CPUs or GPUs, could have different loads at different times due to external processes. This would affect the makespan and thus the training time. To combat this, the makespan time has to be monitored. The monitoring module would detect any drastic performance changes from the average performance, whether it be improved performance or degraded performance. If one of the systems' components, i.e. one of the device's load has increased or decreased, it would affect $f_t(m)$ and thus changes what could be considered $m^*$. In this work, we have added, among other things, a monitoring mechanism and a way to deal with and *adapt* to these changes in the system. Adaptive HTF-MPR would take corrective measures to find a new $m^*$, once the monitoring module sets a trigger. Our monitoring module works with both gradual slow changes [145] and abrupt changes [17].

### Overview of Framework

Adaptive HTF-MPR uses the similar methodology as HTF-MPR with some modifications (see Figure A.17). One modification is the introduction of the Bayesian Optimization [114] step using $f_t(m)$ as the function for performance evaluation. The point of the Bayesian optimizer is to find the local, or neighborhood of the best mappings via intelligent search. Thus, the resulting mappings will be used to construct the makespan predictive model $f_t'(m)$. Another modification is the removal of the initial mappings via Algorithmic approach . This is due to using the results of the Bayesian optimizer as input to the ML to create the makespan predictive model as well as the initial population for the GA. The input or initial start of the Bayesian optimizer is the homogeneous mappings.

### Initial Mappings

Homogeneous mappings are used as an initial step instead of the algorithmically generated mappings (as was described in Subsection A.2.3). The reason is that the Bayesian optimizer will generate those initial mappings required for search. The Bayesian optimizer will therefore construct a less *robust* model given the more concentrated dataset provided. On the other hand, the model will preform better due to the fact that only data-points (mappings) in the more optimized locale will be generated by the GA and therefore only the makespans of said



I = I - C iterations required for training
Generate homogeneous mappings
$m_1$ $m_2$ $m_3$ ... $m_{N_D}$
CPU-0
GPU-0
GPU-1
a
b
c
Baysian opt
Initial search
$m_{New}$
1
3
2
$f_t(m)$
$t_m$
Find next $m$
$m_1$ $m_2$ $m_3$ ... $m_N$
$t_{m_1}$ $t_{m_2}$ $t_{m_3}$ ...
One-hot encoding
X
Y
ML algorithm for training
Create predictive model
I = I - N iterations left
$f_t'(m)$
Continue
I = I - n where n <= N if m* is found
$f_t(m)$
Run $f_{NN}$ using m*
Monitor
Best m*
Return best m according to $t_m$
$m_a$
$m_b$
$m_k$
Select top K mappings, $M_k$ according to $t'_m$
Initial population
Function
Select mutable crossover
Genetic Algorithm
Trigger (C runs)
No
Is training completed?
Yes
End
I = I - K iterations left
$t'_{m_1}$ $t'_{m_2}$ $t'_{m_3}$ ...
$m_1$ $m_2$ $m_3$ ... $m_N$ ... $m_{N+P}$

Figure A.17: Adaptive HTF-MPR Overview: **1.** $N$ initial mappings are generated using Bayesian optimization (Subsections A.2.4 and A.2.4). **2.** Mappings are then run on the TensorFlow graph where their makespans, $f_t(m) \longrightarrow t_m$, are recorded. The number of iterations left to train the model (and therefore get it closer to the final model $f_{NN}$) is $I - N$. **3.** Input data $X$ is turned to one-hot encoding (Subsection A.2.4). Makespan predictive model is constructed (Subsection A.2.3). **4.** Genetic Algorithm is run (Subsection A.2.3) until population size is $P$. **5.** Top mappings are selected according to the predicted makespans. **6.** The top $K$ mappings are then run on the TensorFlow graph to obtain the actual makespans $f_t(m)$. The number of training iterations is advanced by $K$, thus reducing the required runs to $I - N - K$. **6.** The top mapping, $m^*$, is found and used for the rest of the training. The Monitor triggers a rerun of the process if required (Subsection A.2.4).

mappings will be predicted by the more concentrated $f_t'(m)$. It is noted that the target of HTF-MPR and Adaptive HTF-MPR is to beat the default homogeneous mapping, thus the homogeneous mappings are a good starting point for the Bayesian optimizer. Figure A.17 shows the generation of homogeneous mapping of CPU-0 ($m_1$), GPU-0 ($m_2$), and GPU-1 ($m_3$), i.e. $N_D = 3$.

## Bayesian Optimization

Bayesian Optimization is based on *Bayesian reasoning* where the reconstruction of the objective function $f_t(m)$ is updated based on new evidence, i.e. due to evaluation of new data points in $f_t(m)$. The more data-points are evaluated the closer the *surrogate function* is to $f_t(m)$. The *Tree Parzen Estimator* (TPE) [24, 23] is one of the methods for constructing the surrogate function. The target of the Bayesian optimizer is to find the data-point (input) that would result in the minimum of the function. This is done by choosing the next input to be evaluated according to the surrogate function and past results. The surrogate function is described by a probabilistic model approach:

$$P(t|m) \sim \mathcal{N}(\mu(m),\, \sigma(m)^2) \tag{A.16}$$

Where $\mathcal{N}(\mu,\, \sigma^2)$ is the *Normal distribution* with $\mu$ as the expected *mean* function and $\sigma^2$ as the expected *variance* function.

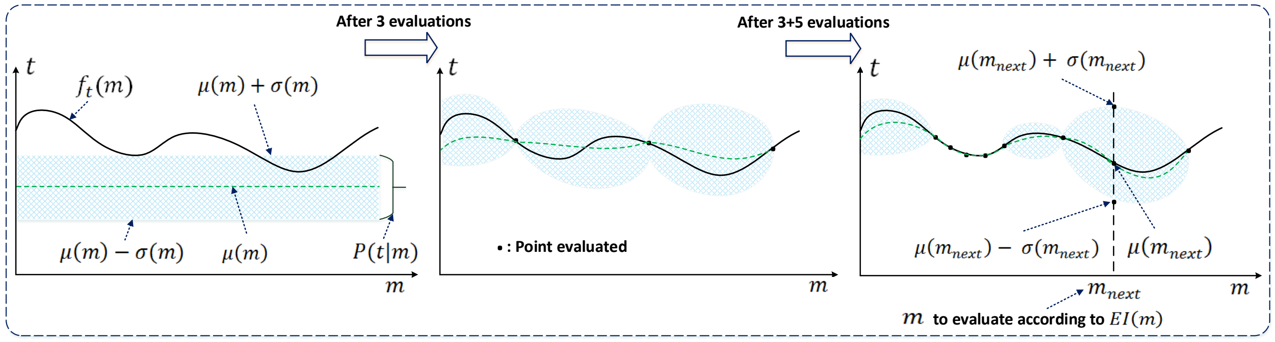


Figure A.18: Bayesian Optimization general method.

The surrogate function is optimized via Bayesian methods by selecting an $m$ that will perform well on $P(t|m)$. Figure A.18 shows a general overview of how increased evaluations affect $P(t|m)$. As an overview, the steps taken by the Bayesian optimizer are:

1. Build $P(t|m)$ according to the already evaluated $f_t(m)$. In our case, the homogeneous

mappings $m_1$,$m_2$,..$m_{N_D}$ and their results on $f_t(m)$ result in an initial $P(t|m)$.

2. Then, the Bayesian optimizer chooses the next $m$ that would be *assumed* to perform well on $P(t|m)$.

3. The chosen $m$ is evaluated with $f_t(m)$.

4. $P(t|m)$ is updated based on the results of $m$ on $f_t(m)$.

Steps 2 - 4 are repeated several times. The reason for the use of $f_t(m)$ rather than $f'_t(m)$ is that Bayesian Optimization is an expensive approach, namely the construction of $P(t|m)$, from history, and choosing the next $m$ to evaluate are expensive. Therefore, the Bayesian optimizer would perform well on expensive functions such as $f_t(m)$, given how the whole Bayesian process is expensive. Using $f'_t(m)$ in the Bayesian optimizer would not be beneficial time wise. Bayesian optimizers are expensive when it comes to computation time yet they require less calls to the objective function compared to other optimizers since they *reason* on what to evaluate next, i.e. use $P(t|m)$, to choose the next $m$ to evaluate.

To decide on which $m$ to evaluate next (step 3), a utility function known as the *acquisition function* [132] is used;

$$EI(m) = E[\max_m(0, f_t(m) - f_t(m_{best}))] \tag{A.17}$$

$$m_{next} =_m EI(m) \tag{A.18}$$

$EI$ is the *Expected Improvement*, a type of acquisition function. $m_{best}$ is the *current* best solution while $m_{next}$ is the next $m$ that would be evaluated. $EI(m)$ is analytically evaluated as follows:

$$EI(m) = \begin{cases} (\mu(m) - f_t(m_{best}))\Phi(Z) & \\ +\sigma(m)\phi(Z) & \sigma(m) > 0 \\ 0 & \sigma(m) \leq 0 \end{cases} \tag{A.19}$$

$$\text{where } Z = \frac{\mu(m) - f_t(m_{best})}{\sigma(m)} \tag{A.20}$$

Where $\mu(m)$ and $\sigma(m)$ are the *mean* and the *standard deviation* of the distribution of $P(t|m)$ at point $m$, respectively (as was mentioned in Equation A.16). While $\Phi$ and $\phi$ are the *cumulative distribution function* and *probability density function* of the *normal distribution*, respectively. Note that the acquisition function is less costly computation wise compared to $f(m)$, i.e. $\mu(m)$ and $\sigma(m)$ are very inexpensive to evaluate. $EI$ would have a high value if the evaluated $m$ is in a known neighborhood that outperforms $m_{best}$ (high $\mu(m)$), or we evaluate in an unknown territory (high $\sigma(m)$). Both approaches of *exploitation* (high $\mu(m)$) and *exploration* (high $\sigma(m)$) are used. For categorical data [52], which is the case with the mapping where the values are devices, the best way to construct the probabilistic surrogate function, and thus evaluate and search, is to use TPE. TPE is used by *constructing* (step 4) the surrogate function $P(t|m)$ by using *Bayes rule*;

$$P(t|m) = \frac{P(m|t)P(t)}{P(m)} \tag{A.21}$$

Where $P(m|t)$ is the probability of a mapping $m$ given an actual makespan $t$.

$$P(m|t) = \begin{cases} l(m) & t < t_{th} \\ g(m) & t \geq t_{th} \end{cases} \tag{A.22}$$

$t_{th}$ is the makespan threshold of the two distributions. $l(m)$ and $g(m)$ both have a normal distribution. With that said, $EI$ would be;

$$EI(m) = \frac{l(m)}{g(m)} \tag{A.23}$$

A selection strategy would be to select $m$ more towards the $l(m)$ distribution given ${}_{m}EI(m)$. As the Bayesian Optimizer progresses in number of iterations, $EI$ converges more towards exploitation rather than exploration, given that $P(t|m)$ gets closer to $f_t(m)$. An overview of the Bayesian Optimizer shown in Figure A.18.

Note that once an $m^*$ is found where $f_t(m^*) < f_t(m_{TF})$ via Bayesian Optimization, the Adaptive HTF-MPR bypasses all the other steps and continues execution using $m^*$, i.e. Run $f_{NN}$ (see Figure A.17).

## One-Hot Encoding

In *one-hot encoding*, a variable is *expanded* to multiple variables. The variables take in either a 0 or a 1. Exactly one of the expanded variables from the original variable has value of 1 while the rest are set to 0. In the case of a mapping $m$ where the size of $m$, or number of variables of $m$, without the one-hot encoding is $|m| = N_{op}$. If one-hot encoding is applied then the size would be the multiple of the number of values each non-one-hot encoding operation would take, i.e. the number of devices. More formally, $|m^{one-hot}| = N_{op}\dot{N}_D$. Figure A.19 illustrates the difference of *integer encoding*, as is the case in with HTF-MPR, and one-hot encoding, as is the case with Adaptive HTF-MPR.

Categorical variables have nominal values, meaning that the values have a qualitative property rather than a quantitative property. With integer encoding (as is the case in HTF-MPR, see Figure A.19) it assumes *order*, that is numbers have order in relation to each other. Thus,

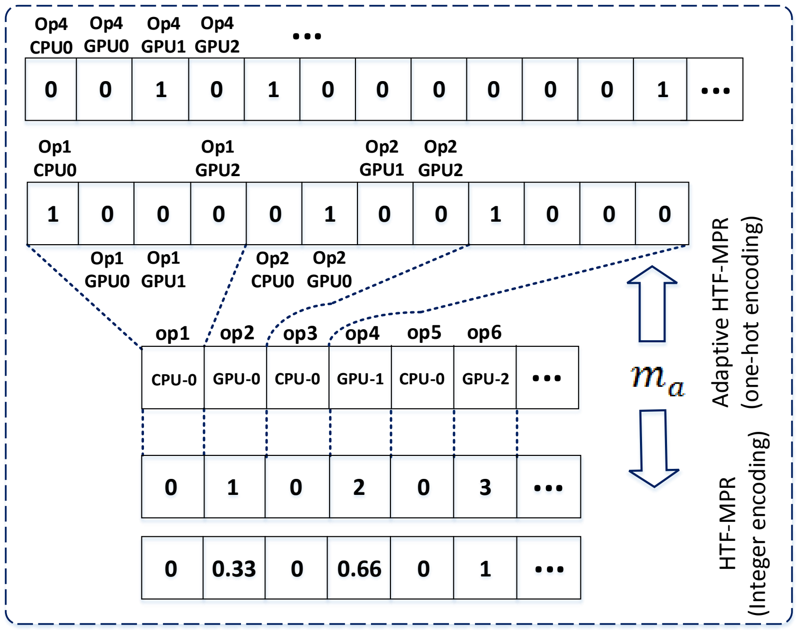


Figure A.19: Encoding: $m_a$ is encoded using integer encoding where CPU-0$\longrightarrow$ 0, GPU-0$\longrightarrow$ 1, GPU-1$\longrightarrow$ 2, and GPU-2$\longrightarrow$ 3. The integers are then *normalized*. The top part illustrates one-hot encoding, where *dummy variables* are used. This increases the number of *features*; in this case a single variable is expanded to four, since there are four devices. Note that CPU-0$\longrightarrow$ 1000, GPU-0$\longrightarrow$ 0100, GPU-1$\longrightarrow$ 0010, and GPU-2$\longrightarrow$ 0001.

CPU-0 does not have a closer relationship to GPU-0 than it does with GPU-1. If the integer 0 is assigned to CPU-0, 1 is assigned to GPU-0, and 2 is assigned to GPU-1 etc, we have implicitly assigned relations. These relations have an affect when used mathematically in the machine learning models. Since no ordinal relationship between the devices exists, one-hot encoding is more befitting.

### Training the Predictive Model

As in HTF-MPR, we train a surrogate function $f_t'(m)$ to be used in the GA. Using the mappings that were generated by the Bayesian optimization evaluations, we train, and thus create a makespan predictive model $f_t'(m)$ using GBR as explained in Subsection A.2.3. Note that the two main differences between the predictive model used in HTF-MPR and Adaptive

HTF-MPR are:

- The training dataset uses mappings that are skewed more towards better performing makespans, i.e.

$$\sum_{m \in M_{Bayesian}} f_t(m) < \sum_{m \in M_{initial}} f_t(m) \tag{A.24}$$

  Where $M_{Bayesian}$ and $M_{initial}$ are the mappings generated by the Bayesian optimizer (Adaptive HTF-MPR) and the Initial Mappings (HTF-MPR), respectively. In addition, $|M_{Bayesian}| = |M_{initial}|$, so as to make the comparison from Equation A.24 fair.

- One-hot encoding is used rather than normalized integer encoding. This is a better fit given the nature of the datatype of the values in the mapping; non-ordinal categorical data.

### Genetic Algorithm Search

As in HTF-MPR, GA is used to search for an optimal mapping that outperforms TensorFlow's default GPU homogeneous mapping using the makespan predictive model $f'_t(m)$ as the surrogate function to evaluate performance of a given solution. The difference here is that:

- The initial population is the mappings from the Bayesian optimizer, meaning a more concentrated search space.

- A makespan predictive model $f'_t(m)$ that is designed to work well within the neighborhood of the search space of the initial population.

**Adaptive-run**

During the run on $m^*$, the average as well as a the standard deviation is taken for a window size of $Q$ iterations of $f_t(m^*)$. If after the $Q$ iterations $f_t(m^*)$ changes to be higher or lower than $\beta$x of the standard deviation then that would cause a trigger to occur. The trigger would start the Adaptive HTF-MPR process again. Note that the number of iterations left for training and reaching the final trained model $f_{NN}$ would be reduced (see Figure A.17).

```
Initialization:
let $\mu_{win} = \frac{1}{Q}\sum_{i=1}^{Q} f_t(m^*)_i$;
let $\sigma_{win} = \sqrt{\frac{\sum_{i=1}^{Q}(f_t(m^*)_i - \mu_{win})^2}{Q}}$;
P_trigger = $\mu_{win} + \beta\sigma_{win}$;
N_trigger = $\mu_{win} - \beta\sigma_{win}$;
i = Q + 1;
Trigger=False;
while $f_{NN}$ still training do
    Advance $f_{NN}$ training;
    i=i+1;
    if N_trigger< $f_t(m^*)_i$ < P_trigger then
        Trigger=True;
        Break from while loop;
    end
end
if Trigger then
    run Adaptive HTF-MPR on $f_{nn}$ from
      iteration i
end
```

**Algorithm 1:** Monitoring Algorithm: The average makespan of each run is taken for a window size of $Q$. The standard deviation is recorded and the triggers are set. While running the Neural Network on mapping $m^*$, we check the current makespan. If the makespan is above the P_trigger or lower than the N_trigger, a trigger is set and Adaptive HTF-MPR is run again.

A trigger would indicate that there was a change in the hardware state; either a drop in performance (gradual or abrupt) or an improvement in performance (again, either gradual or abrupt). In either case this would require a reassessment of the values of $f_t(m)$ and therefore a search for a new $m^*$. Algorithm 1 shows the monitoring mechanism.

### A.2.5 Experimental setup

In order to evaluate the proposed method, a multi-core CPU (Intel(R) Core(TM) i7-7700 CPU 3.60Ghz) and 2 GPUs (Nvidia GeForce GTX 1050 Ti) were used. For the implementation, we used Python 2.7.15 using Anaconda bundled package of libraries. The benchmarks were implemented in GPU-supported TensorFlow 1.9.0, running on CUDA 9.1 and CuDNN v7.1. The GBR makespan predictive model was implemented using scikit-learn 0.19.1 [113] . The Bayesian optimizer was implemented using Hyperopt 0.2 [23].

To evaluate the proposed method three state of art benchmarks were run on HTF-MPR, Adaptive HTF-MPR and default TensorFlow mapper. Table A.1 shows the benchmark list, number of eligible operations for mapping and number of training iterations per benchmark.

| Benchmark | Mapped Operations | Total Operations | Training Iterations |
|---|---|---|---|
| MNIST Softmax | 10 | 99 | 60K |
| ALEXNET | 54 | 294 | 500K |
| VGG-16 | 69 | 376 | 500K |

Table A.1: Benchmarks.

Unigine's SuperPostion benchmarking tool [140] was used to stress-test the system in order to test out the adaptive feature of Adaptive HTF-MPR.

### Mnist Softmax

The *MNIST Softmax* used in our experiment is a simple TensorFlow implementation [20] that trains a classifier for ten-digit grayscale image dataset MNIST [86]. The dataset contains 60,000 training and 10,000 testing images. Each image is 28x28 grayscale and, as the dataset suggests, the classifier has 10 classes. Figure A.20 a shows the graph representation.

Given that the *mappable operations* (operations in the computational graph that are explicitly mentioned in the Python TensorFlow code) are only ten, the total number of possible mappings in this case are $N_D^{N_{OP}} = 3^{10}$. With this small number of mappings, it is possible to generate and evaluate the whole search space and therefore conduct a brute-force analysis to find the *global optimal* mapping. In this section, we will compare the $m^*$ of HTF-MPR,

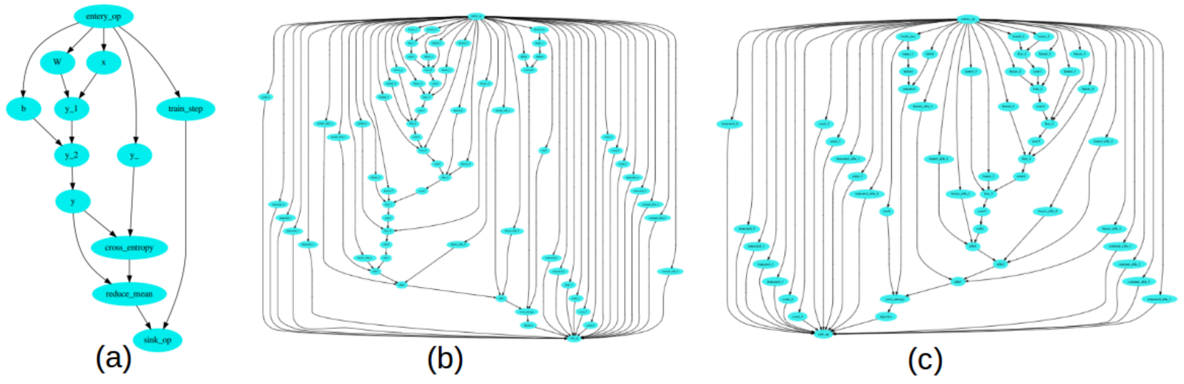


Figure A.20: **a)** MNIST Softmax computational graph. There are 10 mappable operations. The top and bottom nodes are virtual operations and are not mapped to any device. **b)** VGG-16 computational graph with 69 mapable operations. **c)** AlexNet computational graph with 54 mapable operations

Adaptive-HTFMPR, $m_{TF}$, and the global optimal. In addition, we will compare $F_t$ (see Equation A.8) of the policy followed by the three previously mentioned methods.

## AlexNet and VGG-16

AlexNet [80] and VGG-16 [134] are deep convolutional neural networks (CNNs) that are designed to classify images from the ImageNet [123] dataset. The ImageNet training dataset contains 1.2 million labeled images of 1000 labels, i.e. classifications. The input to the neural network is a 3-channel rescale image resolution of 224x224x3. HTF-MPR as well as Adaptive HTF-MPR is tested on both Neural Networks to gauge and evaluate the speedup where we compare the respective $f_t(m^*_{htf.mpr})$ and $f_t(m^*_{A.htf.mpr})$. Also, the total training time of $F_t(\pi_{htf.mpr}, oh)$ and $F_t(\pi_{A.htf.mpr}, oh)$ are measured. The computational graphs for VGG-16 and AlexNet are shown in Figure A.20. For 500,000 training iteration we used batch sizes of 64 and 32 for Alexnet and vgg16.

### A.2.6 Results

#### Mnist Softmax Analysis

We generated all $N_{D\,op}^{N} = 3^{10} = 59049$ mappings for Mnist Softmax. Figure A.21 shows part of the distribution of the makespan, as well as marks the average makespan and the $m_{GPU-0}$ homogeneous mapping's makespan. The makespan values extend to $\tilde{0}.02$ seconds. Makespan distribution beyond 0.002 is not shown in the figure. The three best mappings and three worst mappings are shown in Figure A.22 along with the value of the makespan.

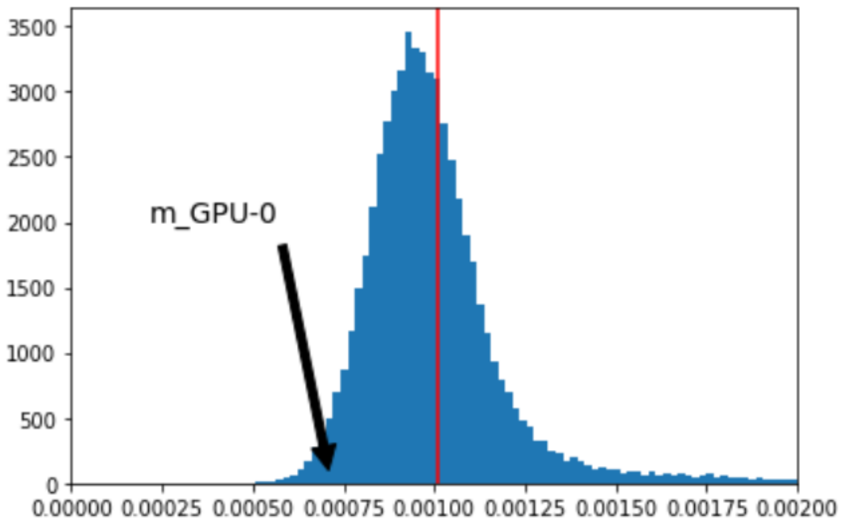


Figure A.21: MNIST Softmax makespan distribution. x-axis shows the makespan and y-axis shows the count for that makespan. Mean of the distribution is shown by the red vertical line. Note that the figure caps at 0.002s, but the distribution has a long tail that extends to 0.02s. Approximately 5% of mappings outperform the default Tensorflow $m_{GPU-0}$ mapping in the Mnist Softmax case.

#### Initial and Bayesian Mappings

We generate 700 mappings ($N$ in Figure A.17) through Bayesian Optimization, where the input to the Bayesian Optimization are the homogeneous mappings. We compared the

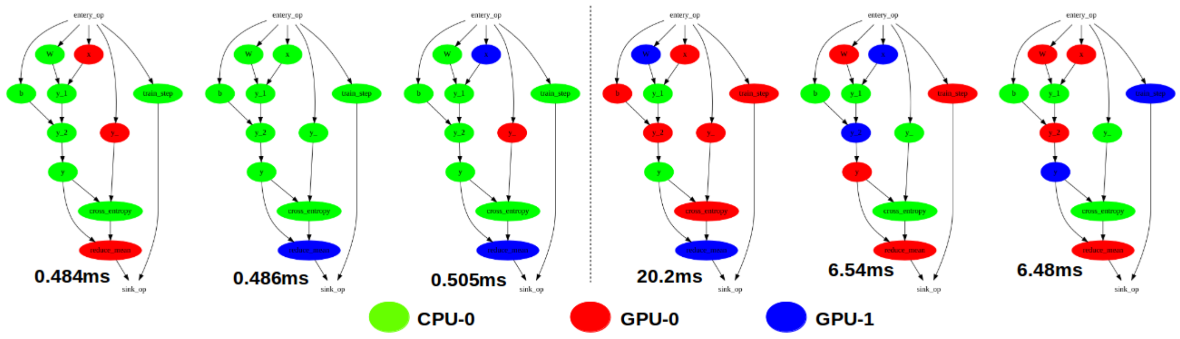


Figure A.22: The three mappings to the left are the top three mappings in terms of makespan. The top most has 7 operations mapped to CPU-0, and 3 operations mapped to GPU-0, with a makespan of 0.484 ms per iteration. The three to the left are the worst mappings, the worst mapping has a makespan of 20.2 ms, with 2 operations mapped to CPU-0, 6 operations mapped to GPU-0, and 2 operations mapped to GPU-1. The mapping $m_{GPU-0}$ has a makespan $f_t(m_{GPU-0})$ =0.72 ms. Note that the worst mappings change devices after each operations incurring high communication costs overhead.

generated mappings of the Bayesian Optimization approach (a), the Algorithmic approach (b),the Genetic Algorithms approach (c), and the Random approach (d) (see Figure A.23). Note that the number of training runs per mapping is equal to 5 in our case, this is in order to mitigate the time overhead due to reconstructing the graph with a different mapping.

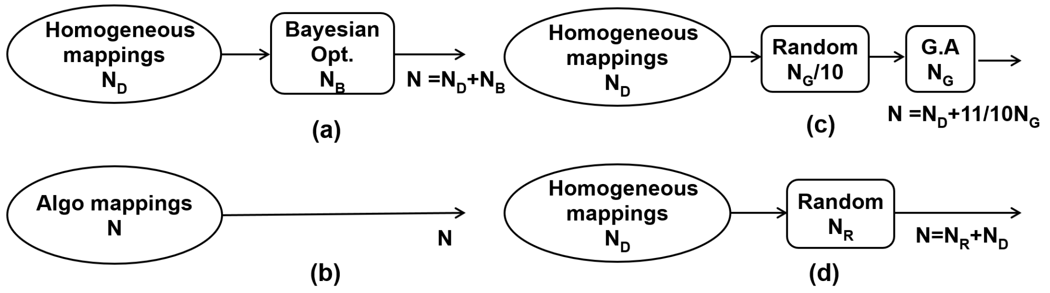


Figure A.23: Configurations of initial mappings. **a)** is the Adaptive HTF-MPR approach while **b)** is the HTF-MPR approach. N is equal for all configurations. The number of mappings generated is N=700 in each case.

In each case, we show the final distribution of makespans of the mappings generated by the different methods and show the the final average of each method. Figure A.24 and

Figure A.25 show the distribution for VGG-16 and Alexnet, respectively, with N=700. Note the Bayesian Optimization method has an overall lower mean and the distribution is skewed to lower makespans.

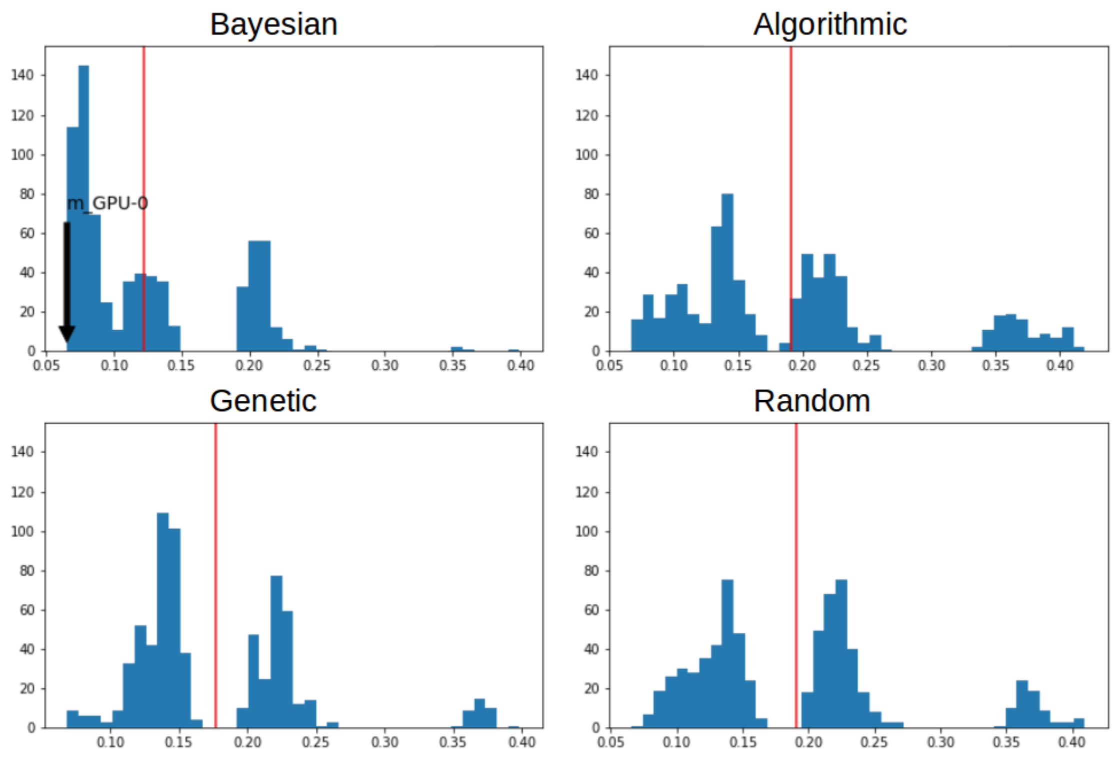


Figure A.24: Makespan distribution for **VGG-16**. The x-axis is the makespan and y-axis is the count of mappings. The vertical red line indicates the average of the distribution. In the Bayesian figure, the Tensorflow default mapping's makespan is indicated with a black arrow labeled $m_{GPU-0}$.

Figure A.26 shows further insight into how the averages of the makespan distribution changes with time. The default mapping in this case is best, for now, given that this the first stage. The Bayesian shows a steady improvement meaning that with each iteration better mappings are found. Same, but slower, trend can be seen by the GA's method. The Algorithmic method starts off with relatively good mappings, but with each iteration does not show much improvement (but finds a good mapping later on which is not shown by the latest average but can be observed in the latest minimum figure), this is indicative of running out of good mapping *ideas*, where *intuition* does not pan out much further.

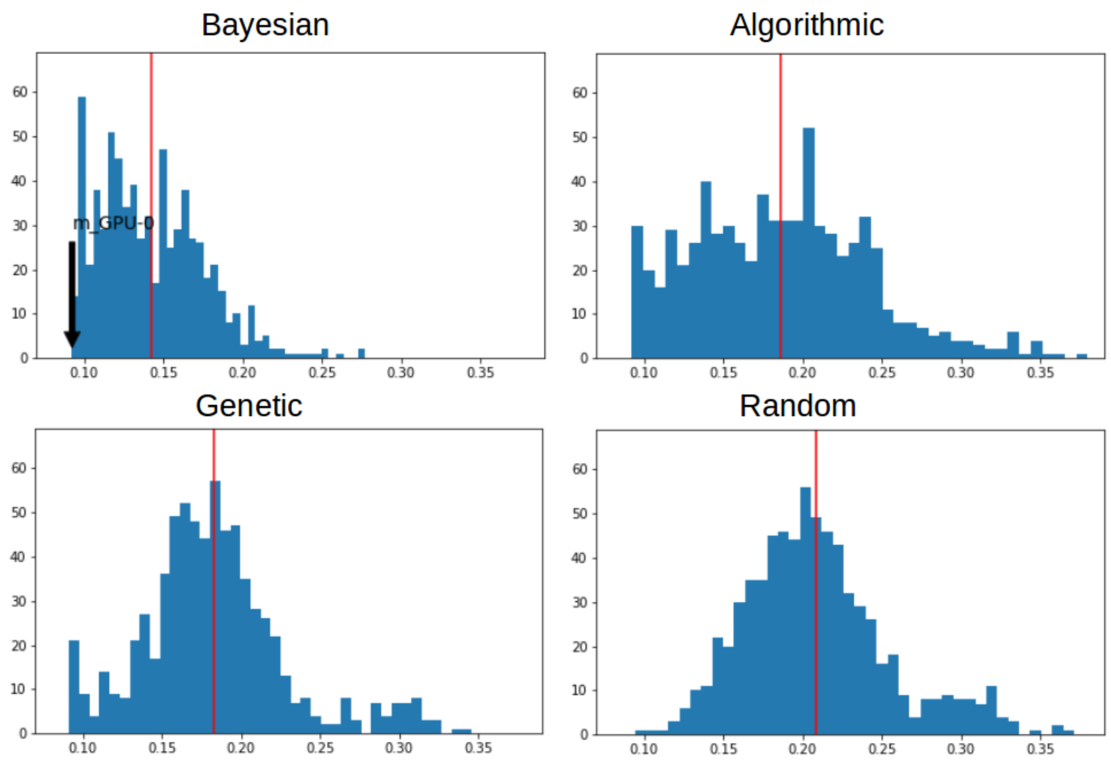


Figure A.25: Makespan distribution for **Alexnet**. The x-axis is the makespan and y-axis is the count of mappings. The vertical red line indicates the average of the distribution. In the Bayesian figure, the Tensorflow default mapping's makespan is indicated with a black arrow labeled $m_{GPU-0}$.

Figure A.27 shows the latest minimum at each iteration. Note that Genetic, Algorithmic and Bayesian all eventually converge within the same neighborhood. Genetic seems to get there quicker while Algorithmic, and Bayesian get there later on iterations.

An important consideration for training time is not only the final makespan that is achieved i.e. $f_t(m^*)$, but the whole process $F_t(\pi, oh_\pi)$. Figure A.28 shows the overall time it takes for the first stage (before the ML stage and running the GA with the predictive model function). Note that in the initial stage the default outperforms the other methods, but when used in conjunctions with the later stages a better mapping is found and therefore a faster overall training time (as shall be shown in Subsection A.2.6) .

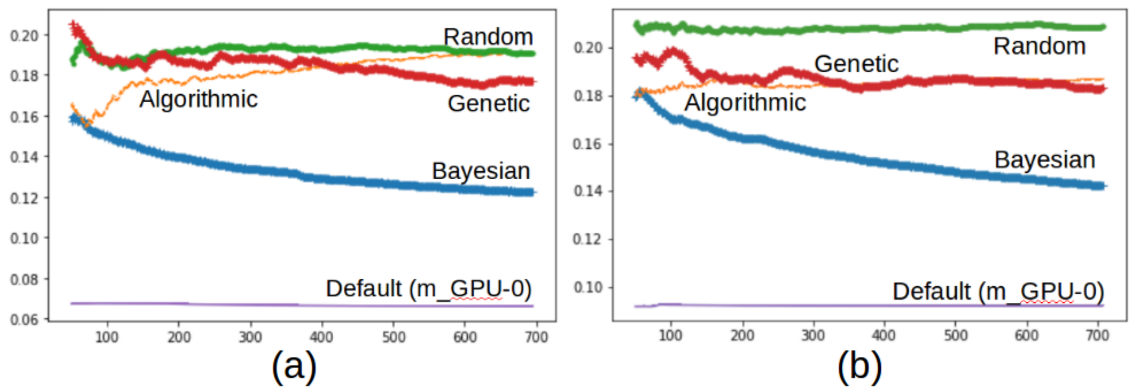


Figure A.26: The latest average with each iteration for a) **VGG-16** and b) **Alexnet**. The x-axis shows iteration count, while the y-axis shows the average makespan. Note that the plot starts from iteration 50. the Bayesian improves with each iteration, same for the GA method.

The GA overhead is comparable to that of the Random method i.e. low overhead. While the Bayesian method is higher than genetic and random. This indicates that when optimizing and searching using the predictive model of the makespan, $f'_t(m)$, it is better to utilize a low-expense optimization method to reduce overhead. Evaluating $f'_t(m)$ is much faster than evaluating $f_t(m)$ by a magnitude of $\tilde{7}$00 and thus it is imperative that many evaluations occur vs *smarter* evaluations. If Bayesian were to be used on $f'_t(m)$ the benefits would vanish due to the costly overhead.

## Makespan Prediction

In this section we compare the performance of using one-hot encoding and integer encoding using the dataset generated by Bayesian and Algorithmic to create the predictive model $f'_t(m)$. To evaluate the performance we use the *Kendall tau rank distance* and used k-fold cross validation method [78] to test the performance . The relative ranking of the mappings, in terms of makespan, and not the actual makespan value is the metric of measurement.

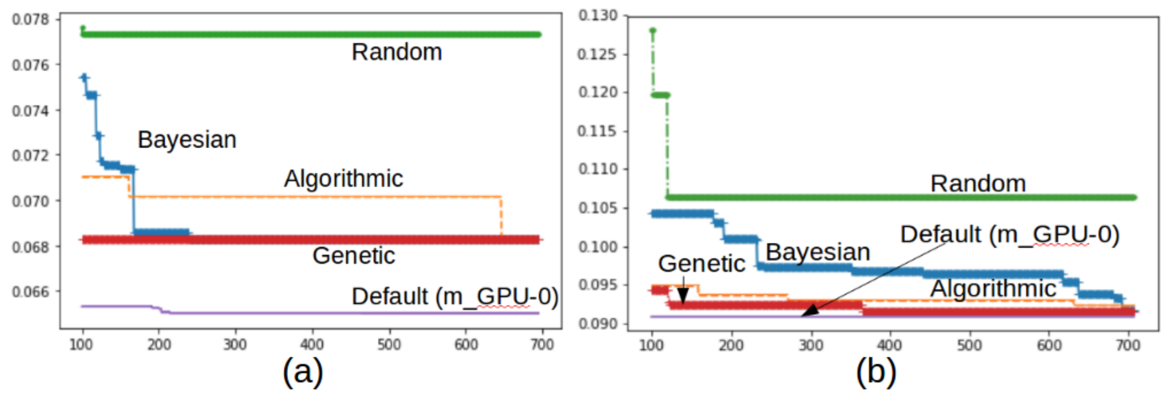


Figure A.27: The latest minimum with each iteration for a) **VGG-16** and b) **Alexnet**. The x-axis shows iteration count, while the y-axis shows the minimum makespan. Note that the plot starts from iteration 100.

Given two mappings $m_a$ and $m_b$, the *Kendall number* is calculated as follows:

$$k(t_a, t_b, t'_a, t'_b) = \begin{cases} 1, & \text{if } t_a < t_b \text{ and } t'_a > t'_b. \\ 1, & \text{if } t_a > t_b \text{ and } t'_a < t'_b. \\ 0, & \text{otherwise.} \end{cases} \tag{A.25}$$

Where $f_t(m_a) \longrightarrow t_a$ and $f_t(m_b) \longrightarrow t_b$ are the actual makespans of mapping $m_a$ and $m_b$, respectively, and $f'_t(m_a) \longrightarrow t'_a$ and $f'_t(m_b) \longrightarrow t'_b$ are the predicted makespans of $m_a$ and $m_b$ respectively. A value of **1** indicates a mismatch in the pair-wise order between the actual and the predictive makespans, and **0** indicates a preserved ordering. The normalized Kendall tau ranking distance is thus;

$$K_{norm} = \sum_i \sum_{j<i} \frac{2 \cdot k(t_i, t'_i, t_j, t'_j)}{N(N-1)} \tag{A.26}$$

Figure A.29 shows an example; five mappings $m_1, m_2, m_3, m_4, m_5$ where the actual makespans are $t_1 < t_3 < t_5 < t_4 < t_2$ and the predicted makespans are $t'_3 < t'_4 < t'_5 < t'_2 < t'_1$.

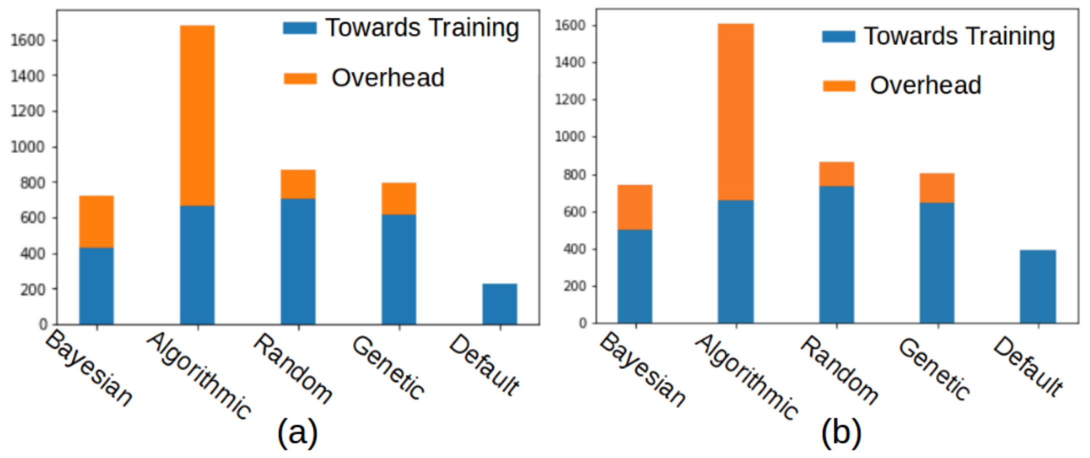
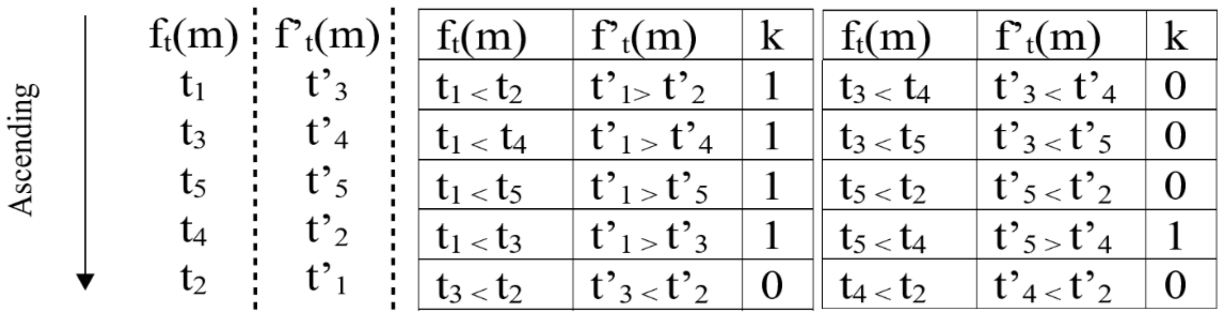


Figure A.28: Total time of first stage (Figure A.23) for a) **VGG-16** and b) **Alexnet**. The total time (seconds) is the sum of the overhead due to search and reconstruction of the graph with each new mapping, and the actual run of the $f_{NN}$(contributes to the reduction of number of training iterations left). With default Tensorflow there is no overhead since there is no reconstruction of the graph given that the mapping is constant. Note that with N=700, there are 5 training iterations per evaluated mapping. Therefore, the Figures show the time for 700x5=3500 training iterations of $f_{NN}$.

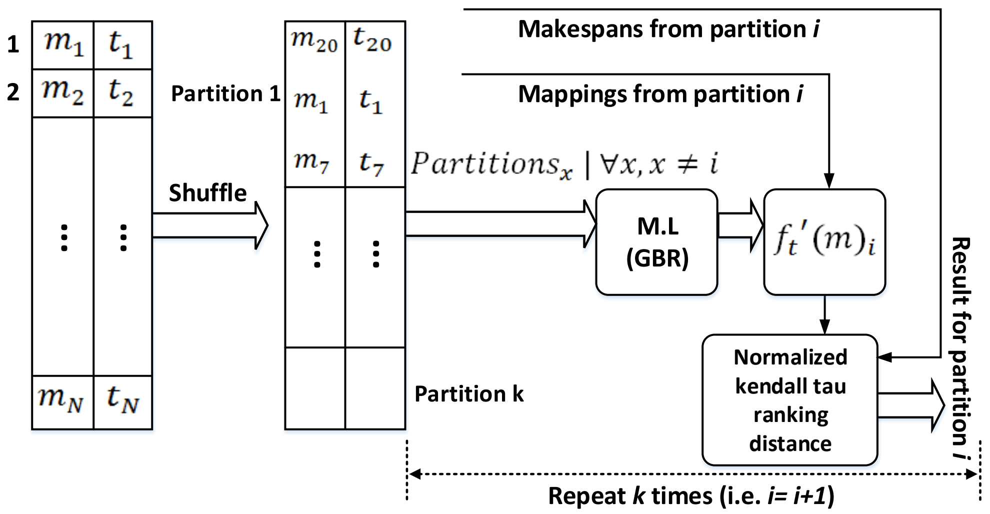


| $f_t(m)$ | $f'_t(m)$ |
|---|---|
| $t_1$ | $t'_3$ |
| $t_3$ | $t'_4$ |
| $t_5$ | $t'_5$ |
| $t_4$ | $t'_2$ |
| $t_2$ | $t'_1$ |

| $f_t(m)$ | $f'_t(m)$ | k | $f_t(m)$ | $f'_t(m)$ | k |
|---|---|---|---|---|---|
| $t_1 < t_2$ | $t'_1 > t'_2$ | 1 | $t_3 < t_4$ | $t'_3 < t'_4$ | 0 |
| $t_1 < t_4$ | $t'_1 > t'_4$ | 1 | $t_3 < t_5$ | $t'_3 < t'_5$ | 0 |
| $t_1 < t_5$ | $t'_1 > t'_5$ | 1 | $t_5 < t_2$ | $t'_5 < t'_2$ | 0 |
| $t_1 < t_3$ | $t'_1 > t'_3$ | 1 | $t_5 < t_4$ | $t'_5 > t'_4$ | 1 |
| $t_3 < t_2$ | $t'_3 < t'_2$ | 0 | $t_4 < t_2$ | $t'_4 < t'_2$ | 0 |

Figure A.29: An example of the kendall values for 5 makespans. The resulting $K_{norm} = 0.5$.

The K-fold method used to validate the predictive model shown in Figure A.30. The mappings-fitness pairing are shuffled then partitioned into k parts. The predictive model $f'_t(m)_i$ is trained using all the partitions except for partition $i$. Partition $i$ is then used as a validation to observe the normalized Kendall tau ranking distance. This process is repeated k times, each time we use a different partition $i$ for the validation. Figure A.30 shows how this process is carried out.



1
2
$m_1$ $t_1$
$m_2$ $t_2$
$m_N$ $t_N$
Shuffle
Partition 1
$m_{20}$ $t_{20}$
$m_1$ $t_1$
$m_7$ $t_7$
Partition k
Makespans from partition *i*
Mappings from partition *i*
$Partitions_x \mid \forall x, x \neq i$
M.L (GBR)
$f_t{}'(m)_i$
Normalized kendall tau ranking distance
Result for partition *i*
Repeat *k* times (i.e. *i*= *i*+1)

Figure A.30: k-fold method of validation. The mappings (input) and the makespan timings (labels) are shuffled. They are then split into k parts. A partition is selected to be the test dataset while the rest of the partitions are used for training the model using GBR. The resulting predictive model is then tested with the test dataset partition. the Normalized Kendal tau ranking is taken and the process is repeated but each time a different partition is used as the test dateset.

The results are shown in Figure A.31. Note that in each case the one-hot encoding outperforms the integer encoding. The performance will affect how many mappings will be chosen to be evaluated. That is, the *top* $K$ mappings after the GA stage.

### Genetic Algorithm on Predictive Model

In this section the results of the GA on the predictive model are presented. The factors that are essential in evaluating the performance of this part are the following;

- The time it takes to search using the GA on the makespan predictive model $f_t'(m)$. That time is indicated by $T_{N+P}$, while the size of the search is $N + P$.
- The results of the search; The first occurrence of a mapping that has a makespan $f_t(m^*)$ better than the default mapping $f_t(m_{TF})$ makespan.

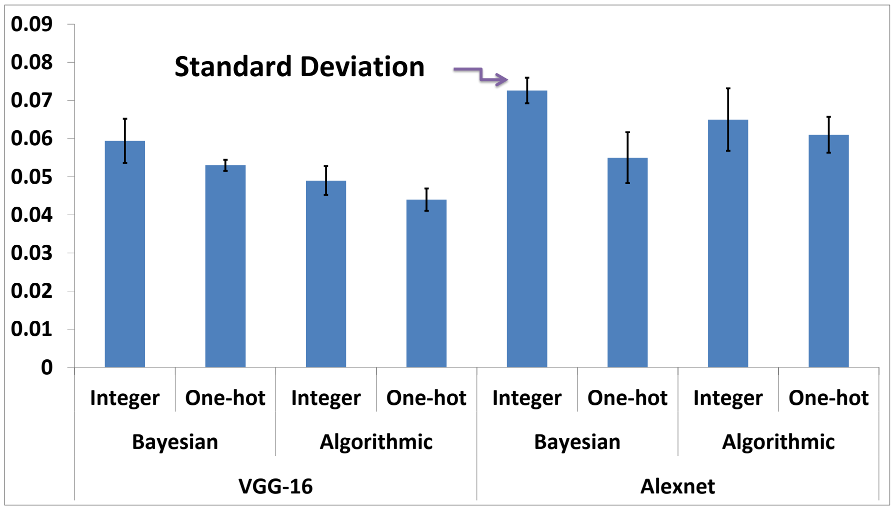


Figure A.31: K-fold results. The y-axis is the normalized Kendall where a lower number indicates a lower error rate. Note that N=700 (number of mappings) and K=5 (number of folds). The bar indicates the average of 5 runs (Normalized Kendall of 5 tested partitions) and the standard deviation shown is due to the difference of the 5 runs.

- What number of evaluations $K$, using $f_t(m)$, are needed to find the best possible makespan $f_t(m^{**})$ among the resulting GA results. Number of evaluations is correlated to time $T_K$. Note that $f_t(m_{TF}) > f_t(m^*) \geq f_t(m^{**})$.

| Model | $N+P$ | $T_{N+P}$ | $K$ | $T_K$ | $i^*$ | $\frac{f_t(m_{TF})}{f_t(m^*)}$ | $rank'(m^*)$ | $rank(m^*)$ | $i^{**}$ | $\frac{f_t(m_{TF})}{f_t(m^{**})}$ | $rank'(m^{**})$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Bay | 10000 | 25.7s | 1000 | 770s | 1021 | 1.04 | 6 | 12 | 7836 | 1.205 | 16 |
| Bay | 100000 | 1012s | 1000 | 768s | 74611 | 1.04 | 1 | 521 | 10026 | 1.209 | 9 |
| Bay | 10000 | 25.36s | 100 | 77.6s | 7428 | 1.036 | 2 | 37 | 7290 | 1.201 | 69 |
| Algo | 10000 | 22.22s | 1000 | 769s | 2935 | 1.04 | 4 | 89 | 9342 | 1.19 | 58 |
| Algo | 100000 | 1099s | 1000 | 743s | 8002 | 1.038 | 7 | 51 | 10247 | 1.204 | 81 |

Table A.2: GA result using predictive model $f_t'(m)$ on **Alexnet**. In this table, "Bay" refers to "Bayesian" and "Algo" refers to "Algorithmic" model.

As indicated in [43], the initial population is an important metric to the GA. Table A.2 and Table A.3 show that the initial population generated by the Bayesian optimizer in both instances outperformed the Algorithmic initial population. As for the size of the search, 10,000 searches in the GA and 100 evaluations was enough to find the best mapping in Alexnet. For VGG-16, the search space is larger and therefore a search of 150,000 is required

| Model | $N+P$ | $T_{N+P}$ | $K$ | $T_K$ | $i^*$ | $\frac{f_t(m_{TF})}{f_t(m^*)}$ | $rank'(m^*)$ | $rank(m^*)$ | $i^{**}$ | $\frac{f_t(m_{TF})}{f_t(m^{**})}$ | $rank'(m^{**})$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Bay | 100000 | 1014s | 1000 | 752s | 1 | 1.00 | 1 | 1 | 1 | 1.0 | 1 |
| Bay | 150000 | 1806s | 1000 | 765s | 100179 | 1.06 | 56 | 3 | 130775 | 1.14 | 87 |
| Algo | 100000 | 920s | 1000 | 698s | 1 | 1.00 | 112 | 1 | 1 | 1.00 | 112 |
| Algo | 150000 | 1846s | 1000 | 703s | 1 | 1.00 | 18 | 1 | 1 | 1.00 | 18 |

Table A.3: GA result using predictive model $f'_t(m)$ on **VGG-16**. In this table, "Bay" refers to "Bayesian" and "Algo" refers to "Algorithmic" models.

$m_{TF}$.

**Run and Adaptivity**

In this section the full run of the TensorFlow default mapper, the HTF-MPR, and the Adaptive HTF-MPR, are presented. In addition, a stress-test is applied on the system and the changes of the makespan are observed. We see how Adaptive HTF-MPR reacts and how it affects the overall training time. The total training time for VGG-16 and Alexnet are shown in Figure A.32. The overhead with Alexnet is low due to the fact that the GA part is not run for long (only 10,000 mappings). The GA gets slower with time and does not have a linear relationship with number of iterations as can be seen from Table A.2 when comparing 10,000 runs and 100,000 runs; the increase in $T_{N_P}$ is 40x while the number of GA iterations increased by only 10x.

Figure A.33 shows what happens to the makespan when a high load is applied.

We applied a high load on GPU-0 for a 30 min duration. The makespan per iteration is shown in Figure A.34. The performance of the predictor went down when Adaptive HTF-MPR was triggered ( due to the GPU-0 high load). The reason for the low performing predictor is the high variance of the makespan (see Figure A.33).

Depending on the load duration and how sporadic the load is, the adaptive part would perform accordingly. In the case of high variance (sporadic) the makespan predictor will not

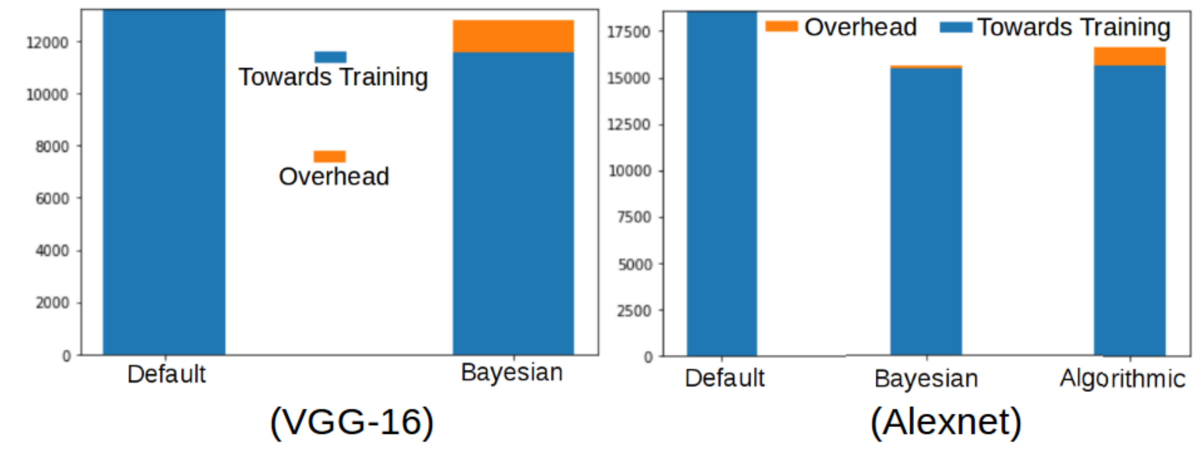


Figure A.32: Total training time (seconds). The Bayesian Optimization approach (Adaptive HTF-MPR) improved the overall time by **3.5%** in VGG-16 and **18.7%** in Alexnet. The overhead in the Bayesian accounts 9.5% of the whole process in VGG-16 while it accounts for 1.1% in Alexnet. Note that the Algorithmic did not find a better mapping for VGG-16 as shown in Table A.3. As for Alexnet, the overall improvement was by 12% and the overhead accounts for 5.6% using the Algorithmic approach.

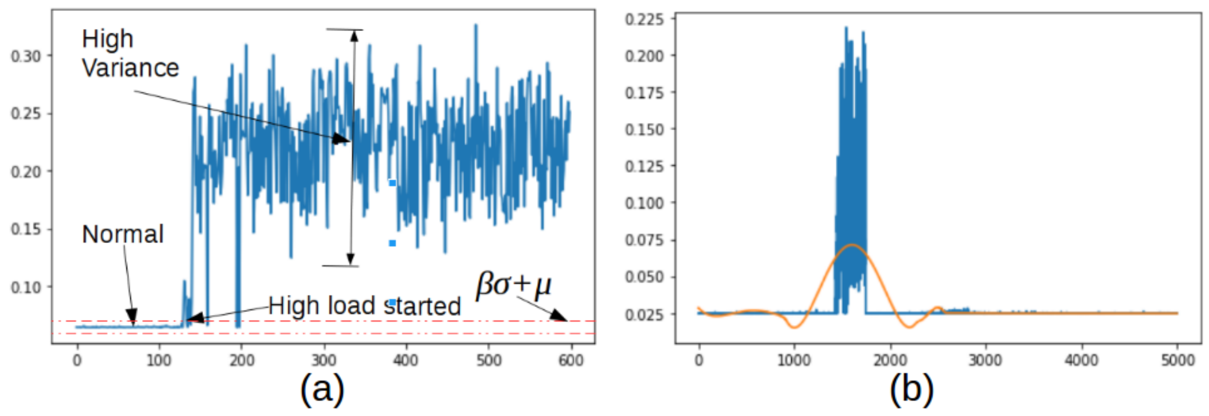


Figure A.33: The TensorFlow default mapping on a) **VGG-16** b) **Alexnet**. The y-axis is the makespan and x-axis is the iteration. The makespan changes when there is high load (using Unigine's SuperPostion benchmarking tool [140]) on the GPU. The red-line shows the threshold for when Adaptive HTF-MPR would be triggered if the default mapping was also the $m^*$ mapping. $\beta = 10$ in this case. Note that we used different loads in both instances. Also, the load has high variance in this case.

be able to get a single point prediction. In case of a bump over or bellow the P_trigger and N_trigger, respectively, Adaptive HTF-MPR would perform as usual.

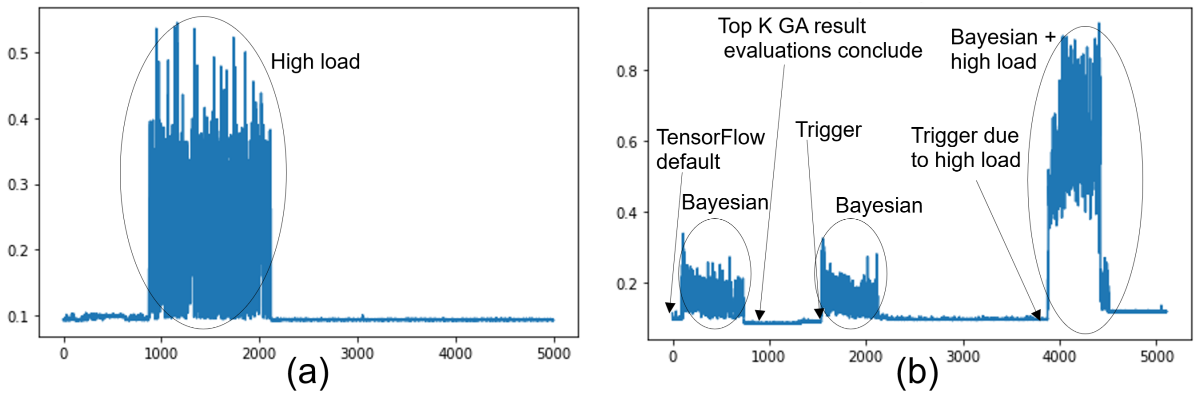


Figure A.34: Alexnet makespan at each iteration **a) without** and **b) with** Adaptive HTF-MPR. Note that GA happens offline (meaning the GA does not contribute to the advancement of the training step) and therefore is not shown. The top K of the resulting GA results are run on $f_{NN}$ and therefore are shown. In this case K=100. The high load is applied for 30 minutes in both cases.

## A.2.7 Summary of Approach Two

In this work, we presented Adaptive HTF-MPR to optimize the mapping of devices to operations in order to improve performance. The proposed framework uses Bayesian Optimization as well as a predictive model on the GA to search for a mapping that outperforms the TensorFlow default mapping. The predictive model is trained using the Bayesian Optimization resulting mappings and makespan observations. The predictive model is constructed using GBR. Experimental results show a substantial overall speedup for the investigated benchmarks. In addition, we presented our analysis of the solution-space using the small benchmark MNIST-softmax. We observed that only a small percentage, 5%, of mappings outperform the default TensorFlow mapping indicating that a successful search scheme is difficult in a large computational graph. The proposed search technique was able to find a mapping that outperforms the default TensorFlow mapping. We also presented the adaptivity mechanism; how it reacts when the system experiences stress.