CSE 6230 - Fall 2014
Term Project Report
Analysing the impact of Processing-In-Memory (PIM) for a memory bound workload

Ajitesh Jain[1], Srinivas Eswar[2]

[1] Ajitesh Jain (ajiteshJain@gatech.edu), GT ID: 903066057.
[2] Srinivas Eswar (seswar3@gatech.edu), GT ID: 902891839.

Acknowledgement

The euphoria that accompanies the successful completion of any task would be incomplete without the mention of the people who made it possible, and whose constant guidance and encouragement helped us in completing the project successfully. We consider it a privilege to express gratitude and respect to all those who guided us throughout the course of the completion of the project.

We would like to thank Professor Richard Vuduc and Jiajia Li for all the help and guide in this term. We express our gratitude to Prashant Nair, PhD. student, ECE department at Georgia Tech whose guidance and support has been invaluable.

Contents

Abstract

Processing in Memory is a concept in which a processing unit is integrated directly with memory. PIM models circumvent the Von Neumann bottleneck by placing memory and logic close to each other. With the emergence of newer technologies like 3D stacked memories and the Hybrid Memory Cube PIM capable systems are becoming a reality. In this project we aim to study the impact of a PIM system on typically memory bound algorithms. Data intensive graph algorithms that are prevalent in several scientific computations is one such class of memory bound workloads.

Introduction

Data intensive computations are becoming increasingly important for modern high performance computing workloads. These workloads have a wide range of applications in the various domains ranging from social sciences, optimization to biology. A common abstraction to analyse and view such interactions is via a graph. Typically solutions to such problems involve classical graph computations like spanning trees, path algorithms, matchings and various other similar computations. It is imperative that we improve existing current infrastructure to cater to these “big-data” applications. [13][14]

Modern computer systems generally have a Von Neumann model of architecture. These systems suffer a performance bottleneck due to the shared bus between program memory and data memory which is used by the processor to step through the computation. Since program memory and data memory cannot be accessed at the same time the throughput of the CPU is much smaller than the rate at which the CPU can work. This is known as the Von Neumann bottleneck. [3]

Process-In-Memory (PIM) is a non-Von Neumann model of computer architecture. It is a decades old concept which proposes the addition of combinational logic on the same chip as memory [1][2]. This allows the PIM logic cores to more efficiently utilize the memory bandwidth afforded to it and effectively scale the throughput barrier. Incorporating both logic and memory in the same die significantly reduces the energy consumption when compared to a traditional off chip CPU and memory units for the same performance. The process of fabricating such a memory commercially was not feasible in the past but with recent advancements in memory technology PIM cores are poised to enter the HPC industry.

We plan to analyse the effect of such PIM cores on a simple graph algorithm.

Memory Architecture

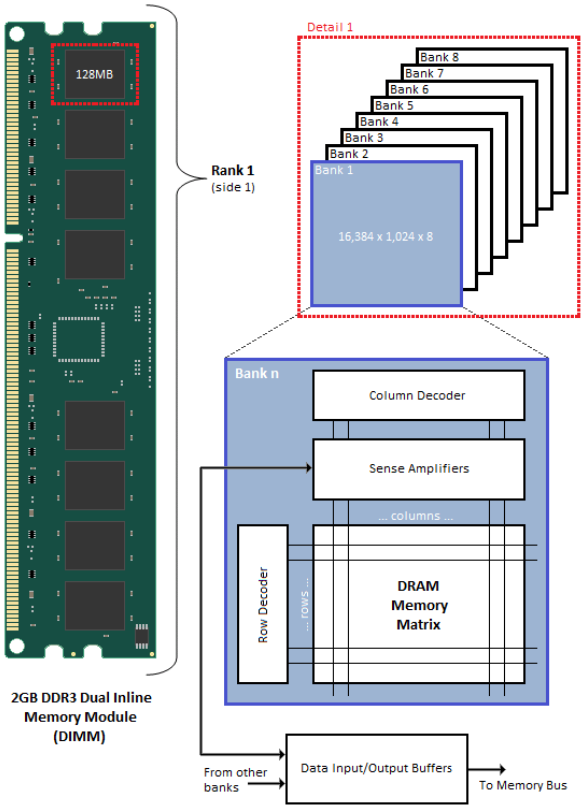


Figure 1 DIMM

Dynamic random-access memory (DRAM) is a type of random access memory that stores each bit of data in a separate capacitor within an integrated circuit. The capacitor can be either charged or discharged; these two states are taken to represent the two values of a bit, conventionally called 0 and 1. Since even "non conducting" transistors always leak a small amount, the capacitors will slowly discharge, and the information eventually fades unless the capacitor charge is refreshed periodically. Because of this refresh requirement, it is a dynamic memory as opposed to SRAM and other static memory.

A DIMM or dual in-line memory module comprises a series of dynamic random-access memory integrated circuits. These modules are mounted on a printed circuit board (PCB). Figure 1 shows the typical architecture of DIMM. It consists of multiple banks on a PCB. Each bank further consists of multiple DRAM memory cells. A memory rank is a set of DRAM chips connected to the same chip select, and which are therefore accessed simultaneously. In practice they also share all of the other command and control signals, and only the data pins for each DRAM are separate (but the data pins are shared across ranks).

Processing in memory was first proposed many years ago to combat the significant problem of computational capabilities far outpacing the improvements in off-chip memory bandwidth and latency. In the decades since, that gap has only increased, and problems have been exacerbated by an increasing amount of power spent on data movement. PIM attempts to solve these problems by embedding computational logic with memory and allowing computation to occur where the data resides, rather than vice versa. By moving computation towards data, systems can benefit from increased bandwidth and reduced energy. One of the main reasons why PIM never gained traction in real-world systems is because logic implemented in a memory process typically was several generations behind contemporary logic processes in terms of performance. However with the recent advancements in 3D die-stacking techniques, programmable processors implemented in a logic process can be coupled with memories implemented in a memory process using through-silicon vias [9]. One such architecture of PIM is Hybrid Memory Cube (HMC) [8].

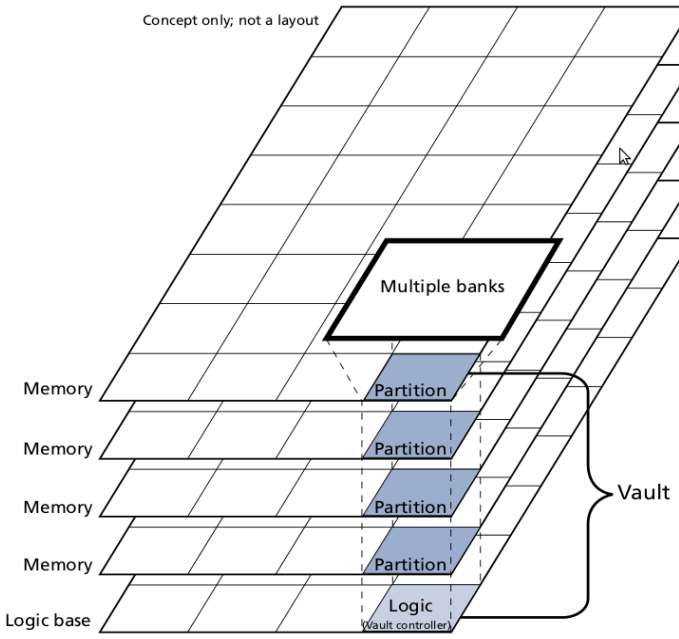


Figure 2 HMC Organization

An HMC consists of a single package containing multiple memory die and one logic die, stacked together, using through-silicon via (TSV) technology (see the example below). Within an HMC, memory is organized into vaults. Each vault is functionally and operationally independent.

Each vault has a memory controller (called a vault controller) in the logic base that manages all memory reference operations within that vault. Each vault controller determines its own timing requirements. Refresh operations are controlled by the vault controller, eliminating this function from the host memory controller. Each vault controller may have a queue that is used to buffer references for that

vault's memory. The vault controller may execute references within that queue based on need rather than order of arrival. Therefore, responses from vault operations back to the external serial I/O links will be out of order. However, requests from a single external serial link to the same vault/bank address are executed in order. Requests from different external serial links to the same vault/bank address are not guaranteed to be executed in a specific order and must be managed by the host controller.

A possible configuration of HMC with processor(s) is shown in the diagram below.

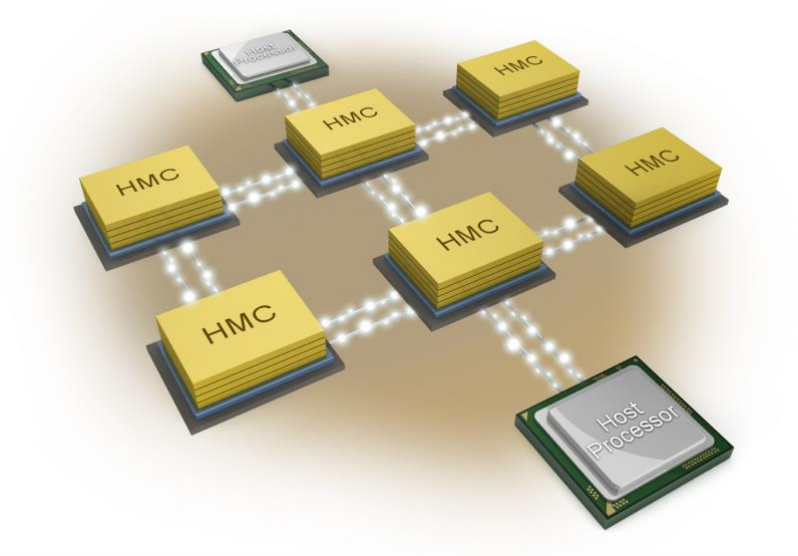


Figure 3 Sample Memory Layout for HMC

Methodology

In our project we had to identify a candidate memory bound algorithm and then set up the necessary infrastructure to perform the analysis. We decided to use USIMM memory simulator to simulate the PIM cores. The final step was considering scenarios to improve performance. In the following sections we will discuss the various choices we made to try and characterize PIM processes.

Benchmark

Standard supercomputing benchmarks like LINPACK TPP, STREAM, DGEMM and others typically test the FLOPS or sustainable memory bandwidth in GUPS. These benchmarks are not representative of the applications which we think would benefit from PIM.

The performance characteristic of the system which we wish to stress in our project is it's communication subsystem. PIM systems, in theory, should have significantly superior communication sub networks than current systems due to the proximity of the logic cores to memory. We wish to validate this claim.

Graph500

Graph500 is a relatively recent benchmark introduced in 2011 [5]. It introduces a new metric for rating computer performance Giga Traversed Edges Per Second (GTEPS). This is a measure of both the communication capabilities and computational power of the machine. This is contrast to FLOPS which gives no weight to the communication capabilities. There are two computation kernels present in the benchmark. The first kernel creates a graph and compresses it into sparse row/column data structures. The second kernel performs a parallel breadth first search on some random vertices in the created graph.

In our project we are interested only in the second kernel of the benchmark. BFS is a widely used graph algorithm and has many applications. Some common use cases are paths of certain length between nodes, semantic analysis and community detection in social networks.

Parallel BFS Implementation

Graph500 has a classic parallel top down implementation of BFS. It follows the top down algorithm shown in Figure 4. The BFS tree is grown from a random root. At every step, all the nodes in the frontier parallely attempt to mark any unvisited neighbours as the next level. The visited neighbours become the next frontier and the process repeats till there are no nodes left in the frontier.

```
function breadth-first-search(graph, key)
  frontier ← {key}
  next ← {}
  tree ← [-1,-1,...-1]
  while frontier ≠ {} do
    top-down-step(frontier, next, tree)
    swap(frontier, next)
    next ← {}
  end while
  return tree
```

```
function top-down-step(frontier, next, tree)
  for v ∈ frontier do
    for n ∈ neighbors(v) do
      if tree(n) = -1 then
        tree(n) ← v
        next ← next ∪ {n}
      end if
    end for
  end for
```

Figure 4 BFS Algorithm

Memory Simulator

Since PIM cores are not yet in production we had to use a memory simulator in order to obtain some first order results on the speed ups which PIM affords.

USIMM

USIMM, the Utah SImulated Memory Module [15], is a DRAM main memory simulator. The simulator consists of scheduler, memory controller and OS modules. The simulator consumes workload traces in the front end and then models a reorder buffer (ROB) for each processor. Memory accesses with each ROB are kept in read and write queues for each channel. The scheduler picks a command for each channel at every memory operate cycle. USIMM models all the DRAM states, timing, performance and power metrics. In addition to the low level controls there exists an OS module which can be used for adding OS page table translation logic.

USIMM is created to simulate standard DRAM configuration. We had to tweak the code a bit to simulate PIM. We visualized our PIM core to resemble a device similar to Micron's Hybrid Memory Cube. The 64 GB RAM device consists of 16 channels or vaults which consists of 8 banks each. Each bank has 65536 rows with 128 columns. Each cell consists of a cache line of 64 bits.

The standard 64 GB DRAM configuration we compared against comprised of 4 channels, 2 ranks, 8 banks with 65536 rows and 256 columns. The cache line was 64 bits long.

Instrumentation

PIN Tool

We created a PIN tool which generated the traces which are required as input to the USIMM simulator. The PIN tool was written such that it collected the instruction pointer, whether the memory access was a read or write operation, the memory address.
The PIN tool was programmed to

I. Collect traces for specific functions specified at as command line arguments. This was required so that we collect traces only for the functions which are executed in parallel in graph500.
II. print the number of cycles spent by CPU between two memory accesses

A sample trace is shown below:

```
[Number of cycles between two thread memory accesses] [read/write] [address] [instruction pointer]
#
# Memory Access Trace Generated By Pin
#
1 W 0x7fff7a3578d0 0x4017ae r
1 W 0x7fff7a3578c8 0x4017b5
1 R 0x7fff7a3578d8 0x4017bc
1 W 0x7fff7a357918 0x4017c3
```

Experiments

A number of possible configurations for PIM are possible. In one possible configuration PIM can be imagined as set of processors having high bandwidth and a host processor offloading parallel task to the network of PIMs. This resembles the GPU architecture where the host processor invokes a kernel on the streaming multiprocessors and each processor executes this in parallel. Other possible use of PIM can be as a co-processor where it can perform some kind of pre-processing on the data before sending it to the main processor. For instance, simple reductions on arrays can be performed by PIM logic. Since the PIM logic is closer to the data, these reductions can be performed with little or no memory bottlenecks.

In our project, we have tried to experiment with PIM treating it like a GPU. We collected traces for the OpenMP version of Graph500 on a jinx 4 core node. Since we collected traces at the thread level we obtain 16 different traces for one run of Graph500. We map each of the processor core's hardware thread level traces to one PIM core in our simulation. We ran the simulator in two different settings.

I. In the first setting we provide the PIM device with no extra OS help. The PIM cores underneath each vault may have to access portions of memory not in its vault. This is akin to just connecting a PIM device to a system and not tune the system in any way.
II. In the second configuration we assume some basic OS support for the PIM device.
    A. We have ensured that the OS maps the addresses accessed by a particular PIM core to the vault directly above it. We modified USIMM's page table mappings to achieve this.

B. This ensures that PIM cores act independently of each other and can exploit the full memory capacity of the stacked memory.
C. This is similar to having a GPU with PIM cores acting like lock-step cores. The real advantage in this scenario is that instead of having data limited to the cache size in GPUs, the PIM cores will have access to large areas of main memory. The overhead of CPUs transferring data to GPUs is also removed.

We also experimented the effect of PIM on another well-known bandwidth bound algorithm – the Fast Fourier Transform (FFT).

Results

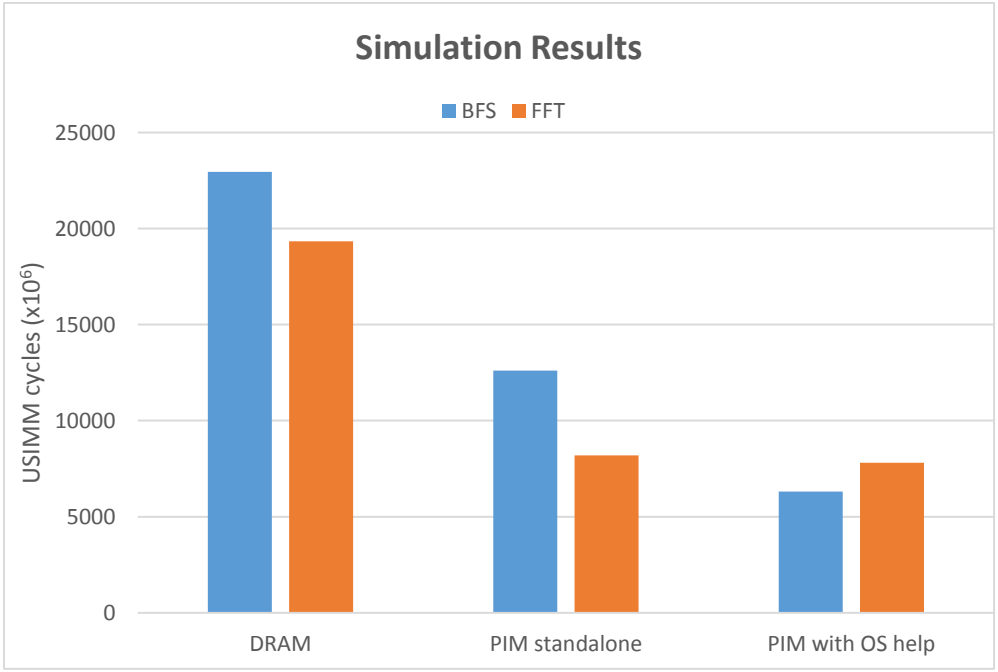
We collected traces for graph500 for an input scale of 16 and an edge factor of 16. These traces were then passed through the 3 different memory configurations of our experiment. The same experiment was repeated with FFT for a matrix size of 65536.

The results were in tune with our expectations. PIM without OS support provided a speed up of 2x. This seems to occur due to the high channel bandwidth afforded to PIM. However with some basic OS support this can be further boosted to 3.6x for BFS and 2.5 for FFT. The OS helps in ensuring that each thread accesses the memory only in its assigned address space.

Table 1. Simulator Readings

	BFS			FFT		
	USIMM Cycles	Speed Up Relative	Speed Up Overall	USIMM Cycles	Speed Up Relative	Speed Up Overall
DRAM	2294607300	1	1	1934259317	1	1
PIM standalone	1260736584	1.82005292	1.82005292	818987430	2.361769236	2.361769236
PIM with OS help	630386621	1.999941848	3.64	781123047	1.048474287	2.476254317

Figure 5 Memory cycles chart



Limitations

As with all simulation based experiments there are a few shortcomings with respect to this project.

1) USIMM is a trace based simulator. Since traces are fixed and don't depend on the simulation timings these results can only be taken as first order approximations.
2) We assume that the OS level mappings to separate vaults exist. This may be true for embarrassingly parallel operations like that of BFS but may not be true for other workloads.

Future Work

For future work we would like to

1. Experiment PIM with different types workloads and algorithms
2. Experiment different configuration of PIM where PIM may behave as a co-processor or model PIM as MPI like architecture where different PIM cores can communicate with each other.
3. Design programming abstractions for PIM like architecture.

References

[1] WILLIAM H. KAUTZ, “Cellular-logic-in-memory arrays," IEEE Trans. Computers- vol C18 pp.719-729, Aug 1969

[2] H. S. Stone. “A Logic-in-Memory Computer”, IEEE Transactions on Computers, 19(1):73–78, January 1970.

[3] Wm. A. Wulf and Sally A. McKee. 1995. “Hitting the memory wall: implications of the obvious”. SIGARCH Comput. Archit. News 23, 1 (March 1995), 20-24. DOI=10.1145/216585.216588 http://doi.acm.org/10.1145/216585.216588

[4] M. Gokhale, B. Holmes, and K. Iobst. “Processing in Memory: The Terasys Massively Parallel PIM Array”, IEEE Computer, 28(42):23–31, 1995.

[5] www.graph500.org

[7] Gabriel H. Loh, Nuwan Jayasena, Mark H. Oskin, Mark Nutter, David Roberts, Mitesh Meswani, Dong Ping Zhang, Mike Ignatowski, AMD Research – Advanced Micro Devices Inc. “A Processing-in-Memory Taxonomy and a Case for Studying Fixed-function PIM”.

[8] Pawlowski, J. Thomas. "Hybrid memory cube (HMC)." In Hotchips, vol. 23, pp. 1-24. 2011.

[9] Topol, Anna W., D. C. La Tulipe, Leathen Shi, David J. Frank, Kerry Bernstein, Steven E. Steen, Arvind Kumar et al. "Three-dimensional integrated circuits." IBM Journal of Research and Development 50, no. 4.5 (2006): 491-506.

[10] High-level Programming Model Abstractions for Processing in Memory, Michael L. Chu Nuwan Jayasena Dong Ping Zhang Mike Ignatowski AMD Research Advanced Micro Devices, Inc.

[11] Pugsley, Seth H., Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li. "NDC: Analyzing the Impact of 3D-Stacked Memory+ Logic Devices on MapReduce Workloads." In International Symposium on Performance Analysis of Systems and Software. 2014.

[12] Low, Yucheng, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. "Distributed GraphLab: a framework for machine learning and data mining in the cloud." Proceedings of the VLDB Endowment 5, no. 8 (2012): 716-727.

[13] Kumar, Vijay, and Eric J. Schwabe. "Improved algorithms and data structures for solving graph problems in external memory." In Parallel and Distributed Processing, 1996, Eighth IEEE Symposium on, pp. 169-176. IEEE, 1996.

[14] Malewicz, Grzegorz, Matthew H. Austern, Aart JC Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. "Pregel: a system for large-scale graph processing." In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pp. 135-146. ACM, 2010.

[15] Chatterjee, Niladrish, Rajeev Balasubramonian, Manjunath Shevgoor, S. Pugsley, A. Udipi, Ali Shafiee, Kshitij Sudan, Manu Awasthi, and Zeshan Chishti. "Usimm: the utah simulated memory module." University of Utah, Tech. Rep (2012).