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.
{mike.chu, nuwan.jayasena, dongping.zhang, mike.ignatowski}@amd.com

Abstract

While the idea of processing in memory (PIM) has been around for decades, both hardware and software limitations have kept it from growing in practical, real-world use. Recent advancements in 3D die-stacking technology have begun to make inroads towards solving some of the implementation issues, but software programmability questions remain. This position paper presents high-level programming models as a solution for PIM programmability by abstracting away many of the low-level architectural details. While we acknowledge that expert programmers still will want low-level, detailed control for optimization, we see high-level programming abstractions as a way to broaden the use of PIM to a larger audience of developers and increase the adoption of PIM architectures in future systems.

1. Introduction

Processing in memory (PIM) 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.

While PIM architectures have been studied extensively in the past [1][2][3][4][5][6][8], they have never gained traction in real-world systems for a variety of reasons. Two significant technical reasons stand out: using a memory process technology to build computational logic resulted in poor performance, and the lack of a coherent programming model for developers made it difficult to exploit the underlying hardware. The first problem was caused because logic implemented in a memory process typically was several generations behind contemporary logic processes in terms of performance. The process problem led to either slow-performing processors or highly specialized PIM solutions geared only towards specific operations, limiting their applicability. The second problem of programmability emerged because PIM is not easy for developers to understand and reason about in the context of their programs. Developers generally are not used to thinking about allocating into different physical memories and coordinating execution corresponding to their data location. Past research on PIM programmability largely focused on compiler techniques or low-level APIs to extract code for execution [4][7].

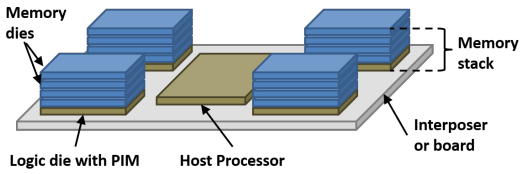

Figure 1: Example compute node design with PIM

While these techniques also abstract away the architectural details, we believe higher-level, library-based abstractions using common data structures and algorithms will increase the adoption of PIM greatly by improving ease of programmability, portability, and the range of supported applications.

Only recently have process technology advancements improved to the point at which solving the first problem is now a possibility. By using 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. These processors can be fully programmable CPUs and GPUs or other types of accelerators. An example of such a PIM system is shown in Figure 1. PIM logic sits below each memory stack associated with a host processor [9].

While solutions to the process technology that would enable the ability to building a fully programmable PIM system are now visible on the horizon, the programmability questions of how to coordinate and execute work with a PIM architecture still remain. This paper focuses on the software side of PIM: how we can improve the programmability aspects and gain traction with developers to increase the adoption of PIM. With parallel programming and GPGPU techniques now common, many of the basic parallelization concepts needed for PIM abstractions are well understood by the software development community. Thus, we advocate for development of high-level programming models to abstract away the low-level architecture details and make the underlying PIM effectively transparent to developers. We propose data structures and algorithms that are fully PIM-aware, and automatically partition them across a PIM system by default, with options allowing expert developers the freedom to fine-tune and optimize their code for increased performance or better energy efficiency.

2. Low-level APIs for PIM

While this paper focuses on the development of high-level programming models for PIM, it assumes a basic set of low-

level APIs that would be necessary to interact with a PIM architecture.

Three basic APIs for PIM can allow for development of a significant amount of high-level abstractions: PIM device discovery, memory allocation/deallocation, and PIM computation launch. Device discovery includes determining the number and querying the capabilities of each PIM processor. Memory allocation requires APIs similar to malloc() and free() to allocate and deallocate on a specific memory. A launch method is necessary to execute computation on the PIM processor.

3. High-level Programming Models for PIM

From a developer's standpoint, the main difficulties in programming for a PIM system are deciding how to partition the data across the memories and, once partitioned, how to coordinate the execution of tasks on their respective data. Both of these decisions are non-trivial and can change drastically the performance and correctness of a program. In many ways, a current high-level programming model for distributed systems such as MPI can map well; however, MPI is targeted towards systems with much more expensive communication, whereas a PIM system is more tightly coupled with lighter-weight communication. Programming models such as OpenMP and OpenACC are much more similar. This paper is not suggesting an alternative to these models, but is advocating ideas and directions for possible incorporation into future versions.

This section provides an example of a C++ template-based solution that is cognizant of the underlying PIM hardware. The basic principles behind this design are to make sensible defaults, so a non-expert programmer can benefit from PIM, and to be robust enough to allow domain experts who understand the architecture to optimize their code.

3.1. PIM Device Discovery

The first step in programming for a PIM-enabled system is device discovery. A programmer needs to be able to query the number and capabilities of the available PIMs. Device discovery is important for PIM software development for code portability towards systems with differing PIM capabilities or even legacy systems with no PIM at all.

```
class pim_info {
public:
  pim_info();
  ~pim_info();

  void initialize(pim_device_type device_type,
                  uint32_t num_entries = UINT_MAX);
  void push(pim_device_id & id);
  uint32_t size();
  pim_device_id * list_of_pims();
  pim_device_id& operator[] (int index);
}
```

Listing 1: A PIM info class to capture PIM devices

Listing 1 shows a class that can encapsulate this information. The pim_info class defaults to initialize all the available PIM devices in the system, but can be modified easily to include only a subset of the PIM devices based on capabilities using the initialize() method.

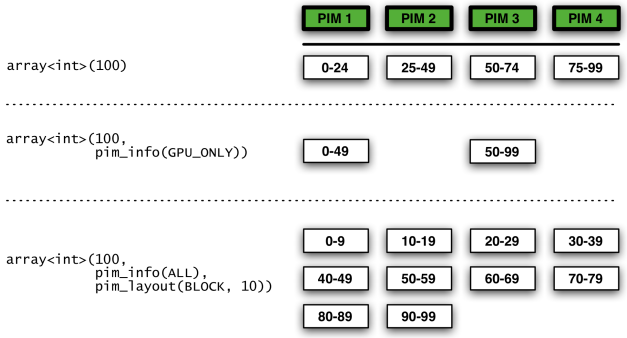

Figure 2: Various array declarations and how their indices would be mapped to the a machine with four PIMs.

This pim_info class can be used as an argument across the PIM data structure constructors and algorithms to specify on which PIM devices to execute. However, in keeping with the principle to have sensible defaults to improve ease of programming, all constructors and algorithms allow the absence of the pim_info declaration. When not specified, the programming model defaults to using all PIMs in the system.

3.2. PIM-aware Data Structures

The next major issue is the partitioning of data across different memories cognizant of the associated PIM memories. We believe that enabling PIM-aware partitioning in familiar data structures is an excellent method to introduce the benefits of PIM to developers.

A basic multi-dimensional array class is presented in Listing 2. This class uses the pim_info class argument to define which PIMs to partition the data across. A pim_layout argument allows the array to be evenly partitioned across each PIM, or in a blocked fashion with a specific chunk size per PIM. Otherwise, the array can be used as expected like any other C++ array. The PIM array class uses the low-level PIM API to allocate memory and keeps track of which array indices are mapped to which separate PIM memory. Figure 2 shows how the PIM array is allocated to the PIM memories given various options available. These different options can give the programmer more fine-grain control over where data is placed based on capabilities of the underlying PIM or access characteristics of their code.

```
template<typename T, int N=1>
class array {
public:
  array(int size, pim_info& info=default_info,
        pim_layout& layout=default_layout);
  array(extent<N>& ext, pim_info& info=default_info,
        pim_layout& layout=default_layout);
  ~array();
  T& operator[] (int index);
  size_t pim_id(int index);
  size_t size();
}
```

Listing 2: A PIM-aware array class

3.3. Coordination of PIM execution

The final challenge for a PIM programmer is the launching and coordination of tasks on individual PIM processors. For best performance and energy efficiency, computation should spawn on the processor corresponding to the data being processed.

Launching work on the PIM processors require two steps from the programmer: specifying the function to execute and initializing the execution. Listing 3 shows an example pim_function class that is the abstract base class for PIM computation. This class has one pure virtual function, run(int index), which is overridden to provide an implementation for what needs to execute per element within a data structure.

```
class pim_function
{
public:
    virtual void run(int index) = 0;
};
```

Listing 3: A PIM function abstract class

Once the computation function has been specified, it needs to be launched. A parallel_for_each algorithm was chosen because it is an easily understood concept among programmers and common in both parallel and GPGPU programming domains. The parallel_for_each algorithm is shown in Listing 4. There are two input arguments to the parallel_for_each: an array, which specifies the domain to iterate over, and the kernel to execute.

```
template<typename T, int N, typename Func>
void parallel_for_each(array<T, N> &domain, Func f,
                       parallel_options opt=PER_PIM);
```

Listing 4: A PIM-aware parallel_for method

By default, the parallel_for_each algorithm spawns a single thread per PIM device on the machine, and each thread executes the run() function on every element of the array associated with that PIM's memory. By using the parallel_options argument, however, a programmer can change this to spawn a separate thread for each element in the array using a PER_ELEMENT parallel_options. The parallel_for_each is currently blocking and thus the host waits at the end of its call for all computations launched to complete. Additional investigation into barriers and synchronization could lead to other models of host/PIM interaction such as asynchronous spawns.

3.4. Example

Listing 5 shows an example of how this programming model could be used to specify a kernel, create data across a PIM-enabled architecture, and launch the kernels accordingly. The process_func class defines the kernel for execution. It specifies that each element of the array should be input into the process() method and written to an output array. The main function then simply needs to create the input and output arrays, initialize the input, and call parallel_for_each.

```
class process_func:public PIM::pim_function {
public:
  process_func(PIM::array<foo> &input,
               PIM::array<foo> &output) :
    in(input),
    out(output) { }

  virtual void run (int i) {
    out[i] = process(in[i])
  }

  PIM::array<foo>& in;
  PIM::array<foo>& out;
};

int main() {
  PIM::array<foo> in(512);
  PIM::array<foo> out(512);
  initialize(in);

  PIM::parallel_for_each(in, process_func(in, out));
}
```

Listing 5: Example use of the PIM high-level model

The high-level model abstracts away the PIM concepts for the programmer; each array is partitioned automatically across the available PIM memories, and the parallel_for_each simply launches the kernel across each of the PIM devices. This was chosen as the default method of partitioning because it is the easiest concept to understand and the most commonly used. However, if programmers had specific knowledge of their algorithm or data mapping, they could fine-tune this program in many ways. For example, a pim_info could specify only PIM devices with certain capabilities and pass that to the array constructor. The parallel_for_each then would launch only on those devices.

4. Future directions

There are many directions to continue this research of PIM-aware high-level models. This prototype was intended as a study of the viability of high-level programming constructs mapping to a low-level PIM API. More sophisticated data structures and algorithms need to be investigated. An array maps fairly easily because its memory generally is contiguous and can be partitioned logically across different memories. How this translates to hashes, trees, or other common irregular data structures remains to be studied. For example, a tree could have an edge- or vertex-based partitioning, but how these work in practice remains to be studied.

Another area of additional study in raising the level of PIM abstractions is in algorithms. A parallel_for_each method maps over to PIM in a straightforward manner; other algorithms like reduce, shuffle, or search need investigation. We envision that many of these can fit into a library that supports common algorithms for PIM.

Another necessary area for future study is PIM-aware synchronization and barriers. The current method, shown in this paper, waits at the end of a parallel_for_each for all the threads to complete. The addition of synchronization primitives could allow for an asynchronous launch and a significant

amount of overlap between host and PIM execution.

The final major area for additional study lies with runtime scheduling for PIM. The model shown in this paper is very programmer-centric when it comes to the scheduling of work by the parallel_for_each method. A more robust system could include a runtime to handle scheduling of work and management of work and data locations. Such a runtime could allow for the execution of irregular task workloads and open up PIM to many new applications.

5. Conclusions

This paper presents a case for research of PIM-aware high-level programming models. As process technology techniques begin to solve issues with poor-performing hardware, research into the software stack can help drive adoption of the architecture. High-level abstractions of underlying hardware using common data structures and algorithms are possible with only a simple set of low-level APIs for querying PIM devices, allocating memory and launching tasks. While this paper shows the start of developing a PIM-oriented high-level model, a significant amount of investigation still lies in data structures, PIM-enabled algorithms and synchronization and runtime scheduling to help raise the abstraction of and lower the barriers to entry for, PIM.

References

[1] D. G. Elliott, M. Stumm, W. M. Snelgrove, C. Cojocaru, and R. McKenzie. Computational RAM: Implementing processors in memory. *IEEE Design & Test of Computers*, 16(1):32–41, 1999.

[2] B. R. Gaeke, P. Husbands, X. S. Li, L. Oliker, K. A. Yelick, and R. Biswas. Memory-intensive benchmarks: IRAM vs. cache-based machines. In *International Parallel & Distributed Processing Symposium*, 2002.

[3] M. Gokhale, W. Holmes, and K. Iobst. Processing in memory: The Terasys massively parallel PIM array. *IEEE Computer*, 28(4):23–31, 1995.

[4] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, and J. Lacoss. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. In *Supercomputing*, 1999.

[5] Y. Kang, M. W. Huang, W. Huang, S. Yoo, Z. Ge, V. Lam, D. Keen, Z. Ce, V. Lain, P. Pattnaik, and J. Torrellas. FlexRAM: Toward an advanced intelligent memory system. In *IEEE International Conference on Computer Design*, pages 192–201, 1999.

[6] P. M. Kogge, J. B. Brockman, T. Sterling, and G. Gao. Processing in memory: Chips to petaflops. In *Workshop on Mixing Logic and DRAM: Chips that Compute and Remember at ISCA '97*, 1997.

[7] J. Lee, Y. Solihin, and J. Torrellas. Automatically mapping code on an intelligent memory architecture. In *IEEE International Symposium on High Performance Computer Architecture*, pages 121–132, 2001.

[8] M. Oskin, F. T. Chong, and T. Sherwood. Active pages: a computation model for intelligent memory. In *ACM/IEEE International Symposium on Computer Architecture*, pages 192–203, 1998.

[9] 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 *Workshop on Memory Systems Performance and Correctness*, 2013.