Dataflow based Near-Data Processing using Coarse Grain Reconfigurable Logic

Charles Shelor, Krishna Kavi, and Shashank Adavally
Department of Computer Science and Engineering
University of North Texas
Denton, TX, USA

***Abstract*—The emergence of 3D-DRAM has rekindled interest in near-data processing research. This article introduces and describes a method for near-data processing using dataflow techniques implemented with coarse grain reconfigurable logic. We provide an initial evaluation of the concept to justify further development. The melding of these technologies produces increased throughput while dramatically reducing energy needs for suitable classes of algorithms. The benchmarks analyzed in this paper show performance speedups of 1.1 to 13.2 with energy reductions of 88% to 99%.**

***Index Terms*—Dataflow, Near-Data Processing, Coarse Grain Reconfigurable Logic, Processing-in-Memory, Big Data, 3D stacked DRAM, Computer Architecture**

I. Introduction

One of the major problems facing today's computer systems is the disparate timing between processor instruction cycles and memory access cycles. This timing mismatch has been tagged the "memory wall" [1]. The cache hierarchy in processors has been used to mitigate the effects of the memory wall by providing fast access to data items on subsequent accesses. However, there are some classes of applications that do not exhibit repeated access to the same data items. These classes of applications achieve little to no benefit from the cache hierarchy. Examples of these classes include streaming and big data applications [2]. One approach to improving the performance of these applications is Near-Data Processing (NDP) also known as Processing-in-Memory (PIM) or Near-Data Computing (NDC). This approach moves the processing closer to the memory to achieve faster access and higher bandwidth.

The recent commercialization of 3D, stacked DRAM [3] provides the opportunity to integrate PIM on the logic layer of the stacked DRAM. While this logic layer provides high bandwidth access to the memory, there are limitations to the size and power of the processing elements. These limitations preclude the use of standard, high performance, out-of-order processors for PIM applications.

The use of dataflow techniques based on Coarse Grain Reconfigurable Logic (CGRL) offers processing capacity and power efficiency suitable for the PIM applications. A dataflow processing-in-memory (DFPIM) structure using CGRL consists of a set of functional blocks with a reconfigurable interconnection. The interconnection of the blocks is configured to implement the dataflow graph of the PIM application. The dataflow paths are synchronized and pipelined such that a new element is computed on every clock cycle. The parallelism and pipelining provide high performance while requiring less energy than out-of-order control flow processors. In this paper, we show two different approaches for identifying segments of programs (or kernels) that are suitable for CGRL dataflow implementation. We can analyze the algorithm or source code to identify kernels. We can also identify frequently executed sequences of instructions from execution traces (no source code is needed). In this paper we will present evaluations using both methods. Preliminary estimates show speedups of 1.1 to 13.2 while showing energy reductions of 88% to 99%.

Section II provides a detailed description of the DFPIM concept. Section III describes the benchmarks used in this evaluation. Section IV describes a methodology of extracting dataflow kernels from benchmark execution traces. Section V provides results and analysis of the benchmarks on DFPIM. Section VI discusses future work on DFPIM. Section VII examines research that is closely related to DFPIM. Section VIII provides summary and conclusions for the DFPIM work presented in this paper.

II. DFPIM Concept

Dataflow Processing-in-Memory (DFPIM) is based on melding three technologies. The dataflow paradigm extracts the available concurrency from the algorithms. Coarse Grain Reconfigurable Logic (CGRL) provides an efficient and flexible method to implement the dataflow processing. 3D-stacked DRAM includes a logic layer on which the CGRL can be implemented and provides low latency, high bandwidth access to memory.

A. Dataflow

Dataflow is a style of computing where the data values "flow" from one operation to the next operation. Dataflow is highly concurrent and self-synchronizing [4]. Generating and analyzing dataflow graphs are integral components to optimizing compilers for high level programming languages. DFPIM utilizes the dataflow graphs of the PIM applications to extract parallelism, detect dependencies, and arrange pipelining of the application. Pure dataflow uses data availability at each operation to determine when to 'fire' the operation and generate a new result. This provides the self-synchronizing characteristic of dataflow. However, this also introduces substantial overhead

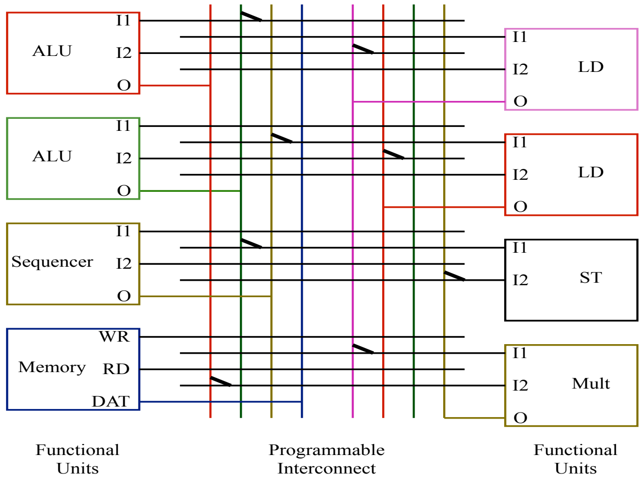

Fig. 1. CGRL Example.

in dataflow processing which has limited its commercial deployment.

Dataflow does not have a concept of memory as only values are utilized. DFPIM uses load units at the dataflow graph inputs to get the needed values from memory. The load units have 2 buffers that each hold a row of DRAM memory. The system will be accessing data from one row buffer while the other row buffer is being filled through a memory access. Similarly, there are store units to write rows back to memory as required. There are delay operations in the dataflow graphs to balance and synchronize the path lengths. The dataflow graph is 'executed' only when all graph inputs are available. This single level of synchronization reduces the dataflow overhead by not requiring synchronization at each operation in the graph.

B. Coarse Grain Reconfigurable Logic

CGRL provides a set of functional blocks that are configured at run-time to implement an algorithm. Each functional block is implemented directly in silicon logic to minimize latency and power. This distinguishes CGRL from Fine Grain Reconfigurable Logic (FGRL) used by standard Field Programmable Gate Array (FPGA) devices. CGRL used as Coarse Grain reconfigurable accelerators (CGRA) provide significant energy efficiency and performance benefits [5]. Figure 1 shows an example CGRL containing two load buffers, LD, one store buffer, ST, two arithmetic logic units, ALU, one sequencer, one memory, and one multiplier. The vertical lines for distribute the output of each block to the inputs of the other blocks on the horizontal lines. The short, diagonal lines represent the reconfigurable connections in a cross bar style interconnection. This arrangement allows one output to drive multiple inputs if a data value is used more than once. The benchmarks used in this paper required 13 to 32 functional blocks to implement the dataflow graph.

C. 3D-Stacked DRAM

3D-Stacked DRAM is a high density, high bandwidth memory subsystem that is created by stacking multiple DRAM

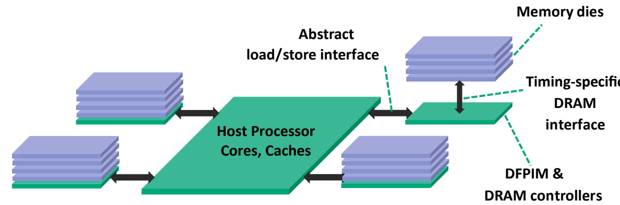

Fig. 2. DFPIM with Stacked DRAM Example.

semiconductor dies vertically and communicating through the stack using Through-Silicon-Vias (TSV) [6]. Chang [7] has shown that the most significant performance benefit of stacked DRAM is increased bandwidth. The high bandwidth is required for high performance DFPIM operation as PIM applications have very high memory demands. The stacked DRAM configuration selected for DFPIM is illustrated in Figure 2. This is the same configuration selected by Zhang [8] and Scrbak [9] in giving the highest bandwidth between PIM and DRAM without thermal issues from the host processor.

The DFPIM instances are included in the base logic layer with the DRAM memory controllers. DFPIM instances are limited to 50% of an 80 mm^2 die with a total dissipated power (TDP) of 10 Watts consistent with Zhang [8].

III. PIM Benchmarks

There are a wide variety of benchmarks that could be used in an evaluation such as this. The purpose of DFPIM is to offload memory intensive kernels from the host processor to utilize the high bandwidth of 3D stacked DRAM and the parallelism of dataflow. We reviewed the map-reduce benchmarks from HiBench [10], the map-reduce benchmarks from PUMA [11], SPEC benchmarks [12], and MiBench benchmarks [13]. We picked 3 benchmarks that had significant differences in the dataflow configuration of the kernels. We were limited to three benchmarks as we currently process the kernels, implement the dataflow diagrams, and evaluate the results manually. Later in section IV, we show how code sequences that are amenable for dataflow implementation can be identified from execution traces. Here we analyze an application to identify functions that could be implemented as dataflow graphs.

The SPEC benchmarks and the PUMA benchmarks both include a version of a histogram application. Our histogram application performs a histogram of the red, blue, and green pixel values of an image. Each pixel is only accessed once and the processing is very simple. Thus, memory bandwidth is the dominant factor in performance of this application. Even with prefetching, the computation with 1 cache line is completed before the next cache line is prefetched. Figure 3 shows the dataflow graph of the histogram application implemented using DFPIM components. The oval labeled 24-bit is the load unit that is extracting the pixels from the image in memory. The load unit is providing 24-bits comprised of the 8-bit red, 8-bit green, and 8-bit blue components of the pixel on each clock cycle. A load unit extracts data from one local row buffer while

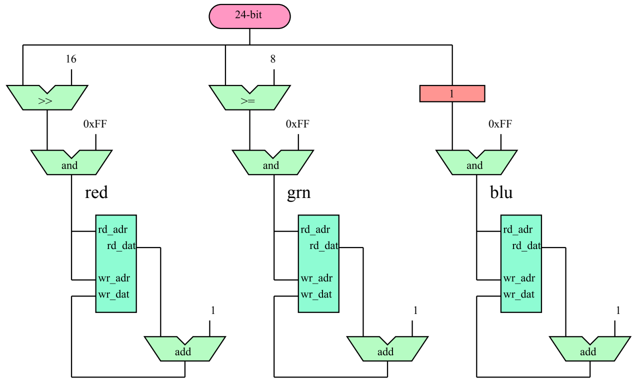

Fig. 3. DFPIM dataflow graph for histogram.

a second row buffer is being filled from memory. If the second row buffer is not filled by the time it is needed the load unit drops the 'data ready' indicator causing the dataflow graph to stop processing until the data becomes available.

The next row of functional units shift the red and green data while delaying the blue data by 1 clock. The third row of blocks are all ALU blocks that AND the input data with the immediate value 0xFF. These three values are used as the addresses to three memories that contain the count of each values' occurrence. The outputs of the memories are incremented with another set of 3 ALU blocks and written back to the memory. The memory block read action is configured to be asynchronous, the ALU operation is configured to be combinatorial, and the memory write operation is synchronous with the next clock edge. This allows the read, increment, and write to be performed in a single clock cycle. The rows are pipelined allowing a complete pixel to be processed every clock cycle. A four-issue processor is shown to take 23 clock cycles per pixel in sectionV.

Both of the map-reduce benchmark suites have a map client that performs a word occurrence count. Actual isolation of the words is a simple process for both control flow processors and dataflow systems. However, implementation of the hash table collision resolution requires more effort in dataflow. The use of CGRL in the dataflow allows a very wide comparator resulting in each word comparison requiring only a single clock cycle greatly increasing application throughput. The DFPIM internal scratch pad memory is used for the hash table and FIFOs separate the word isolation from the hash table operation. The resulting dataflow graphs are not presented due to space limitations.

The third benchmark selected is the FFT implementation from MiBench. FFT differs from the other benchmarks as it does benefit from caches since each element is accessed $\log_2$ (size) times during the algorithm. The data is not accessed in a streaming sequence and there are data dependencies within the loop that restrict the amount of pipelining that can be performed. These differences make the FFT algorithm a poor candidate for PIM, but we chose to analyze it as a "stress test" for DFPIM. The DFPIM scratch pad memory, loop overlap, and 21 operating units still provide adequate throughput.

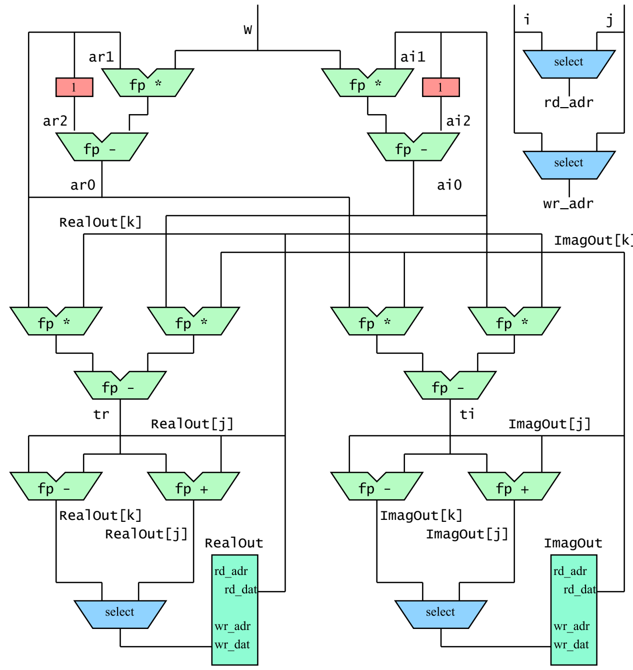

Fig. 4. DFPIM dataflow graph for FFT.

```
0x40153d.0   : @ReverseBits+29.0    : or   eax, eax, ecx             : 2048 counts
0x40153f.0   : @ReverseBits+31.0    : sub   t0d, edx, esi            : 2048 counts
0x401541.0   : @ReverseBits+33.0    : rdip   t1, %ctrl153,           : 2048 counts
0x401541.1   : @ReverseBits+33.1    : limm   t2, 0xffffffffffffffed  : 2048 counts
0x401541.2   : @ReverseBits+33.2    : wrip   , t1, t2                : 2048 counts
0x401530.0   : @ReverseBits+16.0    : lea   ecx, DS                  : 2048 counts
0x401533.0   : @ReverseBits+19.0    : mov   eax, eax, edi            : 2048 counts
0x401535.0   : @ReverseBits+21.0    : limm   t1d, 0x1                : 2048 counts
0x401535.1   : @ReverseBits+21.1    : add   edx, edx, t1d            : 2048 counts
0x401538.0   : @ReverseBits+24.0    : limm   t1d, 0x1                : 2048 counts
0x401538.1   : @ReverseBits+24.1    : and   eax, eax, t1d            : 2048 counts
0x40153b.0   : @ReverseBits+27.0    : srli   edi, edi, 0x1           : 2048 counts
```

Fig. 5. Bit reverse instruction trace.

IV. Benchmark Kernel Extraction

Thus far we have shown how we can construct dataflow graphs for kernels by analyzing the algorithm or source code. We are also working to identify kernels even when source code is not available. By analyzing traces generated by execution runs of programs, we will identify sequences of instructions that are most frequently executed. The traces used in this paper were generated using the Gem5 simulator [14]. We used instruction addresses and counted the number of times each address appeared in the trace. A hash table was used to accumulate the counts and then the table was sorted by the count values. After identifying the most frequently executed sequences from the sorted hash table, we constructed CGRL dataflow graphs manually. We estimated the execution time for the dataflow graph and measured the time for the x86 instruction sequences. We used McPAT [15] to estimate power consumed for the two versions. This section of this paper reports results for the "bit reverse" function of the "FFT" and the "square root" function of the "basicmath" benchmarks from MiBench.

Figure 5 shows a segment of a trace file from Gem5 after annotating the execution count for each instruction. It should be noted that the instructions in the trace file are the x86 micro-ops and not x86 assembly source code. These instructions are the kernel of the integer square root function. The instruction trace was processed into a dataflow graph by using the registers as the data dependencies and the instruction operands as the dataflow nodes. The *LIMM* instructions were replaced with the equivalent immediate value as an input. Any instructions for spilling variables to or from registers were eliminated as those values are held in the dataflow graph arcs. These graphs were then analyzed for timing and energy.

Dataflow graphs generated in this manner will include compiler artifacts and translation overhead unlike the optimized dataflow graphs from section III that are derived directly from the algorithm and source code. However, this dataflow graph extraction approach can be used with any benchmark, even when the source code is not available.

V. Results and Analysis

A. Methodology

This paper is primarily focused on comparing the performance and energy between a standard host processor and a DFPIM implementation. We measured the clock cycles of an out-of-order, 4-issue x86 processor using PAPI [16] for the kernel of each benchmark. We determined the time of execution for each kernel by multiplying the number of clocks by the clock period of a 3.3 GHz processor. The DFPIM clock counts were manually generated by analyzing the dataflow graph to determine the number of clocks needed to process each benchmark kernel. The histogram benchmark is capable of being executed at 1 clock per RGB pixel. The word occurrence count benchmark can isolate words at 1 clock per character, but is provided a 10% hash collision delay for a net rate of 1.1 clocks per character. The FFT benchmark can overlap three iterations using decoupled software pipelining [17]. Each segment has a 2-operation dependency chain with a 4 clock floating point latency. Thus, the FFT dataflow graph requires 16 clocks to initialize and then produces one element every 8 clock cycles. All of the DFPIM timing computations are based on an 800 MHz DFPIM clock using a low power device technology. This value is estimated based on the authors' experience and will be refined based on the results of logic synthesis of the CGRL units in future work.

The energy needed for the x86 processor and the DFPIM dataflow graph to execute each kernel was estimated by using the McPAT power, area, and timing estimator [15]. The processor model used for this evaluation was based on the default Xeon model that is provided with the McPAT version 1.3 installation, modified to use 28 nm technology and 3.3 GHz clock. The area and power of the DFPIM ALU was estimated by a second McPAT configuration using a low operating power device type and 800 MHz clock.

Each of the benchmarks requires some setup and each of the benchmarks has to access memory for the same amount of data for the kernel whether it is executing on an x86 or on a DFPIM. The DFPIM uses less energy for accessing the same amount of data because the external link between the processor and DRAM is not used. The DFPIM gets more data bytes per access so the access overhead energy (precharge, row open, row close) is amortized over more bytes of data. We are not ready to quantify this savings in energy, so we will only compare processor energy and DFPIM energy and assume a constant for the memory subsystem energy. These energy savings will be quantified in future work. These benchmarks are very loop intensive which allows us to ignore the timing and energy of the non-kernel execution.

Benchmark	x86 us	DFPIM us	Speedup	x86 uJ	DFPIM uJ	Savings
Histogram	1811	328	5.52	30,287	197	99.3%
Word count	1866	141	13.23	31,183	249	99.2%
FFT	362	328	1.10	6,055	724	88.0%

TABLE I
x86 Processor and DFPIM Algorithm Derived Comparison.

B. Results

Table I provides the results of our analysis. The histogram benchmark shows an execution time speedup of 5.52. The histogram loop of the x86 takes 23 clocks for each pixel as measured by the PAPI instrumentation. The DFPIM dataflow graph processes a pixel every clock. The speedup is not 23 because the x86 clock is 4.13 times faster than the DFPIM clock. The McPAT analysis of the Xeon processor model shows a total power usage of 16.7 Watts per core when the level 3 cache and networking power is split evenly among the cores. The DFPIM requires 13.75 ALU equivalents totaling 0.522 Watts. The x86 needs 30,267 uJ while the DFPIM uses 197 uJ. This is an energy savings of 99.3%.

The word occurrence count benchmark shows a speedup of 13.23. This is partly due to the pipeline and parallelism of processing one character per clock cycle in the word isolation section of the benchmark and partly to the ability of DFPIM to establish a very wide data path. The wide data path compares 32 characters in a single clock cycle using 4 64-bit ALUs in the hash table section. The wide ALUs and memory for the hash table result in a 46.5 ALU equivalent for size and power. The DFPIM word count configuration uses 1.77 Watts which is about 3 times the power of the histogram benchmark. The processor uses 16.7 Watts for 31,183 uJ. The DFPIM uses only 249 uJ for a 99.2% energy savings.

The FFT benchmark is essentially the same speed in the DFPIM as in the x86 processor with a speedup of only 10%. The DFPIM needs 16.5 ALU equivalents and 14 FPU equivalents. This results in the DFPIM needing 2.21 Watts which is still significantly less than the 16.7 Watts of the x86 processor. The DFPIM provides an 88.0% power savings. FFT does not match the characteristics for a PIM application since it does not access memory in a streaming manner and it does access elements multiple times. The DFPIM provides equivalent performance at 1/8 of the power even in this non-ideal use.

Table II compares the performance and energy values for the execution trace derived dataflow graph. The identified sequences were executed and timed on an x86 processor and then analyzed using dataflow with CGRL nodes. The dataflow graph used 8 ALU's for "bit_reverse" function, 12 ALU's for "square root" function and 25 ALU's for "lex num" function. The lex num dataflow graph has more parallelism and obtains

Benchmark	x86 ns	DFPIM ns	Speedup	x86 nJ	DFPIM nJ	Savings
Bit reverse[FFT]	27.5	22.5	1.22	459	7	98.5%
Square root[Basicmath]	212.0	187.0	1.13	3540	85	97.6%
Lex num[GCC]	136.0	62.5	2.18	2271	59	97.4%

TABLE II
X86 PROCESSOR AND DFPIM TRACE DERIVED COMPARISON.

a larger speedup. These instruction sequences were extracted from the executable without the source code unlike the graphs in section III. Working from the source code allowed optimizations for a dataflow implementation that have not been applied to the execution trace results. Better results are anticipated when optimizations are applied to the instruction trace data.

VI. FUTURE WORK

The preliminary results shown in section V have demonstrated DFPIM is a viable approach to both performance improvements and energy savings for PIM applications. Continued development of DFPIM will include completely defining the set of functional blocks and the quantity of each type of functional block to include in a DFPIM cluster. This definition will include the block's primary functions, list and purpose of all of its inputs and outputs, and configuration options such as latency or operand size. The definition will serve as a reference for generating dataflow graphs and development of the functional blocks. Each of the functional blocks will be modeled in VHDL. A synthesis tool will be used to characterize size, timing, and energy for each of the DFPIM blocks in 2 or 3 different silicon technologies.

A DFPIM simulator will be developed that will read XML representations of the dataflow graphs and allow benchmarks to be executed by the simulator. The simulator will include the size, timing, and energy characteristics of each DFPIM block. This will allow the simulator to compute execution time and energy for the benchmark. The simulator will provide an automated method for analyzing benchmarks which will allow more benchmarks to be evaluated to determine if the preliminary results are consistent for a larger set of benchmarks.

The extraction of benchmark kernels through execution trace processing will be expanded to include additional instruction sets. An automated dataflow graph generation process will be developed that will translate the instruction traces into the XML representation of the DFPIM dataflow graphs for input to the simulator.

VII. RELATED WORK

TOP-PIM [18] is a very similar approach to DFPIM. The principle difference being the use of GPGPU devices as the processor component in the PIM. This study showed a mean decrease in performance of 25% for a 22 nm technology and a mean increase in performance of 8% for a 16 nm technology. The energy savings was shown to be 76% and 86% respectively when including the memory power. The parallelism of dataflow and flexibility of CGRL work to provide better performance at comparable energy savings. There were no benchmarks in common between the two studies so a direct comparison cannot be stated until DFPIM expands its benchmark coverage to include those used by Zhang.

Single Graph Multiple Flows (SGMF) [19] uses a dynamic dataflow paradigm and CGRL to compare to an Nvidia Fermi streaming multiprocessor. The application arena for SGMF is compute intensive applications so it is not suitable as a PIM. The advantages of using dataflow with CGRL is shown in this paper with an average speedup of 2.2 and energy efficiency of 2.6 for the 64 token case.

The use of a low power embedded processor as a PIM is addressed in Scrbak [9]. The embedded processor is limited to memory accesses at a cache line resolution. It also requires energy for instruction fetch and decode and a cache subsystem that is not needed by DFPIM. The parallelism of dataflow CGRL provides higher performance than a single instruction stream processor running at the same clock rate.

The Tesseract PIM in [20] uses multiple in-order processors in a Hyper Memory Cube [3]. The memory bandwidth restrictions of the in-order cores are mitigated by prefetching mechanisms. The internal crossbar network allows the Tesseract processors to communicate without host processor intervention. This allows them to be used for the reduce task workload as well as the map task workload. The Tesseract PIM performance is significant for multi-threaded message passing applications. DFPIM has not been evaluated in these types of applications.

The Near DRAM Accelerator (NDA) [21] utilizes a dataflow network of functional devices to reduce energy by 46% and increase performance by 1.67 speedup. The NDA does not include sequencing functional units nor scratch pad memories which we have shown to be necessary for best performance (up to 13.5 speedup) in some benchmarks. The NDA connects each accelerator to a single DRAM die rather than a 3D-DRAM stack used by DFPIM. This results in a higher accelerator-to-memory cost ratio as a single DFPIM can support 4 or 8 DRAM dies.

VIII. CONCLUSION

In this paper we have used two different techniques for identifying kernels that could be executed by CGRL dataflow PIMs. When the source code or the algorithm for an application is available, it will be easier to extract kernels, analyze the functionality and design optimal dataflow implementations. However when source code is not available, we can extract frequently executed instruction sequences from execution traces, and translate the instruction sequences into dataflow graphs. This approach may not result in optimal dataflow graph implementations. The preliminary results presented here support these conclusions.

More importantly, our preliminary results strongly favor the use of dataflow graphs for exploiting PIM technologies. Such implementations for scale out applications, or applications that do not exhibit temporal localities and applications that require high memory bandwidths can result in significant speedups and substantial energy savings. CGRL dataflow implementations are more efficient than using traditional processing elements or GPUs for these classes of benchmarks. Our work also shows that a hybrid dataflow approach with sequencers, scratch-pad memories, and FIFOs implement multilevel looping and asynchronous interactions within the application kernels without host intervention. These features provide the 13.5 speedup for word occurrence count and complete implementation of the FFT within the accelerator.

Acknowledgment

This work is conducted in part with support from the NSF Net-centric IUCRC and its members. The code for extracting instruction sequences from execution traces was implemented by Yosvany Blanco, Troy King and Margarita Sanchez. These students were supported by NSF REU supplement during summer 2015.

References

[1] W. A. Wulf and S. A. McKee, “Hitting the memory wall: Implications of the obvious,” *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995. [Online]. Available: http://doi.acm.org/10.1145/216585.216588

[2] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. Popescu, A. Ailamaki, and B. Falsafi, “A case for specialized processors for scale-out workloads,” *Micro, IEEE*, vol. 34, no. 3, pp. 31–42, May 2014.

[3] Micron Technology, “Hmc high-performance memory brochure,” nov 2014. [Online]. Available: http://www.micron.com/~/media/documents/products/product-flyer/brochure_hmc.pdf

[4] K. M. Kavi, C. Shelor, and D. Pace, “Concurrency, synchronization, and speculation - the dataflow way,” *Advances in Computers*, vol. 96, pp. 47–104, 2015. [Online]. Available: http://dx.doi.org/10.1016/bs.adcom.2014.10.004

[5] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, “Dyser: Unifying functionality and parallelism specialization for energy-efficient computing,” *Micro, IEEE*, vol. 32, no. 5, pp. 38–51, Sept 2012.

[6] G. Loh, “3d-stacked memory architectures for multi-core processors,” in *Computer Architecture, 2008. ISCA ’08. 35th International Symposium on*, June 2008, pp. 453–464.

[7] D. Chang, G. Byun, H. Kim, M. Ahn, S. Ryu, N. Kim, and M. Schulte, “Reevaluating the latency claims of 3d stacked memories,” in *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*, Jan 2013, pp. 657–662.

[8] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, “Top-pim: Throughput-oriented programmable processing in memory,” in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC ’14. New York, NY, USA: ACM, 2014, pp. 85–98. [Online]. Available: http://doi.acm.org/10.1145/2600212.2600213

[9] M. Scrbak, M. Islam, K. Kavi, M. Ignatowski, and N. Jayasena, “Processing-in-memory: Exploring the design space,” in *Architecture of Computing Systems Ð ARCS 2015*, ser. Lecture Notes in Computer Science, L. M. P. Pinho, W. Karl, A. Cohen, and U. Brinkschulte, Eds. Springer International Publishing, 2015, vol. 9017, pp. 43–54. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-16086-3_4

[10] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, “The hibench benchmark suite: Characterization of the mapreduce-based data analysis,” in *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, March 2010, pp. 41–51.

[11] F. Ahmad, S. Lee, M. Thottethodi, and T. N. Vijaykumar, “Puma: Purdue mapreduce benchmarks suite,” Purdue University, Tech. Rep., 2012. [Online]. Available: http://docs.lib.purdue.edu/ecetr/437

[12] “Spec benchmarks.” [Online]. Available: https://www.spec.org/benchmarks.html

[13] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, “Mibench: A free, commercially representative embedded benchmark suite,” in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, Dec 2001, pp. 3–14.

[14] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 simulator,” *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[15] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, “McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures,” in *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 469–480.

[16] D. Terpstra, H. Jagode, H. You, and J. Dongarra, “Collecting performance data with papi-c,” in *Tools for High Performance Computing 2009*. Springer, 2010, pp. 157–173.

[17] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August, “Parallel-stage decoupled software pipelining,” in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, ser. CGO ’08. New York, NY, USA: ACM, 2008, pp. 114–123. [Online]. Available: http://doi.acm.org/10.1145/1356058.1356074

[18] D. P. Zhang, N. Jayasena, A. Lyashevsky, J. Greathouse, M. Meswani, M. Nutter, and M. Ignatowski, “A new perspective on processing-in-memory architecture design,” in *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, ser. MSPC ’13. New York, NY, USA: ACM, 2013, pp. 7:1–7:3. [Online]. Available: http://doi.acm.org/10.1145/2492408.2492418

[19] D. Voitsechov and Y. Etsion, “Single-graph multiple flows: Energy efficient design alternative for gpgpus,” in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA ’14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 205–216. [Online]. Available: http://dl.acm.org/citation.cfm?id=2665671.2665703

[20] 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*, ser. ISCA ’15. New York, NY, USA: ACM, 2015, pp. 105–117. [Online]. Available: http://doi.acm.org/10.1145/2749469.2750386

[21] 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,” pp. 283–295, 2015. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7056040