A Processing-in-Memory Taxonomy and a Case for Studying Fixed-function PIM

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.
{gabriel.loh, nuwan.jayasena, mark.oskin, mark.nutter, david.roberts, mitesh.meswani, dongping.zhang, mike.ignatowski}@amd.com

Abstract

The emergence of die-stacking technology with mixed logic and memory processes has brought about a renaissance in "processing in memory" (PIM) concepts, first envisioned decades ago. For some, the PIM concept conjures an image of a complete processing unit (e.g., CPU, GPU) integrated directly with memory, perhaps on a logic chip 3D-stacked under one or more memory chips. However, PIM potentially covers a very wide spectrum of compute capabilities embedded in/with the memory. This position paper presents an initial taxonomy for in-memory computing, and advocates for the exploration of simpler computing mechanisms in the memory stack in addition to fully-programmable PIM architectures.

1. Introduction

Processing in memory (PIM) is a decades-old concept of placing computation capabilities directly in memory [11]. The PIM approach can reduce the latency and energy consumption associated with moving data back-and-forth through the cache and memory hierarchy, as well as greatly increase memory bandwidth by sidestepping the conventional memory-package pin-count limitations.

Motivated by the "Memory Wall" [12], PIM research reached a feverish pitch in the mid-to-late 1990's with a variety of proposals and studies [1,4,6–9] and test chips and systems developed [3]. Fabrication limitations and business models worked against PIM being adopted widely in industry; instead, advances in memory interfaces (e.g., RAMBUS, DDRx) have been the main focus of industrial innovation in memory system design.

Recent advances of die-stacking (3D) technology have reignited interest in PIM architectures. A silicon die implemented in a high-performance technology process can be stacked with one or more memory layers. The basic technology building blocks are already in place; for example, Micron's Hybrid Memory Cube technology [10] along with several academic projects [2,5] have already demonstrated the 3D stacking of logic and memory chips.

Past proposals have largely focused on fully-programmable processors (including vector/SIMD and special-purpose). However, there exists a continuum of compute capabilities that can be embedded "in memory". This paper presents a taxonomy of this design space. In addition we argue for continued research in less-than-fully-programmable PIM approaches.

2. A Processing-in-Memory Taxonomy

In this section, we present a taxonomy for different types of, or approaches to, PIM design. Figure 1 shows our taxonomy; the classes are largely divided by how the computing would (likely) interface with software, but they also have implications on the area requirements and power-performance

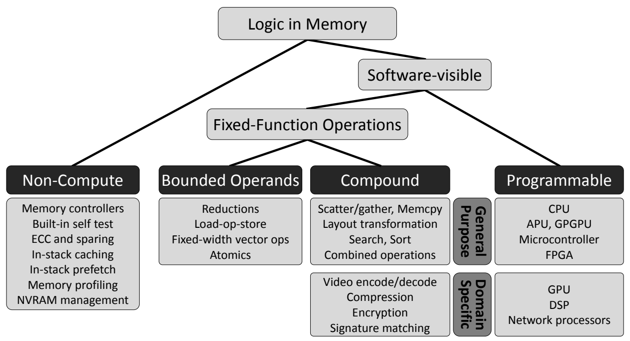


Figure 1: Taxonomy of processing in memory.

efficiency. For the non-compute class, there is no software interface because these all implement software-transparent features. Fixed-function operations with bounded operands map closely to ISA-level instructions and could be accessed via assembly-level intrinsics. Fixed-function operations with compound operands and computations likely would correspond to invoking library calls. Fully-programmable PIMs would be employed with standard (although possibly PIM-enhanced) paradigms such as threading packages, GPGPU programming interfaces (e.g., OpenCL), or whatever is appropriate for the specific type of computing device.

2.1. Non-compute Logic in Memory

For completeness, the first class in our taxonomy covers uses of logic in memory that act invisibly to application software. For example, many of the functions of the Micron Hybrid Memory Cube approach fall under this category, such as the in-stack integration of memory controllers and built-in self-test capabilities [10]. While this may have significant performance, power, or cost benefits, such benefits are largely invisible to the software stack. While there are many promising research directions for such software-transparent applications of logic in memory, this class is not the focus of this paper.

2.2. PIM with Fixed-function Operations

The remaining classes of PIM computation styles are all software-visible. The next level of our taxonomy differentiates based on whether the in-memory computing provides pre-defined or fixed functions, or whether the computation substrate is more generally programmable.

2.2.1. Fixed-function, Bounded-operand PIM Operations

The bounded-operand PIM operations (BPO) can be specified in a manner that is consistent with existing instruction-level memory operand formats. Current load and store instructions specify an address, and the size of the operation is implied/encoded by the opcode (e.g., load byte, store double-word). Simple extensions to this format could encode the PIM operation directly in the opcode, or perhaps as a special prefix in the case of the x86-64 ISA, but no additional fields are required to specify the memory operands. To contrast, a non-bounded operand could consist of a base address and vector length, whereby the size of the data to be manipulated is dynamically determined based on the arguments supplied to the operation, but such an operand format is not typical of modern ISA encodings.[1]

A bounded operand need not be limited to a single word of data. BPOs may operate on multiple words in a limited-vector form, much like how modern x86 SIMD extensions manipulate fixed, pre-defined vectors (i.e., each operand is a single, predetermined, fixed-sized collection of data words). Given the theme of maintaining typical instruction encodings, BPOs likewise would be expected to perform only a single operation (e.g., add) or a very limited, pre-specified set of operations (e.g., test-and-set, multiply-accumulate).

2.2.2. Fixed-function, Compound PIM Operations

Compound PIM operations (CPOs) may access an arbitrary number of memory locations (not-specifically pre-defined) and perform a number of different operations. Some examples include data movement operations such as scatter/gather, list reversal, matrix transpose, and in-memory sorting. In these examples, a non-PIM system would have to read entire data structures all the way into a CPU's registers, simply to have to write all of the contents back out to memory again in a different order. For common patterns and paradigms, simpler fixed-function implementations may provide significantly more area-efficient, lower-power, and higher-performance implementations than performing the equivalent work with a fully-programmable processor in memory.

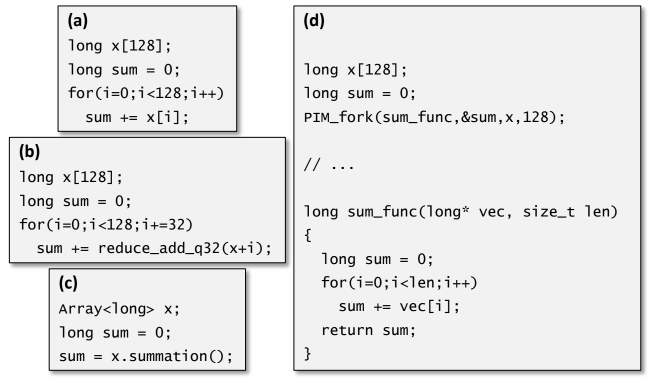
CPOs can cover functionality similar to that provided by BPOs but over more complex sets of data. Whereas an example BPO might perform a reduction (e.g., summation) over a fixed block of data, a CPO could perform a similar reduction over a dynamically specified array, or even a combined operation such as performing an in-memory gather followed by a reduction on the gathered elements. Further, CPOs may be synchronous or asynchronous depending on the nature and expected latency of the computation.

2.3. Fully-programmable PIM

The last class in our taxonomy, and the domain of much previous PIM work, is fully-programmable logic in memory. These solutions provide the expressiveness and flexibility of a conventional processor (or configurable logic device), along with all of the associated overheads except off-chip data migration.

3. Why Fixed-function PIM?

The previous section described our taxonomy for PIM-like computing designs. In general, the different classes varied from one end being fixed-function to the other extreme of full programmability. Past work mostly focused on either end of the spectrum. This position paper argues that there is significant middle ground worth further exploration, for


(a)
```
long x[128];
long sum = 0;
for(i=0;i<128;i++)
  sum += x[i];
```
(b)
```
long x[128];
long sum = 0;
for(i=0;i<128;i+=32)
  sum += reduce_add_q32(x+i);
```
(c)
```
Array<long> x;
long sum = 0;
sum = x.summation();
```
(d)
```

long x[128];
long sum = 0;
PIM_fork(sum_func,&sum,x,128);

// ...

long sum_func(long* vec, size_t len)
{
  long sum = 0;
  for(i=0;i<len;i++)
    sum += vec[i];
  return sum;
}
```

Figure 2: Vector summation code examples using (a) conventional programming, (b) a BPO intrinsic, (c) a CPO in a library call, and (d) a fully-programmable PIM invocation. (Initialization of the array `x` **omitted for brevity.)**

which the proposed taxonomy can serve as a framework for classifying and ultimately better understanding the strengths and weaknesses of the different classes of PIM. In this section, we briefly walk through a simple PIM example to highlight some of the potential tradeoffs of the different approaches, and then discuss key areas for further research explorations.

For our example, we consider an array of 128 integer quad-word (64-bit) values to be added together (a total of 1,024 bytes of data).[2] Figure 2(a) shows conventional C-like code for performing this computation without any PIM functionality. This code would generate 128 load operations (or 16 reads from memory assuming a 64-byte cacheline size) just to compute a single sum.

BPO Implementation: For the sake of illustration, we will assume a BPO summation operation of the form `reduce.add.q32 dest [src]` that performs an addition reduction on the thirty-two quad-word elements (q32) starting at the address pointed to by the `src` register, and the summation is placed in the register `dest`. Figure 2(b) shows the pseudo-code in which `reduce_add_q32()` would be an intrinsic call to invoke the corresponding assembly-level operation. The code is still fairly straightforward, although just as with SIMD operations in conventional processing cores, the highest-performing code will rely on the programmer to write efficient machine- or assembly-level code. This code would require four requests to, and corresponding responses from, main memory (i.e., the four `reduce.add.q32` commands) to compute the final 128-element summation.

The PIM hardware required to support such a BPO is fairly straightforward and quite efficient. Internal to the memory stack, the PIM would only have to implement a hard-wired, 32-way summation tree (which can be relatively fast using Wallace tree-like organizations) to compute the sum. Space-time trade-offs can be made; for example, a 16-way summation tree can be used once on each half of the 32-element array.

CPO Implementation: A possible CPO summation operation could take the form of an `Array` library call `summation()`.

[1]There certainly exist operations that support non-bounded operands, for example, the x86 REP-prefixed instructions, but these are not typically used outside of a few specialized libraries (e.g., memcpy).

[2]This is but a simple example, but even vector summation (and its variants) has many common uses such as computing prefix sums and long-vector dot products.

Figure 2(c) shows the pseudocode invoking such a PIM operation. Compared to the BPO version, this CPO summation hides the PIM-level details from the programmer. Under the covers, the library implementation would map this to a lower-level call that unpacks the array specifications (base address and size) and passes these to the PIM hardware. From a memory-bandwidth perspective, the CPO is very efficient because it generates only a single request to main memory and receives a single response with the final summation.

A key trade-off for the CPO summation is greater hardware complexity compared to BPO. The PIM in this case requires some control logic to iterate through the array accumulating the results. While the logic may add multiple elements in parallel to speed up the process, it is an inherently variable-latency operation depending on the array size.

Full-PIM Implementation: Finally, Figure 2(d) shows a summation operation used with a hypothetical fully-programmable PIM interface ("PIM_fork"). The programmer-level implementation is fairly simple, although it can be argued that this is still quite a bit more work when compared to the CPO and BPO versions. The benefit is that such a framework provides significant flexibility in that the function off-loaded to the PIM need not be restricted to a finite set of predefined, fixed-function operations.

From an efficiency standpoint, a full processor in the PIM incurs additional area, power, and possibly performance overheads compared to the fixed-function implementations. The processor must store the code for the PIM function; fetch, decode, and execute the instructions, likely implemented as a fully-pipelined processor including register files, caches, etc.; potentially support the same ISA as the host processor (if *any* code can be offloaded to the PIM); support virtual memory; and handle thread synchronization and cache coherence with the host (assuming a standard threading model is employed). To be fair, much of this would have to be performed anyway if the summation was *not* offloaded to the PIM at all, but compared to the fixed-function design points, a fully-programmable PIM introduces significantly more complexity, requires more area in the logic-layer of the PIM stack, and consumes more energy.

4. Other Issues and Extensions

The PIM taxonomy presented in this paper is but one possible way to categorize the spectrum of PIM-like architectures. To some extent, the exact taxonomic groupings and specific details are not even that important to the central premise of this position paper, which is to advocate for more explorations in fixed-function PIM approaches. However, we will now briefly discuss some other design concerns that currently are not captured by our taxonomy, and these issues may serve as the basis for extensions or improvements in future PIM taxonomies.

4.1. PIM and Cache Coherence

Whether the PIM hardware supports cache coherence with the host processor(s) is not a design aspect currently covered by our taxonomy, but this is an area that would have significant impacts on the complexity of the PIM implementation. Even with such a simple example as the reduction operation shown in Section 3, all PIM implementations would need some way of ensuring that the memory operands being operated on by the PIM are correct. There are a variety of possible solutions, each with strengths and weaknesses. For example, the PIM could behave just like any other cache-coherent processor, which would require it to perform directory look-ups, issue probes/invalidations, and otherwise generate a potentially significant amount of coherence traffic before proceeding with its intended operation. In the pathological case in which the PIM operation operates on large sets of data that already are mostly resident in the on-chip caches, it may end up taking more time and power to force all of the relevant data to be flushed back to memory for the PIM to work on it than to just let the CPU perform the computation directly out of the caches.

The programmer could instead be forced to explicitly flush the needed operands from the caches to memory prior to the invocation of the PIM operation. For simple sets of data (e.g., base+length), this could be a relatively simple operation (although it may still require a significant amount of coherence traffic). If the data set is highly irregular (e.g., gather/scatter with pointer indirection), such a CPU-pushed cache flush operation becomes more complex. There are other possible approaches that could and should be researched, but this is a cross-cutting PIM design issue that affects both fixed-function and fully-programmable approaches.

4.2. PIM and Virtual Memory

Unless PIM operations are restricted to accessing memory within a single page, some type of virtual memory support will be required. For example, take the simple CPO summation from Section 3, but consider the situation when the target array crosses page boundaries. For this CPO to operate correctly, the PIM logic must compute the *virtual* addresses of the array elements, translate them to physical addresses, and then access and add the data. This could require hardware page-table walkers in the PIM [3] and possibly TLBs for accelerating translations, and incur all of the system complexities that come with supporting virtual memory. However, An intriguing possibility is to integrate this translation support with the translation mechanisms being built for I/O devices and hardware virtual machine and RDMA support.

While array operations potentially could be constrained to operate within page boundaries (which would place an additional burden on the programmer or compiler to manage this), other potentially powerful PIM operations that support pointer-based/indirect memory accesses definitely will require virtual memory translation support. For example, walking a linked-list or other pointer-based data-structure requires converting

[3] PIM-based page-table walkers themselves would be an instance of a Non-Compute operation in our taxonomy, and could be fairly efficient because each step in a multi-level page-table look-up potentially can access page-table data structures directly without going through the traditional on-chip cache hierarchy.

the virtually-addressed pointer to a physical address for every hop in the list traversal. An indirect gather (e.g., gather via an array of pointers) would have similar requirements.

Supporting virtual memory in the PIM also would raise the possibility of invoking page-faults part-way through a PIM operation, requiring the need for precise interrupts on the PIM device itself, as well as a method to signal the operating system. How should system/PIM state be saved/restored when the PIM is half-way through a 1,000-element operation? Aborting the entire operation and restarting after the page-fault has been handled may be grossly inefficient (e.g., having to redo almost all of the work when the fault occurs on the 999,999th element of a million-element array), and could also expose livelock problems (e.g., handling the page fault causing a different portion of the array to be paged out, and then faulting on that page when re-executing the PIM operation, which then forces yet another page to be paged out...).

4.3. Other Issues

Many other possible features, attributes, etc. may be useful in distinguishing different classes of PIM-like architectures. The virtual-memory and cache-coherence issues discussed in this section highlight how even seemingly simple operations (e.g., vector summation) can open up a host of design challenges and potential complexity. Apart from the simple (but necessary) studies needed to quantify the performance, bandwidth, and energy potentials of fixed-function PIM approaches, beneath the surface remain many systems and implementation questions that must be addressed through more research.

5. Conclusions

To be clear, this paper is *not* making the conjecture that fixed-function PIMs are necessarily any better than the fully-programmable approaches (they are not even mutually exclusive). The point of this position paper is to advocate further research into the BPO and CPO classes of PIM designs to better understand the strengths and weaknesses of the different approaches so future PIM architects can make the best trade-offs among implementation costs, performance, power, flexibility, and complexity. To this end, we advocate more research in at least the following areas: (1) application analysis to identify which functions are worth moving to memory, and for each whether the function is best implemented as a BPO, CPO, or a fully-programmable PIM; (2) area, power, and performance trade-offs of fixed-function versus fully-programmable PIM architectures; (3) programmability studies to determine the best ways to expose (or hide) PIM functionality to (or from) the application writers; and, (4) studies of the operating system issues surrounding the support of different classes of PIM computing.

References

[1] D. G. Elliott, W. M. Snelgrove, and M. Stumm. Computational RAM: A Memory-SIMD Hybrid and its Application to DSP. In *Proc. of the Custom Integrated Circuits Conference*, Boston, MA, May 1992.

[2] D. Fick, R. Dreslinski, B. Giridhar, G. Kim, S. Seo, M. Fojtik, S. Satpathy, Y. Lee, D. Kim, N. Liu, M. Wieckowski, G. Chen, T. Mudge, D. Sylvester, and D. Blaauw. Centip3De: A 3930 DMIPS/W Configurable Near-Threshold 3D Stacked System With 64 ARM Cortex-M3 Cores. In *Proc. of the Intl. Solid-State Circuits Conference*, pages 190–191, San Francisco, CA, February 2012.

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

[4] 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 *SC*, 1999.

[5] D. H. Kim, K. Athikulwongse, M. B. Healy, M. M. Hossain, M. Jung, I. Khorosh, G. Kumar, Y.-J. Lee, D. L. Lewis, T.-W. Lin, C. Liu, S. Panth, M. Pathak, M. Ren, G. Shen, T. Song, D. H. Woo, X. Zhao, J. Kim, H. C., G. H. Loh, H.-H. S. Lee, and S. K. Lim. 3D-MAPS: 3D Massively Parallel Processor with Stacked Memory. In *Proc. of the Intl. Solid-State Circuits Conference*, pages 188–190, San Francisco, CA, February 2012.

[6] Y. Kim, T.-D. Han, S.-D. Kim, and S.-B. Yang. An Effective Memory-Processor Integrated Architecture for Computer Vision. In *Proc. of the Intl. Conf. on Parallel Processing*, pages 266–269, Bloomington, IL, August 1997.

[7] R. C. Murphy, P. M. Kogge, and A. Rodrigues. The Characterization of Data Intensive Memory Workloads on Distributed PIM Systems. In *the Second International Workshop on Intelligent Memory Systems*, 2000.

[8] M. Oskin, F. T. Chong, and T. Sherwood. Active Pages: A Computation Model for Intelligent Memory. In *Proc. of the 25th Intl. Symp. on Computer Architecture*, pages 192–203, Barcelona, Spain, June 1998.

[9] D. A. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A Case for Intelligent RAM. *IEEE Computer*, Mar/Apr 1997.

[10] J. T. Pawlowski. Hybrid Memory Cube: Breakthrough DRAM Performance with a Fundamentally Re-Architected DRAM Subsystem. In *Hot Chips 23*, 2011.

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

[12] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *Computer Architecture News*, 23(1):20–24, March 1995.