In-memory data rearrangement for irregular, data intensive computing

Scott Lloyd and Maya Gokhale
Lawrence Livermore National Laboratory
gokhale2@llnl.gov

***Abstract*—As CPU core counts continue to increase, the gap between compute power and available memory bandwidth has widened. A larger and deeper cache hierarchy benefits locality-friendly computation, but offers limited improvement to irregular, data intensive applications. In this work we explore a novel approach to accelerating these applications through in-memory data restructuring. Unlike other proposed processing-in-memory architectures, the rearrangement hardware performs data reduction, not compute offload. Using a custom FPGA emulator, we quantitatively evaluate performance and energy benefits of near-memory hardware structures that dynamically restructure in-memory data to cache-friendly layout, minimizing wasted memory bandwidth. Our results on representative irregular benchmarks using the Micron Hybrid Memory Cube memory model show speedup, bandwidth savings, and energy reduction in all cases. Application speedup ranges from 1.24X to 4.15X. The number of bytes transferred is reduced by up to 11.69X, reflecting the efficiency of data rearrangement. Energy improvement ranges from 1.49X to 2.7X. We analyze the effect of memory access at an 8-byte granularity, and find energy reduction possible of up to 7.84X.**

1. Introduction

The memory wall is perhaps the most prominent obstacle threatening our ability to analyze expanding data volumes. While CPU innovations deliver Teraflop compute nodes, irregular memory access prevents many data intensive workloads from achieving corresponding performance improvements. It remains a continuing challenge to keep the CPU cores doing useful work since memory bandwidth improvements mainly benefit regular, streaming access patterns. If only one eighth of every cache line fetched from memory is used – 8 bytes of a 64 byte cache line – memory bandwidth and power are wasted, and performance suffers.

We have developed a novel method to dynamically lay out data in memory in the form that the application needs, when the application needs it. Our proposed Data Rearrangement Engine (DRE) dynamically rearranges data in memory to create a cache-optimized layout so that the CPU uses every byte of every cache line in the rearranged data structure. Our approach uses near-memory hardware logic implementable in a logic layer of emerging 3D memory packages such as the Hybrid Memory Cube [1]. It effectively exploits vast bandwidth internal to the memory to maximize use of limited off-package bandwidth.

Using an FPGA-based emulator built for this purpose, we present quantitative assessment of performance and energy improvements through in-memory data rearrangement on representative data intensive analytics benchmarks. Our system architecture using simple DMA and gather/scatter hardware delivers application speedup from 1.24X to 4.15X, memory bandwidth reduction greater than an order of magnitude, and energy improvement from 1.49X to 2.7X. Remarkably, energy reduction of up to 7.84X is possible when memory can be accessed in 8-byte units.

1.1. Irregular applications and cache

As CPU clock frequency plateaus, multi- and many-core architectures with heterogeneous compute units have emerged as the norm, enabling continued improvement in peak FLOPS through spatial parallelism. To mitigate the gap between greatly increased peak FLOPs and more modest improvement in memory bandwidth, many-core CPUs incorporate deep cache hierarchies to increase the likelihood that applications' memory accesses will be satisfied in cache. However, memory intensive applications with little spatial or temporal access locality may not benefit from cache hierarchies. For these applications, the compute units often sit idle waiting for data.

For example, sparse matrix/vector operations form the core of the popular PageRank algorithm, which traverses a web graph to locate frequently referenced web pages. As shown in Figure 1, the algorithm accesses random locations in the graph and pagerank vector. Ideally the CPU cache will only hold pagerank vector entries corresponding to a vertex's edgelist, as shown on the left.

DMA and gather/scatter hardware integrated with the CPU allows the CPU to initiate data structure rearrangement, but the full data structure must still traverse the memory bus. In our memory-integrated approach, DREs are on the memory package, and only the restructured data traverses the memory bus.

1.2. Memory-integrated computing

The concept of integrating logic with memory has been investigated for many years [2], [3], [4], [5], [6]. With this

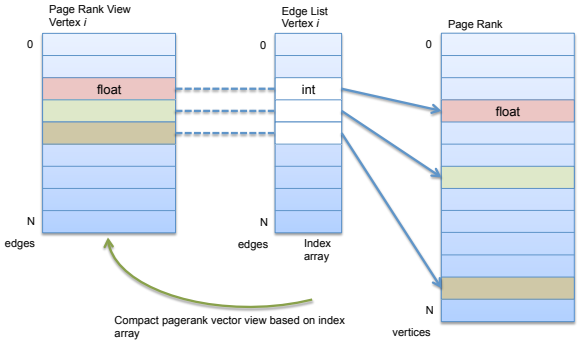


Figure 1: View assembly: The DRE assembles a page rank view based on the adjacency list for vertex i which is an index array into the full page rank vector.

technology, computation that is typically handled by a CPU is performed within the memory system. Performance is improved and energy reduced because processing is done close to the data without having to move data across chip interconnects from memory to the processor.

There have been many proposed designs for compute elements in memory. However, previous designs were predicated on the integration of logic in the same fabrication process as DRAM cells, and did not become commercially viable due to cost. With the advent of 2.5D and 3D packaging, placing compute elements directly within the DRAM is no longer necessary. Stacking allows logic to be placed on a separate layer in a separate fabrication process, which makes memory-integrated computing structures more attractive than in a purely DRAM fabrication.

Computing with 3D stacked memory has been explored using the logic layer of the Hybrid Memory Cube to hold processor arrays [7]. The fundamental HMC architecture consists of a stack of DRAM layers connected to a base logic layer with through silicon vias. The logic layer contains an integrated memory controller. In the HMC design, the memory controller receives and transmits packetized data over a custom link protocol to an external (off-package) unit such as a CPU. In proposed HMC processor designs, throughput-oriented processors are located in the logic layer and operate on streaming data directly from DRAM banks. In contrast, we perform data reduction, not compute offload, in memory, similarly to FPGA-based data reorganization proposed by Diniz et al. [8].

1.3. Data rearrangement engines

Unlike previous proposals that place full functionality processor arrays in memory, we propose lightweight data rearrangement engines (DRE) that assist the main CPU by creating cache-friendly data blocks on demand in an in-memory buffer. In our design, load and store requests from the CPU traverse the normal cache hierarchy. DREs are explicitly invoked by applications to rearrange data within the memory. When a DRE command finishes, the application accesses the rearranged data buffer, which then traverses the cache hierarchy instead of the original data blocks. For example, upon request, a DRE would create a "gathered" subset of the pagerank vector in memory containing entries for a vertex's edge list, and the application would access that reduced vector. Since the application needs every byte transferred, cache lines are fully utilized. As an additional benefit, computation on the restructured data can be vectorized, which isn't possible when the data is scattered in memory.

2. Architecture

In our scheme an application is partitioned between restructuring data in memory and computing with data in the CPU. During the course of execution, a DRE library routine assembles the desired view of a data structure fragment in a memory-resident SRAM buffer, the main application in the CPU reads and updates the buffer, and the DRE stores from the buffer back to DRAM. The DRE's application-specific restructuring creates or stores a compact view of a data structure, and the CPU program loads, computes on, and stores that compact view. The DRE microarchitecture and associated API support this workflow.

2.1. Microarchitecture

The DRE has been designed to be compatible with the HMC memory organization. On the HMC, DRAM banks are laid out in vertically structured *vaults*. A vault atomically accesses a 32-byte unit. The vaults are interconnected with each other and with a link controller that is responsible for serializing and de-serializing the data going to or coming from the memory package. The HMC link protocol can handle multiple data packet lengths from 128 bytes down to 16 bytes. The HMC has multiple high speed links to connect to the CPU and potentially to other HMC devices for greater capacity. In the latter configuration, inbound memory requests are either routed to an internal vault or forwarded to the destination on a pass-thru link. The current HMC packet protocol only supports requests coming from an external host; however, a possible extension to the protocol and interconnect would allow a DRE to originate memory requests from within a device and access data structures transparently across a large distributed memory.

Figure 2 shows a notional diagram of an HMC-like package in which data rearrangement engines are also attached to the interconnect. A DRE consists of a programmable DMA unit (load/store unit or LSU) along with a microcontroller (MCU) that executes a simple set of commands as directed by an application. The MCU program orchestrates LSU actions by generating command messages consisting of addresses and lengths for the LSU to fetch/store in either a fixed stride pattern or as specified in an associated stream of indexes. The LSU uses an SRAM buffer as a scratchpad. This buffer can also be directly addressed by the main

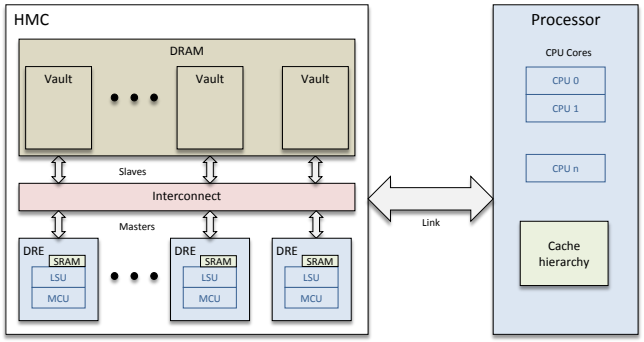


Figure 2: Data rearrangement engines connect to the internal interconnect. Each DRE holds a programmable reorder engine and an SRAM buffer.

CPU and serves as a shared view buffer for communication between CPU and DRE.

2.2. API

Upon request, an application process acquires a DRE. The application specifies a microcontrol unit (MCU) program, and the operating system loads the program into MCU instruction memory, which is also in the logic layer. The application and MCU communicate with small messages: the application issues commands and receives completion notification by writing and reading a memory-mapped address range. Commands include:

setup to load parameters, such as base addresses and either DMA size and stride for DMA operations, or index vector size and base address for gather/scatter;

fill to copy from DRAM to the SRAM buffer according to the access pattern established during setup;

drain to copy from the SRAM buffer into DRAM according to the access pattern established during setup.

2.3. CPU interaction

The CPU and DRE exchange control messages: the CPU issues commands and awaits completion, and similarly the DRE waits for commands, executes them, and notifies completion. A range of reserved memory addresses is used to communicate parameters and completion flags. Polling is used to check for completion.

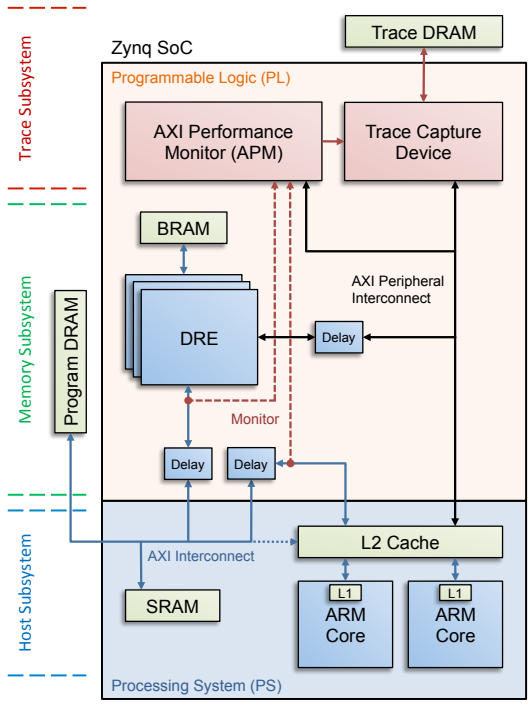
The CPU and DRE components cooperate to maintain cache/DRAM consistency by issuing cache flush and invalidate operations at well defined synchronization points such as preceding and following **fill** operations. Cache consistency operations are explicitly invoked by application code and target the cache in either the CPU or in the DRE's microcontrol unit. The flush/invalidate is done to the entire cache or to an address range, depending on the size of the updated region. Our implementations select the most efficient option. The overhead of maintaining consistency must be factored into evaluating the potential benefit of DREs.

The CPU's Memory Management Unit (MMU) translates process virtual addresses to physical memory addresses. Memory requests from the DRE must also be translated, which requires that the DRE have its own address translation table. A general mechanism of mirroring the CPU MMU is done in graphics processors, high performance networking such as Infiniband, and some processing-in-memory proposals [7]. We propose a simpler, albeit more restrictive approach and require that the application allocate data to be accessed by the DRE in contiguous physical pages. This can be accomplished by using a custom allocator to gather a large contiguous physical range, as is being developed in transparent large page support in the OS, and by pinning pages (commonly done by network interfaces in High Performance Computing systems) to prevent subsequent relocation. The **setup** command gives the base physical page address, and the DRE adds the base to each address being loaded or stored. This requires minimal additional hardware, adds no performance overhead as the address can be assembled as it is loaded onto the request queue, and virtually no energy overhead since it involves a simple concatenation of bit fields. Addresses outside that range would then trigger return of control to the application to assemble a new continguous range for the DRE to access.

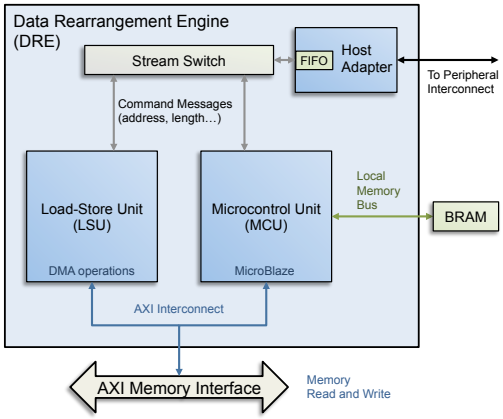
3. Emulator

3.1. Data flow

To quantitatively evaluate the performance and energy usage of in-memory data reordering engines, we have developed an FPGA emulator modeling a CPU and DRE. The emulator is implemented on a Xilinx Zynq 7000 System-on-Chip. The Zynq block diagram and emulation framework are shown in Figure 3a. The Zynq SoC has two main components, the Processing System (PS) and Programmable Logic (PL). In the emulation, the ARM A9 cores in the PS run the application and use a dedicated memory, a 1 GB DRAM (labeled Program DRAM) that holds instructions and data. The DREs (Figure 3b) are implemented in FPGA logic in the PL, which also holds the emulation infrastructure to non-intrusively capture memory traffic. Although there is a direct path on the Zynq from the ARMs to Program DRAM, during emulation memory requests from the ARMs not satisfied in cache are routed through the PL. This enables the trace subsystem to monitor the ARMs' memory accesses to Program DRAM. The DREs also issue memory requests to Program DRAM, and these are also captured. The AXI Performance Monitor filters the memory requests and forwards them to the Trace Capture Device for storage in the 1 GB Trace DRAM. The SRAM (on the PS side) is used to emulate the SRAM scratchpad in the DRE by looping SRAM accesses through the PL as well. As the AXI Performance Monitor passively reads transactions on the system bus, it does not perturb memory request timing.



(a) Zynq SoC with emulation framework



(b) Data Rearrangement Engine detail

Figure 3: Emulation architecture

3.2. Clock Frequencies

The clock system on the Zynq platform supports many configurable clocks that can span a wide range of frequencies. The A9 cores on the Zynq 7000 can run up to 800 MHz and down to under 1 MHz. The programmable logic clock frequency depends on the specific design placed on the FPGA but is typically around 200 MHz. The DDR program memory runs at 1066 MT/s. Since the DRE runs in slower programmable logic, the CPU is also slowed to run at a frequency with a ratio comparable to the target system. For example if the DRE runs at 100 MHz and the CPU at 200 MHz, program run times when scaled by a factor of 20 represent components running at 1 GHz and 2 GHz respectively. The DRAM clocks are not slowed; therefore, memory requests are routed through a set of programmable delay units (labeled Delay in Figure 3a) to emulate memory latencies consistent with the CPU and DRE frequency. These delay units also allow emulation of a wide range of memory latencies encompassing current and future technologies. CPU and DRE clock frequencies along with delay parameters are set with values that maintain consistent ratios needed to emulate various CPU and active memory configurations.

Our use of an SoC to emulate a system offers efficiency and challenges. Using hard IP modules such as the ARM cores, the on-chip scratchpad, and the memory architecture of the development board saves FPGA logic and development time. However, these fixed components also limit host design space exploration and require coordination of multiple clocks to accurately model the desired system.

4. Experiments

To evaluate the potential benefits of DRE-assisted computation, irregular, data intensive benchmarks were run over a range of emulated CPU-memory latencies.

4.1. Emulation parameters

The emulation targets a standard CPU core and an HMC-like memory. Datapath widths and memory bandwidths conform to standard configurations. The processor is a hypothetical 32-bit ARM A9 core running at 2.57 GHz with 5 GB/s memory bandwidth. Since the applications are memory bound, using a 32-bit rather than 64-bit processor was not found to affect run time. The 1.25 GHz LSU has a 64-bit internal data path and a bandwidth of 10 GB/s. The 1.25 GHz MCU has a 32-bit data path and 5 GB/s of bandwidth. The control path of sending commands to the DRE has a round trip latency of 340ns. Measuring performance using a single CPU core and a single DRE enables precise measurement of the application's memory access characteristics.

Memory parameters are derived from measurement on an Arira Design Gen 2 HMC evaluation board [9] which has instrumentation to capture latency, bandwidth and power. Based on these measurements, the latency to access the DRAM array is set to 45ns, reflecting the effects of random access in applications lacking long sequential data bursts. On the HMC there may be additional latency due to congestion in vault request and response queues within the memory package. Since our benchmarks run in isolation, we model the effects of interference from other jobs in a workload by emulating congestion delay at three different rates, 0ns (no delay) for a light load, 20ns for medium load, and 40ns for heavy load. This latency affects both CPU and DRE memory accesses. SRAM latency for the DRE's scratchpad is set to 10ns. SRAM latencies vary widely; this value represents

an average among latencies reported in the literature. The link latency to transfer packets between memory package and CPU is set to 24ns, again derived from measurement. The DRAM energy is modeled at 19.4 pJ/bit, the SRAM is 1 pJ/bit, and the link is 10.3 pJ/bit. DRAM and link energy estimates are obtained from measurement on the HMC board. SRAM energy is estimated from reports in the literature [10].

Irregular applications often access random 8-byte data values. To study the effects of hardware support for this behavior, we additionally evaluate the energy impact of a "narrow vault" architecture modeled on the HMC in which the DRAM can be accessed internally in 8-byte units. On the HMC, the DRAM is accessed in 32-byte units, and even a 16-byte request will touch 32 bytes within a vault.

For these configurations, detailed evaluation is conducted on three representative benchmarks.

4.2. Benchmarks

There are two forms of each benchmark, CPU only and DRE-assisted. The CPU version is the benchmark in its original form. In the DRE-assisted version, the CPU program communicates with the DRE to load and store an SRAM "view buffer," but all computation is performed by the CPU. In each benchmark, restructured data in the view buffer is in a compact form that allows the compiler to vectorize CPU computations. However, this is not possible in the CPU-only version when the data is scattered in memory. Both forms of benchmarks are serial; in the DRE-assisted version, the CPU core waits for DRE command completion to perform computation and does not double buffer to hide the DRE latency. The benchmarks run standalone with a one-to-one virtual to physical mapping to enable accurate measurement of each phase of DRE operation.

The benchmarks are as follows:

RandomAccess [11] uses the DRE gather and scatter hardware. It is the best example of extreme irregular applications, and is designed to measure memory performance in the presence of a completely random access pattern in combination with minimal compute. The benchmark reads, modifies and writes back random elements of a table that occupies up to half the total memory size. In our benchmark the table is of size .5 GB. The benchmark iteratively performs the computation

```
T[ran[j] & (TableSize-1)] ^= ran[j];
```

where `ran` is a sequence of random 64-bit numbers. Like the Graph500 breadth first search benchmark, RandomAccess encapsulates the core of one class of irregular, data-intensive applications.

PageRank uses the DRE gather hardware. It is a popular data intensive irregular benchmark characterized by floating point computation on a sparse matrix (graph) and vector (pagerank). We use a synthetic scale-free input graph with 2^{22} vertices and an adjacency list representation of the graph. The algorithm iterates through the list of vertices and updates each vertex's rank. Indirect, scattered accesses to the page rank vector are replaced by direct, contiguous accesses into the edgelist's page rank view.

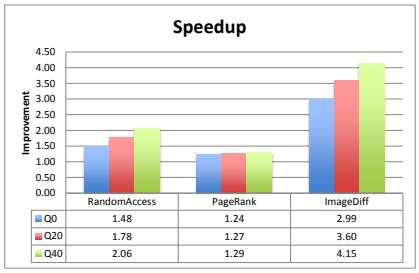
Image differencing of reduced resolution imagery uses strided DMA. We include this benchmark to demonstrate that even a regular streaming access can appear irregular from the viewpoint of cache re-use, enabling effective DRE assist when the stride exceeds cache line length. The benchmark loads two high resolution 2D images into memory and, given a decimation factor, subtracts corresponding pixels in reduced resolution views in both x and y dimensions. We use decimation factor 16. In the DRE-assisted implementation, the DRE loads two view buffers with corresponding blocks of the decimated images. The CPU performs the image difference and stores the differenced image to memory.

All three benchmarks exercise the synchronization and cache consistency management methods described in Section 2. The CPU part of the application issues setup, fill, and drain commands by sending messages to the DRE through the special memory addresses. The DRE executes the command and returns a completion message to the CPU. To maintain memory consistency, the CPU issues cache invalidate to update the cache with the DRE's fill of the SRAM view buffer, and issues cache flush to write its updated view buffer contents to the memory so that the DRE can drain the buffer. The DRE issues corresponding cache flush and invalidate operations to update its cache if needed. The time to perform these operations is included in the performance evaluation.

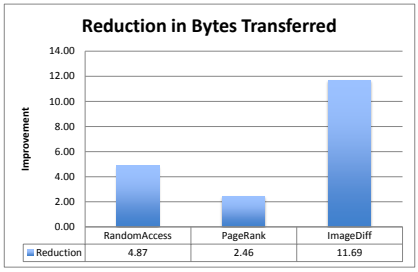
4.3. Evaluation

Figure 4 summarizes benchmark speedup, reduction in memory bandwidth required, and reduction in energy used. For these benchmarks, it is always advantageous to use a DRE for performance, memory bandwidth savings, and energy reduction. The speedup results highlight the effects of intra-package queue delays. Using the DRE gives speedup in the contrived case of exclusive access to memory in serial execution, but there is even more speedup in the more normal case of interfering memory requests, as shown in the Q40 row (queue delay of 40ns). Memory bandwidth savings is at least 2.46X and is 11.69X at best, showing the data reduction afforded by using the DRE. Energy use is modeled in two scenarios, HMC and Narrow. The HMC row reflects the access unit of the HMC, requiring 32-byte access even when only 8 bytes are requested. The Narrow row reflects potential energy savings if it were possible to access down to an 8-byte granularity. In HMC mode, energy savings is at least 49% and is as high as 2.7X. If narrow mode were available, the energy savings jumps to 7.80X.

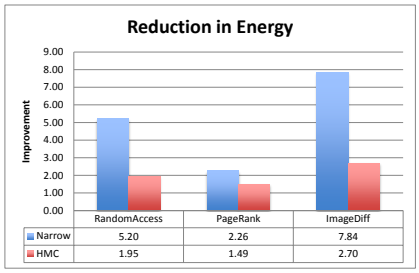
PageRank shows speedup ranging from 24% – 29%. The more modest speedup compared to the other benchmarks is due to the large number of vertices with few edges, which is characteristic of a scale-free graph. Because of synchronization overhead between CPU and DRE, it is only profitable to process long edgelists on the DRE. The number of "hub" vertices and the edgelist lengths increase with scale, so larger graphs are expected to show greater speedup.



(a) Speedup for three queue delay parameters.



(b) Reduction in bytes transferred for DRE-assisted versions of benchmarks.



(c) Reduction in energy for a hypothetical narrow access memory and for the HMC.

Figure 4: Performance of benchmark applications.

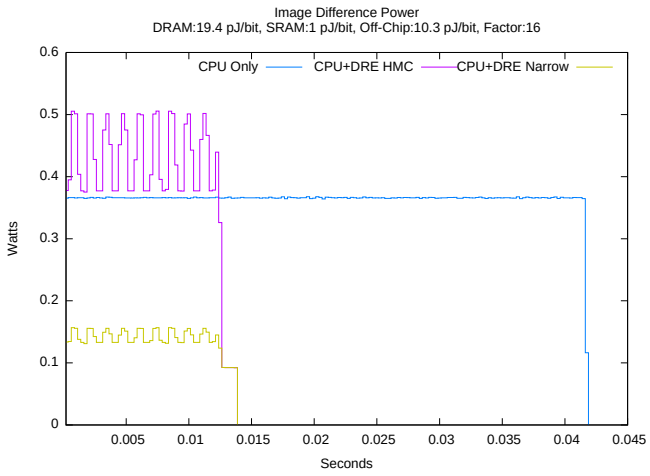
Figure 5 shows a running power profile captured from ImageDiff memory traces (due to space limitations, the other energy profiles are not shown). The full trace shows that the DRE-assisted version with HMC model (32-byte access) uses more power than the CPU only version, but for a much shorter time. The narrow model uses significantly less power. Both models show overall energy savings (Figure 4c). The enlarged segment shows the handoff between CPU and DRE in the DRE-assisted version (red for CPU and either green or yellow for DRE depending on HMC or Narrow model). The blue line is CPU only.

5. Discussion

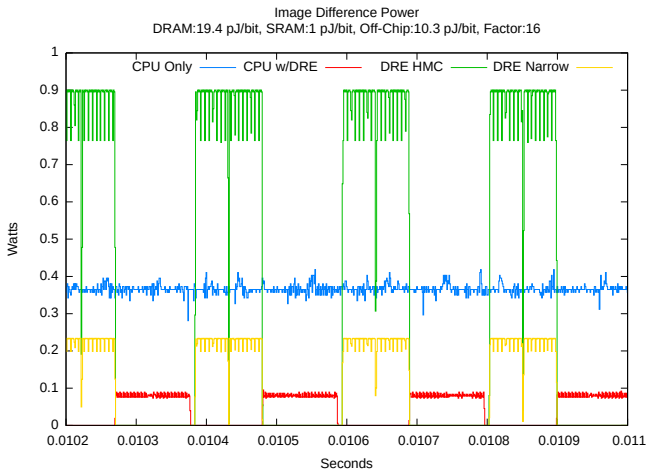
Data rearrangement in active memory provides substantial performance and energy improvement in benchmarks whose access patterns are representative of irregular, data intensive analytics workloads. The improvement is particularly significant considering that the DRE, in contrast to other processing-in-memory designs, performs data reduction, not compute offload. The improvements are enabled by transforming cache unfriendly data layout into a tightly packed, locality enhancing format before the data traverses the memory bus. Overheads to using the DRE include coherence transactions to maintain memory consistency, and communication between CPU and DRE. Despite these overheads, the proposed approach using shared scratchpad demonstrates improvement in all metrics evaluated.

Another aspect to consider is the access granularity at the memory bank. For irregular applications, 8-byte granularity would save substantial energy for DRE-assisted irregular applications. While reducing access size introduces perhaps unacceptable complexity into DRAM, it has been suggested that such an organization is better suited to future persistent memories [12].

In this evaluation we have written the DRE-assisted versions of the benchmarks manually to closely control the low level hardware interfaces for effective co-design of hardware and API. With promising performance and energy results from the evaluation, it is now time to built higher level tools. These include encapsulating the view buffer interactions in libraries similarly to communication libraries,



(a) The entire run.



(b) Enlarged segment of the run.

Figure 5: ImageDiff power profile.

hiding DRE interaction in high level language classes that use the libraries, and using compiler pragmas to indicate DRE interaction.

While the focus of this work has been a single CPU core interacting with a single DRE, a more comprehensive use case would include multiple DREs, managed by one or multiple cores. Each DRE would have its own view buffer, and, as for the single DRE/CPU case, the application would have to coordinate synchronization and cache consistency if data structures are shared. Another important aspect is whether/how the reduction in bandwidth used by the irregular part of the workload can be exploited by other more regular applications. While this is certainly plausible, more work must be done to evaluate the effect on full system throughput.

6. Conclusions

This work follows a data reduction approach to memory-integrated computing that leverages simple memory movement hardware to directly address the memory bandwidth problem. Our implementation is compatible with the Hybrid Memory Cube and is applicable to other integrated memory/logic technologies. The DRE uses a simple API to perform application-specific transformations on data structures, enabling cache-friendly buffers to be assembled. Our benchmarks, which use this API, exemplify irregular, data intensive access patterns, and demonstrate how to use data rearrangement hardware effectively.

To assess quantitatively the impact to applications of using DREs, we have designed and implemented an emulator with DRE instantiated in programmable hardware blocks. We have assessed performance and energy of the benchmark applications over a range of memory parameters. DRE-assisted performance improvement ranges from 1.24X to 4.15X on irregular, data intensive access patterns. Reduction in bandwidth usage ranges from 2.46X to 11.69X. Energy is reduced by up to 2.7X for the current generation HMC and up to 7.84X for a proposed narrow access organization. We find that data rearrangement in memory offers significant advantage to applications with irregular, data intensive access patterns.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract No. DE-AC52-07NA27344. We thank Roger Pearce for the original pagerank benchmark.

References

[1] "Hybrid Memory Cube," http://www.hybridmemorycube.org/, Hybrid Memory Cube Consortium, 2011. [Online]. Available: http://www.hybridmemorycube.org/

[2] M. Gokhale, W. Holmes, and K. Iobst, "Processing in memory: The Terasys massively parallel PIM array," *IEEE Computer*, vol. 28, no. 4, pp. 23–31, Apr 1995.

[3] J. B. Brockman, S. Thoziyoor, S. K. Kuntz, and P. M. Kogge, "A low cost, multithreaded processing-in-memory system," in *Workshop on Memory Performance Issues at ISCA*. ACM, 2004, pp. 16–22.

[4] J. Draper, J. T. Barrett, J. Sondeen, S. Mediratta, C. W. Kang, I. Kim, and G. Daglikoca, "A prototype processing-in-memory (PIM) chip for the data-intensive architecture (diva) system," *The Journal of VLSI Signal Processing*, vol. 40, pp. 73–84, 2005.

[5] J. Gebis, S. Williams, C. Kozyrakis, and D. Patterson, "VIRAM1: A mediaoriented vector processor with embedded DRAM," in *Student Design Contest, DAC*, 2004.

[6] L.Zhang, Z.Fang, M.A.Parker, B.K.Mathew, L.Schaelicke, J.B.Carter, W.C.Hsieh, and S.A.McKee, "The impulse memory controller," Nov. 2001, pp. 1117–1132.

[7] R. Nair, "Active memory cube," Dec. 2014, www.cs.utah.edu/wondp/Nair.pdf.

[8] P. C. Diniz and J. Park, "Data reorganization engines for the next generation of system-on-a-chip fpgas," in *Proceedings of the 2002 ACM/SIGDA Tenth International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '02. New York, NY, USA: ACM, 2002, pp. 237–244.

[9] "Arira design," 2015, www.ariradesign.com.

[10] B. Rooseleer, S. Cosemans, and W. Dehaene, "A 65 nm, 850 mhz, 256 kbit, 4.3 pj/access, ultra low leakage power memory using dynamic cell stability and a dual swing data link," in *ESSCIRC*, Sept 2011, pp. 519–522.

[11] "HPC challenge randomaccess," accessed 01/2015, http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/.

[12] J. Meza, J. Li, and O. Mutlu, "A case for small row buffers in non-volatile main memories," in *ICCD*. IEEE, Oct. 2012, pp. 484–485.