1. Introduction

It is common practice to refer to the central processor unit (CPU) of a computer system as the main or only processing unit of the computer. The CPU is the component of the computer through which data is fed and changed while it is performing calculations.

The ability of a system to do arithmetic operations is frequently used as a performance metric for systems. However, the computing capacity of a system is not only dependent on the number of calculations that it is able to do in a single second. It is also dependent on the system's capability and efficiency in staging data to these calculations. This includes the references to auxiliary tables in addition to inputs and parameters.

Today, all of this data that a computation requires is loaded into a central location on a computer that is referred to as the memory of the device.

These modern applications significantly overwhelm the data storage and movement resources of a modern computer because of their growing dependence on manipulating and mining through large sets of data. The main memory of a modern computer is incapable of carrying out any operations on the data it stores. Therefore, in order to perform any operation on data that is stored in memory, the data needs to be moved from memory to the CPU via the memory channel, which is a pin-limited off-chip bus. The central processing unit (CPU) is required to send a request to the memory controller before it is possible to move the data. The memory controller will then send commands to the DRAM module that is responsible for storing the data across the memory channel. Following this, the DRAM module will read the data and return it across the memory channel. The data will then proceed through the cache hierarchy before being stored in a CPU cache. After the data has been loaded from the cache into a CPU register, the CPU will be able to perform operations on the data.

Unfortunately, the large amounts of data that need to move across the memory channel create a large data movement congestion in the computing system. This congestion affects both modern applications and emerging applications. The delay in the movement of data imposes a significant cost, not only in terms of performance but also in terms of the amount of energy demand.

Access to memory, however, gradually slowed down to the point where it lagged behind the rate at which computations could be performed as a consequence of both the increasing speed of computers and the growing complexity of the issues being solved.

This resulted in the development of concepts such as data-flow architectures, in which the formula for the calculation did not consist of a sequential series of steps and in which the data were not centralised but rather were dispersed in parallel across the processing units of the computer. These ideas were developed as a direct consequence of the aforementioned situation. Because computer engineers were able to find techniques to foresee the instructions and data that the computational elements would need and locate them suitably in time for them to be processed on the various units of the machine, such approaches were not successful commercially. As a result,

modern computers have a set of local registers and one or more levels of cache in a memory hierarchy in addition to the computer's main memory.
The machines that have resulted from this have gotten fairly smart and complex, and the community is once again questioning whether it is efficient to hold data in a central location and move it to processing units that are growing further and further away. As long as Moore's law maintained consistently rising transistor densities and Dennard scaling kept power requirements low, cost and energy were not major problems for the semiconductor industry. Dennard scaling also kept power requirements low. Now that Dennard scaling has been discontinued, the attention of those concerned is shifting to the amount of energy that is used in the process of transporting data to and from memory that is shared by several processors across huge systems. As a consequence of this, there is a movement towards splitting the memory over many compute nodes and carrying out multiple activities in parallel, in a manner that is comparable to the traditional data-flow concepts. It would appear that the computational requirements of workloads that deal with vast amounts of unstructured data are met particularly well by this paradigm.

2. Concept of PIM

PIM has been discussed for at least the past four decades, but earlier efforts to implement the concept were not widely adopted due to the difficulty of integrating processing elements for computation with DRAM. The ability to perform logic operations using memory cells themselves inside a memory chip and the emergence of potentially more computation-friendly resistive memory technologies provide new opportunities to embed general-purpose computation directly within the memory. Examples of these innovations include three-dimensional (3D)-stacked memory dies, which combine a logic layer with DRAM layers.

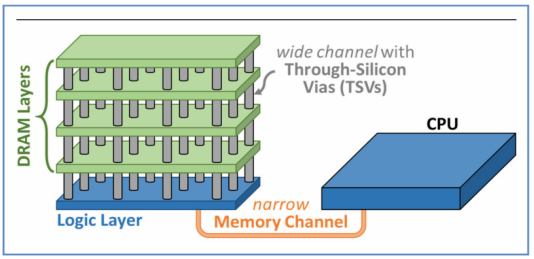


Fig. 1 : overview of a 3D-stacked DRAM architecture

The main memory of a modern computer is made up of DRAM, and is capable of carrying out any activities that are related to the data it stores. As a result, in order to perform any action on data stored in memory, the data must first transfer from memory to the CPU via a channel known as the memory channel. After passing through the various levels of the cache hierarchy, the data is finally saved in the CPU cache. After the data has been loaded from the cache into a CPU register, the CPU will be able to perform operations on it.
Unfortunately, the vast amounts of data that need to be moved generate a large data movement bottleneck in the computing system, which hinders the performance of modern applications. This results in a significant reduction in both performance and the amount of energy required. The cost of moving large amounts of data is the single largest contributor to the total cost of

computation for data-intensive applications in the current era. This is true both in terms of performance and energy consumption. PIM, also known as near-data processing, is a computing paradigm that involves integrating processing capabilities directly into the memory subsystem of a computer system. This is in contrast to the traditional von Neumann architecture, in which processing and memory are two separate entities. Recent advancements in memory design have made it possible for architects to avoid unnecessary data movement by performing PIM. PIM, also known as "near-data processing," is a computing paradigm that involves integrating processing capabilities directly into the memory subsystem of a computer system.

Despite the fact that PIM can prevent many data-intensive applications from shifting data from memory to the CPU, it poses additional difficulties for system architects and programmers.

3. Current Architecture

Von Neumann argued that it was useful to have a single type of memory with interchangeable parts to accommodate the various requirements in different applications of the various functions mentioned above. This was the case even though the nature of the operations that were required for the various tasks was different and even though the locations in the machine where they were needed were different. Von Neumann's theory was that it was useful to have a single type of memory. In 1949, when it was eventually constructed, the EDVAC, in addition to the external tape unit, included a memory that could hold 1024 words of 44 bits each. The memory was a sequential access mercury delay-line memory with a throughput of one word every 48 seconds from each of 128 delay lines. The average latency was 192 seconds, and the memory had a capacity of 128 delay lines. In contrast, there was one addition carried out for every 864 seconds that elapsed.

These high-speed memories were prohibitively expensive and fussy, rendering them unfit for use in a commercial setting. Drum memory was the sort of memory that commercial systems elected to use instead, since it was a more cost-effective option. The drum itself was a metal cylinder that was coated with a magnetic recording substance. The maximum number of ten-digit words that could be stored in the drum memory of the original IBM 650 machines was 2,000, and the machines had an average access time of 2.4 milliseconds. The maximum rotational speed of the drum was limited to 12,500 revolutions per minute. It was able to conduct 200 additions of ten-digit numbers per second. The high-speed delay line memory of the EDVAC was supplemented in 1954 by the addition of a drum memory that contained 4608 words. Drum Memories had a brief run as main memory because of the rotational latency, but they were still in common use until the 1960s as secondary memory or storage. Although DRAM memories had a short run as main memory, they were still in widespread usage.

	Power5	Power6	Power7	Power7+	Power8
Technology	130 nm	65 nm	45 nm	32 nm	22 nm
Cores per chip	2	2	8	8	12
L1 cache per core	32 + 32 KB	64 + 64 KB	32 + 32 KB	32 + 32 KB	64 + 32 KB
L2 cache per core	⇓	⇓	256 KB	256 KB	512 KB
Last-level cache per chip	1.9 MB	8 MB	32 MB	80 MB	96 MB
Off-chip cache per socket	36 MB	32 MB	None	None	128 MB
Max. Memory per socket	64 GB	192 GB	256 GB	512 GB	1 TB
Bandwidth to Memory	15 GB/s	30 GB/s	100 GB/s	100 GB/s	230 GB/s

Fig. 3. Cache and memory capacities for IBM's Power series of computers

But with advancement, DRAM scalability is being pushed to its practical limits as a result of modern applications' growing demand for bigger memory systems. Densification, latency reduction, and energy reduction in standard DRAM architectures are getting more challenging.

4. Near Data Processing

In general, algorithms are created with the intention of locating necessary data close to where they are used, both spatially and temporally. When algorithms display this locality, hardware implementations are developed for the situation. It gets more challenging to satisfy locality requirements at the algorithmic and hardware levels as an algorithm uses larger data and access to data occurs in unpredictable ways.

Hence, instead of moving the data through all the levels to the CPU and then again back to memory, it would be better in terms of both latency and energy to process the data where it is located.

An example of such a concept is IBM Netezza, as illustrated in Fig. 3. This system performs a variety of computations near to the disc, therefore, it delivers processed data to the processor rather than raw data from the disc. There are many different calculations that can be done on the disc, from straightforward filtering and data compression to complex query processing. The advantages of carrying out such operations closer to the disc are numerous: energy is conserved by avoiding the need to transport data via the memory hierarchy; memory bandwidth needs are decreased; query response time is decreased; and the host processor can carry out other duties.

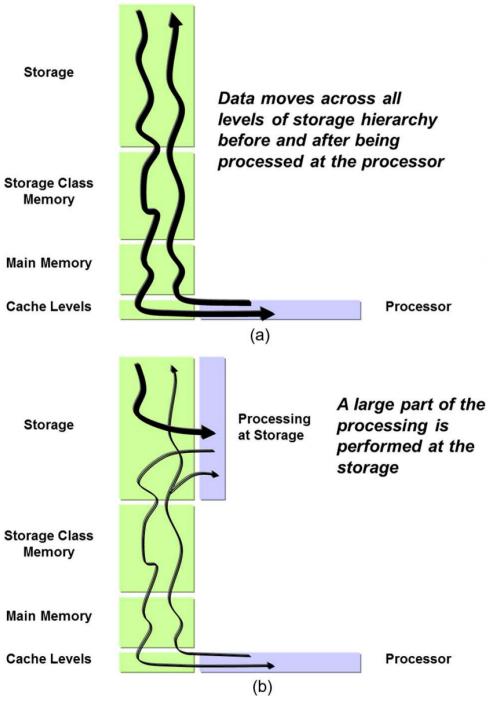


Fig1. Moving computation to data. (a) Traditional processing (b) The Netezza Paradigm

When a complex query is delivered to the processor, the portions of the query that can be spread and processed in parallel close to the discs are provided to each disc. Field-programmable gate arrays (FPGAs), which are included in the Netezza system's disc controller and are depicted in Fig. 4, enable the "computation near data" to be tailored for each application. The The main

processor receives the incomplete results of such localized processing and combines them with other results of a similar nature from other discs to carry out additional operations.

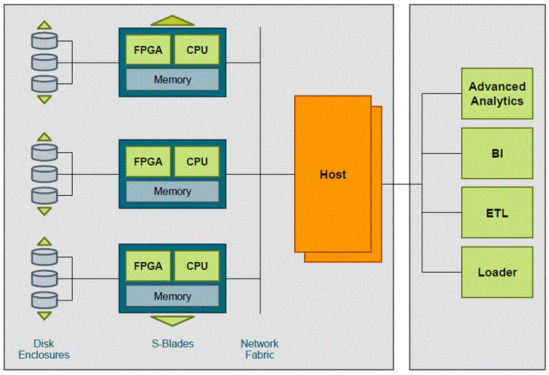


Fig. 4. Netezza TwinFin appliance architecture

These techniques can also be implemented on SSDs. The latency of the flash drives will make it easier to perform operations near the SSD.

A near-data processing architecture can help save bandwidth, decrease latency by up to 65%, and manage energy more efficiently.

This paradigm holds a lot of potential for leading us into the future of near-data processing.

5. Processing in memory

Architects incorporate logic within DRAM to carry out data-parallel operations in the first family, which includes NON-VON, computa- tional RAM, EXECUBE, Terasys, and IRAM. In the second family of evolution, programmers propose more adaptable substrates that tightly incorporate logic and reconfigurability within DRAM itself to boost flexibility and available computational power. Some of these machines are Active Pages, FlexRAM, Smart Memories, and DIVA.

The main concept was to combine processing and memory on one chip in order to utilize the available bandwidth rather than being restricted by it through the pin interfaces of several processors.

These proposals demonstrated effective performance and low power consumption, but the limitations of current memory technology prevented their practical implementation.

5.1 UPMEM PIM Architecture

Fig. 5 depicts a UPMEM PIM architecture with

1. A host CPU
2. Standard Main Memory (DPU Modules)
3. PIM- enabled memory or UPMEM modules

A UPMEM module is a typical DDR4-2400 DIMM (module) that has an array of PIM chips.

An UPMEM chip consists of 8 DPUs, with each DPU having access to a 64 MB DRAM cell, or MRAM, 24kb IRAM and a 64kb WRAM.

The host CPU can access MRAM for copying input data and retrieving results from and to main memory.

If the buffers transmitted from or to all MRAM banks are the same size, then CPU-DPU and DPU-CPU data transfers can be carried out continuously across several MRAM banks in parallel. If not, data transfers are sent sequentially (i.e., a transfer from or to one MRAM bank begins after the transfer to or from another MRAM bank is finished).

Every inter-DPU communication goes through the host CPU, which copies data from the CPU to the DPU and retrieves results from the DPU to the CPU. At this time, in UPMEM-based PIM systems, the host CPU and the

DPU cannot access the same MRAM bank.

Data layouts for main memory and PIM-enabled memory differ. While PIM-enabled memory requires full 64-bit words to be mapped onto the same MRAM bank (on one PIM chip), main memory uses the traditional horizontal DRAM mapping, which maps successive 8-bit words onto successive DRAM chips. This unique data architecture in PIM-enabled memory is necessary since each DPU can only access a single MRAM bank while working with data types up to 64 bits.

To carry out the required data shuffle when transferring data between main memory and MRAM banks, the UPMEM SDK contains a transposition library. These data layout transformations are transparent to programmers. The serial/parallel/broadcast CPU-DPU and serial/parallel DPUCPU transfer functions included in the UPMEM SDK make internal calls to the transposition library, which transforming the data layout as necessary).

To run a DPU function or kernel, the host CPU can allocate the desired number of DPUs. The host CPU will then begin the DPU kernel, either synchronously or asynchronously. The host CPU thread is put on hold during synchronous execution until the DPU set has finished running the kernel. Asynchronous execution quickly transfers control back to the host CPU thread so that it can later check the progress of the operation.

In the current UPMEM-based PIM system configuration, the maximum number of UPMEM DIMMs is 20, which can contain up to 2,560 DPUs, which amounts to 160 GB of PIM-capable memory.

In practical, a UPMEM system contains 2,556 DPUs and a total of 159.75 GB of MRAM. With 128 DPUs each, the DPUs are divided into 20 double-rank DIMMs. Each DPU operates at 350 MHz. A dual x86 socket with two memory controllers per socket houses the 20 UPMEM DIMMs. Three memory channels are present on each memory controller. Two traditional DRAM DIMMs are installed in each socket, each of which serves as the host CPU's primary memory channel.

5.2 PIM in tensor flow

The results of our constraint analysis and the application of our systematic PIM target tool flow lead us to the conclusion that PIM is an effective solution for a number of important current workloads. In particular, we have discovered that workloads associated with machine learning and data analytics lend themselves particularly well to PIM. This is because these types of workloads frequently consist of compute-intensive and memory-intensive application stages. These workloads gain a significant advantage from PIM when just the memory-intensive PIM targets (those that are compatible with the limits of our system) are offloaded to the PIM logic. Examples of these types of workloads include neural network inference, graph analytics, and hybrid transactional/analytical processing database workloads.

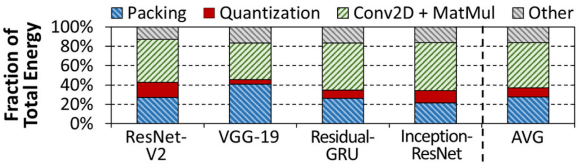


Fig. 5: Energy breakdown during

TensorFlow Lite inference execution on four input networks.

TensorFlow Lite is a version of Google's TensorFlow machine learning library that is specifically optimized for mobile and embedded systems. As a case study, we give a complete analysis utilizing our PIM target identification approach for TensorFlow Lite. TensorFlow Lite makes it possible to accomplish a range of tasks, including image classification, face recognition, and the instant visual translation that Google Translate offers. These applications all execute inference on consumer devices by utilizing a convolutional neural network that was pre trained on cloud servers. In this particular case study, we will be focusing on a processing-near-memory platform. To do this, we will be integrating either fixed-function PIM accelerators or small in-order PIM cores into the logic layer of a 3D-stacked DRAM. We model a 3D-stacked DRAM that is analogous to the Hybrid Memory Cube. This version of the memory has 16 vaults, which may also be thought of as vertical slices of DRAM. We add either one PIM core or one PIM accelerator to each vault, taking care that the combined surface area of both the PIM core and the PIM accelerator does not exceed the total area that is available for logic in each vault (which ranges from 3.5 to 4.4 mm2). Each PIM core and PIM accelerator are only capable of simultaneously executing a single PIM target. Previous work that we have done has provided information regarding the specifics of our methodology as well as the parameters of the platform that we are aiming towards.

The first step in inference is to provide a neural network with some input data (for example, an image). A directed acyclic graph with numerous layers makes up a neural network, which can be thought of as a type of graph. Each layer is responsible for a series of calculations, the results of which are then passed on to the following layer. The calculation might be different for each layer if we take into account the different kinds of layers. In order to extract high-level characteristics from the input data, a fully connected layer will conduct a matrix multiplication (MatMul) on those values. In order to extract low-level features from the input data, a 2-D convolution layer will apply a convolution filter known as Conv2D across the data. An output layer is the very last layer of a neural network. This layer is responsible for doing classification in order to provide a prediction based on the input data.

5.2.1 PIM effectiveness for quantization

TensorFlow Lite does the Conv2D operation twice for every time it performs quantization. Before beginning Conv2D, the 32-bit input matrix is first subjected to quantization. This step, which decreases the width of each matrix element to 8 bits and, as a result, the complexity of the

Operations that must be done by the CPU in order to complete Conv2D are conducted before Conv2D. After that, Conv2D is executed, and while it is running, gemmlowp produces a 32-bit

result matrix. On this result matrix, quantization is carried out for the second time (the process that takes place here is known as re-quantization). Invoking Conv2D more frequently (which happens when there are more 2-D convolution layers in a network) leads to increased quantization overheads. This is because of how the algorithm works. As an illustration, VGG-19 only needs 19 Conv2D processes, which results in a negligible amount of quantization overhead. Quantization, on the other hand, takes 16.1% of the entire system's energy and 16.8% of the execution time. This is because ResNet-v2 requires 156 Conv2D operations. It is reasonable to anticipate that the quantization overheads will increase along with the depth of neural networks, given that a deeper network will require a more extensive matrix.

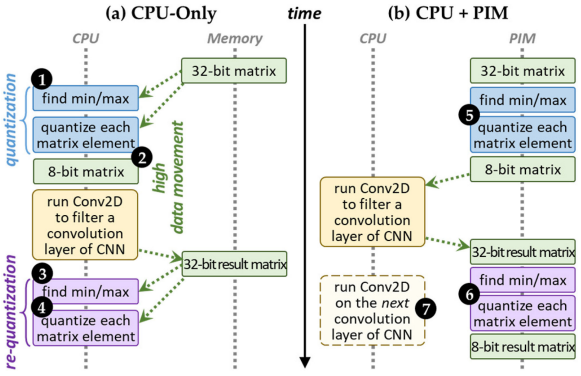


Fig. 6: Quantization on (a) CPU versus (b) PIM

6. Challenges in PIM

There are a number of challenges that need to be addressed before it is possible to construct PIM architectures that are widely usable by the majority of programmers and are adopted by them. In this essay, we will talk about two of the most critical issues that PIM is currently facing.

When building PIM logic, the first step is for programmers to be able to determine which parts of an application are appropriate for PIM, and architects need to be aware of the limitations imposed by various substrates in order to design PIM logic effectively. Second, after opportunities for PIM have been discovered and PIM architectures have been created, programmers need a means to reap the benefits of PIM without having to resort to sophisticated programming models in order to do so. Despite the fact that these two issues constitute some of the most significant barriers to widespread adoption of PIM, a number of other significant challenges still exist.

6.1 Design Constraints

The computations that are eligible to receive benefits from PIM are subject to a number of fundamental constraints as a result of the target architecture. PIM can be implemented in one of two ways: processing-near-memory or processing-using-memory. Every approach comes with its own set of constraints on the types of logical operations that can be carried out in memory in an effective and efficient manner.

In the case of processing-near-memory, the PIM logic needs to be added close to the memory. This can be done either in the logic layer of a 3D-stacked memory chip or in the same package. Because of this, The amount of PIM logic that may be introduced is constrained. For example, in an HMC-like 3D-stacked memory architecture that is implemented using a manufacturing processing technology of 22 nm, we estimate that there is around

50–60 mm2 of area available for architects to incorporate new circuitry into the DRAM logic layer. This estimate is based on the fact that an HMC-like 3D-stacked memory architecture uses a stacked memory design. The architecture of the memory can also place further constraints on the amount of space that is available. For example, the 3D-stacked memory is segmented into a number of vaults in HMC. Vaults are vertical slices of 3D-stacked DRAM.
Because the logic is directly connected to the memory in the vault by the TSVs, logic that is placed in a vault has quick access to the data that is stored in the memory layers of the same vault; however, gaining access to the data that is stored in a different vault requires much more time. As a consequence of this, architects frequently duplicate PIM logic within each vault in order to reduce the amount of time that PIM operations take. The quantity of space that can be stored each vault is consequently reduced to a substantial extent as a result of this: There is approximately 3.5–4.4 mm2 of area available for PIM logic on a memory chip that has a 3D-stacked architecture and has 32 vaults.
In addition to space limitations, a number of the computing platforms being targeted have other restrictions. For example, consumer devices like smartphones, tablets, and netbooks have extremely severe requirements for any new hardware enhancements in terms of the amount of space and energy budget that they can accommodate. When it comes to consumer electronics, the addition of even a little bit more logic to the memory could result in a large price increase. In point of fact, unlike PIM logic that is introduced to server or desktop settings, consumer devices may not be able to afford the addition of full-blown general-purpose PIM cores, GPU PIM cores, or complicated PIM accelerators to 3D-stacked memory. This is in contrast to PIM logic that is added to server or desktop environments. As a consequence of this, one of the most significant difficulties associated with enabling PIMin consumer devices is to determine the type of in-memory logic that can simultaneously maximise energy efficiency and be implemented at the lowest possible cost. Another drawback in 3D-stacked memory is its thermal dissipation, which is caused by the fact that adding PIMlogic to the logic layer has the ability to elevate the DRAM temperature to values that are unacceptable.
In the case of processing-using-memory, the cells and memory arrays themselves are what are used to implement the PIM logic. To enable logic operations on the cells or in the memory array, or to provide more specialised functionality beyond what the cells and memory array themselves can readily execute (for example, dedicated adders or shifters), it may be necessary to include additional logic in the controller and/or in the array itself.

6.2 Hardware implementation

Once the constraints of what kinds of hardware could potentially be implemented in memory have been established, the properties of the programme itself are a critical indicator for determining whether or not certain parts of an application can benefit from PIM. It may be a naive assumption to

totally move memory-intensive applications to PIM logic, but this is not necessarily the case. However, we have discovered that there are some instances in which certain components of these apps continue to reap benefits from being kept on the CPU. For instance, numerous suggestions for PIM designs include the addition of low-power general-purpose cores located close to memory. These cores are referred to as PIM cores. PIM cores are typically ISA-compatible with the CPU and are able to execute any part of the application; but, due to space, energy, and thermal limits, they are unable to afford to have massive cache hierarchies that span multiple levels or execution logic that is as complicated as that of the CPU. PIM cores can execute any part of the application. PIM cores frequently do not have any caches or only have a tiny amount of cache, which limits the amount of temporal locality that they are able to exploit. Additionally, PIM cores typically do not have any advanced aggressive out-of-order or superscalar execution logic, which limits their capacity to extract instruction-level parallelism. As a consequence of this, parts of an application that either require a lot of computing power or are amenable to being stored in cache should be kept on the larger, more advanced CPU cores.

In view of these constraints, we have determined that it is critical to decide which parts of an application are appropriate for PIM. PIM targets are sections that meet these criteria. Even though PIM targets can be chosen manually by a programmer, doing so would involve a large amount of effort on the part of the programmer in addition to a comprehensive grasp of the hardware tradeoffs that exist between CPU cores and PIM cores. The tradeoffs between CPU cores and PIM accelerators may not be known before determining which parts of the application are PIM targets for architects who are adding custom PIM logic to memory (for example, fixed-function accelerators, which we call PIM accelerators). This is because PIM accelerators are tailored for the PIM targets.

We design a systematic toolflow for identifying PIM targets in an application in order to decrease the effort of manually identifying PIM targets. This will help reduce the load of manually identifying PIM targets. In order to determine whether or not each PIM target satisfies the requirements of the system that is being taken into consideration, this toolflow makes use of a system that runs the complete application on the CPU. For instance, when we examine workloads for consumer devices, we use hardware performance counters in conjunction with our energy model to isolate candidate functions that have the potential to be PIM targets. In a consumer device, a function is considered to be a PIM target candidate if and only if it satisfies the following conditions:

1) Out of all the functions in the workload, it is the one that uses the most energy, despite the fact that one of the key goals of consumer workloads is to reduce energy use.

2) The transport of its data uses up a sizeable portion (say, more than twenty percent) of the entire amount of energy required by the workload in order to make the most of the possible energy

savings that can be gained by offloading to PIM.
3) It requires a significant amount of memory (i.e., its last-level cache misses per kilo instruction, or MPKI is greater than 10, as the potential for PIM to save more energy is increased when a greater number of data movements are eliminated.
4) The transfer of data represents the single most significant contributor to the total amount of energy required by the function.
After that, we use two criteria to determine whether or not each potential function is capable of having PIM logic implemented. First, we get rid of any PIM targets that suffer any kind of performance hit when they are executed on basic PIM logic (such as PIM core or PIM accelerator). Second, if any PIM targets demand more space than what is now available in the logic layer of the 3D-stacked memory, we eliminate them from consideration. Note that in the case of pre-built PIM architectures that have a fixed PIM logic, we will instead get rid of any PIM targets that are unable to be processed on the PIM logic that is already in existence.
Although our toolflow was originally developed to determine PIM goals for consumer devices, the toolflow can be changed to fit any other hardware restrictions that may be present. For example, in our research on how to lower the cost of maintaining cache, coherence in PIM architectures), and we take into account the amount of data sharing, which may be defined as the total number of cache lines that are read simultaneously by the CPU and by PIM logic. When we did that work, one of the things that we did was get rid of any potential PIM targets that would result in a high amount of data sharing if the target were offloaded to a PIM core. This is because doing so would induce a large amount of cache coherence traffic between the CPU and PIM logic, which would cancel out the savings that would be gained from fewer data transfers.

6.3 Sharing data between CPU and PIM logic

In a computer system that is capable of PIM, the PIM logic should be able to run concurrently with the CPUs, much like a multithreaded computer does. This will allow for the most efficient use of the system's resources. When using a typical paradigm of multithreaded execution that makes use of shared memory between threads, it is necessary for many cores to align their memory writes in order to ensure that threads do not operate on data values that have become obsolete. When one core of a When a CPU writes data to a memory address, cached copies of the data kept within the caches of other cores must be updated or invalidated in order to maintain what is known as cache coherence. This is because CPUs use caches that are specific to each core of the processor. A protocol is necessary for cache coherence. This protocol is meant to manage write permissions for each core, invalidations, and updates, as well as arbitration when multiple cores request exclusive access to the same memory address. Over a shared connection, the per-core caches that are contained within a chip multiprocessor are able to carry out coherence operations.
The coherence of the cache presents a significant problem for the system when

it comes to enabling PIM designs as general-purpose execution engines. If the PIM processing logic is coherent with the CPU, the PIM programming model is reasonably straightforward since it stays similar to conventional shared memory multithreaded programming. This similarity makes it easier for general-purpose systems to adopt PIM architectures. Therefore, making it possible for PIM processing logic to keep using such a straightforward and time-honored shared memory programming style can make it easier for PIM to gain universal acceptance. However, using standard fine-grained cache coherence for PIM (for example, using a cache-block-based MESI protocol) causes a significant amount of coherence messages to traverse the narrow memory channel. This has the potential to nullify the benefits of high-bandwidth and low-latency PIM execution. Unfortunately, the proposed solutions for coherence in earlier PIM works either set certain limits on the programming paradigm (by doing away with coherence and mandating programming based on message passing) or limit the performance and energy gains that may be achieved by a PIM design.

We present a coherence mechanism for PIM called CoNDA, which does not need to issue a coherence request for every memory access in order to keep existing programming patterns while simultaneously optimizing improvements in both performance and energy consumption. Instead, as shown in Fig., CoNDA enables efficient coherence by having the PIM logic do the following:

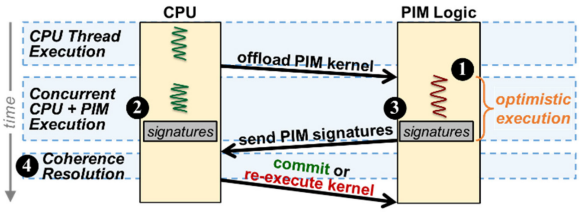


Fig. 7: High-level operation of CoNDA

1) speculatively acquire coherence permissions for multiple memory operations over a given period of time (this is what we call optimistic execution; in fig. 7);

2) batch the coherence requests from the multiple memory operations into a set of compressed coherence signatures.

3) send the signatures to the CPU to determine whether the speculation violated any coherence semantics.

7. Future Scopes of PIM

Memory-driven classical computing today has a significant disadvantage, which takes a long cycle (read-write) of memory access, which is ten times more, than the executing time of a logical operation. So a proposed solution to eliminate this is the implementation of memory-driven computing in quantum (photonic) transactions on the atomic structure of electrons eliminates the low speed of memory access.

The creation of memory-driven quantum computing uses write-read operations on electron memory, where binary states are realized, for example, in the spin moments of electrons. Simulating parallel quantum algorithms on classical computers gives a significant increase in performance. The traditional memory has several problems, like:

1) the inability to execute arithmetic-logical operations in memory that forces developers to create specialized ALUs for processing data based on the use of a bus, adversely affecting the performance
2) the inability to execute read-write data in parallel by simultaneously accessing all cells in the address memory
3) the inability to create addressable memory free of hardware-complex address decoders, which make memory an expensive product
4) the inability to create a computer free of logical operations that require the development of specialized ALU blocks outside the main memory.
These shortcomings can be eliminated by implementing quantum memory-driven computing, in which there is no need for a separate ALU and the instructions are processed in parallel, and all the operations can be accessing read-write transactions.

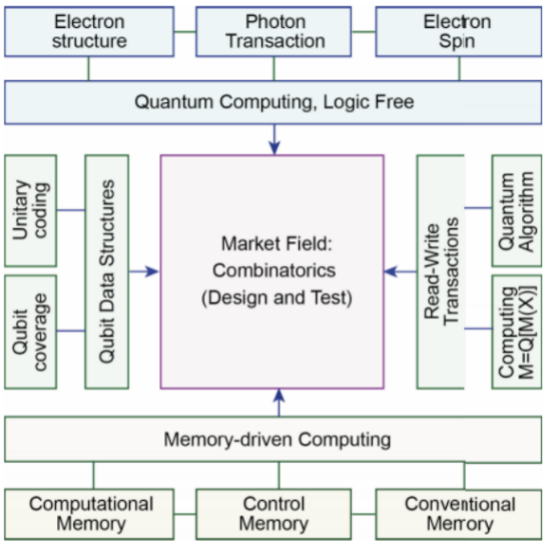
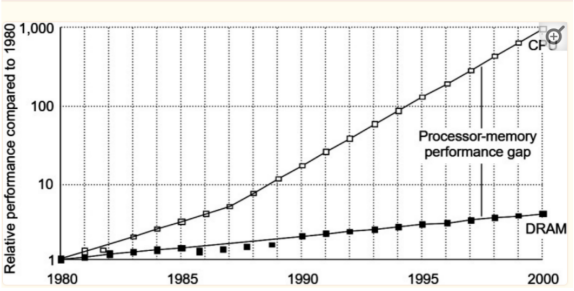


Fig. 8 : Quantum design and test PIM

Another approach could be the use of the Sparse Matrix Vector Multiplication accelerator for processing in memory. It is designed to hide the memory access latency to non-local banks, which increases the speed by 13.5 times and saves 87% of energy. In this, we distribute non-zero elements across the memory banks and operate in them using the processing elements. The input/ and output vectors in the sparse matrix are statically distributed on all the DRAM layers. The separation of the storage allows the processing elements to operate the sparse matrix in a manner that maximizes read bandwidth.
We can use the computing-in-memory concept, which integrates computing and storage capabilities into a single unit. Data does not need to migrate between the processor and memory. The technology breaks the bottleneck of traditional computer architecture, which is considered a significant development trend for future breakthroughs in computing power. Traditional architecture has several problems, like: data migration across the bus consumes much energy; the performance difference between processor and memory The processor's performance increased by 50% per year, whereas for memory, it was 7% per year. The mismatch between the processor and memory results in slow processing and wasted resources.



Computing-in-memory technology breaks through the traditional computing architecture, eliminating the need for a separate computing unit to process the data. The technology can be divided into charge-based and resistor-based storage technologies. Most charge-based storage technology is based on adding a computational circuit to a conventional memory cell. Resistor-based circuits mainly use new non-volatile memories, such as RRAM, PCM, and FRAM. Advanced memory combined with an across-array structure enables complex computations such as vector-matrix multiplication, greatly accelerating computation speed and reducing operational power consumption.

8. Conclusion

Early PIM implementations failed due to the complexity of combining compute processing units with DRAM. Memory-integrated general-purpose computing is possible now that memory cells can perform logic operations and resistive memory technologies are more computation-friendly. 3D-stacked memory dies combine logic and DRAM.
Modern DRAM main memory handles any data task. The memory channel must convey data to the CPU before use. The CPU cache holds cache hierarchy data. Cache-processing CPU registers.
The computing system's massive data flow bottleneck hinders applications. Performance decreases. Data mobility consumes most compute in data-intensive applications. Performance decreases. Near-data processing (PIM) processes in computer memory. Von Neumann design divides computation and memory. PIM helps architects prevent data transport waste. PIM, or "near-data processing," incorporates computation into a computer's memory subsystem.
Near Data Processing helps to reduce bandwidth, reduce energy usage, and increase the performance of a system,
PIM prevents data-intensive programs from transferring data from memory to the CPU, but system architects and programmers face other challenges. The challenges may include both software and hardware implementation. It may be design constraints of hardware, lack of optimized program to implement parallel processing in memory, poor design of PIM modules etc. Despite all the challenge, PIM is an effective way towards the advancement in the evolution of architectures.

9. References

[1] Ghose, S., Boroumand, A., Kim, J. S., Gomez-Luna, J., & Mutlu, O. (2019, November 1). Processing-in-memory: A workload-driven perspective. *IBM Journal of Research and Development*, *63*(6), 3:1-3:19. https://doi.org/10.1147/jrd.2019.2934048

[2] *Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System*. (n.d.). Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System | IEEE Journals & Magazine | IEEE Xplore. https://ieeexplore.ieee.org/abstract/document/9771457/

[3] Nair, R. (2015, August). Evolution of Memory Architecture. *Proceedings of the IEEE*, *103*(8), 1331–1345. https://doi.org/10.1109/jproc.2015.2435018

[4] Xie, X., Liang, Z., Gu, P., Basak, A., Deng, L., Liang, L., Hu, X., & Xie, Y. (2021, February). SpaceA: Sparse Matrix Vector Multiplication on Processing-in-Memory Accelerator. *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/hpca51647.2021.00055

[5] *Quantum memory-driven computing for test synthesis*. (n.d.). Quantum Memory-driven Computing for Test Synthesis | IEEE Conference Publication | IEEE Xplore. https://ieeexplore.ieee.org/abstract/document/811014

[6]