UNIVERSITY OF THESSALY

ELECTRICAL AND COMPUTER ENGINEERING

Diploma Thesis:

Process-In-Memory System Simulation using the Gem5 Platform

Papalekas Dimitrios

Supervisor:

Stamoulis Georgios

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ Η/Υ

Διπλωματική Εργασία:

Προσομοίωση Συστημάτων Επεξεργασίας στη Μνήμη με τη Χρήση της Πλατφόρμας Gem5

Παπαλέκας Δημήτριος

Επιβλέπων:

Σταμούλης Γεώργιος

Examination Committee:

Prof. Dimitriou Georgios

Prof. Stamoulis Georgios

Prof. Sotiriou Christos

University Of Thessaly, 24/2/2022

Acknowledgments

Primarily, I would like to express my gratitude to my supervisor, Prof. Georgios Dimitriu. He introduced me to uncharted research subjects, inspired my interest and supported me with his deep knowledge and work ethics, despite all the setbacks that emerged along the way. His enthusiasm for novelty and his reassuring personality are traits that I really look up to.

I would also like to thank Prof. George Stamoulis. His guidance, arising from his valuable experience and selfless mentorship, throughout my whole studies kept me always on track. His initiative to introduce me to the Electronics lab of University of Thessaly was the starting point of a collaboration and honest friendship with its leaders. Dr. Antoniadis Charalambos, Dr. Garyfallou Dimitrios and Dr. Floros George. Their blend of hard work, dedication and generous support refined my character.

Last but not least, I would like to thank my family. Among others, my mom Mary equipped me with self-belief, my father Costas with methodical mindset and my step-father Nikos with honesty and patience. Also, I am grateful to my siblings, Andreas and Olympia, with whom we share admiration and love. And of course, my soulmate Marini, who tirelessly loves me and keeps me focused on my goals no matter what the circumstances are.

Abstract

Contemporary CPU capabilities are more than enough to process demanding and intense computational tasks. On the contrary, the challenges that impede the conventional DRAM memory scaling renders it the main bottleneck of today's computer systems, diminishing that way their total processing power. The aforementioned is severely felt when dealing with data-intensive applications such as Machine Learning algorithms, Graph Processing or heavy consumer workloads. To overcome the DRAM evolution obstacles system architects shifted their research focus to Near-Data-Processing (NDP) architectures, where the memory die is accompanied with logic capable of processing data where they reside, alleviating the need for costly data transactions between the CPU and the main memory. The emergence of 3D-Stack Memory models made the realization of such a venture feasible. However, only a small amount of these models is commercially available and accessible.

In order to surpass the challenges that remain until the broad adoption of NDP architectures, practical and easy-to-use NDP simulation environments must be developed and distributed to enable contributions. Among a confined range of selection, we opted for DAMOV, a general-purpose kernel-offloading PIM simulator based on zsim and Ramulator along with its benchmark suite that contains a broad-scoped selection of memory intensive applications.

The current study can be divided into two parts. The first part depicts a comprehensive image of the NDP trends that define the contemporary research field, their branches, and the range of application of them respectively. The second part presents a methodology for determining the suitability of workloads for NDP execution function-offloading granularity, after observing key metrics that are extracted from the simulator. By carefully applying the metrics in a specific sequence, we classify the workloads into categories that can determine a priori if NDP can result to execution speedup

Περίληψη

Οι δυνατότητες των σύγχρονων επεξεργαστών είναι επαρκείς ώστε να εκτελέσουν απαιτητικές και έντονες υπολογιστικών διαδικασίες. Αντιθέτως, λόγω των εμποδίων που δυσχεραίνουν την εξέλιξη της DRAM, η μνήμη καθίσταται κύρια αιτία υπολογιστικής συμφόρησης. Το γεγονός αυτό είναι περισσότερο προφανές όταν αντιμετωπίζονται εφαρμογές με μεγάλες απαιτήσεις δεδομένων, όπως είναι οι αλγόριθμοι Μηχανικής Μάθησης και η επεξεργασία γραφημάτων. Προκειμένου να ξεπεραστούν οι δυσκολίες στην ανάπτυξη της DRAM, οι επιστημονική κοινότητα έχει στρέψει το ενδιαφέρον της σε αρχιτεκτονικές Επεξεργασίας Κοντά στη Μνήμη. Σε αυτού του τύπου τις αρχιτεκτονικές το κύκλωμα της μνήμης συνοδεύεται από ένα λογικό κύκλωμα ικανό να επεξεργαστεί τα δεδομένα κοντά στο χώρο αποθήκευσης τους, ελαχιστοποιώντας με αυτόν τον τρόπο την ανάγκη για ενεργοβόρες μεταφορές δεδομένων ανάμεσα στην κύρια μνήμη και τον επεξεργαστή. Η πραγματοποίηση τέτοιων εγχειρημάτων κατέστη δυνατή μετά την ανάπτυξη τρισδιάστατων μνημών. Παρόλα αυτά, ελάχιστα μοντέλα είναι διαθέσιμα για εμπορική χρήση.

Προκειμένου να διευκολυνθεί η έρευνα που θα καθιερώσει την E.K.M., γίνεται προσπάθεια να αναπτυχθούν προσομοιωτές λογισμικού ανοικτού κώδικα και εύκολοι στη χρήση. Μέσα από περιορισμένες επιλογές, επιλέξαμε τον DAMOV, έναν επεξεργαστή για E.K.M γενικού σκοπού επιπέδου συνάρτησης, βασισμένο στους zsim και ramulator, μαζί με τα προτεινόμενα benchmarks που τον συνοδεύουν, και αφορούν ένα μεγάλο εύρος εφαρμογών.

Η παρούσα μελέτη διαχωρίζεται σε δύο τμήματα. Το πρώτο τμήμα παρουσιάζει μια συμπυκνωμένη και κατανοητή ανάλυση των κύριων τομέων E.K.M, των κλάδων αυτών και του πεδίου εφαρμογής για κάθε έναν. Το δεύτερο τμήμα παρουσιάζει μια μεθοδολογία ικανή να καθορίσει την αποτελεσματικότητα της E.K.M σε επίπεδο συνάρτησης. Η μεθοδολογία στηρίζεται στη δημιουργία κατηγοριών που περιέχουν εφαρμογές με παρόμοια συμπεριφορά, και δημιουργούνται μέσω της εφαρμογής ειδικών μετρητικών κριτηρίων που προκύπτουν από τη διεξαγωγή των πειραμάτων.

Table Of Contents

Contents

1 Introduction

For decades now, the performance of CPUs has been improving at a very fast rate, minimizing the energy and time cost to perform demanding arithmetic operations. To execute these computations, all the data involved must be located in the core's cache memory. Data movement from the main memory to the processing cores can be up to four orders of magnitudes slower, hence energy consuming also. Nowadays the datasets of modern applications are immensely growing, increasing the occurrences of cache misses, thus the need for data movement from the RAM to CPU [1, 2]. The aforementioned transfer happens via a narrow-bandwidth memory bus. To increase performance, systems' evolution led to the inclusion of mechanisms focused to the alleviation of the impact of data movement such as prefetchers and deep cache hierarchies. This strategy, nevertheless comes with significant hardware cost and does not fit well to the wide range of applications [3, 4, 5, 6]. To keep up with modern workload needs, typical computer systems must either increase cache capacity by adding more cores or include a main memory system that scales efficiently in terms of performance, energy and capacity altogether. The first aforementioned assumption is ruled out by the dark silicon effect [7] and by the fact that by sharing the last level cache memory, many CPU cores tend to access the same memory addresses causing cache contention, or fail to exploit the caches due to little data reusability [8]. The second is considered infeasible by many studies [9, 10] as, understandably, the improvements of a category come on the expense of the others. As a result, the focus on computer systems research is shifted from the Von-Neumann processor-centric model to the Near-Data-Processing (NDP) paradigm [11, 12, 13, 14], a paradigm that suggest the processing of data near or even inside the memory, where they reside. This paradigm aims to function in an effective way for applications that are characterized by irregular access patterns, little memory locality and larger working sets [15 16, 17, 18, 19]. Although NDP has been proposed for more than 50 years [11, 20], hardware limitations prevented its materialization. Recent breakthroughs such as 3D-stacked memory [21, 22] led to the realization of the concept.

NDP, also called Processing-In-Memory (PIM) can be divided into two main approaches. The Processing-Using-Memory, that aims to exploit analog, bulk operations inside memory cells such as simple bitwise operations [23, 24] and the Processing-Near-Memory that suggests the integration of 3D-stacked memory along with a logic layer that operates as a CPU directly underneath the main memory. Because the current bandwidth is limited by the numbers of I/O pins available in DRAM and the capabilities of the memory bus, more bandwidth is now provided to the cores of the NDP logic layer. The latter has been explored by means of application-specific kernels [15, 16, 18] with individual instruction offloading [25, 26] and general-purpose cores that encapsulate entire function-offloading capabilities [27]. Application-specific kernels are PIM modules that try to exploit the characteristics of a specific category of applications usually only offloading certain instructions for NDP. A classic example is recognizing and offloading the data manipulation tasks of graph processing algorithms. Despite the fact that application-specific solutions achieve important speedups when implemented for crucial workload categories, the strict offloading conditions they oblige in are a barrier for their widespread adoption. On the other hand, PUM and instruction offloading require either a compiler or profiler to determine every instruction's PIM suitability, usually adding a time overhead, or the programmer's knowledge and PIM-specific code handling.

In our opinion, the most viable solution is in the direction of general-purpose function-offloading. Offloading the whole function to the PIM consists of the mechanism with the less demanding adjustments from the programmer's endpoint, and general-purpose approach renders the widespread adoption of the PIM paradigm more feasible by its panoramic perspective.

In order to defend our proposition and to perform a holistic review on the matter, we present a comprehensive analysis of the most prolific trends in the NDP research field, analyzing the benefits and the liabilities of each and every one. Later on, we narrow our focus to function-offloading PIM execution and propose an empirically derived methodology on how to predict whether applying PIM in the application under examination would result to a desired execution speedup. We then try to highlight the significance of open-source general-purpose function-

offloading software simulation of NDP systems that succeed to assist the exploration of a) various architecture proposals b) offload-candidate workloads at no hardware cost and demand no special programming adjustments.

Among the available options we chose the DAMOV simulator environment [28], a simulator that combines zsim [29] and Ramulator [30] using the HMC 3D-stacked memory model, coupled with a comprehensive benchmark suite compiled by a plethora of data-intensive functions to examine potential benefits by executing them on the PIM. We carry on by comparing the so far proposed workload profiling methods and select the most accurate in terms of efficiency and execution simplicity. We then attempt to simplify some of the suggested profiling methods and keep the process solely in the simulator's environment. By examining prior works and juxtaposing their extracted propositions with the results of the experiments we conducted, we infer on the most precise and easy-to-use metrics that assist to identify the usefulness of NDP. We showcase that two of the most prominent metrics are Arithmetic Intensity (AI) and cache Misses Per Kilo Instructions (MPKI).

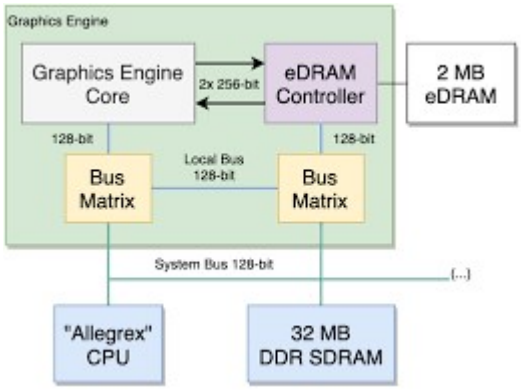


Figure 1 PSP eDRAM

2 Background

2.1 3-Dimensional Integrated Memory Circuits

The term 3D IC refers to the method of vertically stacking integrated circuits for purposes of space conservation and performance improvement. Despite the efforts being made for both computational units and memory modules to conform to this paradigm, the necessity for the memories' development to overcome specific obstacles concerning space scaling and bandwidth improvement made them the focal point of the 3D IC research field. The first technical parameter that needed to be encountered was the means of communication and interconnection among the several vertically stacked layers of the 3D chip. The two salient manufacturing technologies that were explored were recrystallization and wafer bonding [31]. In the course of time, wafer bonding was identified as the prominent one and a specific sub-technique of it, Through-Silicon-Vias (TSVs) became the approach of choice for the key manufacturing companies. TSVs are electrical interconnects made of copper that completely penetrate a silicon wafer in order to support communication between two vertically stacked layers. They are characterized as a high-performance medium, the roots of which can be traced back to 1958 [32, 33].

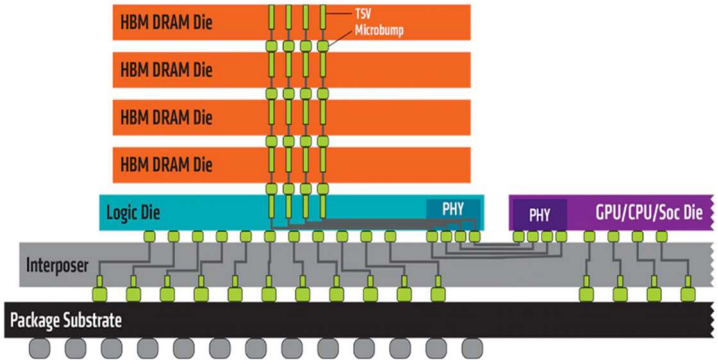


Figure 2 High Bandwidth Memory [21]

The first major corporation to achieve the realization of a 3D IC was Fujitsu in 1983 [34], later followed by Mitsubishi [35]. The aforementioned efforts did not use TSVs as their interconnections technology, which were introduced to the industry in 1981 and were not integrated into the chip's design until 2008 [36]. Intel's 2004 initiative to produce a 3D version of its Pentium 4 resurged the community's interest to stacked chipsets, an interest that later entrained renowned companies like Samsung and Toshiba to the development of commercial 3D Memories such as PSPs eDRAM [37] [Figure 1].

Currently, the two main reference modules in terms of 3D memories are AMD's and Samsung's High Bandwidth Memory (HBM) [Figure 2] and Micron's (later supported by Samsung) Hybrid Memory Cube. HBM implements stacked DRAM dies connected by TSVs to produce a larger amount of bandwidth, up to 665 GB/s [22] by enabling parallel transfer of data to and from the memory dies. This is feasible because HBM hosts two 256-bit memory channels per block of 4 DRAMs, leading to a 4096 bits-wide bus for a standard HBM cube.

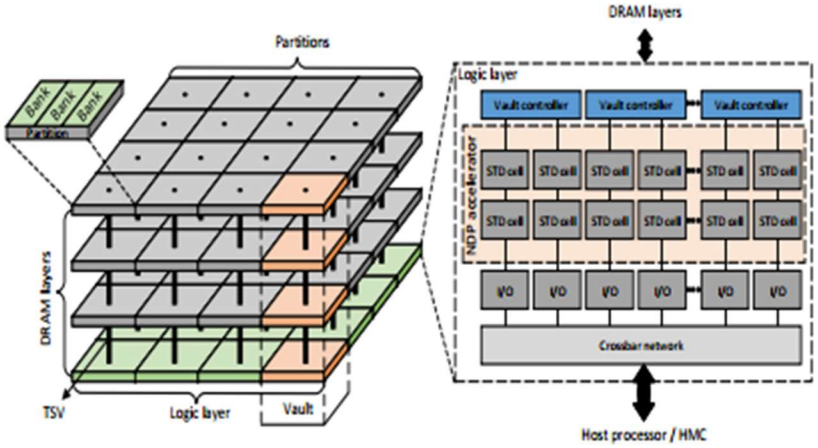


Figure 3 Hybrid Memory Cube [20]

2.2 Hybrid Memory Cube

The rapid development of 3D stacked memories acted as the key for incorporating processing opportunities inside the memory. Having identified the memory wall as the prevailing bottleneck of today's computer systems' performance, the potential of reducing data movement by transferring the computation to the memory rather than the opposite forced the corporations and researchers to the development of chips that integrate logic as well as memory. Hybrid Memory Cube (HMC) [Figure 3] is one of the most popular 3D-stacked memory architectures available. HMC is the result of the collaborative work of Samsung and Micron. It consists of up to eight DRAM dies and one logic die. These vertical layers are connected with each other by Through-Silicon-Vias (TSVs), vertical, high-performance interconnects designed to slice through the silicon die, providing communication to between the layers. Within the cube, memory is vertically divided to partitions named vaults (see picture). Underneath each vault lies a vault controller, a logic unit that operates directly on the data present at the corresponding vault and governs read and write requests to the vault. By developing a dedicated controller for each vault, HMC succeeds to provide independent access to every vault thus achieving parallelism. Referring to as much as 8 logic layers altogether every request can consume data on a vertical axis and reach a maximum bandwidth of 80Gb/s per vault. The increase of the accessible bandwidth compared to modern DRAM technologies manifests that the use of HMC can attenuate the effects of bandwidth induced latency for bandwidth-bound applications when these applications execute on a traditional DRAM module.

Later on, we showcase a number of applications that appertain to this class and the speedup that is achieved when they are offloaded for execution in a PIM that includes HMC as the memory model of choice. One drawback to this independent memory access pattern is that there is no specific memory requests timing and the response can be obtained in a different order than the request was made because every vault executes as an independent memory module. Every request from the Host CPU to the HMC memory module is first directed to the corresponding vault and then stored to a buffer in the vault controller. Every such buffer retains a FIFO protocol, so requests to a specific vault will be executed in order. Generally, though, the vaults generally reorder their internal requests to optimize bandwidths and to reduce average latencies, and the vaults do not communicate with each other so there is no guarantee that the series of requests will be preserved. As a result, the host processor must implement a coherence protocol, same to the network package that enumerate their requests to secure that data is continuous and of the same order as on the protocols request, a matter which poses as a drawback to a general-purpose PIM approach.

Additionally, the HMC specifies its own PIM commands [Figure 4] that implement simple Atomic Instructions, commonly recurring bitwise operations that the logic layer can execute straight on the correlated data. This type of commands has been the area of exploitation for various instruction-level compiler-based PIM proposals because they do not encounter the aforesaid data coherency issues and can be a very fast solution for workloads that make extensive use of simple bitwise commands. Below, we present a table containing HMC's most basic commands

32-byte WRITE request
32-byte READ request
Dual 8-Byte Add Immediate
Single 16-Byte Add Immediate
8-byte increment
AND16
NAND16
NOR16
OR16
XOR16
Compare and Swap if Greater Than
Compare and Swap if Less Than

Figure 4 HMC Basic Commands

3 NDP Trends

3.1 Overview

Since NDP was first introduced as an idea about five decades ago there have been many different methodologies proposed, each of them aiming to address a separate aspect of the processor-centric Von Neumann design that acted as a performance bottleneck. The majority of the most significant of them converge to eliminating every unnecessary data movement among memory and processing components. This way they aim to override the usage of the time and energy intensive memory off-chip link as well as minimize the improper leveraging of the caching hierarchy. In order to study these works we categorize them based on whether they operate directly on the memory cells or use an added logic layer to Process-Near-Memory, based on their range of application to general-purpose or application-specific and finally based on their proposed offloading granularity to kernel offloading or instruction-level offloading.

The term offloading granularity is used to describe the level of insight of the candidate section of code. In instruction-level offloading the system focuses on executing individual commands on the PIM subsystem as opposed to kernel offloading where a whole function is sent to PIM submodule to be executed. The first approach nullifies unnecessary data transfers because only the instruction is sent to PIM where the data consist. The later approach prevails regarding software development simplicity and usage abstraction, as the same hardware and software architecture can support every offloading candidate function.

The following analysis can be considered as a coherent assessment of some of the most significant propositions regarding PIM, nevertheless concentrates on the offloading candidate's selection methodologies which they suggest, examining the issue more from the programmers' standpoint, since it is one of the most constraining concerns in the direction of PIM adoption.

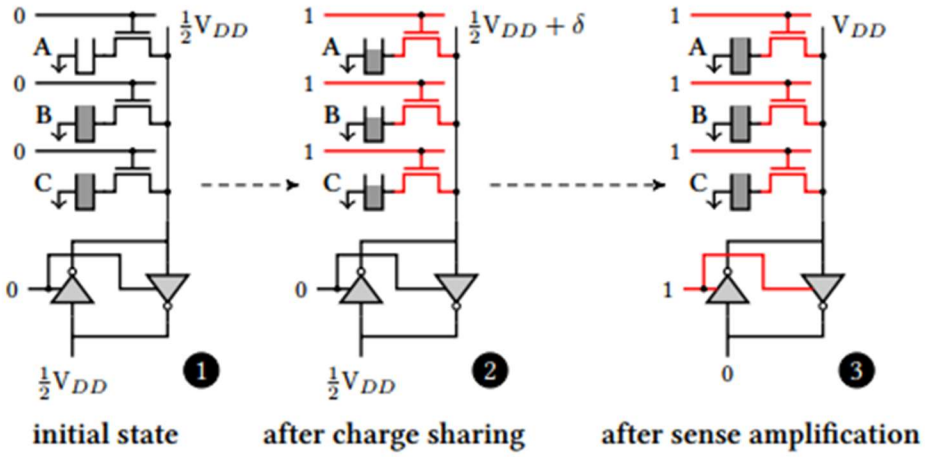


Figure 5 Triple Row Activation [39]

3.2 Bulk In-Memory Operations

Although 3D-stacked memory architectures revolutionized the Processing-In-Memory research field, there have been proposed approaches that attempt to take advantage of existing DRAM architecture and operations to induce simple computation capabilities by implementing minimal changes to the memory chips. Amongst the number of data intensive applications there is a fraction of them that comprises almost exclusively of bulk data movement operations. With the term bulk data movement operations, we refer to these blocks of instructions that require no computation on the processing end, such as batch initialization of a memory block (initializing an array to null) or bulk data copy from a memory address to another.

Investigating the existing DRAM internal organization, we observe that a DRAM chip is divided to multiple DRAM banks. These banks communicate with each other and with the I/O via a shared internal bus that is itself divided to subarrays. Each subarray is responsible for a number of DRAM rows, the columns of which are connected together across the multiple rows using bitlines. RowClone [38] implements a mechanism that issues a row-open request to multiple rows of a subarray and activates the source and the destination row back-to-back. Then the Pipelined Serial Mode transfers a large number of bytes from the source to the destination

row. This way one can manage the initialization of several rows of DRAM to a specific number by reducing the number of requests needed

Ambit [39] is an extension of RowClone that tries to include bitwise operations in its range of application. Using a Triple-Row-Activation, as shown in figure 5, the cells of the first two rows can participate as operand in bitwise majority functions and return the result to the third row. Additionally, the sense amplifiers that are present to the DRAM subarrays are equipped with an inverter. The results of each row's NOT function is captured and stored to a special, newly designed row, ready to be read when a NOT operation is issued. Tested on bitwise operations both RowClone and Ambit achieved a 11.6x speedup and 44x operation throughput respectively compared to traditional DRAM models and the NVIDIA GTX 745 GPU. Despite of their results their applicability is limited to the aforementioned operations and their success relies on carefully carving the requests to fit well with the memory row size, a problem that lies into the programmers or the compilers consideration.

3.3 Instruction Level Offloading

Many of the examinations that were made on workloads which are known to put the memory bandwidth under significant pressure led to the conclusion that the vast majority of the computational instructions are simple Read-Write-Modify operations such as integer addition or equality checks. Also, very little amount of these instructions was related to the data chunks that are being transferred, rendering up to the 90% of the data transfer as nonessential.

A classic example of these patterns are graph manipulation applications which are also applicable to a broad domain. [39, 40]. The support of RWM operations from the HMC via Atomic Instructions made this category of applications the prevailing subject of instruction-offloading PIM research. In this section we review some of the most compelling propositions regarding this category.

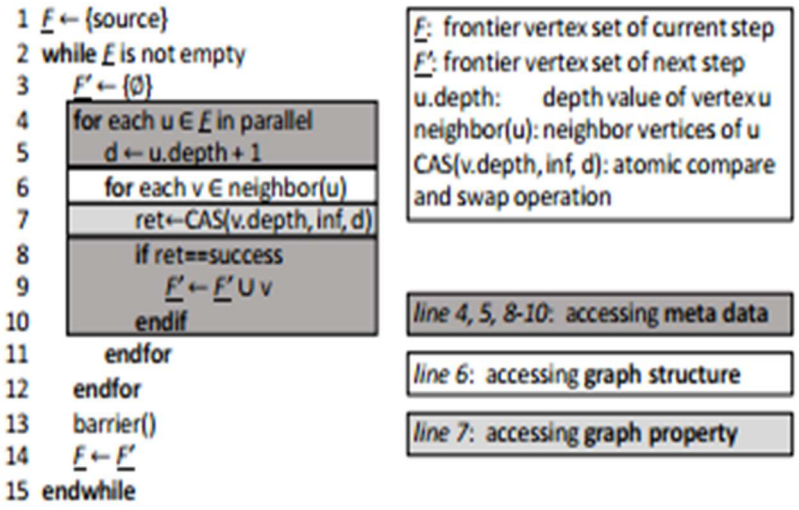


Figure 6 GraphPIM Instruction offloading [41]

3.3.1 GraphPIM

Baseline PIM architectures take advantage of the absence of time-consuming data transfers using a memory bus, because both memory and the PIM logic are on the same chip. Cache misses and data requests from the memory are still present but due to the amplified bandwidth their impact on performance and energy consumption is minified. Graph traversal and computational algorithms are infamous to invoke a significant amount of cache misses because of the random memory access patterns they produce when traversing from one graph vertex to another. Neither spatial nor temporal locality could be guaranteed. Understandably this family of applications are immediately candidates for execution on a PIM.

Graph PIM [41] attempted to enlarge the performance gains by splitting the graph algorithm to three parts. [figure 6] The first one and the third one refers to the vertex's metadata so spatial locality and data presence in cache are presumable. The second one though is related to accessing other nodes and is the one that produces the randomness in the access patterns. Most of the time it consists of simple Read-Modify-Write commands such as comparisons and

additions. GraphPIM implements an instruction-level Offloading mechanism that uses the HMC-Atomic commands to perform these RMW operations directly on the memory and bypass the need for data transfers. This methodology can entrain a 2.5x speedup over the baseline PIM model and a 37% reduction on power consumption. The downside that prevents a wider adoption of this methodology is the lack of exploration of candidate selection and requires from the programmer to analyze the algorithm to identify the RMW commands and limits itself on applications of which the majority of commands are RMW.

3.3.2 CAIRO

CAIRO [42] in essence functions like an extension of GraphPIM. Identifying the underlying programming burden to assess instruction-level offloading suitability, CAIRO work proposed a compile-time mechanism to automatically identify PIM candidate instruction by performing five tests per instruction. The first test ensures that the translation of a block of code produces only RWM commands that themselves translate to HMC Atomic commands. The second test ensures that the selected atomic instruction is included in the atomic instruction set provided by the HMC. The third is a direct access test, meaning that the addresses referenced in the atomic command must be embedded in the instruction code and no register read is required to access a memory location. E.g. lw $2 ($11) fails this test because register $11 that contains the address to load is not accessible from the PIM. The fourth test ensures that the size of the data that we ask the PIM to compute do not exceed 16 bytes. Lastly, a density test establishes that there are several instruction candidates in a block of code so that the effect of PIM computation will not be negligible.

We chose to analyze the tests mentioned above in order to showcase that instruction-level offloading is based on sound mathematical foundations but requires either the programmer to analyze the algorithm or strict offloading conditions to be met, affecting only a small group of instructions.

3.4 Function Offloading

All of these works are very accurate and efficient but rely their speedup results to the application's translation to HMC atomic instructions and so forth being heavily reliant to the memory model's support of these commands. It should be also noted that all of the above are idealized configurations that ignore the factor of maintaining cache coherency.

In order to facilitate NDP offloading from the programmer's standpoint, many efforts concentrate to the creation of tools that will be able to automatically distinguish Processing in Memory candidate code blocks. A way to achieve the identification of such blocks is to run a profiling tool beforehand and configure the PIM execution based on the experiments result. For example, Intel's V-Tune [43] can effectively detect if the application under examination is compute or memory bound and provide results which can help the user identify if it is bandwidth or cache bound. By inspecting those numbers and by combining them with proposed research metrics, one can carry out more effective experiments. An alternative way is via the creation of compiler-based techniques that aim to unify the process of both identifying and executing PIM code blocks and completely dismiss the user – programmer from any extra effort. Albeit, as far as the latter proposition is concerned, extending the compiler's functionality to include these capabilities requires several modifications to be made in the existing hardware which naturally stands as an obstacle to widespread PIM adoption.

Although all proposed techniques converge to the fact that transaction bandwidth should be the main criterion of choice, none of them has been successful to systematically recognize the best fits for Near Data Processing among bandwidth-bound applications. So far application categorization is based on experimental analysis and practical observations of execution results, so many research groups attempt to provide a large domain of tested benchmarks and conclude based on their output.

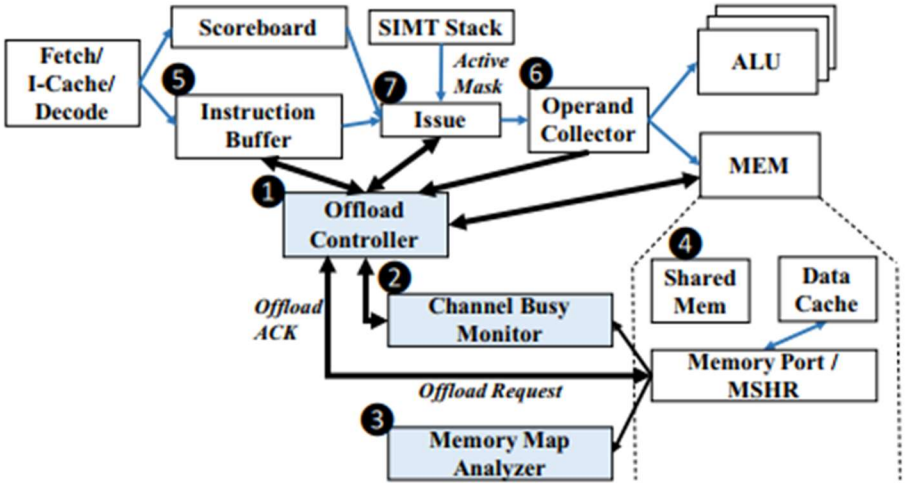


Figure 7 TOM Hardware Selection Support [26]

3.4.1 TOM – Transparent Offloading and Mapping

Until now, due to their potential to handle multiple functions at the same time, Graphic Processing Units (GPUs) have been used as an execution alternative for tasks that support high parallelism and can be divided to small individual tasks. Examples of such tasks are Image Processing, Big Data applications and Machine Learning algorithms. One thing that the above categories have in common is that they are memory bound applications, meaning that they require only small, repeatedly computations to be made in a large portion of data. So far, the restricted off-chip pin bandwidth could not provide enough data portions to the GPU running threads. Combining the GPU cores with emerging 3D-stacked memory technologies by placing GPU processing units on their logic layer can be seen as a promising way to leverage their performance by reducing DRAM access overheads.

Transparent Offloading and Mapping (TOM) [26] attempts to explore the latter proposed architecture of developing compiler-based techniques to offload bandwidth-intensive computations in GPUs. The paper suggests the development of two mechanism. The first one is a compiler extension that can identify loops which have the potential to be memory-bound. After identifying the loop, the decision of whether the block should be executed on a PIM core is made by an algebraic equation that essentially operates by scrutinizing the number of load and

store commands the block will issue to the memory. The actual offloading is performed by a series of added hardware components [figure 7] that utilize the aforementioned run-time information, decide whether the block should be marked as an NDP candidate, and instruct the GPU kernels to wait until the block is executed on a PIM. Additionally, based on the observation that loops generate paternal memory accesses, often with spatial locality, a second mechanism is responsible for copying the memory pages that the code blocks accessed in its first repetitions inside the 3D-stacked memory to minimize the memory access time of these addresses. TOM promises a 1.3x average performance improvement in comparison with baseline GPU models that are not equipped with code offloading functionality.

3.4.2 Damov

To assist the widespread adoption of PIM techniques in modern memory systems, a more general approach of workload identification emerged. Ideally, the methodology should not involve any programmer's knowledge of the underlying offloaded algorithm. Despite the customary reasons that could lead an algorithm to benefit from NDP, such as the aforementioned big workload size, there was no specific understanding of the exact properties of an application that could produce a data movement bottleneck. DAMOV consists a comprehensive analysis in this direction. First of all, the DAMOV team developed the first open-source function-offloading granularity simulator based on Zsim[] and Ramulator. Essentially, what the simulator does is configuring the Ramulator to ignore any latency produced by the memory requests that were initiated from the offloaded code block. A more detailed analysis of the simulator is included in later chapter.

Secondly, the research uses external profiling tools and well-established metrics, such as the roofline model [44] to extract information about an application's dependency on the memory system. The main focus of their work is to methodically understand the reason behind any memory related slowdowns an algorithm may experience. For example, after traversing through a number of processing cores ranging from 4 to 256, they investigate the total cache misses. If this number of misses decreases as the cores count increases, that means that the

application makes a good utilization of the caches but is dependent on the cache size. Similarly, apps that continue to produce a big amount of caches misses are bounded by the main memory's off-chip link bandwidth. They continue accordingly until they end up generating six function categories based on the source of the data transfer bottleneck. Finally, the team provides an extended suite of benchmarks that are well-suited for NDP experiments, a benchmarking suite that we also use in our work.

Despite being essential to identify the exact reason of slowdown, an extensive methodology like this is not necessary if the primary goal is to define if the application can exploit NDP to improve its performance. To this extent, the methodology should be as simple as it gets, using only the minimum number of metrics and classification steps needed.

4 Methodology

4.1 Overview

All the above-mentioned works have attempted to come up with a proposed methodology that determines if a selected workload is suitable for NDP. This decision is usually made by using different profiling tools, the results of which assist to identify the root causes of the data movement bottleneck. Nevertheless, they often use isolated metrics that could potentially be combined to produce an empirical, yet more methodical way to recognize which algorithm can benefit by moving its computation to the memory chip. Another limiting factor of their analysis is that they concentrate on a specific application or an individual category of applications, not allowing for generic and applicable conclusions to be made. There is therefore created a need to constructively categorize applications, so that a programmer can decide on whether the algorithm could benefit from PIM just by identifying in which class it belongs. There are many ways to perform such classifications. Firstly, it can be based on whether the functions in question are compute or memory bound. Then it can be derived from the application's memory access pattern and whether it can manage to effectively exploit the CPU caching mechanisms. Many functions produce irregular access patters, rendering, that way, the

caches hierarchy ineffective. In addition, existing algorithm classifications can be simplified to make the classification process faster and comprehensive.

Prior works have come up with indicative metrics that can be observed as a guide for efficient workload profiling. Such metrics are the Arithmetic Intensity (AI), the Last Level Cache Misses Per Kilo Instruction (MPKI) and the Last-To-First Miss Ratio (LFMR). These metrics, when read in some specific order, can lead to conclusion concerning the effect of NDP of an algorithm. This sequence must be carefully fabricated to reduce any possible inaccuracy each metric produces when used by itself. It also can be read in a way that every metric assigns every application in a category, resulting in distinctive classes that a programmer can rely on to make approximations on the behavior of the relying algorithm and it's NDP suitability. For example, we expect functions with low AI to benefit from the NDP system. Equivalently, a higher MPKI acts as indicator that the application puts a lot of pressure to the memory subsystem, and we expect a speedup when functions with high MPKI are offloaded to the PIM unit.

The goal of our methodology is first of all to accurately extract the above-mentioned metrics for every algorithm under examination, and then to provide a combination of these metric that is able to produce exact classes of functions with similar behavior when offloaded to PIM. The methodology we developed can be broken down to three major steps. The first step is the profiling of each application via the extraction of its key metrics. This is done by feeding the source code of the function accompanied with the corresponding workload to the simulator and subsequently filtering the resulting log files through our python facilitation scripts. These scripts are carved in a way to automatically output all needed metrics for every different configuration of the environment such as the number of cores or the distinct workloads applied. The second step is the independent analysis of the AI and MPKI metrics in order to conclude on whether they can serve as sufficient indicators for the usefulness of the PIM execution for the function under consideration. As the analysis of the second step suggests that certain applications do not comply with these assumptions, we propose a two-step workload classification method which relies on dividing the application firstly by their MPKI metric and then by their AI. Although classification techniques have also been suggested in the past, we find our two-step approach to provide a sufficient trade-off between accuracy and simplicity. Its

simplicity can be a key factor for a future transcription of this work as a real-time methodology that can be executed by the compiler, further alleviating any programmer's involvement.

In our evaluation we make a considerate selection of illustrative algorithms for every classification category in order to allow the generalization of our results and to verify the conclusions emerging at previous studies. We provide scripts and techniques that facilitate the user to perform the whole analysis inside the simulator's environment, without the need of installing and deploying external profiling tools. We, finally, select a careful sequence of the aforementioned metrics' evaluation, based on which a programmer can determine a priori and with a narrow error percentage the effectiveness of the NDP execution of the application in question.

Specifically, we observed that all the applications that produced an MPKI value that surpassed a threshold value went on to achieve a performance improvement from the NDP. For the experiments conducted, this threshold was calculated to the value of 10. While correctly provisioning for these applications, MPKI failed to be consistent for the functions that lay under the threshold value. Eventually, applying the AI metric for this group of algorithms resulted in a sound classification of these apps. All low MPKI apps that generated an AI more than 40, performed better on the host platform than on the PIM. In conclusion, this two-step evaluation creates three groups of functions with predictable outcome as far as NDP speedup is concerned.

The experiments were conducted on a broad set of application, that can serve for the generalization of the results and the easier adoption of the proposed classification. The DAMOV benchmarking suite consists of a representative collection of applications from various popular domains such as graph processing […], machine learning […], databases […], vector arithmetic and other commonly used workloads.

4.2 Proposed Metrics

4.2.1 Arithmetic Intensity (AI)

The metric of arithmetic intensity indicates whether the application under examination is considered compute-bound or memory-bound. For an application to be considered compute-bound, the total time consumed performing computations must outweigh the time spent for data transfers by a significant amount. Approaching the issue from another point of view, a certain portion of data that is copied from the main memory to the cache must be exploited and reused for several computations. Consequently, it is safe to define Arithmetic Intensity as the quotient of instructions a CPU performs divided by the total bytes accessed in the main memory. Successfully performing a number of operations without the need to issue a memory request implies a compute-bound function, and, conversely, issuing many requests for a small number of instructions indicates a memory-bound function. A less error-prone definition should suggest replacing the denominator by the number of bytes accessed per cache line, but since we are trying to unify the process of evaluating the metrics, we adopt the simpler evaluation that is also facilitated by the simulator. Predictably, we expect an application with high compute intensity to not suffer from severe data movement bottlenecks, as demonstrated by prior work [45].

This metric can be used by its own from the programmer to predict the algorithm's suitability for NDP execution with a satisfactory accuracy. During our evaluation we came across functions that do not oblige completely to the aforementioned criterion, but the majority of the applications that were characterized as memory-bound went on to achieve speedups while on an NPD environment. Some of the various potential sources of memory boundedness are cache misses, cache coherence traffic, and long queuing latencies. Due to the existence of functions that do not appertain to this rule, a need for complementary metrics emerges

4.2.2 Last Level Cache Misses Per Kilo Instruction

In order for a memory request to be directed to the main memory, the required data must not reside in the last level cache. Only after a last level cache miss the CPU initiates a transaction

to the memory. Last Level Cache Misses Per Kilo Instructions (LLC-MPKI) is used as an indicator [2, 16, 46, 47] of an application's dependance from the memory system. A high MPKI value can either mean that the app produces accesses to memory addresses that are far from one another, making it harder for the system to collect all the data needed inside the cache hierarchy, or that the app operates on a significant amount of data, the size of which exceeds the caches capacity by a lot. LLCMPKI is, accordingly, proportional to the applications memory-boundness and a sufficient estimation of an algorithms data locality.

As its name suggests, MPKI can be derived as the quotient of the number of the cache misses occurred during the execution divided by the average instructions among all processing cores of the system. Practically, the log file of zsim includes both statistics, so a carefully developed python script can be enough to extract the metric. Since last-level cache, in the present case the L3, is shared among all processing units, inspecting the L3 cache is more architecturally independent than focusing on L1 or L2 caches that scale accordingly to the number of CPU cores.

5 Evaluation

5.1 DAMOV Simulator

As stated before, the real hardware for HMC is not commercially available. In order to verify the aforementioned theoretical estimations and come up to our conclusions, we perform software simulation. To simulate the host, as well as the NDP processing cores we use zsim, a multicore x86 open-source simulator. Zsim is configured through a .cfg configuration file that determines the needed execution parameters and generates memory trace files. Each line of these files represents a memory request to a specific memory address. These trace files are later fed to Ramulator, a cycle-accurate DRAM simulator that supports a wide range of commercial or even academic memory standards, such as DDR4 and HMC. Coupled with an extensive benchmarking suite and expanded to support software instrumentation, the package is called

DAMOV simulator. Software Instrumentation is the process of adding pre-declared hooks inside the code that denote that a specific sector of code most be treated differently. For example, inside the DAMOV the hooks are used before and after a function to declare that its execution will be offloaded to PIM. The full tutorial is included in the Appendix section.

In order to simulate the host system ZSim is modified to produce filtered memory traces. This means that Ramulator will be aware only of the requests that reached the memory controller, ignoring the requests that were served by the systems cache, thus inside the ZSim. For the PIM system, unfiltered traces are obtained. Consequently, when the host's pipeline issues a request, it is directly fed to the main memory, with the difference that Ramulator is specifically instructed to use the HMC memory model and ignore the overhead metrics that are related to the off-chip link. That way it mimics that the processing unit lies directly underneath the main memory module. As far as software instrumentation is concerned, while in PIM mode, any memory requests that originate from the section of code that is denoted for NDP, are directly fed to the Ramulator.

It is clear that the selected software simulation environment has several advantages, with the most prevailing of them being its simplicity. ZSim and Ramulator are easily configurable regarding the caches size, hierarchy, supported memory models, model of CPU execution (in-order, out-of-order) and other top-level options. Nevertheless, assuming that just by subtracting any latency induced by the off-chip links constitutes an efficient approximation, fails to cope with significant issues such as memory coherence. In reality, the vaults of the HMC are not equipped by any internal communication mechanism, so coherence must be taken care of from the host CPU, as it is already been said. Implying that there is no memory bus shrinks the problem down to essentially just a bandwidth problem, which is generally true, but to a smaller extent.

5.1.1 Simulator Architecture

In order to evaluate the performance of our experiments we have configured our simulator to resemble a Host CPU with private L1 (32 kB) and L2(256 kB) caches and a shared

L3 (8 MB) cache hierarchy. In the NDP configuration, the processing core is only equipped with an L1 cache that serves as a buffer for the main memory.

Although current computer systems are equipped with classical 2-D DRAMs, in order to narrow down our focus to only the data movement aspect of the execution, we oblige to prior works that suggest keeping every other system component the same. To that extent, even the Host CPU model is equipped with a 3-D HMC main memory module. In order to compare possible PIM execution to the todays traditional Van Neumann architecture, one could instruct Ramulator to use the DRAM model instead.

Based on previous studies [27, 49], we do not perform extensive scalability analysis as far as the number of processing cores is concerned, as it is shown that you need at least 64 cores to fully exploit the bandwidth provided by the HMC. Hence, we vary the core count between 64 and 256. Except for specific cases, the majority of the applications behave in a similar way when increasing the number of the processing units, so we only present the exceptions in this work.

5.2 Results

5.2.1 Retrieving the Metrics

The goal of our work is to provide a way for a unified analysis that can be contained exclusively inside the environment of the simulator. To this end we combine the statistics files from both zsim and Ramulator with our python scripts. To retrieve the AI we use the get_ai.py script. Inside every zsim.out file there is a line that describes the number of instructions executed at every CPU core. Our python program at first takes the zsim.stat file of every experiment as an input and returns the average number of instructions executed in all processing cores. The error produced by the assumption of perfect parallelism that is induced from calculating an average value compensates for the alleviation of the need for external software. Without loss of generality, the same metric can be derived by taking the total instructions executed in all processing cores as the denominator. The difference that it would produce is the extraction of a

different threshold value for AI. Then, the same script reads the Ramulator's output file to calculate the total bytes read from the main memory. Lastly it determines the AI as the ratio of the instructions' average divided by the total memory transaction bytes issued to Ramulator

For the extraction of the MPKI metric, the DAMOV simulator has already provided the get_stats_per_app.py script. The program requires exclusively the zsim output to determine the number of L3 cache misses and the divide this number with the average instructions performed by the host CPU. In the Appendix section, we present some modifications we made to the source script in order to support burst execution of experiments with various configuration options such as platform (host, pim), processing cores number or workload size.

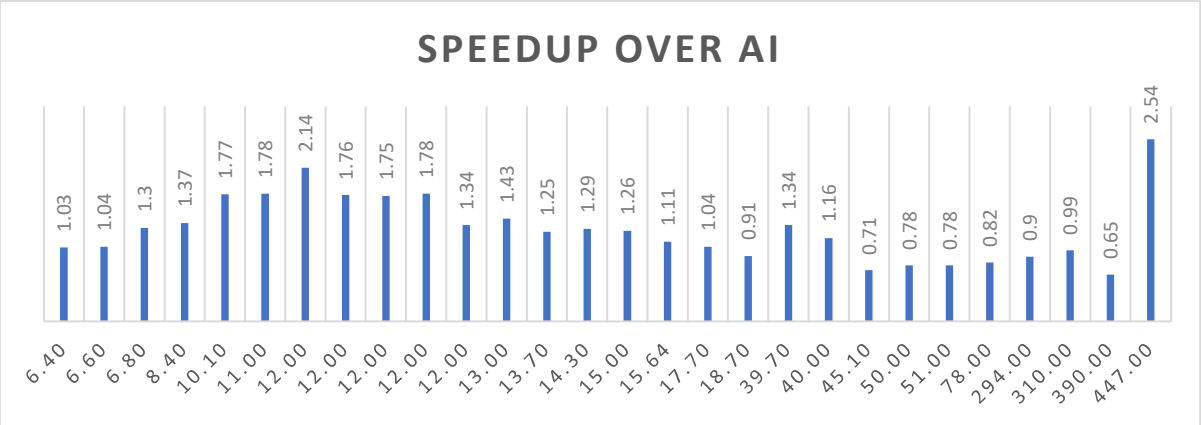


Figure 8 Speedup Over AI

5.2.2 Arithmetic Intensity Analysis

In order to conclude on whether the metric of Arithmetic Intensity is a suitable one, we extract its value for 28 representative applications [Figure 8]. As stated before, we should observe that functions with low AI could benefit from their PIM offloading resulting to speedup. After asserting this issue, we should search for a threshold value that will act as an indicator for the programmer to decide upon. A first observation that can be made is that there is no proportionality between the AI value and the resulted speedup from the NDP. This divergence on the results hints to ser to understand that there is another characteristic metric that defines

the result. The highest average speedup is calculated to be x1.83 in the area between 10.1 and 12 of AI [Figure 9]. If we extend the area of our focus until we reach the first slow down, the apps with AI between 6.8 and 15 result to an average speedup of 1.55.

A promising observation to the direction of our research is that all functions that are subjected to a performance slow down are concentrated in the area between 45.10 and 390. There are two exceptions in opposite directions. The Bezier Kernel calculation performs better on the host CPU, but is not included in the aforementioned area. Additionally, the Padding algorithm that holds the highest speedup calculated at x2.54, also holds the highest AI value (447).

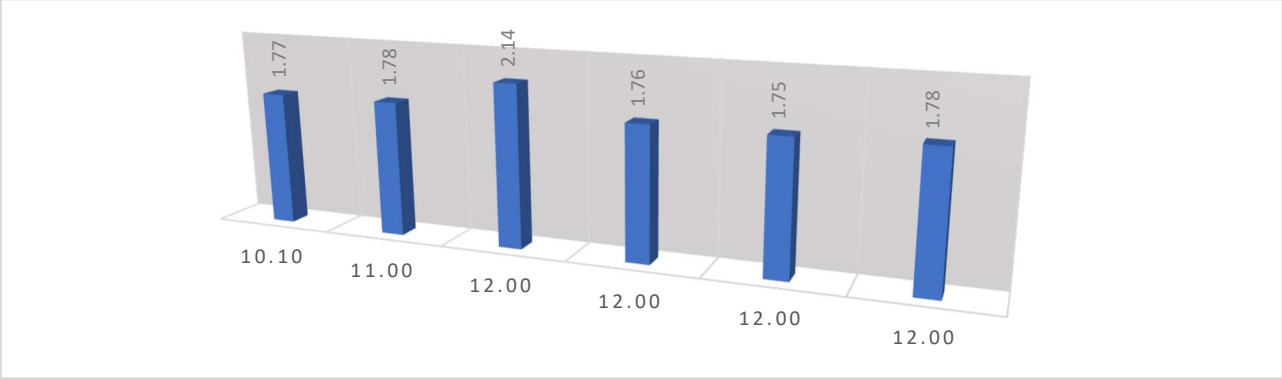


Figure 9 AI area with highest speedup average

AI Group	Average Speedup
AI < 40	1.41
AI > 40	0.81

Figure 10 Average Speedup based on AI Threshold

That being said, an AI value of 40 segregates the functions into two groups [Figure 10]. The first group consists of all apps with AI equal or less than 40. Inside this group only one

function does not reduce its execution time when offloaded to PIM (x0.91). Inside the second group that contains the functions with AI greater than 40, all applications perform better on the host platform except for one (x2.54). In conclusion, the AI value of 40 manages to classify the apps with a 92% efficiency.

Although being an adequate percentage, that 92% could be improved if we include this AI threshold analysis as a part of our two-step categorization method that we later present.

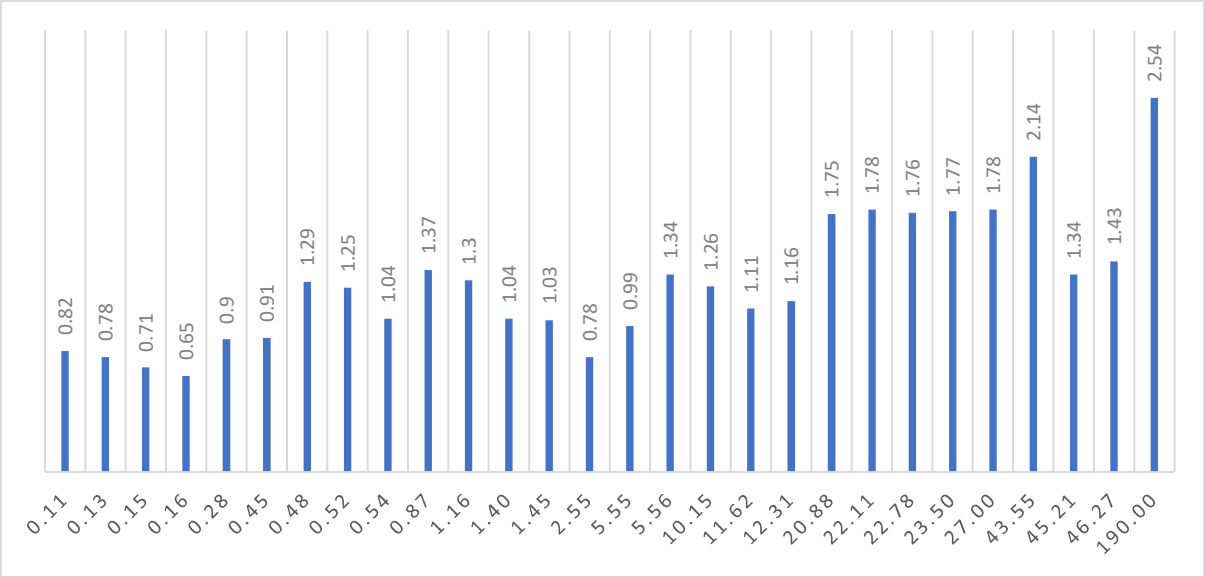


Figure 11 Speedup Over MPKI

5.2.3 MPKI analysis

Our previous analysis stipulates that MPKI is potentially a more accurate metric than AI, as prior works have shown that every application that results in a high MPKI value can improve its performance if offloaded to PIM. After simulating the same 28 apps as above [Figure 11], the output can verify this theory for a threshold value of 10. Hence, all the functions with an MPKI value greater than 10 show execution time enhancements when they are executed in the PIM. For our experiments, the average speedup these applications deliver is x1.76

Nevertheless, the metric is incapable of showing the existence of functions that can benefit from the NDP even though they have a smaller MPKI result. We observe a random speedup pattern for the applications that resulted in an MPKI value less than 10. Using the knowledge of the underlying algorithm, we can notice that many of the apps that can benefit from the PIM despite their small MPKI value perform only trivial algebraic calculations (e.g. triple matrix multiplication). These observations lead us to assume that MPKI can be coupled with AI to provide a more precise characterization method.

For example, in our experiments they are three database MapReduce algorithms. The nature of these applications suggests that although they utilize the caches in an effective manner, the increased size of their workload require a significant number of requests to be issued to the main memory. The bandwidth provided by the HMC (39.7 GB for Word_Count , 43.7 for Linear_Regression) can alleviate the costs of the necessary data movements.

5.2.4 Applications Classification

Taking the above metric analysis into account, we observe that a sequential deployment of both could produce an accurate technique to determine the NDP suitability of a workload. Exploiting the fact that MPKI is accurate for applications that reside above the threshold value, we firstly divide the applications into two groups. The *group A* [Figure 12] consists of all the functions that produce a speedup as a result of their high MPKI value. In the present case, group A includes 12 applications with an average speedup of x1.76.

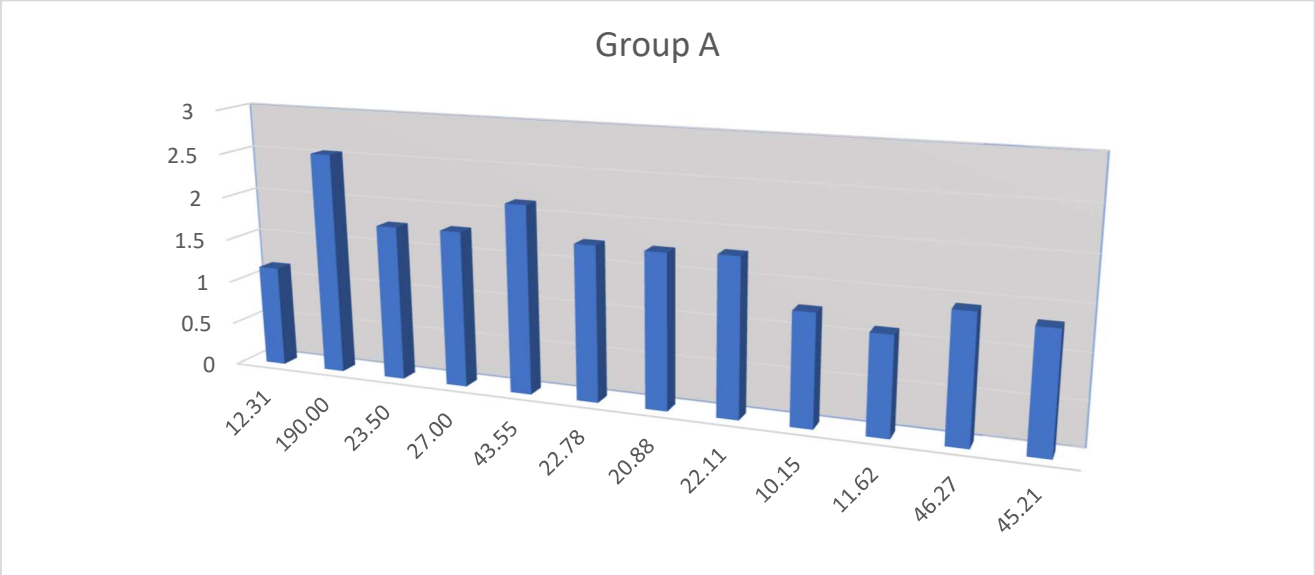
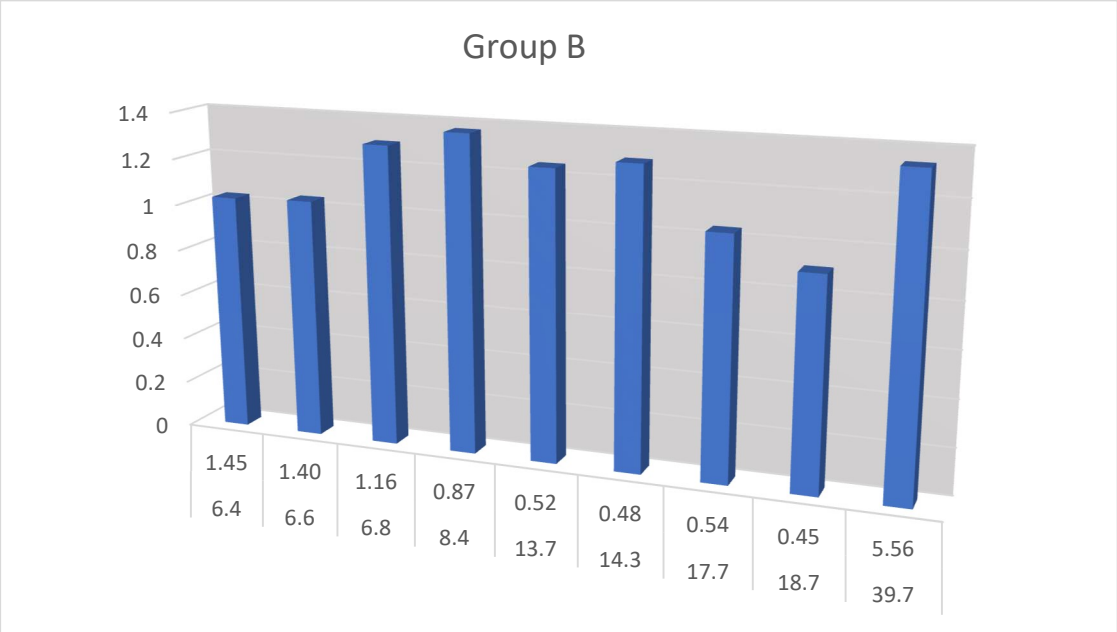


Figure 12 Group A applications

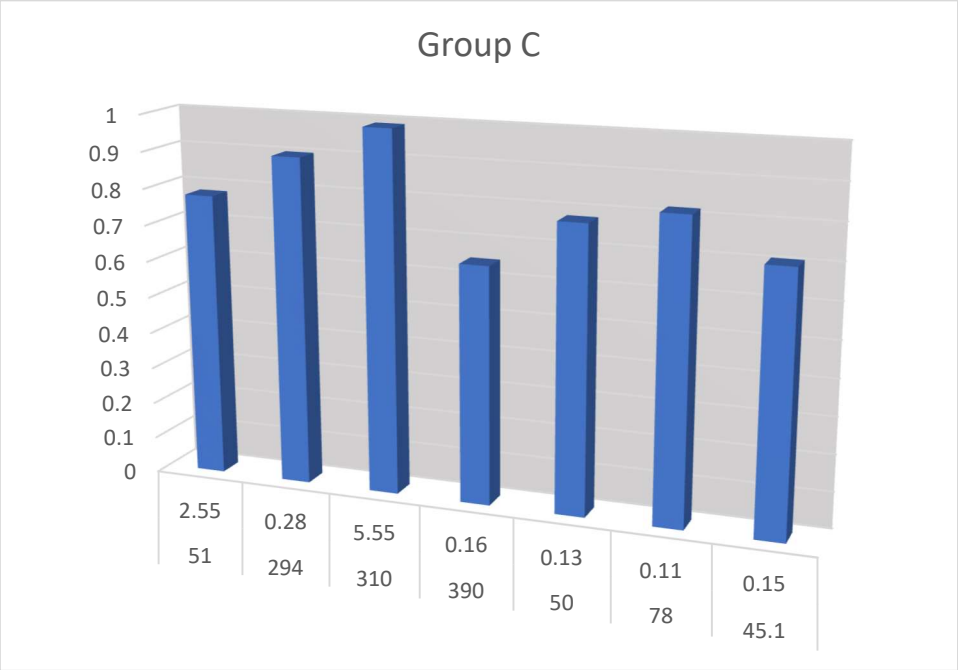
We then apply the AI metric filtering to the *group B* which contains the applications that the MPKI failed to accurately address. Among the low MPKI apps, the AI succeeds to address the potential performance improvements dividing the application into *B* and *C* groups. B functions can make good use of the PIM due to their memory boundness. In B group we classify the functions that produce MPKI < 10 but AI < 40 [Figure 13]. Their average speedup value is x1.20, less than the average value of the A group, as expected.



Group B	MPKI	AI	Speedup	Group
polybench_linear-algebra_3mm	1.45	6.4	1.03	B
polybench_linear-algebra_gemm	1.4	6.6	1.04	B
polybench_linear-algebra_gemver	1.16	6.8	1.3	B
polybench_stencil_convolution-2d	0.87	8.4	1.37	B
phoenix_WordCount_main	0.52	13.7	1.25	B
phoenix_Linearregression	0.48	14.3	1.29	B
polybench_linear-algebra_doitgen	0.54	17.7	1.04	B
chai_BS_BEZIER_KERNEL	0.45	18.7	0.91	B
phoenix_PCA_main	5.56	39.7	1.34	B

Figure 13 Group B applications

The compute bound applications of the remaining group C [Figure 14] are more suited for execution on the Host CPU. In this last group we classify the apps with MPKI < 10 and also AI > 40. The average speed up for the functions of this group is x0.804.



Group C	MPKI	AI	Speedup
chai_BFS_BFS	2.55	51	0.78
phoenix_Kmeans	0.28	294	0.9
chai_TRNS_CPU	5.55	310	0.99
ligra_BC_edgeMapSparseRmat	0.16	390	0.65
hpcg_HPCG_ComputeProlongation	0.13	50	0.78
hpcg_HPCG_ComputeSYMGS	0.11	78	0.82
hpcg_HPCG_ComputeRestriction	0.15	45.1	0.71

Figure 14 Group C applications

The resulting classification diagram is as follows

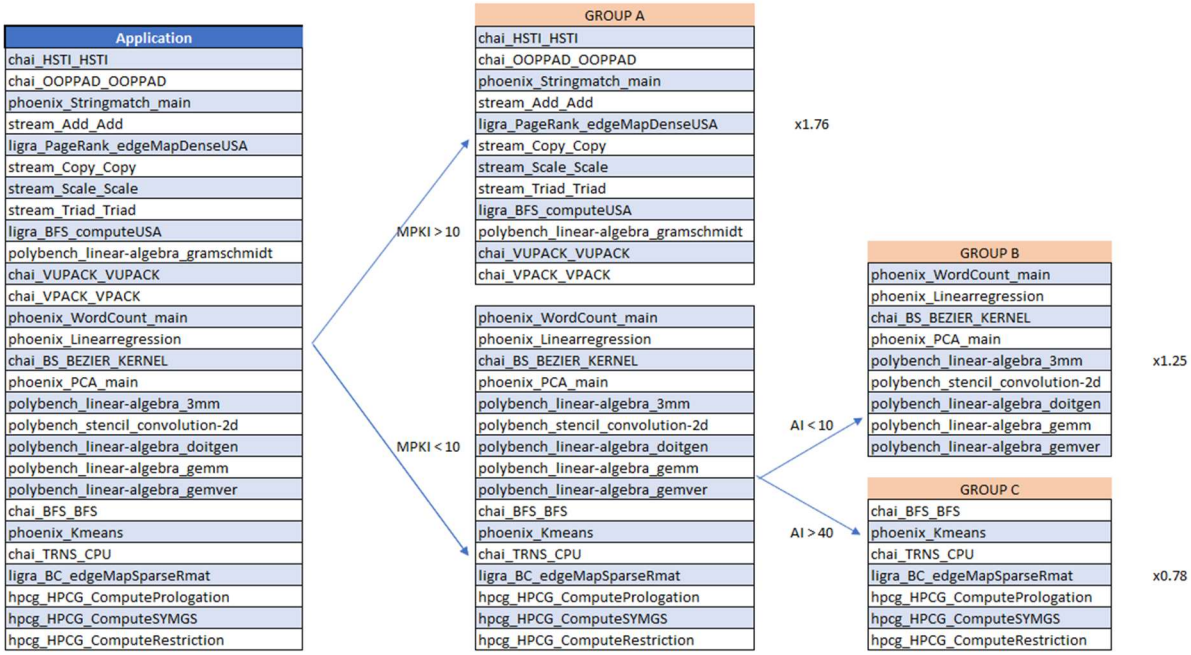


Figure 15 Classification Diagram

6 Conclusions and Future Work

In this work we deducted a comprehensive review analysis of the most prevalent categories in the domain of NDP research. We presented the range of application for every one and highlighted their advantages and disadvantages. In order to support the unrestricted research about the adoption of PIM, even though its commercialization is not on a desirable level, we narrowed down our focus to software simulation using the DAMOV simulator. For the facilitation of the programmer, we concentrated our efforts to function-offloading NDP approaches with no knowledge of the underlying algorithm. In this direction we presented a simplified, yet accurate two-step classification method that is based on two prevalent metrics and successfully divides applications based on their NDP execution speedup. These metrics are AI and LLC MPKI.

The future of NDP will eventually deal with significant challenges, such as cache coherence. But based on this thesis, we could point out two main direction that can shape future research. We believe that the future of NDP exploration should highlight whether the generic approach of function-offloading will be beneficial enough to prevail against the faster but confined instruction-offloading. We also propose that the simplicity of this analysis could be a good starting point for the development of a real-time compiler-based offloading technique.

7 Appendix

7.1 Installation

Due to the several and fragile dependencies of the project, it is recommended to install the simulator on a clean image. We tested the installation process using a Virtual Machine and we verified that it works but takes its toll performance wise. Also, after several trial-and-error cycles, we strongly recommend to use an Ubuntu 18.04 image. The DAMOV simulator team has lately provided a script that takes care of the packages' versioning but is based on the Ubuntu 18.04 environment. To install the simulator you should navigate inside the scripts folder and run ./setup.sh and ./compile.sh. If the installation produces any errors referring to the compilers version, or the version of the Python installed, please configure the gcc alternatives to use a gcc-6+ compiler as follows:

- Remove the current gcc alternative priorities
    - sudo update-alternatives --remove-all gcc
    - sudo update-alternatives --remove-all g++

- Install the gcc version
    - sudo apt-get install gcc-6.* g++-6.*

- Update the alternatives
    - sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-6.* 10
    - sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-6.* 10
    - sudo update-alternatives --install /usr/bin/cc cc /usr/bin/gcc 30
    - sudo update-alternatives --set cc /usr/bin/gcc
    - sudo update-alternatives --install /usr/bin/c++ c++ /usr/bin/g++ 30
    - sudo update-alternatives --set c++ /usr/bin/g++

- Configure the alternatives

- sudo update-alternatives --config gcc
- sudo update-alternatives --config g++

After successfully configuring both the compiler and the python versions, you can follow the commands provided at the simulators repository [https://github.com/CMU-SAFARI/DAMOV].

a.

7.2 Tutorial

The simulator's suite provides plenty of workload benchmarks carefully carved to fit the simulation environment of the PIM capabilities that were added to the simulator. Nevertheless one can create its own benchmark written in C or C++ to conduct specific experiments. In order for zsim to understand which code segment is under examination, the special zsim hooks must be included. A simple demo of a trivial application that just initializes an array inside the memory is presented below in figure 16. After the successful compilation of the workload the user must produce the configuration file for the simulator.

In order to simultaneously generate multiple files for various configurations, that are also organized according to the simulators file system, a python script is provided as shown in the figure 17 below. By alternating the array 'number_of_cores' the user can specify the experiments scalability. After line 104, by manually commenting out the commands that correspond to any unnecessary CPU configurations, you can exclude them from the simulation. The functions in-between are responsible to create the suitable file system for the configuration files.

The next step is running the experiment for both the Host and the PIM platform and compare the output stats. The commands are provided below in figures 18 and 19 respectivelly. In order to decrease the idle time and automate the process, we created two python scripts. The first script is '*run_and_get.py*' [Figure 20] that unifies the process of running and reading the output of an experiment. The second is the '*batch_run.py*' [Figure 21] that instructs the previous script to execute based on the combinatorics of choice, such as the number of cores, the processing platform and the specific workload.

```c
// Print numbers from 1 to 10
#include <stdio.h>
#include "../zsim_hooks.h"

int main() {
    int i;
    int arr[1000000];

    zsim_begin();
    for (i = 0; i < 1000000; ++i) {
        arr[i] = i;
    };

    for (i = 0; i < 1000000; ++i) {
        arr[i]++ ;
    };
    zsim_end();
    return 0;
}
```

```
cecild@[redacted]kpap ~/DAMOV/workloads/test_workload 09:09:37 0
> gcc -Wall my_workload.c -o my_workload
```

Figure 16 Create & Compile Custom Workload

```python
import sys
import os
import errno

os.chdir("../workloads")
PIM_ROOT = os.getcwd() +"/"
os.chdir("../simulator")
ROOT = os.getcwd() +"/"

def mkdir_p(directory):
    try:
        os.makedirs(directory)
    except OSError as exc:  # Python >2.5
        if exc.errno == errno.EEXIST and os.path.isdir(directory):
            pass
        else:
            raise

def create_host_configs_no_prefetch(benchmark, application, function, command, version):
    number_of_cores = [1, 4, 16, 64, 256]

    for cores in number_of_cores:
        mkdir_p(ROOT+"config_files/host_"+version+"/no_prefetch/"+benchmark+"/"+str(cores)+"/")

    for cores in number_of_cores:
        mkdir_p(ROOT+"zsim_stats/host_"+version+"/no_prefetch/"+str(cores)+"/")

    for cores in number_of_cores:
        with open(ROOT+"templates/template_host_"+version+".cfg", "r") as ins:
            config_file = open(ROOT+"config_files/host_"+version+"/no_prefetch/"+benchmark+"/"+str(cores)+"/"+application+"_"+function+".cfg","w")
            for line in ins:
                line = line.replace("NUMBER_CORES", str(cores))
                line = line.replace("STATS_PATH", "zsim_stats/host_"+version+"/no_prefetch/"+str(cores)+"/"+benchmark+"_"+application+"_"+function)
                line = line.replace("COMMAND_STRING", "\"" + command + "\";")
                line = line.replace("THREADS", str(cores))
                line = line.replace("PIM_ROOT",PIM_ROOT)

                config_file.write(line)
            config_file.close()
        ins.close()

def create_host_configs_prefetch(benchmark, application, function, command, version):
    number_of_cores = [1, 4, 16, 64, 256]

    for cores in number_of_cores:
        mkdir_p(ROOT+"config_files/host_"+version+"/prefetch/"+benchmark+"/"+str(cores)+"/")

    for cores in number_of_cores:
        mkdir_p(ROOT+"zsim_stats/host_"+version+"/prefetch/"+str(cores)+"/")

    for cores in number_of_cores:
        with open(ROOT+"templates/template_host_prefetch_"+version+".cfg", "r") as ins:
            config_file = open(ROOT+"config_files/host_"+version+"/prefetch/"+benchmark+"/"+str(cores)+"/"+application+"_"+function+".cfg","w")
            for line in ins:
                line = line.replace("NUMBER_CORES", str(cores))
                line = line.replace("STATS_PATH", "zsim_stats/host_"+version+"/prefetch/"+str(cores)+"/"+benchmark+"_"+application+"_"+function)
                line = line.replace("COMMAND_STRING", "\"" + command + "\";")
                line = line.replace("THREADS", str(cores))
                line = line.replace("PIM_ROOT",PIM_ROOT)

                config_file.write(line)
            config_file.close()
        ins.close()

def create_pim_configs(benchmark, application, function, command, version):
    number_of_cores = [1, 4, 16, 64, 256]

    for cores in number_of_cores:
        mkdir_p(ROOT+"config_files/pim_"+version+"/"+benchmark+"/"+str(cores)+"/")

    for cores in number_of_cores:
        mkdir_p(ROOT+"zsim_stats/pim_"+version+"/"+str(cores)+"/")

    for cores in number_of_cores:
        with open(ROOT+"templates/template_pim_"+version+".cfg", "r") as ins:
            config_file = open(ROOT+"config_files/pim_"+version+"/"+benchmark+"/"+str(cores)+"/"+application+"_"+function+".cfg","w")
            for line in ins:
                line = line.replace("NUMBER_CORES", str(cores))
                line = line.replace("STATS_PATH", "zsim_stats/pim_"+version+"/"+str(cores)+"/"+benchmark+"_"+application+"_"+function)
                line = line.replace("COMMAND_STRING", "\"" + command + "\";")
                line = line.replace("THREADS", str(cores))
                line = line.replace("PIM_ROOT",PIM_ROOT)

                config_file.write(line)
            config_file.close()
        ins.close()


```

```python
if(len(sys.argv) < 2):
    print "Usage python generate_config_files.py command_file"
    print "command_file: benckmark,applicationm,function,command"
    exit(1)

with open(sys.argv[1], "r") as command_file:
    for line in command_file:
        line = line.split(",")
        benchmark = line[0]
        application = line[1]
        function = line[2]
        command = line[3]
        print line
        command = command.replace('\n','')

        ### Fixed LLC Size
        create_host_configs_no_prefetch(benchmark, application, function, command, "inorder")
        create_host_configs_prefetch(benchmark, application, function, command, "inorder")
        create_pim_configs(benchmark, application, function, command,"inorder")

        ### Fixed LLC Size
        create_host_configs_no_prefetch(benchmark, application, function, command, "ooo")
        create_host_configs_prefetch(benchmark, application, function, command, "ooo")
        create_pim_configs(benchmark, application, function, command,"ooo")

        create_host_configs_no_prefetch(benchmark, application, function, command, "accelerator")
        create_host_configs_prefetch(benchmark, application, function, command, "accelerator")
        create_pim_configs(benchmark, application, function, command,"accelerator")


```

```
cecild@[redacted]kpap ~/DAMOV/simulator 09:19:58 0
> python scripts/my_test_generate_config_files.py command_files/my_workload_cf
```

Figure 17 Configuration File Generation

```
cecild@[redacted]kpap ~/DAMOV/simulator 09:27:28 130
> ./build/opt/zsim config_files/host_inorder/no_prefetch/my_workload/256/my_workload_my_workload.cfg
```

```
cecild@[redacted]kpap ~/DAMOV/simulator 09:27:28 130
> python scripts/get_stats_per_app.py zsim_stats/host_inorder/no_prefetch/64/my_workload_my_workload_my_
workload.zsim.out
```

```
 ------------------ Summary -------
Instructions: 625189008
Cycles: 634335262
IPC: 0.985581356504
L3 Miss Rate (%): 50.6108161072
L2 Miss Rate (%): 99.8423279267
L1 Miss Rate (%): 0.381357690347
L3 MPKI: 0.102447418589
LFMR: 0.505704562861
```

Figure 18 Run & Read Stats from host

```
cecild@kpap ~/DAMOV/simulator 09:32:29 0
> python scripts/get_stats_per_app.py zsim_stats/pim_inorder/64/my_workload_my_workload_my_workload.zsim
.out
```

```
 ------------------ Summary ------
Instructions: 625189164
Cycles: 630885209
IPC: 0.990971344836
L3 Miss Rate (%): 0.0
L2 Miss Rate (%): 0.0
L1 Miss Rate (%): 0.433666967636
L3 MPKI: 0.0
LFMR: 0.0
```

Figure 19 Run & Read Stats from PIM

```python
import os
import sys

str = sys.argv[1].split('/')
str = str[:-1] + str[-1].split('.')

print str
if 'pim' in sys.argv[1]:
    statstr = str[1] + '/' + str[3] + '/' + str[2] + '_' + str[4] + '.zsim.out'
else:
    statstr = str[1] + '/' + str[2] + '/' + str[4] + '/' + str[3] + '_' + str[5] + '.zsim.out'

buildcommand = './build/opt/zsim ' + sys.argv[1]
statcommand = 'python scripts/get_stats_per_app.py zsim_stats/' + statstr

#print buildcommand
#print statcommand
os.system(buildcommand)
os.system(statcommand)

```

Figure 20 Run & Get

```python
import os
import sys

str = sys.argv[1].split('/')
str = str[:-1] + str[-1].split('.')

print str

#python batchrun.py config_files/host_ooo_large/no_prefetch/ligra/64/BFS_computeUSA.cfg
bench = str[3]
config_file = str[5] + '.cfg'
#ben_size = ['small', 'large']
cores = ['64', '256']


for c in cores:
    print('config_files/host_inorder/no_prefetch/'+bench+'/'+c+'/'+config_file)
    os.system('python runandget.py config_files/host_inorder/no_prefetch/'+bench+'/'+c+'/'+config_file)
    print('config_files/pim_inorder/'+bench+'/'+c+'/'+config_file)
    os.system('python runandget.py config_files/pim_inorder/'+bench+'/'+c+'/'+config_file)
```

Figure 21 Batch Run

7.3 Configuration

In this section we examine the configuration file [Figure 22] to inspect to what extend we can alternate the simulators configuration. The first sector of the script is about the CPU cores. The most significant change we can make is modifying the number of cores. As far as the caches are concerned, you can change the size, the associativity, and the latency in cycles. It is very simple to change the memory model, among the available models inside Ramulator. In the 'sim' section the user can specify the path for the output file. The max total instructions number can also be modified to increase or decrease the simulation time. The selected instructions count is calculated to be the best trade-off between speed and credibility. Finally, the user selects the command that points to the workloads executable.

```
sys = {
    lineSize = 64;
    frequency = 2400;

    cores = {
        core = {
            type = "Timing";
            cores = 64;
            icache = "l1i";
            dcache = "l1d";
        };
    };

    caches = {
        l1d = {
            caches = 64;
            size = 32768;
            array = {
                type = "SetAssoc";
                ways = 8;
            };
            latency = 4;
        };

        l1i = {
            caches = 64;
            size = 32768;
            array = {
                type = "SetAssoc";
                ways = 4;
            };
            latency = 3;
        };


        l2 = {
            caches = 64;
            size = 262144;
            latency = 7;
            array = {
                type = "SetAssoc";
                ways = 8;
            };
            children = "l1i|l1d";
        };
```

```
47
48          l3 = {
49              type = "Timing";
50              caches = 1;
51              banks = 16;
52              size = 8388608;
53              latency = 27;
54
55              array = {
56                  type = "SetAssoc";
57                  hash = "H3";
58                  ways = 16;
59              };
60
61              children = "l2";
62          };
63      };
64
65      mem = {
66          type = "Ramulator";
67          ramulatorConfig = "ramulator-configs/HMC-config.cfg";
68          latency = 1;
69      };
70  };
71
72  sim = {
73      pimMode = false;
74      stats = "zsim_stats/host_inorder/no_prefetch/64/my_workload_my_workload_my_workload";
75      phaseLength = 1000;
76      maxOffloadInstrs = 1000000000L;
77      maxTotalInstrs = 1000000000L;
78      statsPhaseInterval = 1000;
79      printHierarchy = true;
80      gmMBytes = 8192;
81      pinOptions = "-ifeellucky";
82      deadlockDetection = false;
83  };
84
85  process0 = {
86      command = "~/DAMOV/workloads/test_workload/my_workload";
87      startFastForwarded = True;
88  };
```

Figure 22 The Config File

8 Bibliography

2. Mutlu, Onur, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. "Processing data where it makes sense: Enabling in-memory computation." *Microprocessors and Microsystems* 67 (2019): 28-41.

3. Boroumand, Amirali, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim et al. "Google workloads for consumer devices: Mitigating data movement bottlenecks." In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 316-331. 2018.

4. Jia, Zhen, Jianfeng Zhan, Lei Wang, Chunjie Luo, Wanling Gao, Yi Jin, Rui Han, and Lixin Zhang. "Understanding big data analytics workloads on modern processors." *IEEE Transactions on Parallel and Distributed Systems* 28, no. 6 (2016): 1797-1810.

5. R. Sites, “It’s the Memory, Stupid!” *MPR*, 1996

6. Tsai, Po-An, Yee Ling Gan, and Daniel Sanchez. "Rethinking the memory hierarchy for modern languages." In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 203-216. IEEE, 2018.

7. Qureshi, Moinuddin K., M. Aater Suleman, and Yale N. Patt. "Line distillation: Increasing cache capacity by filtering unused words in cache lines." In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pp. 250-259. IEEE, 2007.

8. Tziouvaras, Athanasios. "Design space exploration in near-data co-processors for general-purpose acceleration, in high-performance and low-power processing environments." PhD diss., University of Thessaly.Electrical and Computer Engineering, 2021.

9. Qureshi, Moinuddin K., Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. "Adaptive insertion policies for high performance caching." *ACM SIGARCH Computer Architecture News* 35, no. 2 (2007): 381-391.

10. Ferdman, Michael, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. "Clearing the clouds: a study of emerging scale-out workloads on modern hardware." *Acm sigplan notices* 47, no. 4 (2012): 37-48.

11. McKee, Sally A. "Reflections on the memory wall." In *Proceedings of the 1st conference on Computing frontiers*, p. 162. 2004.

12. Stone, Harold S. "A logic-in-memory computer." *IEEE Transactions on Computers* 100, no. 1 (1970): 73-78.

13. Elliott, Duncan G., W. Martin Snelgrove, and Michael Stumm. "Computational RAM: A memory-SIMD hybrid and its application to DSP." In *1992 Proceedings of the IEEE Custom Integrated Circuits Conference*, pp. 30-6. IEEE, 1992.

14. Patterson, David, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. "A case for intelligent RAM." *IEEE micro* 17, no. 2 (1997): 34-44.

15. Zhu, Qiuling, Tobias Graf, H. Ekin Sumbul, Larry Pileggi, and Franz Franchetti. "Accelerating sparse matrix-matrix multiplication with 3D-stacked logic-in-memory hardware." In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1-6. IEEE, 2013.

16. Ahn, Junwhan, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. "A scalable processing-in-memory accelerator for parallel graph processing." In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pp. 105-117. 2015.

17. Ghose, Saugata, Amirali Boroumand, Jeremie S. Kim, Juan Gómez-Luna, and Onur Mutlu. "Processing-in-memory: A workload-driven perspective." *IBM Journal of Research and Development* 63, no. 6 (2019): 3-1.

18. Hsieh, Kevin, Samira Khan, Nandita Vijaykumar, Kevin K. Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. "Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation." In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pp. 25-32. IEEE, 2016.

19. Liu, Pei, Ahmed Hemani, Kolin Paul, Christian Weis, Matthias Jung, and Norbert Wehn. "3D-stacked many-core architecture for biological sequence analysis problems." *International Journal of Parallel Programming* 45, no. 6 (2017): 1420-1460.

20. Shaw, David Elliot, Salvatore J. Stolfo, Hussein Ibrahim, Bruce Hillyer, Gio Wiederhold, and J. A. Andrews. "The NON-VON database machine: A brief overview." *IEEE Database Eng. Bull.* 4, no. 2 (1981): 41-52.

21. Memory Cube Consortium, "Hybrid Memory Cube Specification Rev. 2.0," http://www.hybridmemorycube.
Hybrid org/

22. JEDEC, High Bandwidth Memory (HBM) DRAM, Standard No. JESD235 (2013).

23. V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, T. C. Mowry, Buddy-RAM: Improving the Performance and Efficiency of Bulk Bitwise Operations Using DRAM, arXiv:1611.09988 [cs:AR] (2016)

24. V. Seshadri, O. Mutlu, Simple Operations in Memory to Reduce Data Movement, in: Advances in Computers, Volume 106, 2017.

25. J. Ahn, S. Hong, S. Yoo, O. Mutlu, K. Choi, A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing, in: ISCA, 2015.

26. J. Ahn, S. Yoo, O. Mutlu, K. Choi, PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture, in: ISCA, 2015

27. K. Hsieh et al., "Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems," 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016, pp. 204-216, doi: 10.1109/ISCA.2016.27.

28. Geraldo F. Oliveira, Juan Gómez-Luna, Lois Orosa, Saugata Ghose, Nandita Vijaykumar, Ivan Fernandez, Mohammad Sadrosadati, Onur Mutlu, "DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks". arXiv:2105.03725 [cs.AR], 2021.

29. Sanchez, Daniel & Kozyrakis, Christos. (2013). ZSim: fast and accurate microarchitectural simulation of thousand-core systems. ACM SIGARCH Computer Architecture News. 41. 475. 10.1145/2508148.2485963.

30. Y. Kim, W. Yang, O. Mutlu. "Ramulator: A Fast and Extensible DRAM Simulator". In IEEE Computer Architecture Letters, March 2015.

31. Reif, Rafael; Tan, Chuan Seng; Fan, Andy; Chen, Kuan-Neng; Das, Shamik; Checka, Nisha (2002). "3-D Interconnects Using Cu Wafer Bonding: Technology and Applications" (PDF). Advanced Metallization Conference

32. William Shockley, “Semiconductive Wafer and Method of Making the Same”, US Patent # 3,044,909, filed on October 23, 1958 and granted on July 17, 1962

33. Merlin Smith and Emanuel Stern, “Methods of Making Thru-Connections in Semiconductor Wafers”, US Patent # 3,343,256, filed on December 28, 1964 and granted on September 26, 1967.

34. Kawamura, S.; Sasaki, Nobuo; Iwai, T.; Mukai, R.; Nakano, M.; Takagi, M. (1984). "3-Dimensional Gate Array with Vertically Stacked Dual SOI/CMOS Structure Fabricated by Beam Recrystallization". 1984 Symposium on VLSI Technology. Digest of Technical Papers: 44–45

35. Akasaka, Yoichi; Nishimura, T. (December 1986). "Concept and basic technologies for 3-D IC structure". 1986 International Electron Devices Meeting: 488–491.

36. Kada, Morihiro (2015). "Research and Development History of Three-Dimensional Integration Technology" (PDF). Three-Dimensional Integration of Semiconductors: Processing, Materials, and Applications. Springer. pp. 8–13. ISBN 9783319186757.

37. James, Dick (2014). "3D ICs in the real world". 25th Annual SEMI Advanced Semiconductor Manufacturing Conference (ASMC 2014): 113–119.

38. V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, M. A. Kozuch, P. B. Gibbons, T. C. Mowry, RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization, in: MICRO, 2013

39. V. Seshadri, K. Hsieh, A. Boroumand, D. Lee, M. A. Kozuch, O. Mutlu, P. B. Gibbons, T. C. Mowry, Fast Bulk Bitwise AND and OR in DRAM, CAL (2015)

40. Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, J. M. Hellerstein, Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud, VLDB Endowment (2012)

41. Fidel, Adam & Amato, Nancy & Rauchwerger, Lawrence. (2014). KLA: A new algorithmic paradigm for parallel graph computations. Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT. 10.1145/2628071.2628091.

42. L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar and H. Kim, "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2017, pp. 457-468, doi: 10.1109/HPCA.2017.54.

43. Hadidi, Ramyad & Nai, Lifeng & Kim, Hyojong & Kim, Hyesoon. (2017). CAIRO: A Compiler-Assisted Technique for Enabling Instruction-Level Offloading of Processing-In-Memory. ACM Transactions on Architecture and Code Optimization. 14. 1-25. 10.1145/3155287.

44. Santos, Paulo C., Marco AZ Alves, Matthias Diener, Luigi Carro, and Philippe OA Navaux. "Exploring cache size and core count tradeoffs in systems with reduced memory access latency." In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pp. 388-392. IEEE, 2016

45. Intel Corp., "Intel VTune Profiler User Guide," https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/, 2021

46. Williams, Samuel, Andrew Waterman, and David Patterson. "Roofline: an insightful visual performance model for multicore architectures." *Communications of the ACM* 52, no. 4 (2009): 65-76.

47. Doerfler, Douglas, Jack Deslippe, Samuel Williams, Leonid Oliker, Brandon Cook, Thorsten Kurth, Mathieu Lobet, Tareq Malas, Jean-Luc Vay, and Henri Vincenti. "Applying the roofline performance model to the intel xeon phi knights landing processor." In *International Conference on High Performance Computing*, pp. 339-353. Springer, Cham, 2016.

48. Kim, Hyojong, Hyesoon Kim, Sudhakar Yalamanchili, and Arun F. Rodrigues. "Understanding energy aspects of processing-near-memory for HPC workloads." In *Proceedings of the 2015 International Symposium on Memory Systems*, pp. 276-282. 2015.

49. Nai, Lifeng, and Hyesoon Kim. "Instruction offloading with hmc 2.0 standard: A case study for graph traversals." In *Proceedings of the 2015 International Symposium on Memory Systems*, pp. 258-261. 2015.

50. Santos, Paulo C., Marco AZ Alves, Matthias Diener, Luigi Carro, and Philippe OA Navaux. "Exploring cache size and core count tradeoffs in systems with reduced memory access latency." In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pp. 388-392. IEEE, 2016.