A Survey on *Near-Data Computing*: a promising incarnation of *Processing-in-Memory*

Feng Shi[1]

***Abstract*— (importance section, rephrased partially) In the past two decades, with the exponential growth in the volume of structured and unstructured data, data-intensive applications continue to experience rising adoption on both desktop and server systems. Clearly, the conventional computing model is incapable of scaling with the increasing volume of data. (why? use parallelism? citation?)**

As a branch of Processing-in-Memory (PIM), the renewed concept of Near-Data Computing (NDC) locates large-scale data processing in close proximity to the memory array. NDC improves the computational efficiency not only in terms of increased data throughput (by which factor? order of magnitude?), but also decreases the energy usage. As a key technology, 3D integration has matured over the past decades to such a point that it renders the implementation and practicality of NDC feasible.

This paper is a survey on (1) the taxonomy of a NDC system from both architectural and technical perspectives, (2) the performance and energy efficiency properties of such a system, and (3) the algorithmic optimizations for specific architectures in the scope of case studies.

ACKNOWLEDGEMENT

This paper as a submission to the Writing Qualification Exam is written solely by Feng Shi.

I. INTRODUCTION

In the past decades, along with the ease of collecting and accessing massive volume of data through the Internet services, people attempt to mine intelligence hidden inside these raw data (intelligence is often associated with a thinking being. Knowledge? Information? Patterns?). Therefore, big data applications and machine learning algorithms became (are becoming?) more and more (increasingly) valuable to mainstream society (mainstream? just society?) and companies. At the same time, analytics on high-volume data become more and more (avoid more and more)sophisticated not only because of data's high dimension but also due to their irregularity of access patterns and limited reusability. Therefore, researchers and system designers have to face challenges on both an application- and a computer-architecture level.(this is not clear. random access patterns corresponds to computer architecture? application level to high-volume data?)

A. Challenges in Data-Intensive Applications

The computing paradigm has been shifted from computation-intensive to data-intensive or data-centric; therefore the bottleneck of a system is no longer the capability of computational elements, but data movements (transfer?) contribute the dominant cost in performance and power consumption of the computer system. Due to the unique characteristics (it is not entirely clear what those characterstics are. Perharps, define them first) of data-intensive applications, they must overcome the following challenges:

- **Topology-driven computation.** Data-intensive applications not only consist of high-volume data (an application is code. Does it consist of data? task? is this common terminology?) but also contain implicit structures of computation in the underlying topology (topology of what exactly? Of the data?). As a result, allocating computational resources to each partition (partition of what? Partitions over the data?) is variable from one data partition to another. Therefore, mapping computational resources to data partitions is hard to exploit parallelism(wording.).
- **Poor locality.** Most data-intensive applications are usually highly irregular and not explicitly structured (I'm confused by the definition of an application. An application is irregular?): such as in graph application, some vertices only have a small number of neighbors, whereas others may connect to a very large number of neighbors (this structure is explicit, or not? I see poor locality now.); furthermore, the neighbors of a vertex

[1]Feng Shi is a Ph.D. student with Department of Computer Science, University of California Los Angeles, California 90095 `shi.feng at cs.ucla.edu`

are not always stored consecutively in memory. Therefore, such poor locality is not friendly to the traditional cache-based memory subsystem.

- **High I/O to computation ratio.** This challenge is closely related to previous one. Due to the poor locality which causes cache-based architecture inefficiencies, data-intensive applications usually exhibit higher cost in accessing the data than performing the computation itself. Memory fetch and network communication can turn into the critical bottlenecks of performance and the notable source of power consumption.
- **Iterative Execution.** Many data-intensive algorithms exhibit an iterative computation structures which requires iterative-friendly computational models. Moreover, the intermediate results generated between iterations necessitate a substantial amount of memory and storage. Further, the workloads between iterations are usually not evenly distributed, e.g., in a convolutional neural network, convolution layers share the same structure but they have different scales, that is, each layer generates feature maps in different size from those generated by other layers, and these feature maps are the intermediate results which need large amounts of storages.

B. Two architectural "Walls" faced by Data-Intensive Applications

Besides the challenges presented in the previous section, two "walls" must be tackled in a computer system. First, the modern computer system has encountered the issue of a "*Memory wall*" (define of memory wall first, then cache as solution, shorten sentence), e.g. On-chip cache memory, which is 10 to 100 times faster than off-chip DRAM, was supposed to knock down the memory wall. In order to bridge the speed gap between processor and off-chip memory, cache memory has to prefetch the data from off-chip memory before processor can perform computations. However, most of the data-intensive applications lack locality in memory, therefore, cache misses happen very often, and energy wastes are ineluctable. Second, due to data movement consumes energy, data-intensive applications issue dramatic amount of memory access requests during their executions.(rewrite. applications require countless random memory access. this is causing data transfers. transfers consume energy. results in high overall memory consumption.) In such case, computer system running data-intensive application hits the "*Power wall*" (define. Is the system consuming too much energy and can not diffuse the heat or is the high energy consumption economically not efficient?) rapidly.

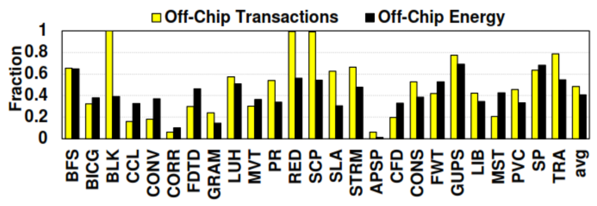

Fig. 1. Data movement and system energy consumption caused by off-chip memory accesses

Figure 1 illustrates the data movement and energy consumption overheads of transferring data between memory and compute units across 25 applications in a modern computer system, by showing: (1) the fraction of all data movement in the system stemming from off-chip transactions between memory and compute units, and (2) the fraction of total system energy consumption caused by this off-chip data movement.(wordiness) We notice that memory accesses result in 49% of all data movement and are responsible for 41% of the energy consumption of the system.

As such, it is of importance to develop a more advanced architecture to address the issues caused by data movement in the traditional von Neumann architecture. A promising approach is to move the computation closer to the memory to achieve faster access and higher bandwidth. Therefore, Processing-in-Memory and Near-Data Computing, which comply with such principle, come into sight. We discuss this concept in the following sections.

C. Resurgence of *Near-Data Computing*

The concept of *Processing in Memory* (PIM) or *in-Memory Processing* is not novel, the original idea of implementing PIM dates back to early 1990's [1], [2], [3], [4], [5], [6], and multiple PIM prototypes were proposed and demonstrated the potential for significantly improved performance and reduced power consumption in many application categories. Among them, EXECUBE [1], IRAM [4], DIVA [3], FlexRAM [5] etc. are the representative early proposals. Efforts such as IRAM [4] integrated embedded DRAM on logic chips. However, this approach was constrained by high-cost and low-density of the embedded DRAM, therefore it could not accommodate sufficient memory capacity and this drawback hindered its employment on high-performance systems. Efforts such as DIVA [3] and FlexRAM [5] integrated logic on memory dies. However, due to the generation gap between DRAM processes and contemporary logic processes, the performance of logic implemented in DRAM processes was drastically reduced.

After a dormant decade for the PIM, recent advances in die-stacking technology extend memory systems from 2D planar structure to 3D stack structure [7], [8], [9], which dramatically alleviate the cost limitation and practicality concerns of PIM. The success in commercialization of 3D stack memory system could cause the reincarnation of PIM [10], [11]. And this results in an upward trend in the *Near-Data Computing* (NDC) research field. Today, NDC has assumed a scope broader than that captured by the early PIM research of the 1990s.

II. Architecture of Near-Data Computing

Near-Data Computing (NDC), as its name suggested, applies the underlying principle *processing in the proximity of memory* to minimize the data transfer cost, where monolithic compute units are physically placed closer to monolithic memories. 3D stacking technology is the most promising solution to bring computation as close as possible to data. Therefore, recently-proposed NDC hardware architectures use a similar base design, and a baseline node configuration suitable for *Near-Data Computing* is shown in Figure 2 from a high-level view.

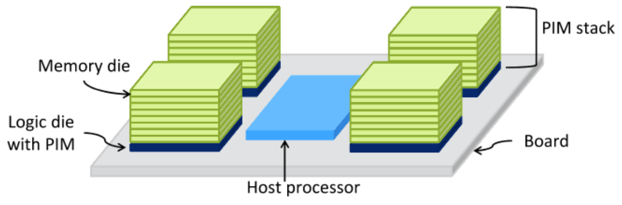

Fig. 2. An baseline configuration of compute node with PIM

An individual compute node of a *Near-Data Computing* system consists of three entities: (1) multiple memory stacks constitute the memory sub-system; (2) a logic layer consisting of low-power *compute units* beneath each memory stack acts as an accelerator; (3) and a high-end host processor chip with high-performance cores taking charge of coordination among entities. The layers in a memory stack and the logic layer under this memory stack are integrated vertically by Through-Silicon-Via (TSV) technology. TSVs are used to ship data from the DRAM dies to the logic layer. Due to the extension to the third dimension, it allows the system to eliminate the energy overheads of moving data over long board traces, and also provides an order of magnitude higher bandwidth between the stacked memory banks and the compute units in the logic layer. Moreover, the logic layer implements high-speed signaling circuits so that it can interface with the host processor chip through fast and wide serial links. For power-efficient execution of complex analytics workloads like large-scale graph processing algorithms, it is best to use as large a number of low energy-per-instruction (EPI) cores as possible in logic layer. This will maximize the number of instructions that are executed per joule, and will also maximize the number of instructions executed per unit time, within a given power budget. We call these power-efficient cores in a logic layer the *Near-Data Cores* or *NDCores*. While the high-end host processor is packed together with those memory stacks on the same board. The high-end host processor is taking the charge of dispatching the tasks to NDCores and schedules the ordering of task dispatches.

Now let us look at more details of each subsystem in the following subsections.

A. **Memory Stack**

The two most prominent and successfully commercialized 3D stacked memory technologies today are Micron's Hybrid Memory Cube (HMC) [12] and JEDEC's High-Bandwidth Memory (HBM) [13]. Although these two memory technologies have a few differences in their physical structure and process. But from an architectural and logical view they adopt the same memory organization, both HMC and HBM integrate a logic die with multiple DRAM chips in a single stack, which is divided into multiple independent channels (typically 8 to 16), as shown in Figure 3.

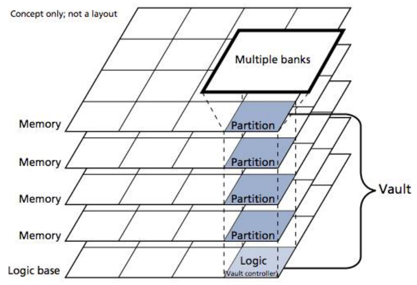

Fig. 3. Architectural view of 3D stacked memory with logic layer

We consider HBM as an example: it exposes each channel as a raw DDR-like interface; HMC implements

DRAM controller in the logic layer as well as SerDes links for off-stack communication. The term *vault* is used to describe the vertical channel in HMC, including the memory banks and its separate DRAM controller. 3D-stacked memory provides high bandwidth through low-power TSV-based channels, while latency is close to normal DDR3 and DDR4 chips due to their similar DRAM core structures [14].

TABLE I
MEMORY TECHNOLOGY COMPARISON (BASED ON DATA PROVIDED BY MICRON [15])

	Pins	Bandwidth	Power
DDR3	143	12.8 GB/s	6.2 W
DDR4	148	25.6 GB/s	8.4 W
HMC	128	80.0 GB/s	13.4 W

1) **Physical Characteristics of 3D-stacked Memory:** The Table I illustrates a comparison between DDR3, DDR4, and HMC-style baseline designs from three aspects, in terms of power, bandwidth, and pin-count. 3D stacked memory, which has better bandwidth-per-pin and bandwidth-per-watt characteristics than either DDR3 or DDR4, is optimized for high-bandwidth operation and targets workloads that are bandwidth-limited. According to the HMC 1.0 specification [15], a single HMC provides up to 320 GB/s of *external* memory bandwidth through eight high-speed serial links. On the other hand, a 64-bit vertical interface for each DRAM partition or *vault*, 32 vaults per cube, and 2 Gb/s of TSV signaling rate [16] together achieve an *internal* memory bandwidth of 512 GB/s per cube. Moreover, this gap between external and internal memory bandwidth becomes much wider as the memory capacity increases with the used of more HMCs. The key objective of adopting near-memory processing is not solely to provide high memory bandwidth, but especially to achieve *memory-capacity-proportional* bandwidth. Considering a system composed of 16 8GB HMCs as an example, conventional processors are still limited to 320 GB/s of memory bandwidth assuming that the CPU chip has the same number of off-chip links as that of an HMC. In contrast, a NDC system exposes 8 TB/s (=16 $\times$ 512 GB/s) of aggregate internal bandwidth to the in-memory computation units. This memory-capacity-proportional bandwidth facilitates scaling the system performance with increasing amount of data in a cost-effective way, which is a key concern in irregular data access patterns, such as graph processing [17].

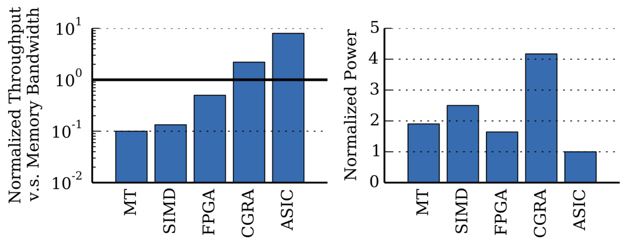

Fig. 4. Computational throughput and power consumption for different logic types implementing the graph processing kernel: multi-threaded (MT) cores, SIMD cores, FPGAs, CGRAs, and custom units (ASICs).

B. Logic Layer and Compute Units

The logic layer at the bottom of 3D-stacked memory contains the *Near-Data Cores* (or *NDCores*) and is the key component to fulfill the *Near-Data Computing*. The NDCores can characterize the quality of a system design in the following manner. These NDCores should (1) be *area-efficient* in order to provide sufficient computational throughput to match the high bandwidth available through 3D stacking; (2) be *power-efficient* in order to reduce total energy consumption, and to avoid causing thermal issues in the DRAM stacks; (3) provide *high flexibility* to allow for reuse across multiple applications and applications domains. Recently proposed designs implement the compute units inside a *NDCore* with simple cores, SIMD units, GPUs, FPGA blocks, CGRA arrays, or custom accelerators (ASICs).

Figure 4 compares different logic options available to NDC systems on which a typical kernel of graph processing is executed [18]. To be fair in this experiment, all these options utilize the same memory configuration. Each memory stack possesses 8 vaults (or channels) and supplies up to 128 GBps peak bandwidth in total, but, once memory controllers and interfaces are accounted for, only about 50 mm^2 of the logic layer can be used for compute units [19]. Whenever the area and power budgets allow it ought to put as many logic units as possible in the stack. The computational throughput in Figure 4 is normalized to the maximum memory bandwidth, indicated by the solid line. Matching this line implies a balanced system with sufficient compute capability to take full advantage of the high bandwidth through TSVs. Once the memory bandwidth is saturated there is no point in scaling the computational throughput any further.

1) **General-purpose Processor as NDCore:** This category of *NDCores* design takes the same philosophy as *many-core processor* design. The logic layer under the 3D-stacked memory contains one simple processor core for each vault; as we previously mentioned, one stack has 32 vaults, therefore, each stack holds 32 cores [16], [17], a typical design is shown in Figure 5. **The architecture of NDCore.** Most designs which fall into this category use simple in-order cores similar to ARM Cortex-A series as *NDCores* [20], [21], such as the one shown in Figure 5 (c). Wide-issue or Out-of-Order cores are not necessary due to the lack of temporal locality and abundant instruction-level parallelism in code, nor are they practical given the stringent power and area constraints. For some workloads – particularly for irregular and unbalanced workloads, such as graph processing – some NDCores may suffer from under-utilization as they are often stalled waiting for data processed by other NDCores. Therefore, fine-grained *multi-threading* (cycle-by-cycle) should be used as a cost-effective way to improve the utilization of simple cores given a considerable amount of memory bandwidth available within each stack [22], [23].

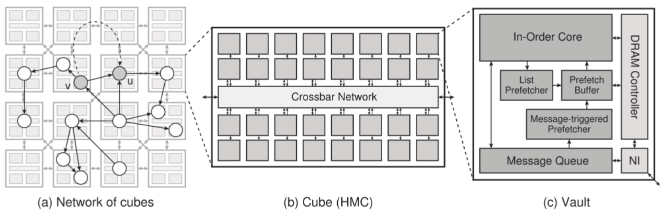

Fig. 5. A typical NDC system with general-purpose processor cores as *NDCores* [17].

The communication among NDCores. Another matter for *NDCores* of this category is the handling of communication among cores internally in a stack and externally between stacks, respectively. Many real-world applications, such as graph processing and deep learning, require complex communication between hundreds to thousands of threads [24], [25]. Relying on the host processor to manage all the communication will not only turn it into the performance bottleneck but will also waste energy moving data between the host processor and memory stacks.

To solve this issue, some elegant solutions are proposed. Tesseract [17] employs a low-cost message passing mechanism plus prefetching for communication between *NDCores*. Tesseract chooses message passing to (1) avoid cache coherence issues, (2) eliminate the need for locks to guarantee atomic updates of shared data, and (3) overlap the message passing with computation to hide the latency of remote access. While, another design [23] proposes a per-core *remote load buffer* (RLB) mechanism, that allows sequential prefetching and buffering of a few cache lines of *read-only* data. This is manageable because flushes are only necessary at synchronization points such as barriers, as a result no writeback is needed.

2) **Accelerator as NDCore:** *Near-Data* cores composed by SIMDs (mostly GPUs) or ASICs are considered as accelerators. Both SIMD or ASIC architectures are suitable for specific fields of applications opposite to general purpose ones, thus it is appropriate to put them into the accelerator category.

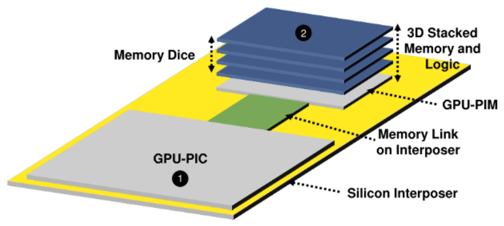

Fig. 6. NDC-assisted GPU architecture [26]

GPU core as NDCore. A GPU-based NDC system consists of at least one 3D-stacked memory chip and a powerful GPU chip placed adjacent to memory stack. The 3D-stacked memory integrates a base logic layer, housing a chip with GPU cores. Both chips are interconnected through a high-speed serial memory link on an interposer, as demonstrated in Figure 6. This architecture contains two types of compute engines: (1) large and powerful primary GPU cores, which is similar to modern GPU cores, and (2) smaller and less powerful GPU cores, which are placed in the logic layer under 3D-stacked memory and assist in performing the computation. The GPU integrated into memory stack is more power-efficient than the main GPU and this organization has considerably lower thermal constraints and is more scalable regarding memory capacity. In order to fully and automatically exploit the performance and energy-efficiency of the system, mechanisms of how to automatically identify the code segments to be offloaded to the GPU cores in memory stack are needed. To maximize the concurrency, a smart strategy has to be adopted for scheduling multiple kernels on the main GPU and the cores in memory. For example, computations that are memory-intensive and can tolerate the lower parallelism presenting in the logic layer of GPUs are likely better executed on GPU cores in memory stack instead of on main GPU.

ASIC as NDCore. In Figure 4, custom accelerators (ASICs) show an extreme as they can easily saturate the memory channels but lack programmability. The extra compute capability provides marginal benefits [18], [27]. ASICs are usually made for digital signal processor (DSP), and they provide very small silicon footprint and much lower power consumption compared to other types of NDCores.

Due to limited functionalities of ASICs, there is only a small number of designs considering to adopt ASICs in NDC system, however their stunning performance and low power potential still attracts research on some specific applications. A typical example is the customized compute engine designed for MapReduce framework. Both *Mapper* and *Reducer* phases need large quantities of sorting operations, so a customized sorting engine will offer great help. [28] presents the hardware accelerator for merge sort and bitonic sort. The merge sorter follows the *divide and conquer* paradigm and is internally organized like a heap, and each cycle the smallest of two input key-value pairs will be chosen to advance to the next level; the input stream buffers must keep track of the sorted input lists. Merge sorters are mainly used for small size input, each merge sorter works on an input with 2 to 8 elements. On the other hand, A bitonic sorter can sort 1024 inputs, and it is either used as a finishing step to finalize the sorting that has been partially accomplished by a hardware range partitioning unit, or used as a pre-processing step to prepare an input to be operated on by a group of eight-input hardware merge sorters.

3) **Reconfigurable Compute Unit as NDCore:** The third category of logic units available to be used in a logic layer are reconfigurable units, they provide a good trade-off between performance and flexibility. There are two common reconfigurable logic types, FPGA and CGRA.

Architectural View of Reconfigurable Compute Unit. FPGA achieves low power consumption through customization but suffers from low clock speed and high area overheads due to the bit-level configurable blocks and routing fabric, which are too fine grained for arithmetic operations, The DSP blocks in modern FPGAs help unless their number and width match emerging big data applications. This is one drawback shown in the left diagram of Figure 4, it indicates that FPGA may fail to saturate memory bandwidth with high performance under the area constraints.

Another type of reconfigurable compute units, called *Coarse-Grain Reconfigurable Accelerators* (or CGRAs), can be deployed to the logic layer of a memory stack. A CGRA is typically comprised of a

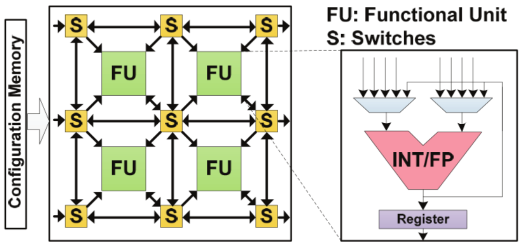

Fig. 7. A CGRA with a grid of 2×2 functional units [29].

large number of coarse-grained functional units (FUs) in a grid which perform various arithmetic and logic operations, which are connected by a configurable interconnect fabric (Figure 7); CGRAs also significantly reduce the power and area overheads for computation. However, due to the limitation of the simple interconnect, traditional CGRAs only target applications with regular computation patterns, such as matrix multiplication or image processing. Recent NDC systems [29], [18] used some new types of CGRA with either circuit-switch routing network [29] or CLB [18] as a powerful interconnect to support more complicated data and control flows. The CLBs in [18] (Figure 8, which the author calls *Heterogeneous Reconfigurable Logic*), are similar to those in FPGAs, they utilize lookup tables (LUTs) for arbitrary, bit-level logic. With the help of these CLBs, the NDCores can implement control units which are less regular, e.g. load and store unit with indirect addressing mode used in data access pattern of sparse structure.

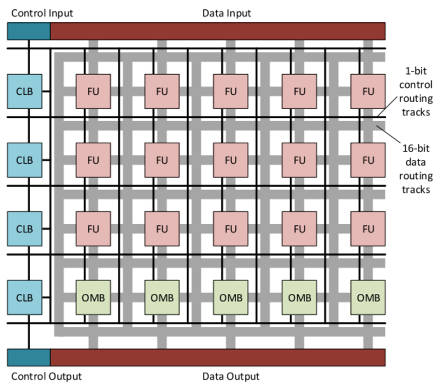

Fig. 8. The organization of the Heterogeneous Reconfigurable Logic (HRL) array [18]

Computational Model of Reconfigurable Compute Unit. Data-intensive and compute-intensive application kernels can be decomposed into *dataflow graphs*. A *dataflow graph*, containing nodes (operations) and edges (data transfer or communication), is mapped to the FPGA's DSP or CGRA's FUs and interconnect either manually or using automated techniques. The host processor triggers FPGA or CGRA reconfiguration at runtime to implement different kernels at different times, replacing the content of configuration memory with the new dataflow graph's configuration data. Exploiting spatial data parallelism in application kernels and efficiently processing kernel's dataflow graphs; reconfigurable compute unit, e.g. CGRAs, considerably improve performance and energy consumption compared to conventional processors, as shown in Figure 4. In particular, CGRAs can partially eliminate the large energy overheads of fetching and scheduling instructions in conventional out-of-order processors. While, FPGA alike logic, e.g. CLBs in FPGA, provides more flexible programmability in interconnect.

C. Host

Most NDC systems use a high-end processor with out-of-order (OoO) cores as host, while the GPU NDC system uses advanced and powerful GPU as host. As stated in previous section, most of NDCores don't support virtual memory, nevertheless, host processors can still use virtual addressing in their main memory since they use separate DRAM devices. Since host processors have access to the entire memory space of system, including 3D-stacked memory, it is up to the host processors to distribute workloads across HMC vaults.

III. Performance and Energy Analysis

The work in [23] shows the experiments on power and performance comparison. The authors used a conventional DDR3 system as a baseline (Conv-DDR3). They made a comparison among the system of 3D stacking without logic layer (Conv-3D), the base NDC system, and an advanced NDC system (2 threads per vault, 16 vaults per stack, 8 stacks, 1024 threads in total). Figure 9 illustrates the performance and energy comparison between the above four systems. The Conv-3D system supplies significantly higher bandwidth than the DDR3 baseline due to the use of high-bandwidth 3D memory stacks. However, the two NDC systems deliver significant improvement over both the Conv-DDR3 and the Conv-3D systems in terms of performance and energy. The base-NDC system achieves around 3.5$\times$ faster 3.4$\times$ more energy efficient over DDR3 baseline. The advanced NDC system attains another 2.5$\times$ improvement over the base-NDC and accomplishes 3-16$\times$ better performance and 4-16$\times$ less energy over the base-DDR3 system.

Both NDC systems consume significantly low energy due to that 3D-stack memory removes the distance of data transfer across long traces on board. Another factor affects energy rising from DDR memory's dynamic recharging; highly-threaded multi-core/many-core architecture can greatly reduced the total cycles of execution, hence, NDC systems eliminate much more stalled cycles where dynamic energy dissipation happen. The improvement in speed for both NDC systems is obvious, *higher bandwidth* and *shorter distance* for data transfer play crucial roles.

IV. Algorithmic Level Optimization

Balancing workloads in vaults and reducing nontrivial data transfers among vaults are two critical issues which seriously affect the performance and power consumption of NDC systems. Workload balancing concerns how computation is distributed to each vault and how data is partitioned and processed then stored in each vault. Whenever it is necessary, one vault may need to access the data stored in another vault to continue its own computation; each request and response pair for data transfer involves non-trivial communication through memory banks and a serial link between memory stacks and costs power consumption while transmission of data and signaling. So workload balancing and non-trivial communication reduction are very important for NDC system design as well as in software level.

A. Data Partition and Workload Balancing

A better data partitioning for balancing workload on each NDCore can effectively improve the overall performance of system by (1) getting a higher concurrency, (2) reducing the total amount time when some NDCores are idle while others are engaged in performing some tasks, and (3) minimizing the number of data transfers between different vaults.

The workload balancing can be broadly classified into two categories: ***static*** and ***dynamic***. Static workload balancing strategy is simple to realize with the help of the compiler and the OS, for many practical cases, relatively inexpensive heuristics provide fairly acceptable approximate solutions to the optimal static workload balancing problem. While dynamic balancing strategy can be more flexible and versatile, especially when static strategy may result in a highly imbalanced distribution of data and workload partition. However, it is more complicated to realize a dynamic strategy and also requires some hardware modifications to support those mechanisms used in the dynamic strategy.

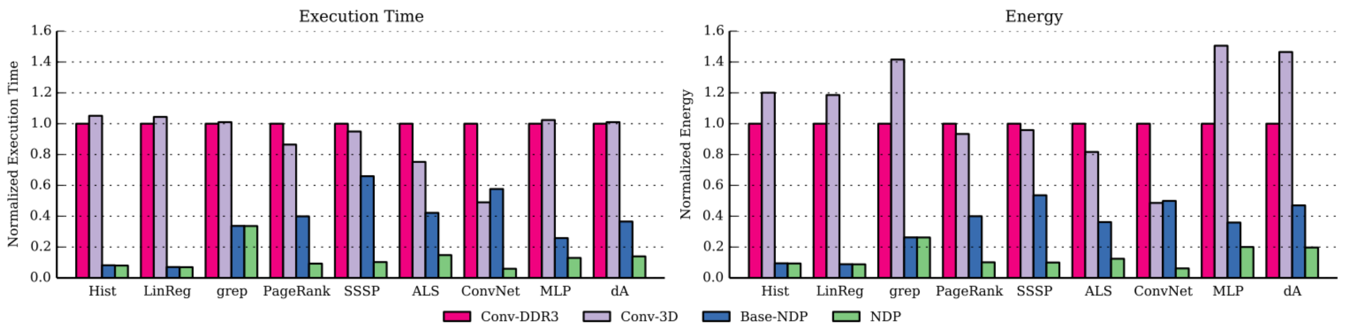

Fig. 9. Performance and energy comparison between Conv-DDR3, Conv-3D, Base-DNC and advanced DNC systems [23].

B. Case studies

In the following subsection, we give some case studies to show how workload balancing and algorithmic level optimization affect the performance and power consumption of NDC system.

1) **Case 1: Static workload balancing in graph processing:** A sequential memory access pattern provides the most energy efficient way to exploit spatial locality, further maximizes the cache utilization, avoids remote data transfer between NDCores, and also reduces the energy overhead of fetching cache line; these merits are all expected by graph processing. Figure 10 compares the performance, energy, cache line utilization, and DRAM row utilization of two implementations of a graph framework (The kernel of the framework is an implementation of *Single Source Shortest Path* (or *SSSP*) algorithm). The edge-centric implementation provides a 2.9× improvement in both performance and energy over the vertex-centric. The key advantage is that the edge-centric scheme enhances spatial locality (streaming, sequential accesses). The cache line utilization histogram indicates that the higher spatial locality is converted into a higher fraction of the data used within each cache line read from memory, and a lower total number of cache lines that need to be fetched [23].

From the algorithmic perspective, in each iteration SSSP searches for the edge to a vertex that appears closest to a source. Vertex-centric graph partitioning divides the set of vertices (adjacent lists) into several partitions then loads them in the vaults of the memory stack. SSSP needs to frequently access the weight information of edges. When two vertices of an edge were dispatched in different vaults (as shown in Figure 5), the remote request for weight information sent to these two vaults is inevitable. The action of sending request costs delay while waiting for the reply and eventually the degradation of performance. It will be even worse when too many remote requests are issued then the resulting communication congestion deteriorates performance and power consumption. On the other hand, edge-centric implementation can avoid unnecessary remote requests for edge information since the necessary information stored in the same vault, therefore cross-vault communications issued by an edge-centric implementation are less than the counterpart. Of course, edge-centric implementation has its own drawback, that is, the data structure used by edge-centric implementation requires redundant storage. Therefore, this overhead occupies more storage space in each vault.

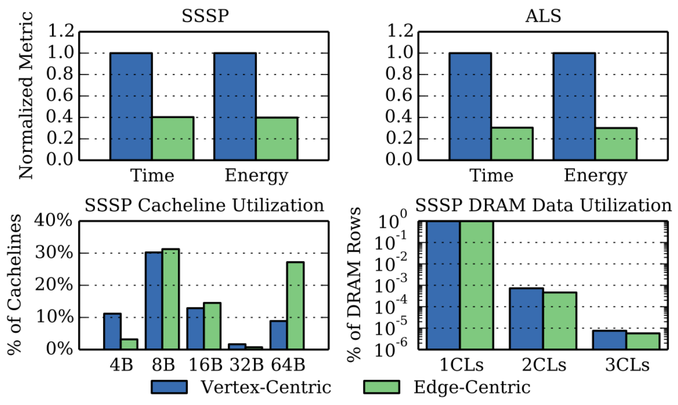

Fig. 10. Vertex-centric and edge-centric graph frameworks. [23]

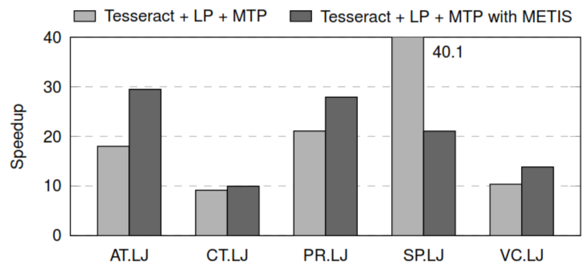

Fig. 11. Performance improvement after graph partitioning. [30]

Elaborate graph partitioning schemes should be employed for balancing the workload in each vault. A better data partitioning can minimize communication between different vaults and between different memory stacks. Figure 11 shows the performance improvement of Tesseract [30] where the input graphs are evenly distributed across vaults based on an optimized graph partitioning algorithm. For the purpose of shortening the critical path (the more workload are balanced, the shorter the critical path), the author employs METIS [31] to perform 512-way multi-constraint partitioning to balance the number of vertices, outgoing edges, and incoming edges of each partition. Therefore, employing better graph distribution can further improve the performance of NDC system. This is because graph partitioning minimizes the number of edges crossing between different partitions (53% fewer edges cuts compared to random partitioning in LJ as shown in Figure 11), and thus, reduces off-chip network traffics.

2) **case 2: Dynamic Strategy of Workload Balancing:** Dynamic workload balancing techniques are usually used for the applications with the unpredictable workload at runtime. And in some cases, the workload can also be generated dynamically, e.g. reinforcement learning problems. In the work of [30], the author proposed a dynamic strategy which offloads workload to NDC-assisted GPU platform. Their strategy first divides the workload according to the *computational kernels*, these kernels will be executed concurrently. Furthermore, each kernel takes different execution time on NDCore from on main GPU (recall in section II.B.2, the GPU-based NDC system consists of a powerful main GPU (or GPU-PIC, PIC stands for Processing in core), and a single memory stack or several memory stacks integrated with low-power simple GPU cores (NDCores or GPU-PIM, PIM stands for processing in memory)). Therefore, to make the workload executed fast and efficiently on this GPU-based NDC system, we need to conceive an algorithm to offload each kernel of the workload to an appropriate compute node, either on main GPU or on the NDCores.

Figure 12 illustrates the advantage of kernel offloading over application offloading for FDTD (Finite-Difference Time-Domain) application. Scenario-I and Scenario II are the two possible application offloading strategies executing the entire FDTD application on GPU-PIM or GPU-PIC, respectively. On the other hand, in the kernel offloading strategy (Scenario-III), each *kernel* of FDTD is offloaded to the computation engine where its execution time is lower (i.e., each kernel is offloaded to a unit it has *affinity* towards). Therefore, Kernel-K1 is offloaded to GPU-PIM and the other two kernels are offloaded to GPU-PIC. Kernel offloading saves many execution cycles (Ⓐ) even beating the best application offloading strategy. However, the key challenge of kernel offloading is in identifying the *affinity* of each kernel. Another benefit of the offloading

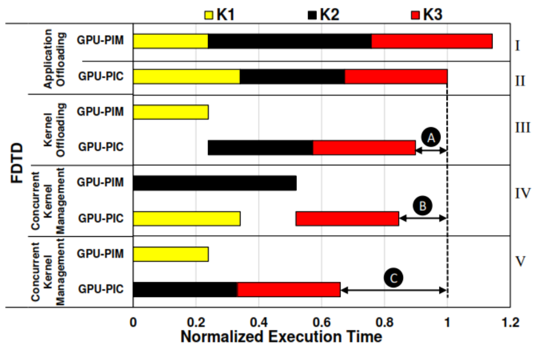

Fig. 12. Performance advantages of kernel offloading (III) and concurrent kernel management (IV and V) mechanisms using the FDTD application as an example.

workload in kernel level is that we can manage the kernel scheduling at fine-grained granularity. Whenever the kernels are independent, they can be executed concurrently by utilizing as many compute units as possible. Figure 12 demonstrates an example of this in Scenario IV and Scenario V. In FDTD, Kernel-K3 depends on both Kernel-K1 and Kernel-K2, so it can begin only when both Kernel-K1 and Kernel-K2 have done their executions. However, Kernel-K1 and Kernel-K2 can execute in parallel. Due to the concurrent execution of kernels, overall application execution time is decreased (Ⓑ), compared to the best application-offloading scenario, in scenario-IV. Scenario-V in Figure 12 illustrates that kernel affinity, i.e., scheduling each kernel on the execution engine that is best for the kernel's performance matters: Kernels K1 and K2 have affinity for GPU-PIM and GPU-PIC, respectively. Further, executing them concurrently on these engines leads to even higher overall application execution time savings (Ⓒ) than in Scenario-IV where the same kernels are scheduled onto the opposite engines.

V. CONCLUSION

(importance section, phrase nicely) In this paper we revisit the *Near-Data Computing* (NDC) system from architectural and technological perspectives. We also present case studies demonstrating performance optimization. As 3D integration technology becomes more and more(avoid more and more) mature and reliable, it is no longer the technical adoption barrier for implementing computer architecture with near memory computation ability. NDC offers a promising approach to overcome the challenges posed by emerging data-intensive applications. Moreover, a viable NDC system should be constructed by collaborative efforts between technologies(IC designers and engineers are persons, but technologies?), IC designers and system engineers, as what we have demonstrated in case study section. The simulation results show that NDC significantly improves the performance of a wide range of evaluated applications (3.2-60.6$\times$) and reduces their total energy consumption (63-96%).

REFERENCES

[1] P. M. Kogge, "Execube-a new architecture for scaleable mpps," in *Proceedings of the 1994 International Conference on Parallel Processing - Volume 01*, ICPP '94, (Washington, DC, USA), pp. 77–84, IEEE Computer Society, 1994.

[2] M. Gokhale, B. Holmes, and K. Iobst, "Processing in memory: The terasys massively parallel pim array," *Computer*, vol. 28, pp. 23–31, Apr. 1995.

[3] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park, "Mapping irregular applications to diva, a pim-based data-intensive architecture," in *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, SC '99, (New York, NY, USA), ACM, 1999.

[4] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent ram," *IEEE Micro*, vol. 17, pp. 34–44, Mar. 1997.

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

[6] D. G. Elliott, M. Stumm, W. M. Snelgrove, C. Cojocaru, and R. Mckenzie, "Computational ram: implementing processors in memory," *IEEE Design Test of Computers*, vol. 16, pp. 32–41, Jan 1999.

[7] A. Papanikolaou, D. Soudris, and R. Radojcic, *Three Dimensional System Integration: IC Stacking Process and Design*. Springer Publishing Company, Incorporated, 1st ed., 2010.

[8] Y. Liu, W. Luk, and D. Friedman, "A compact low-power 3d i/o in 45nm cmos," in *2012 IEEE International Solid-State Circuits Conference*, pp. 142–144, Feb 2012.

[9] M. G. Farooq, T. L. Graves-Abe, W. F. Landers, C. Kothandaraman, B. A. Himmel, P. S. Andry, C. K. Tsang, E. Sprogis, R. P. Volant, K. S. Petrarca, K. R. Winstel, J. M. Safran, T. D. Sullivan, F. Chen, M. J. Shapiro, R. Hannon, R. Liptak, D. Berger, and S. S. Iyer, "3d copper tsv integration, testing and reliability," in *2011 International Electron Devices Meeting*, pp. 7.1.1–7.1.4, Dec 2011.

[10] Micron, "Hybrid Memory Cube - a revolution in memory," 2014.

[11] AMD, "High Bandwidth Memory - reinventing memory technology," 2015.

[12] H. M. C. Consortium, "Hybrid memory cube specification 2.1," 2014.

[13] J. Standard, "High Bandwidth Memory (HBM) DRAM - jesd235a," 2013.

[14] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin, J. H. Cho, K. H. Kwon, M. J. Kim, J. Lee, K. W. Park, B. Chung, and S. Hong, "25.2 a 1.2v 8gb 8-channel 128gb/s high-bandwidth memory (hbm) stacked dram with effective microbump i/o test methods using 29nm process and tsv," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 432–433, Feb 2014.

[15] H. M. C. Consortium, "Hybrid memory cube specification 1.0," 2013.

[16] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "Ndc: Analyzing the impact of 3d-stacked memory+logic devices on mapreduce workloads," *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 190–200, 2014.

[17] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 105–117, ACM, 2015.

[18] M. Gao and C. Kozyrakis, "Hrl: Efficient and flexible reconfigurable logic for near-data processing," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 126–137, March 2016.

[19] J. Jeddeloh and B. Keeth, "Hybrid memory cube new dram architecture increases density and performance," in *2012 Symposium on VLSI Technology (VLSIT)*, pp. 87–88, June 2012.

[20] A. limited, "Cortex-A5: The cortex-a5 processor is the smallest, lowest power armv7 application processor.," 2013.

[21] A. limited, "Cortex-A7: The arm cortex-a7 processor is the most efficient armv7-a processor.," 2014.

[22] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The tera computer system," in *Proceedings of the 4th International Conference on Supercomputing*, ICS '90, (New York, NY, USA), pp. 1–6, ACM, 1990.

[23] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT '15, (Washington, DC, USA), pp. 113–124, IEEE Computer Society, 2015.

[24] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, (Berkeley, CA, USA), pp. 17–30, USENIX Association, 2012.

[25] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, (Berkeley, CA, USA), pp. 571–582, USENIX Association, 2014.

[26] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das, "Scheduling techniques for gpu architectures with processing-in-memory capabilities," in *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pp. 31–44, Sept 2016.

[27] G. H. Loh, N. Jayasena, M. H. Oskin, M. Nutter, D. Roberts, M. Meswani, D. P. Zhang, and M. Ignatowski, "A processing-

in-memory taxonomy and a case for studying fixed-function pim," 2013.

[28] S. H. Pugsley, A. Deb, R. Balasubramonian, and F. Li, "Fixed-function hardware sorting accelerators for near data mapreduce execution," *2015 33rd IEEE International Conference on Computer Design (ICCD)*, pp. 439–442, 2015.

[29] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 283–295, Feb 2015.

[30] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 105–117, June 2015.

[31] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, pp. 359–392, Dec. 1998.