**Processing-In-Memory Baseline Architecture and API**

June 18, 2013

Executive Summary

Processing-in-memory (PIM), which moves computation closer to where the data resides, has long been recognized as having the potential to improve both the performance and energy costs associated with memory access. Previous research in this area focused on implementing processing capabilities directly on DRAM chips. This has yielded poor results due to incompatibilities between the CMOS processes for DRAM and high performance processor logic. With the advent of 3D stacking, it has become possible to implement processors in high performance CMOS as one layer in a larger 3D memory stack. This research report investigates the opportunities for exploiting such PIM capabilities for future exascale systems.

Rather than focus on details of a hardware design proposal, our PIM research takes a programmability-centric view. We believe that rational choices about future hardware implementations for PIM first require a more thorough understanding of the programmability requirements, coupled with research into what features of a PIM device lead to the most cost effective improvements in performance and energy efficiency for targeted applications.

This report outlines a baseline architecture that incorporates an accelerated processing unit (APU) for PIM sharing a cache coherent global address space with the host processor and other PIM devices in the same node. This architecture greatly simplifies the process of porting, debugging and optimizing applications for execution on PIMs at the cost of some additional hardware complexity. We use this baseline model as a starting point for evaluating the architecture and programming model.

We describe ways to extend familiar programming languages to incorporate PIM using API layers on top of C-based programming models. Device discovery, memory allocation, and dispatching work to PIMs are the key functions enabled by the APIs. The APIs could potentially be extended to other high level programming languages, and can serve as the basis for higher level programming abstractions for use by a broader set of application developers. The APIs have been written to be ISA independent to enable broader applicability to a variety of systems.

We adapt the mechanisms of the Heterogeneous System Architecture (HSA) as the framework for dispatching work to PIM devices. HSA was initially developed at AMD as a framework for efficient integration of programmable accelerators and has been adopted by a variety of vendors through an industry consortium.

While the base architecture enables a single SMP operating system to control the scheduling of tasks on the host processors and PIMs, we have found that using a device driver model with a light weight kernel (such as *Kitten* from Sandia National Laboratories) managing tasks on each PIM is likely to be much more efficient.

The overall power/performance benefits of PIMs are highly dependent on the hardware implementation and application characteristics. However, our estimates show that the energy efficiency (bytes/nano-Joule) for handling a cache miss within a PIM is 1.5x-2x better than for a host processor accessing stacked memory on an interposer, and ~2.5x better than an optimized case for accessing stacked memory over board wiring using ultra-short-range SerDes.

The cost–benefit of implementing hardware cache coherence for a PIM is also highly dependent on the hardware implementation and application characteristics. We examined a number of options for implementing both host-PIM and PIM-PIM coherence, and concluded that suitable coarse-grain tracking methods, such as region coherence, can be implemented with low hardware overhead and minimal coherence traffic.

Incorporating PIM into system management functions, instrumentation, checkpoint restart, RAS and debug-ability tools is also addressed. An outline of a projected path for adoption of PIM functions is defined.

Table of Contents

[1 Introduction 1](#_Toc359310255)

[2 Background 3](#_Toc359310256)

[3 Baseline PIM Architecture 4](#_Toc359310257)

[3.1 Host Memory Interface 5](#_Toc359310258)

[3.2 APU-in-Memory 5](#_Toc359310259)

[3.2.1 Initiating Operations on PIM 6](#_Toc359310260)

[3.2.2 Receiving Results from PIM 8](#_Toc359310261)

[3.2.3 Reporting Completion and Error Codes 8](#_Toc359310262)

[3.2.4 Unified Virtual Address Space 8](#_Toc359310263)

[3.2.5 Node-Level Global Memory Access 9](#_Toc359310264)

[3.2.6 Inter-PIM Interconnect 10](#_Toc359310265)

[3.2.7 Instruction Storage 11](#_Toc359310266)

[3.2.8 Cache Hierarchy Considerations 12](#_Toc359310267)

[3.2.9 Cache Coherence Cost/Benefit Considerations 15](#_Toc359310268)

[3.2.10 Address Striping 22](#_Toc359310269)

[3.3 Hardware Parameters 24](#_Toc359310270)

[3.4 Specialized Operations 26](#_Toc359310271)

[4 User-level API 28](#_Toc359310272)

[4.1 Scope 28](#_Toc359310273)

[4.1.1 Discovering Available PIM Resources 29](#_Toc359310274)

[4.1.2 Orchestrating Memory Data Layout 29](#_Toc359310275)

[4.1.3 Dispatching PIM Computations 29](#_Toc359310276)

[4.1.4 Providing Visibility of Intended Memory Usage 29](#_Toc359310277)

[4.2 API Specification 29](#_Toc359310278)

[4.2.1 pim\_get\_device\_ids 30](#_Toc359310279)

[4.2.2 pim\_get\_device\_info 32](#_Toc359310280)

[4.2.3 pim\_malloc 35](#_Toc359310281)

[4.2.4 pim\_reloc 37](#_Toc359310282)

[4.2.5 pim\_free 38](#_Toc359310283)

[4.2.6 pim\_map 39](#_Toc359310284)

[4.2.7 pim\_unmap 40](#_Toc359310285)

[4.2.8 pim\_spawn 40](#_Toc359310286)

[4.2.9 pim\_get\_id 42](#_Toc359310287)

[4.3 Execution Model and Synchronization 42](#_Toc359310288)

[4.4 Platform-Specific Semantics 43](#_Toc359310289)

[4.4.1 POSIX Threads 43](#_Toc359310290)

[4.4.2 OpenCL 44](#_Toc359310291)

[5 System Integration 47](#_Toc359310292)

[5.1 System Software Considerations 47](#_Toc359310293)

[5.2 RAS, Checkpoint Restart and Debug 49](#_Toc359310294)

[5.3 System Management 50](#_Toc359310295)

[5.4 Instrumentation 50](#_Toc359310296)

[6 A Potential Path for PIM Adoption 52](#_Toc359310297)

[7 Case Studies 56](#_Toc359310298)

[7.1 Simple Tests 56](#_Toc359310299)

[8 References 58](#_Toc359310300)

# Introduction

This report is provided in partial fulfillment of the M1 milestone described in the *Extreme-Scale Computing Memory Research and Development Statement of Work*. The milestone is reproduced below, and this architecture and interface description report deliverable is given in italic. To complete the fulfillment of the M1 milestone, a PIM emulator software package and a report documenting its operation are provided separately.

**Milestone M1 (Architected Programming Interface & Emulator for PIM)**

The Subcontractor shall develop an architected programming interface for PIM. This programming interface shall be in the form of a report setting forth the following:

* Methods to initiate operations on a PIM and receive results consistent with the HSA architecture
* Methods to report various types of completion codes and errors
* Methods to integrate PIM with system management, checkpoint restart, debug, and instrumentation
* Cost/benefit analysis of cache-coherent PIM implementations

Additionally, the Subcontractor shall develop PIM emulator software to emulate APUs-in-memory on standard hardware.

**Deliverables**:

* *A report documenting newly created architecture and interface descriptions with supporting cost/benefit analysis*
* APU-in-Memory Emulator Software
* A report that documents the operation and internal functioning of the Emulator Software

Improvements in bandwidth and latency to off-chip memory have not kept up with rapid increases in computational capabilities that have resulted from the advancement of processor implementation technology and architectures. Further, an increasing proportion of power in computing systems is being spent on data movement, with off-chip memory accesses being a major contributor. These problems are exacerbated for emerging workloads that exhibit memory-intensive behaviors with irregular access patterns and limited data reuse as well as for traditional HPC applications with high bandwidth to computation ratios. Moving computation closer to where the data resides has the potential to improve both performance bottlenecks and energy costs associated with memory accesses.

We investigate the benefits of reducing the aggregate distance of data movement, and the associated energy consumption, through processing-in-memory (PIM) implemented using 3D die stacking. We present a baseline architecture that incorporates an accelerated processing unit (APU) for PIM and serves as a starting point for evaluating the hardware architecture design space for die-stacked in-memory processing. We also describe an application programming interface (API) that allows programmers to map applications to aPIM-enabled node architecture.

While there are a variety of implementation challenges to be addressed in stacking processing dies with memory, the industry is well on the path to addressing these as evidenced by progress on various emerging die-stacked memory standards and consortia [1][2][3]. Understanding the implications of the new capabilities and challenges presented by PIM for programmability and usability of computing systems, however, is in its infancy. Therefore, our PIM research takes a programmability-centric view. We explore ways to extend familiar programming languages and models to incorporate PIM and to encompass PIM capabilities within emerging frameworks for heterogeneous computing. Our API layers on top of C-based programming models and enables us to support and evaluate PIM within those programming models. The APIs could potentially be extended to other high level programming languages, and can serve as the basis for higher level programming abstractions for use by a broader set of application developers. Our initial efforts focus on POSIX threads for task-based applications and OpenCL for data-parallel applications. We adapt the mechanisms of the Heterogeneous System Architecture (HSA) [4] as the framework for dispatching work to PIM devices. HSA was initially developed at AMD as a framework for efficient integration of programmable accelerators and has been adopted by a variety of vendors through an industry consortium [5].

# Background

PIM has been a topic of interest in the research community for several decades. The early proposals explored the integration of caches and computation at a time when large scale integration was in its infancy [6]. PIM architectures integrating DRAM and computation attracted significant attention in the research community around the turn of the century [7][8][9][10][11][12]. Many of those efforts focused on one of two approaches. Projects such as IRAM [8] integrated embedded DRAM on logic chips. However, this approach could not cost-effectively accommodate sufficient memory capacity to form the primary memory of mainstream high-performance systems due to the reduced density of embedded memory. Efforts such as ActivePages [9], FlexRAM [10], and DIVA [11] integrated logic on memory dies. However, due to the reduced performance of logic implemented in DRAM processes (typically multiple process generations behind leading-edge logic processes), such approaches resulted in in-memory processors with reduced performance or highly specialized architectures geared only for limited sets of operations. This in turn limited the applicability and programmability of such PIM devices which, along with the cost implications of reduced DRAM density due to the presence of compute logic, limited the adoption of such solutions.

In this project, we revisit PIM concepts utilizing 3D die stacking to tightly couple processors implemented in a logic process with memories implemented in a memory process using through-silicon via (TSV) technology. This approach presents a different tradeoff in the PIM design space that has somewhat lower bandwidth than processors and memory implemented on the same die but circumvents the issues of logic performance and memory density encountered by prior approaches. As the processing is implemented in a logic process, this approach enables the incorporation of general-purpose processors “in” memory, providing the opportunity to support familiar programming models and lower the programmability and usability barriers. Further, as the memory dies themselves need not incorporate the in-memory compute units, this approach enables the use of commercially-viable memory dies.

Other recent research efforts [13] as well as retrospectives by some researchers who had led previous PIM research projects [14][15] have advocated 3D die stacking for PIM, validating the viability of this approach. We anticipate our research to add unique value with respect to other contemporary research in this space in two ways. First, we explore PIM architectures with APUs that integrate both GPUs and CPUs, enabling us to evaluate the impact of PIM on both throughput-bound computations and latency-bound computations (and applications with phases that exhibit both). Second, such anarchitecture enables us to support fully-general programming models allowing us to explore the benefit of executing arbitrary user-defined code in memory (as opposed to limiting in-memory execution to pre-defined sets of operations).

# Baseline PIM Architecture

Figure 1 shows an overview of AMD’s proposed FastForward node organization. This baseline architecture integrates an Exascale Heterogeneous Processor (EHP) capable of performing 12 TFLOPS and eight stacks of high-bandwidth DRAM capable of providing an aggregate 4TB/s to the EHP in a single package. Additional high-speed interfaces are provided to the system interconnect and off-package non-volatile memory (NVM) providing an aggregate capacity of 1TB.

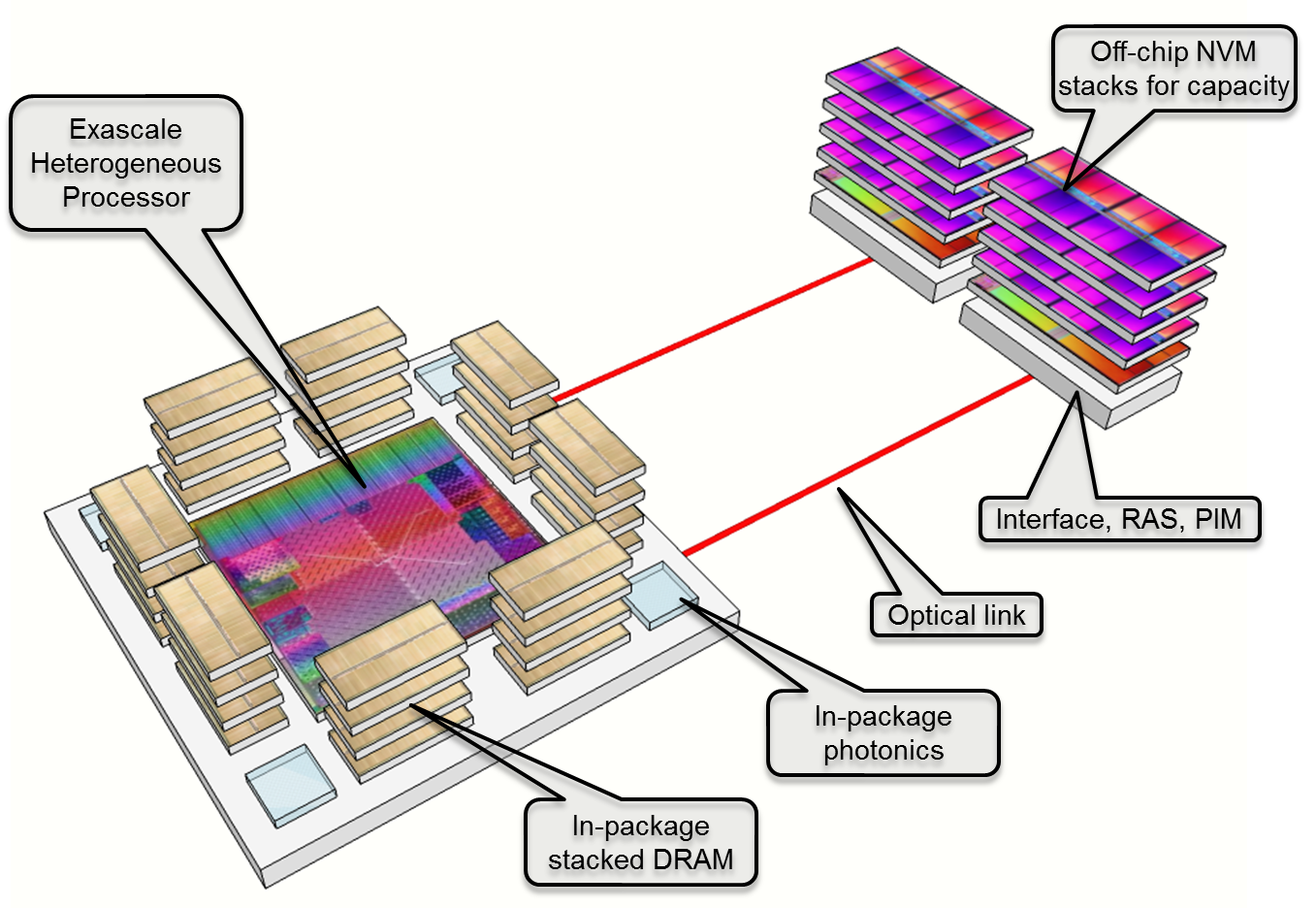


Figure 1: AMD FastForward Proposed Baseline Node Architecture

The PIM project considers the impact of integrating processing capabilities within the memory stacks. These in-memory processing devices will function under the control of the EHP as a host processor.

In the current phase of the project leading up to the M1 milestone, we have developed a baseline PIM architecture and infrastructure necessary to evaluate the performance and energy implications of implementing PIM capabilities in logic layers stacked with main memory. While our architecture and infrastructure are designed to easily accommodate PIM stacked with either or both of the main memory types shown in Figure 1 – in-package DRAM and off-chip non-volatile memory (NVM) – our efforts to date have primarily focused on PIM stacked with DRAM.

This section describes our baseline PIM architecture.

## Host Memory Interface

The memory dies of each memory stack must service read/write requests received via the external host interface as well as from in-stack PIM. In order to enforce adequate fairness and quality of service policies among these requestors, the DRAM or NVM controller(s) must be implemented within the memory stack, conceivably on the PIM die or another logic die within the memory stack. Therefore, the host interface does not explicitly expose the timing nor the exact physical structure of the memories. Instead, host requests are issued as high-level commands at the granularity of loads and stores (instead of RAS/CAS/precharge etc., as is the case with today’s commodity DRAM interfaces) and responses from the memory stack occur after an arbitrary latency.

Note that the above organization maps well to memory interface standardization activity ongoing within the memory community. JEDEC HBM task group is defining solutions for interfacing a processor die to a memory (DRAM) stack [1]. Such an interface could be used to stack PIM dies under HBM memory stacks. An interface with abstract host access characteristics is already in use by the Hybrid Memory Cube (HMC) DRAM architecture , which was initially proposed by Micron and is being standardized through an industry consortium . Early PIM designs may elect to adopt variants of such emerging or existing interface standards.

Please refer to Section 6 for a discussion of the tradeoff space in host interface (and host architecture in general) specialization for PIM and the corresponding capabilities enabled.

## APU-in-Memory

Our baseline architecture implements an APU as the in-memory processor in each of the memory stacks. This enables the execution of both CPU- and GPU-oriented general-purpose code in the memory stacks. This fully-programmable architecture has several advantages over more specialized PIM architectures:

* **Lower barrier to initial adoption by developers**: Unlike the highly specialized processors used in some prior PIM research, the APU enables execution of arbitrary code on a PIM device. Therefore, no extensive rewrite is necessary to make use of PIM – merely identifying and annotating which code segments are to run on PIM is sufficient from a correctness point of view.
* **Availability of development tools**: The rich set of development tools (compilers, debuggers, profilers etc.) available for general-purpose processors can be readily exploited for PIM development.
* **Reduced round-trips between host and PIM devices**: Entire tasks or kernel graphs (i.e., collections of tasks and kernels) can be dispatched to PIM without having to return control back to the host processor after each individual task or kernel executed on a PIM device.
* **Incremental and targeted optimizations**: Even within the code that is to execute on PIM devices, developers can focus their optimization efforts on sections that have the most impact on performance and energy while leaving other parts of code unchanged.
* **Broader applicability**: Full programmability enables PIM capabilities to be exploited in emerging applications that may not have been anticipated at PIM design time.
* **Economies of scale**: An APU-based PIM device (along with the attached memory stack) may be usable as a standalone, low-power, integrated processing solution in a variety of non-HPC markets.

While we adopt an APU as our baseline PIM architecture, we have intentionally minimized dependencies on specific instruction set architecture (ISA) features or other vendor-specific capabilities to the extent possible. As such, we expect that the architectural concepts and programming interfaces described in this document to be widely applicable across different ISAs and processor architectures.

### Initiating Operations on PIM

As the PIM devices are programmable APUs, a number of general purpose work dispatch methods may be used. Here we look at three options, consider their suitability for performance, scalability, and virtualization, and lastly examine paths for extensions or future studies. Note that the discussion here considers the tradeoffs in underlying implementations and the details discussed here are not exposed to application programmers. The API described in Section 4 provides a higher-level abstraction for users to manage work dispatch and relates aspects.

One option for work dispatch is to use an inter-processor interrupt (IPI) to signal work from the host, or signal completions from the PIM device.   Depending on the operating system (OS) and runtime environment, the IPI processing path length can be long, and the burden of handling asynchronous interrupts from many PIM devices can add uncertainty to host operation.  This option may be undesirable in an environment with many PIM devices, or where frequent work dispatch is required, and would be incompatible with work dispatch on HSA. Therefore we do not consider this option further here.

Another option would be for the host to dispatch work by storing to memory-mapped registers or buffers on a PIM device (a.k.a., MMIO).  An MMIO interface would be consistent with PCIe attached devices, which may not share an address space with the program running on the host.  The MMIO option may be unattractive for performance because MMIO loads/stores are typically uncached, resulting in lower throughput and higher latency than cacheable memory accesses.  The MMIO option may also be undesirable in an environment requiring virtualization as it requires extra effort from the host OS, so we do not consider MMIO further here.

In our baseline architecture, PIM devices share page tables with the host (Section 3.2.4 provides a detailed discussion of this aspect). So the third option we explore is work dispatch via shared memory.  This option is attractive versus IPI because it offers scalable, low-overhead communication.  This option is attractive versus MMIO because load/store to cacheable shared memory is efficient, and backing pages can be virtualized by the host OS.  Because unstructured communication via shared memory lacks architectural definition, we leverage HSA's user-mode-queue (UMQ) facility [4].  The HSA UMQ is an architected, low-overhead interface for work dispatch that allows user-level software to submit command requests into a circular buffer in shared memory.  We explore the UMQ option in more detail here.

From a software perspective under the UMQ model, tasks to be executed on a PIM are placed on a task queue in user-space memory and the PIM is notified of the availability of new work.[[1]](#footnote-1) Each entry on a PIM task queue, at a minimum, contains:

* **Identifier of function to execute**: In case of user-defined code, this may be a function pointer or kernel identifier.
* **Pointer to data arguments**: Pointer to one or more buffers in user-memory that contain values of, or pointers to, arguments for the operation.

In addition to the above, each entry may contain implementation-dependent meta-data.

As is the case with HSA, each application or process creates its own PIM dispatch queues. Each queue is tied to a specific PIM device, and in our baseline, the placement of work on a specific queue uniquely determines which device that work will execute on.[[2]](#footnote-2) As with HSA, the hardware architecture supports a single application or process creating multiple queues per PIM device. However, for simplicity our initial user-level API described in Section 4 only supports a single queue per PIM device per application.

A key challenge with queued communication is the determination by the consumer of the availability of new information on the queue. Polling has well-understood inefficiencies. Further, polling can also lead to reduced responsiveness in cases where a large number of queues may need to be monitored. Instead, modern processors incorporate *monitor/wait* mechanisms that allows the processor to specify an address on which to wait for the availability of new data (e.g., producer writes new queue entries in the UMQ while the consumer executes monitor/wait). While the hardware-based resources used to implement such monitor/wait mechanisms are limited in today’s processors, we envision processor vendors provisioning a sufficient number of such resources in future hardware which can then be further augmented via virtualization and/or remapping in software to provide a solution that scales to arbitrary numbers of queues to be monitored.**[\*]** We adopt these monitor/wait mechanisms for signaling from the host to the PIM devices as well as from PIM to the host for UMQ-based communication.

Quantitatively characterizing proxy-applications that run on PIM, and exploring options for efficient UMQ dispatch to collections of PIM devices, as well as the impact of various signaling capabilities back to the host, may be paths for future studies to further improve work dispatch efficiency.**[\*]**

### Receiving Results from PIM

PIM devices share a single virtual address space with the host in our baseline architecture as discussed in Sections 3.2.4 and 3.2.5. Therefore, applications may pass arbitrary data structures by reference as input arguments to PIM computations via the API described in Section 4. This enables host and PIM devices to communicate arguments and results with the full flexibility of a shared-memory multi-processor.

### Reporting Completion and Error Codes

As is envisioned for HSA, UMQ can also be used for communicating status and error information back to the host. Such information can be placed by the PIM on separate queues monitored by the host. These may include application-level status and result return queues monitored by user code as well as queues owned and monitored by the runtime software framework for system management, completion-based dependency handling, and error handling purposes.

Certain classes of high-priority system-level exceptions may be communicated to the host via dedicated side-band channels. For example, memory exceptions (e.g., page faults) incurred by the PIM devices may need to be communicated to the host under certain OS configurations (see Section 5.1 for a discussion of OS implications of PIM). As handling these exceptions often requires kernel intervention, UMQ would not necessarily reduce the overhead of communicating these. As such, they may be communicated to the host using standard interrupt mechanisms.

### Unified Virtual Address Space

Our baseline architecture implements a single unified address space among the host and PIM devices. This decision enables several desirable capabilities, including:

* **“A pointer is a pointer”**: Pointers can be de-referenced correctly on any of the processors, enabling the passing of pointers and pointer-based data structures among the host and PIM devices. This is a crucial enabler for some classes of applications with irregular data structures.
* **User-mode work dispatch**: As all devices within the node have access to the same address space, passing of data structures and control information between the host and PIM devices can be done through memory with full generality without the need for OS intervention. This reduces the overhead associated with work dispatch, thereby allowing finer granularity task dispatch and frequent communication among the devices. This is a crucial capability for enabling a wide range of complex and irregular applications to use PIM effectively. User-mode work dispatch is also an important aspect of HSA.
* **System software organization flexibility**: A shared address space enables multi-processor OS capabilities to be extended to PIM devices. Please refer to Section 5.1 for a more detailed discussion of system software implications.
* **No memory pinning**: As the PIM processors in our architecture are able to operate on virtual addresses and handle page faults, we avoid the need to pin memory or impose other restrictions on the residency of memory pages or regions operated on by PIM processors.

While a single, shared virtual address space enables a range of desirable capabilities, providing this abstraction requires the host and PIM devices to share a single set of page tables for virtual to physical address translations. The specific implementation of shared page tables will depend on system-level considerations. As discussed in Section 5.1, the approach that would require the least evolution in host OS infrastructure would be to expose the PIM processors as accelerators via a device driver model (as is the case with early HSA accelerators). To support such an approach, our baseline architecture incorporates an *IO memory management unit* (IOMMU) similar to the ones used with HSA accelerators [17]. An IOMMU provides PIM devices the necessary mechanisms to read a shared set of page tables maintained by the host and to request modifications to them via memory exception handling requests. In our baseline architecture, we incorporate an IOMMU per memory stack.

A further optimization is to distribute the page tables among the memory stacks such that a PIM device has not only the data it frequently accesses but also the corresponding page table entries in the local stack. This allows the in-stack IOMMU to perform the translations without the need for significant off-stack accesses.

While the above device-driver-based baseline allows the PIM devices read access to the page tables with minimal requirements on OS evolution, any updates to the page tables must be performed by the host[[3]](#footnote-3). In this model, a PIM device that needs a page table update (e.g., handling a page fault) must interrupt the host and request the desired update. However, more advanced implementations that allow the PIM devices to autonomously update page tables can be envisioned, and may be desirable in cases where PIM devices may cause frequent page table modifications. Such implementations may range from designs that allow each PIM to manage a subset of the page table entries (e.g., a sub-tree of the page table hierarchy rooted at a page directory entry) to fully general implementations that allow any PIM to update any part of the page tables. Understanding the system-level implications of such schemes requires further research and is not considered in detail in this milestone. Such schemes are also deemed less relevant for HPC workloads where memory allocation is carefully managed to minimize or eliminate page faults.

### Node-Level Global Memory Access

As this document discusses the architecture within a node, we use the terms “global memory” and “node-level global memory” interchangeably to refer to all of the main memory within a node.

Given the high bandwidth, low-energy access PIM devices have to memory within the local stack, a key aspect of optimizing an application to use PIM is to partition performance-critical data structures among the memory stacks within the node such that most PIM accesses are to the local memories. However, constraining a PIM device to only access its local memory stack significantly raises the barrier to programmability. Such a constraint would require developers to guarantee that all memory accessed by any code executed on a PIM device is resident in its local memory stack. Implications of this range from replicating shared data (e.g., tables, constants) in all stacks and added complexity and capacity pressure in data layouts (e.g., ensuring halos for stencil computations are replicated in the local stack) to significantly greater code complexity, reduced PIM dispatch granularity, increased data movement among memory stacks, and increased synchronization (e.g., for input-dependent or dynamic irregular data structures).

In keeping with the programmability focus of our work, we provision capabilities for any PIM device to access any main memory within the node. Naturally, access to memory within the local stack provides greater bandwidth and consumes less energy. And, therefore, performance-critical code should ensure that the majority of accesses from a PIM device are to the local memory stack. However, the availability of node-level global memory access from PIM devices avoids the complexities mentioned above for infrequently accessed data and corner cases allowing the developers to focus their optimization efforts on performance-critical sections of code.

Architectural mechanisms for enabling global memory access from PIM devices are discussed in Sections 3.2.4 and 3.2.6.

### Inter-PIM Interconnect

As discussed in Section 3.2.5, the proposed baseline node architecture allows any PIM to access any main memory within the node. From an implementation perspective, this requirement implies the need for a communication medium among the PIM devices and memory stacks within any given level of the memory system.[[4]](#footnote-4) At a high-level, this may be achieved via one of two approaches:

* **A dedicated “inter-stack” (or “inter-PIM”) interconnect**: This implements a dedicated network among the memory stacks of a level of the main memory hierarchy. While this adds the overhead of implementing such an interconnect, it reduces the need for extensive changes to the host processor architecture to support PIM (please see Section 6 for a more detailed discussion of the implications of PIM on host processor architecture). It is worth noting that the HMC architecture [16] already incorporates the ability to support an interconnection network among multiple memory stacks. While the HMC interconnect is geared for a different usage scenario (memory system scaling) and does not incorporate all features that would be desirable for an inter-PIM interconnect, this design provides a proof-point on the viability of such an approach and the opportunity for PIM systems to build upon existing and emerging industry-standard capabilities.
* **Adapt existing interconnect**: This approach re-uses existing interconnect, typically at the next higher level in the memory hierarchy. For example, PIM at the first level memory (DRAM in our proposed node) may send non-local access requests to the host to be routed to the appropriate memory stack via the host’s on-die memory switch. Similarly, PIM at the second level (NVRAM in our proposed node) may request handling of the communication from the first level memory (which, in turn, may send the request to the host as described above). This approach minimizes additional interconnect resources but requires extensive changes to the host. In other words, the host must be able to service autonomous requests from the memory system while today’s processors only receive read responses from memory modules. Please see Section 6 for a more detailed discussion of the implications of PIM on host processor architecture.

We adopt the first solution above as our baseline. A simplified diagram of such an organization with four memory stacks and a ring interconnect is shown in Figure 2. We anticipate bandwidth needs of the inter-stack interconnect to be one to two orders of magnitude lower than local stack memory bandwidth per PIM. However, broad-based application studies are needed to determine specific bandwidth requirements as well as potential interconnect topology optimizations.**[\*]**

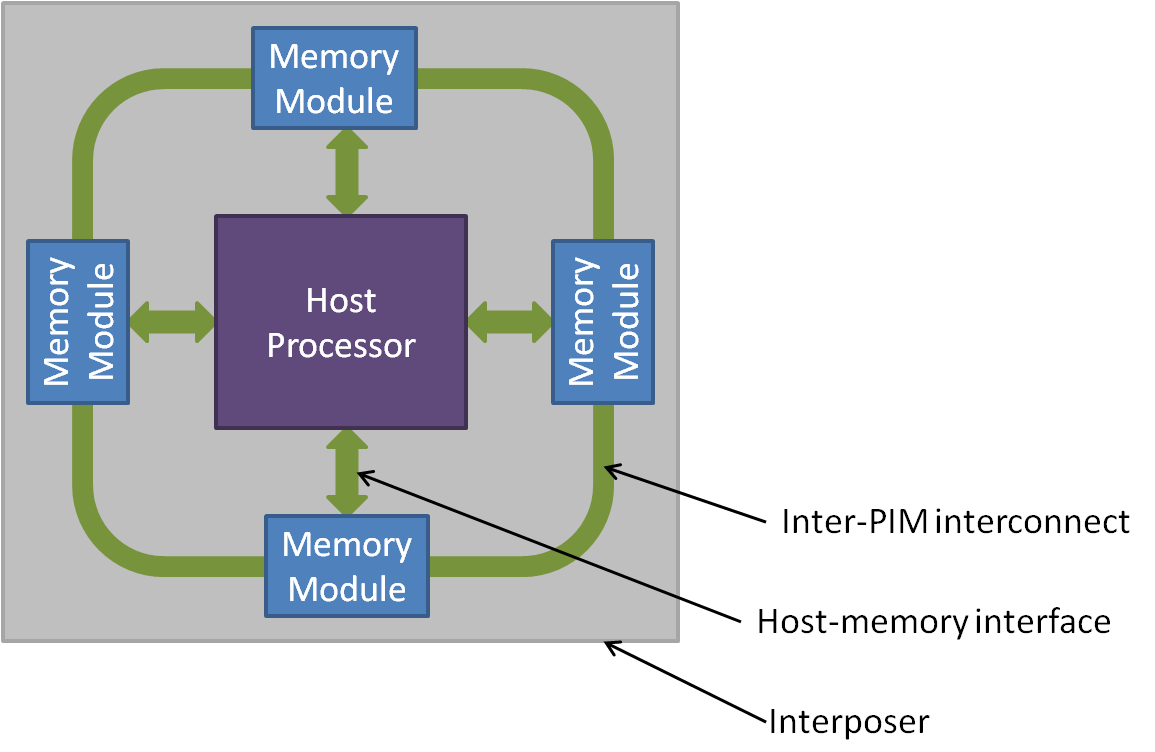


Figure 2: Simplified example of inter-stack (or inter-PIM) interconnect

### Instruction Storage

Ensuring PIM execution fetches instructions from local memory reduces the performance and energy overheads associated with instruction fetch. However, as our baseline hardware architecture allows node-level global memory accesses to be performed by any PIM device, it does not dictate any specific placement of instructions for PIM. We defer to software layers for techniques to ensure common-case PIM instruction fetch from local memory stacks.

System software optimized for PIM-enabled systems can play a crucial role here. As an example, on the first invocation of an OpenCL kernel *K* on a PIM device *P*, a copy of the compiled code for *K* could be created by the run-time software in the memory stacked with *P*. Memory space devoted to kernel storage in such a system may be bounded by imposing an (application-specific) upper limit and managing that space as a software-managed instruction cache under the control of the runtime system. Replacement of kernels in such a cache may adopt policies based on profiling that allow optimization for specific applications or may use simpler application-agnostic policies (such as *least-recently-used*).

### Cache Hierarchy Considerations

Some key candidate application patterns for PIM acceleration are those with memory access patterns that are not amenable to effective caching and, therefore, cannot make use of the deep and large cache hierarchies often found on high-performance host processors. However, even data access patterns that are typically considered uncacheable can often benefit from the coalescing, bandwidth amplification, and energy reduction effects a small- to medium-sized cache can provide. Therefore, we consider L1 and L2 cache organization an important aspect of PIM design. Large L3 (or last-level) caches, however, are likely to be less effective for PIM processors given the high bandwidth and lower latency to memory as well as the nature of applications likely to be mapped to PIM devices.

One possible organization of L1 and L2 PIM caches is to closely mirror the architecture of a conventional processor (or more specifically in our baseline, an APU) such as the EHP. Figure 3 shows a high-level overview of such an organization. This figure assumes GPU cluster caches similar to the EHP [18]. Further, although the CPU cores for PIM are expected to use low-power processors instead of the high-performance variants expected in the EHP, they too incorporate conventional cache hierarchies. This provides for a well-understood organization as well as design reuse with non-PIM APU designs.

Note that, in the organization shown in Figure 3, there are no shared caches among the different types of compute units of the PIM. Therefore suitable cache coherence mechanisms must be implemented among the L2 caches and such coherence policies must preferably mirror those among the CPU and GPU cores of the host to ensure uniformity of programming models[[5]](#footnote-5). Further, any global data sharing among the units must occur through DRAM (or NVM, as the case may be) or via cache coherence mechanisms. This organization also has implications for the implementation of fixed-function accelerators. Such accelerator hardware must either be associated with the CPU or GPU in order to gain the use of a cache or must implement their own caches. While the incorporation of a shared L3 solves some of these issues, it adds area overhead and adds to the memory latency and cache complexity of PIM, even though most PIM-friendly use cases may not be amenable to high L3 cache utilization.

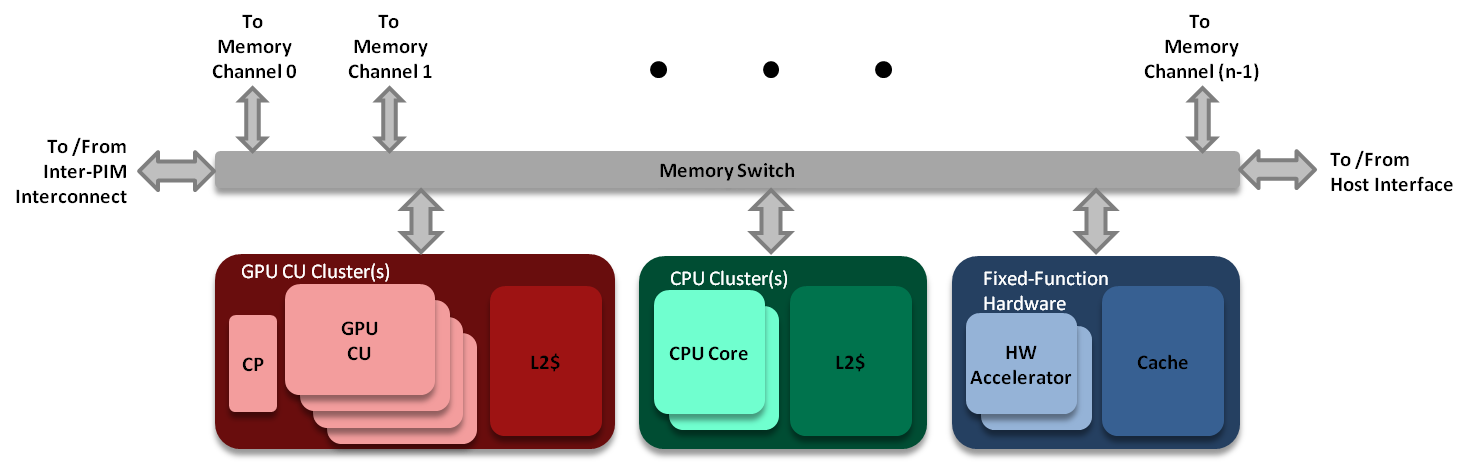


Figure 3: PIM organization with conventional processor-attached L2 Caches (L1 caches within GPU CUs and CPU cores not shown for simplicity; optional fixed-function hardware shown)

Figure 4 shows an alternative cache organization where the second level caches are aligned with memory channels and are shared among all units of the PIM to enable data sharing via the L2. This banked, shared implementation results in a slight increase in latency for PIM CPU L2 accesses and potentially higher memory switch bandwidth demands for GPU accesses. However, this organization unifies the switching requirements of a banked L2 cache with those of the memory switch, resulting in potential implementation overhead reductions. Further, in cache coherent implementations, the directory structures may also be banked along memory channels providing localized coherency operations associated with an L2 bank (please refer to Section 3.2.9 for a detailed discussion of cache coherence options). Further, fixed function units may also share the single L2 cache with minimal additional overheads. And the single logical L2 alleviates the need to keep multiple L2s within a PIM coherent. A key downside of this approach is the potential increase in energy due to increased memory switch traffic. However, that overhead is partially offset by the smaller size of each channel-attached L2 cache accessed per request. Additionally, L2 cache organization of PIM devices differ significantly in this scheme from mainstream processor designs, making design reuse more challenging.

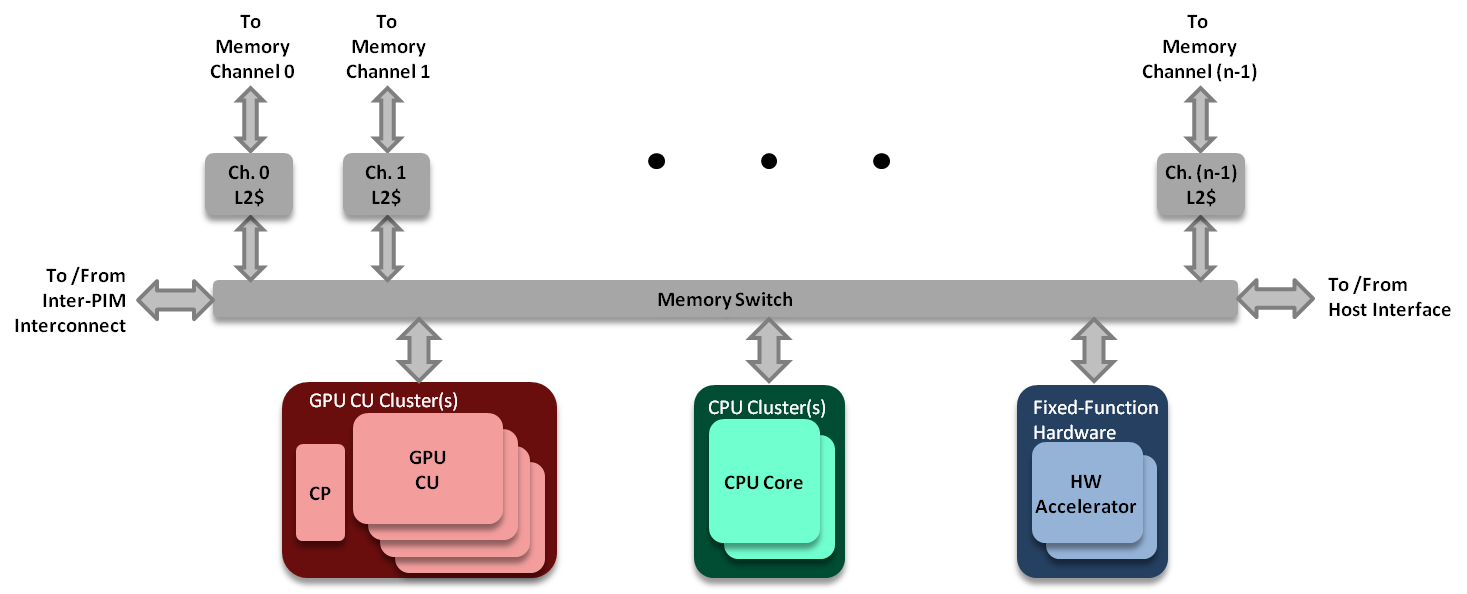


Figure 4: PIM with memory-channel-sliced, shared L2 cache (L1 caches within GPU CUs and CPU cores not shown for simplicity; optional fixed-function hardware shown)

The above cache organizations also have implications for specialized, fine-grain in-memory arithmetic operations (e.g., add-and-store). While we do not explore such operations in our baseline architecture, they may be of interest in future explorations**[\*]**. The organization shown in Figure 3 requires fine-grain in-memory operations to use uncached memory locations or implementations of the arithmetic capabilities separately at each L2 cache (and possibly at each L1 data cache). The organization shown in Figure 4, on the other hand, allows a unified implementation of such operations at the shared L2 cache (and possibly at each L1 data cache).

Table 1 compares the tradeoffs of the two L2 cache organization options discussed above, along with the option of not implementing any L2 caches for the PIM devices. Each of the L2 organizations provide different sets of desirable features. Determining a specific PIM cache hierarchy organization depends on in-depth studies of a wide set of PIM-candidate applications and is beyond the scope of this milestone.

Table 1: Tradeoffs among PIM L2 cache organization options

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Processor-attached L2 Cache(s)** | **Channel-attached L2 Cache(s)** | **No L2 Cache(s)** |
| **GPU performance** | Unaffected | Unaffected (latency tolerant) | Moderately reduced due to no inter-CU sharing via caches |
| **CPU performance** | Ideal | Moderately reduced due to longer L2 access latency | Reduced |
| **Memory switch traffic and energy** | Moderate | High | High |
| **L2 cache interference** | Minimal (separate L2s; no interference except due to coherence) | Medium (shared L2; interference reduction possible via partitioning etc.) | Not applicable |
| **CPU-GPU(-fixed-function) data sharing** | Through DRAM/NVRAM (or coherence schemes) | Through L2 cache | Through DRAM/NVRAM |
| **Hardware overhead** | High (separate L2 caches even when only CPU or GPU is in use) | Medium (shared L2 caches)[[6]](#footnote-6) | Low |
| **L2 cache coherence overheads** | High (must keep separate L2s coherent) | Low (shared L2; only L1 invalidations) | None |
| **Standalone fixed-function accelerator implications** | High area or performance overhead (separate L2 cache or no L2 cache) | Low overhead (shares existing L2 caches) | High performance overhead (no caching) |
| **Design reuse** | Easy (similar cache hierarchy to mainstream processors) | Difficult (need customizations to cache hierarchy of mainstream processors) | Difficult (need customizations to cache hierarchy of mainstream processors) |

### Cache Coherence Cost/Benefit Considerations

Cache coherence provides a unified view of memory contents to multiple computing units with separate caches. In the context of PIM in AMD’s proposed FastForward node architecture, there are two crucial cache coherence interfaces. The first is cache coherence between the host processor and each of the PIM processors on a node (*host-PIM coherence*). The second is among the multiple PIM processors within the node (*inter-PIM coherence*). The following sub-sections discuss the tradeoffs associated with these.

Host-PIM Coherence

The cost/benefit tradeoff of host-PIM cache coherence is very much application-dependent. Consider the following use-case scenarios of PIM and their coherence requirements:

* One class of prime candidates for PIM computation are applications that do not cache well due to some combination of large data set sizes and lack of locality in access patterns. For these applications, moving the computation close to memory provides higher bandwidth and lower latency on the inevitable large numbers of cache misses. Caches are of limited use for these applications and conventional solutions that provide cache coherence across the entire address space (at significant hardware and energy cost) appear to be overkill.
* Another class of candidates for PIM includes application phases with low compute to memory bandwidth ratios. However, PIM is only beneficial if the data only resides in memory and not in on-chip host caches (i.e., if the application can be blocked such that the inputs to such phases are already in host caches, the computation can be performed on the host itself). Therefore, again, conventional cache coherence may be overkill as the data accessed by PIM is unlikely to be in host processor caches.
* A third use case for PIM is to augment the overall compute capability of the node (i.e., more total FLOPS within a fixed interposer area or fixed physical volume). In such cases, the host and PIM devices may communicate at fine granularity and conventional cache coherence could be essential to enabling efficient communication between the host and PIM devices and for ease of programmability.

At a high-level, cache coherence implementations can be considered to fall in to three primary categories:

* **Software-only coherence**: These schemes rely on software techniques to flush hardware caches at appropriate synchronization points to ensure a consistent view of memory across multiple computing units operating in the same address space. A consistent view is only guaranteed at the points where caches are flushed. Such a scheme is used between discrete GPUs and host (CPU) processors today in non-HSA systems. Due to the overhead inherent in cache flushes, such schemes are only efficient when the need to share data is infrequent.
* **Hardware coherence**: These schemes provide a consistent view of memory at all times enabling the least complex and most flexible programming models. However, this desirable abstraction comes at the expense of significant hardware and energy overheads. Schemes based on cache snooping, coherence directories, coherence tokens etc. fall in to this category and encompass the vast majority of cache coherent commercial systems implemented to date.
* **Hybrid coherence schemes**: These combine hardware and software techniques to provide some degree of coherence (without having to flush caches) with reduced hardware overheads. An example of such an approach is *Acoherence* [19] that requires software to annotate points in execution where a consistent view of memory is needed and implements efficient hardware mechanisms to support that view (i.e., not a full cache flush). In general, these schemes may also provide regions of memory that implement differing levels of coherence support. The burden of identifying which data structures in an application may need what level of coherence falls on the programmer (either the application programmers or the developers of compile- or run-time tools to manage these decisions).

The key drawback of software-only schemes – the need to flush caches – is exacerbated in our design due to potentially large cache capacities of the host (for example, the EHP host of our FastForward node architecture baseline has cumulative totals of 32MB of CPU L3 caches, 24MB of GPU L2 caches, and 32MB of shared last-level caches). Flushing all of the host caches any time a PIM device needs a consistent view of memory is neither scalable with the number of PIMs nor desirable. Techniques that rely on the programmer to annotate which data is accessed by both the host and PIM, and only flushing that data from caches, may be a viable adaptation for some applications.

Purely snooping-based hardware cache coherence schemes have well-understood challenges for high-bandwidth systems (as is the case with the host and the PIMs in our proposed architecture) and for scaling beyond small numbers of processors. Therefore, we do not consider this a viable option for host-PIM coherence. The key drawback of directory-based[[7]](#footnote-7) traditional hardware cache coherence schemes is the storage overhead necessary to store the directory to track which caches contain copies of any given piece of data. Adopting such a scheme would require implementing a directory at each PIM to avoid probing host caches on each PIM memory access (frequently probing host caches would negate a significant fraction of the benefits of in-memory processing). The directory would track which data is cached in the host and only generate coherence traffic to the host for data that is cached there. However, a traditional directory, which tracks cached data at the granularity of individual cache lines, would require on the order of 2MB to 4MB per PIM to implement. Such a structure implemented in a memory stack could consume 10% or more of a logic die’s total area (for typical anticipated DRAM die stack footprints). Coherence schemes that track caching at coarser *region* granularities [20] could pay significant dividends in reducing directory storage requirements as well as in reducing coherence traffic. An implementation of region coherence that tracks sharing at 4KB granularity has been shown to reduce storage requirements by 50-75% for general-purpose workloads [21].

Region coherence has been proposed, and is being researched in detail, in the context of the proposed EHP architecture within AMD’s FastForward processor research. The PIM project intends to incorporate additional enhancements to this approach resulting from that work. Early internal evaluations of region coherence with 4KB regions have shown cache coherence traffic reductions averaging 98% across a set of general-purpose benchmarks [18]. Further, some region coherence schemes are able to gracefully degrade to cache-line-granularity tracking of sharing for parts of memory where fine-grain sharing is detected [18][21]. We anticipate that a coarse-grain tracking method, such as region coherence, meets the needs of the example use cases identified above that have minimal needs or opportunities for cache-level data sharing while still enabling the use cases that do require efficient host-PIM sharing. Therefore, we expect that the potential for area, energy, and coherence traffic reductions from region coherence may be even greater for PIM use cases than for general-purpose workloads due to the very large data sets with well-defined sharing patterns that are likely to be typical of PIM workloads.

Hybrid coherence schemes that incorporate aspects of both hardware and software coherence schemes are another option. These have the potential to provide significant flexibility to tailor coherence support to application requirements. For example, such a scheme may provide multiple segments of memory, each with its own choice among no coherence (for data used only by the host or PIM but not both), software-like-coherence (for data that is not likely to cache well, as discussed in some use cases above), or hardware coherence (for synchronization variables, fine-grain shared data etc.). Hardware support for each segment can be implemented as appropriate to achieve desired performance levels without prohibitive area and cost overheads. However, programmers are required to annotate each data structure with its desired level of coherence and incorrect annotations can lead to not just reduced performance but incorrect results.

Some features of the PIM API described in Section 4 have already been conceived to enable hybrid coherence schemes. *Map* (Section 4.2.6) and *unmap* (Section 4.2.7) API calls can serve to annotate coherence points between host and PIM devices. A specialized PIM memory allocation routine (Section 4.2.3) takes in a set of flag arguments that specify which devices in the node are expected to read and/or write the allocated data. These flags can drive optimizations on which subsets of caches and cache lines need to be flushed and which ones need to be invalidated at coherence points.

Table 2 summarizes the advantages and disadvantages of the viable flavors of each of the above coherence types. Both region coherence and hybrid coherence have desirable characteristics and are promising candidates for host-PIM coherence. A quantitative evaluation of these options requires the availability of a sufficiently broad range of applications ported to the PIM environment as well as tools geared towards detailed cache analysis of PIM. Therefore, such a study is not feasible within the scope of the current milestone, but this is an important area of future research in developing PIM solutions.**[\*]**

Table 2: Tradeoffs among viable host-PIM coherence options

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Selective Software Coherence** | **Fine-grain Hardware Coherence** | **Region Hardware Coherence** | **Hybrid Coherence** |
| **Host cache flushes** | At coherence points (any data that may be shared) | No | No | At coherence points (limited only to true host-cached, shared data) |
| **PIM cache flushes** | At coherence points (any data that may be shared) | No | No | At coherence points (limited only to true PIM-cached, shared data) |
| **Programmer overhead for annotations** | High | None | None | Selectable |
| **Overhead at coherence points** | High (due to cache flushes) | None | None | Medium (due to selective cache flushes) |
| **Hardware overhead** | Minimal | High | Medium | Low |
| **Coherence traffic** | None | High | Medium | Low |
| **Energy overhead** | Medium (due to cache flushes) | High (due to coherence traffic and coherence structures) | Moderate (due to region coherence traffic and structures) | Moderate (selective cache flushes and limited coherence) |
| **Pathological corner cases** | Degrades to full cache flushes at coherence points | High coherence traffic | Degrades to fine-grain coherence | Efficiently managed with sufficient user hints |

Inter-PIM Coherence

Methods discussed above for host-PIM cache coherence that have limited hardware overheads, such as region coherence and hybrid coherence, can be extended to encompass inter-PIM cache coherence as well. Hybrid coherence, in particular, may be extended such that software can annotate data that requires inter-PIM coherence similarly to the way data that requires host-PIM coherence may be annotated. Encompassing inter-PIM coherence adds incrementally to any hardware tracking structures for two reasons:

* Additional state is needed to track the contents of PIM caches in addition to host caches.
* Encoding of sharer sets (i.e., which entities have valid copies of the data) needs additional fields as PIMs become potential sharers.

Exploiting the fact that optimized PIM computations must operate primarily out of the local memory stack to realize the benefits of in-memory compute can allow further reductions in inter-PIM coherence overheads. While access to all of memory within the node from any PIM is supported, such accesses are meant to alleviate the need to spend optimization effort on non-performance-critical aspects of computations. As a result, we anticipate accesses from PIMs to other memory stacks to be infrequent and off the performance critical path in optimized code. Consequently, inter-PIM coherence requirements may be more amenable to relaxation (from a performance point of view) to reduce hardware and energy overheads. As such, the following techniques that further reduce hardware overheads may also be viable for inter-PIM coherence:

* **No remote caching:** PIM caches can only hold data from the local memory stack. This avoids the issue of inter-PIM coherence altogether by disallowing multiple copies of a given line among the PIM caches. The key downside is the need to traverse the inter-PIM interconnect on every non-local access, and the inability to exploit even immediate spatial locality (at a cache-line granularity). While this may potentially be desirable for sparse gather/scatter operations, it can greatly increase the number of independent requests on the inter-PIM interconnect.
* **Bounded-lifetime remote caching**: PIM devices are allowed to cache data from other memory stacks for short, bounded periods of time and the cached copies are invalidated at the end of that period. This enables some amount of short-term cache-line-granularity coalescing of data, leading to potential improvements in interconnect utilization and requirements. However, there is no need for explicit coherence messages as all cached copies in the system will self-invalidate within a bounded (short) period of time. This idea is described in [22] and is related to recent research on intra-GPU coherence [23].
* **Bounded-capacity remote caching**: Each PIM device allows a limited (small) number of cache lines worth of data from the memory stacked with it to be remotely cached by other PIM devices. Once the full allotment of remotely cached lines is reached, each new remotely cached access results in the “recall” of a previously remotely cached line. This bounds the overhead for tracking remote copies as well as the number of coherence messages needed. While this is in some ways an extreme case of coherence directory caches [24], this application is explicitly geared for very infrequent data sharing. The efficiency of this technique can be further enhanced by API extensions to enable application-level input and/or hardware predictors (either at the requestor or the home PIM device or both) to select which subsets of data needs cache coherent among PIM devices.

The tradeoffs of the above approaches for inter-PIM coherence are summarized in Table 3. While the ideal solution depends heavily on application characteristics, bounded-lifetime remote caching is especially promising for anticipated PIM usage scenarios where non-local accesses from PIM devices are expected to be rare. Similar to host-PIM coherence, quantitative evaluations of these tradeoffs require porting a large number of applications and developing specialized tools. While this is not feasible within the current milestone, this, again, promises to be an interesting area of future research.**[\*]**

Table 3: Tradeoffs among viable inter-PIM coherence options

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Encompass Inter-PIM Coherence in Host-PIM Coherence** | **No Remote Caching** | **Bounded-Lifetime Remote Caching** | **Bounded-Capacity Remote Caching** |
| **Interconnect traffic efficiency for dense access patterns** | Efficient (cache line transfers) | Inefficient (single-word transfers) | Efficient (cache line transfers) | Efficient (cache line transfers) |
| **Interconnect traffic efficiency for sparse access patterns** | Inefficient — full cache line reads and possibly writes | Optimal (interconnect can be optimized for short transfers) | Inefficient —full cache line reads (writes may only transfer valid data) | Inefficient —full cache line reads (writes may only transfer valid data) |
| **Temporal locality capture** | Optimal (true caching efficiency) | None | Limited (only during lifetime) | Limited (only a small number of lines outstanding) |
| **Hardware overhead** | Medium (extends host-PIM coherence structures) | None | Minimal | Minimal |
| **Coherence traffic** | Medium (assumes region or hybrid coherence) | None | None | Medium (coherence for a limited number of lines) |
| **Energy overhead (assuming mostly dense access patterns)** | Medium (due to coherence traffic and coherence structures) | High (due to no spatial or temporal locality capture) | Medium (due to limited temporal locality capture) | Medium (due to limited temporal locality capture) |
| **Scalability with number of PIM devices** | Max number of supported devices must be known at design time | Independent of the number of devices | Independent of the number of devices | Max number of supported devices must be known at design time |

### Address Striping

Architectures with multiple memory modules (e.g., the eight DRAM stacks in our proposed EHP node architecture) have a choice in how the physical address space is distributed among those modules. At one extreme, the addresses can be interleaved at a fine granularity, such as a single cache-line, among the modules. At the other extreme, each module can be assigned a contiguous address range equivalent to the capacity of the module to achieve an extremely coarse interleaving.

Modern throughput-oriented architectures typically employ fine grain (e.g., 1-8 cache line) interleaving among memory modules (or channels). For access patterns with spatial locality, this provides improved memory traffic distribution across all available memory modules, resulting in higher aggregate bandwidth (conversely, a coarse distribution will lead to heavy accesses to one module for some time while other modules idle, then heavy accesses to another module and so on). For this reason, data accessed by the host can often benefit from fine grain address interleaving among the DRAM stacks in our node architecture.

PIM, on the other hand, has a diametrically opposed requirement. As the realization of the benefits of PIM require the majority of accesses to be to the local memory stack, PIM often benefits from the allocation of entire large data structures within a single stack. In other words, PIM often requires an effectively very coarse-grain interleaving. In a paged virtual memory system, this requirement implies that address interleaving among memory modules must be no finer than the smallest OS page size. The interleaving need not be any coarser than an OS page size as PIM-accessed data structures can be allocated contiguously in virtual address space and a PIM-aware OS can ensure all of those virtual pages map to physical memory within a single memory stack.

We assume our baseline node architecture implements address interleaving among memory modules at an OS page granularity.

A more advanced approach that can meet both of the conflicting requirements on address interleaving is to partition the physical address space in to two (or more) segments, one of which is interleaved finely among memory modules and the other interleaved at coarse granularity. Figure 5 shows an example address distribution across four modules for two segments – one with a fine granularity and the other with a coarse granularity.

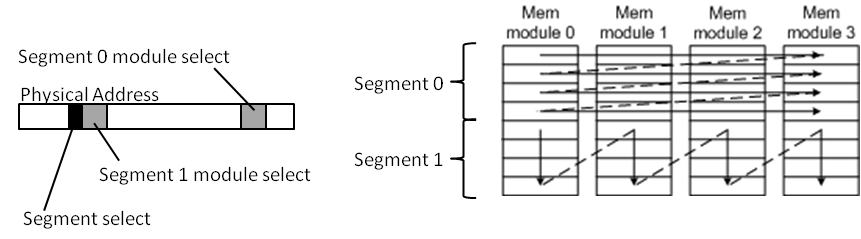


Figure 5: Example multi-granularity address distribution and mapping of physical address bits for segment and module selection

Given segments with fine and coarse (i.e., ≥ OS page) granularity address interleaving, data accessed primarily by the host may be allocated in the fine-grain distributed segment while data accessed primarily by the PIM may be allocated in the coarse-grain distributed segment. While the relative sizes of the segments may be dynamically modified, we do not currently envision the dynamic variation of interleaving granularity of already-allocated and populated data structures. If the interleaving granularity of a data structure needs to be changed, it may be physically copied from one segment to another. Note, however, that such migrations can be done entirely within the physical address space while keeping the virtual addresses unchanged, making these migrations transparent to the application (this is especially true if the data structure is page-aligned and padded to an integer number of pages to ensure no side-effects of the migration on other data).

## Hardware Parameters

Table 4 describes high-level performance parameters for two proposed configurations of PIM under DRAM stacks for AMD’s FastForward node architecture. For these estimates, we assume the PIM dies must have roughly the same footprint as the DRAM dies.[[8]](#footnote-8)

For the estimates in Table 4 we also assume the PIM devices are implemented on dedicated logic dies under memory stacks (i.e., any necessary DRAM support logic, such as test circuitry, will reside on a first logic die while PIM is on a second logic die). This is primarily to factor out the impact of memory-vendor-specific logic die features and to enable practical design partitioning between memory and processor vendors. However, technologically, it should be feasible to implement PIM and DRAM support logic on a single logic die (possibly with reduced PIM capabilities relative what is shown in Table 4 due to area constraints).

Table 4: Example PIM configurations for DRAM stacks

|  |  |  |
| --- | --- | --- |
|  | **Example 1 (Baseline)** | **Example 2 (Aggressive)** |
| DRAM die footprint (mm2) | 80 | 120 |
| Number of GPU CUs per DRAM stack[[9]](#footnote-9) | 16 | 32 |
| Peak PIM GPU CU frequency (MHz) | 750 | 750 |
| Number of CPU cores per DRAM stack | 2 | 2 |
| Issue width per CPU core | 2 | 2 |
| Peak PIM CPU frequency (MHz) | 2000 | 2000 |
| Peak compute per DRAM stack (TFLOPS) | 0.768 | 1.536 |
| Fraction of host compute in 8 PIM devices | 50% | 100% |
| Memory bandwidth per stack (TB/s)[[10]](#footnote-10) | 1 to 2 | 1 to 2 |
| Resource-balanced point (FLOPS/byte) | 0.384 to 0.768 | 0.768 to 1.536 |

The host processor in AMD’s FastForward proposed baseline node architecture (EHP) has a peak compute rate of 12 TFLOPS (double-precision) and peak aggregate DRAM bandwidth of 4TB/s, resulting in an optimal resource balance at 3 FLOPS/byte. As can be seen from Table 4, PIM devices are balanced for computations with up to an order of magnitude lower computation per byte of memory accessed. Therefore, a system consisting of EHP and PIM devices can near-optimally support a broader range of compute profiles than a system with only one or the other. Further, the aggregate compute capabilities of PIM devices in the eight memory stacks match a significant fraction of the EHP’s compute capabilities, allowing PIM-only computations to still reach high throughput rates.

Table 5 lists high-level energy estimates for memory accesses from PIM (in the *PIM* column) and host (in the *Host on Interposer* column) in our baseline node configuration. We also show two additional host access types for comparison. The *Enhanced Host on Interposer* column shows energy for a host access in a system that uses an API like the one proposed for PIM in this report to control data layout and compute placement within the host to ensure that data access to each stack originates from the quadrant of the host die closest to the memory stack, reducing traversal distance on the host die. The *SerDes Host* column shows access energy for a more traditional organization (scaled to near-exascale timeframe expectations) that uses a high-speed serial interface and board-level (electrical) traces to access memory. This last configuration is meant to provide a reference point to what is likely to be a more mainstream configuration. The estimates in Table 5 are based on internal AMD data and conservative projections based on the International Technology Roadmap for Semiconductors (ITRS) [25].

Table 5: Memory system energy estimates

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Energy Component**  **(pJ/b unless stated otherwise)** | **PIM** | **Host on Interposer** | **Enhanced Host on Interposer** | **SerDes Host** |
| DRAM array access | 3 | 3 | 3 | 3 |
| TSV with ESD (fJ/b) | 110 | 110 | 110 | 110 |
| In-stack 10mm traversal[[11]](#footnote-11) | 1.1 | 1.1 | 1.1 | 1.1 |
| Interposer 2mm traversal (fJ/b) |  | 55 | 55 |  |
| Ultra-short-range SerDes |  |  |  | 2.5 |
| On host die 30mm traversal[[12]](#footnote-12) |  | 3.3 |  | 3.3 |
| On host die 15mm traversal[[13]](#footnote-13) |  |  | 1.65 |  |
| PIM L2 (2MB) allocate energy[[14]](#footnote-14) | 0.25 |  |  |  |
| Host L2 (2MB) allocate energy[[15]](#footnote-15) |  | 0.25 | 0.25 | 0.25 |
| Host LLC (4MB) allocate energy[[16]](#footnote-16) |  | 0.4 | 0.4 | 0.4 |
| Total miss handling energy, including cache allocation (nJ per 64B block) | 1.1 | 2.1 | 1.7 | 2.7 |
| **Miss handling energy normalized to PIM** | **1x** | **~2x** | **~1.5x** | **~2.5x** |

## Specialized Operations

While our baseline PIM architecture consists of an APU stacked with memory, there are a variety of specialized capabilities that may benefit greatly from custom hardware that can provide in-memory acceleration with very low overheads. Examples of such operations may include:

* Fine-grain augmented memory operations such as op-and-store, which can effectively move read-modify-write operations to the memory. These provide the additional benefit of eliminating the need for locks on updates to a single shared variable.
* Compound memory operations that touch an arbitrary range of memory addresses but whose operation is well-defined and can be expressed in a small number of parameters. These include strided or indirect gather/scatter operations, pointer chasing, and data layout changes.
* Standardized or otherwise well-defined computation operations on a range of memory operations (e.g., compression, encryption, reductions).

Determining which specialized operations are beneficial to accelerate in hardware requires wide-ranging application studies that are beyond the scope of the current milestone. As a result, our baseline architecture currently does not incorporate such features. However, we envision the arbitrary code offload API described in Section being a useful framework for initially expressing hardware acceleration candidates in software and estimating the potential impact of additional hardware acceleration. This is especially true for coarse-grain operations that operate on large amounts of data and for which UMQ-based dispatch is a viable approach. A detailed evaluation of such specialized accelerators is specified as part of the next PIM milestone (M2) deliverables.

# User-level API

This section describes the API for accessing the PIM devices from a user-level application.

## Scope

As the PIM devices consist of APUs that are capable of executing general-purpose code, the API need not to provide additional capabilities to express the computation for execution via PIM. Therefore, the goals of the user-level API are:

* Discovering available PIM resources
* Orchestrating memory data layout
* Dispatching PIM computations
* Providing visibility of intended memory usage

The following sub-sections (4.1.1 through 4.1.4) discuss the significance of each of these goals, and API calls for achieving these goals are described in Section 4.2.

The API presented here is designed to augment existing C-based programming frameworks. Example implementations of the API for POSIX threads and OpenCL are described in Sections 4.4.1 and 4.4.2, respectively. However, the API does not rely on capabilities unique to C and the insights gained using this API as well as implementation of comparable API features should generally extend readily to other high-level programming languages.

This API is geared towards facilitating early-stage PIM evaluations by expert programmers. Therefore, the interface provides a high degree of visibility into the hardware configuration and requires manual control over data placement and compute-data co-location. Naturally, we envision the need for higher levels of abstractions for use by the broader developer community. An example of such a higher-level abstraction can be found in the PIM-related aspects of the proposed OpenMP pragma extensions discussed in the AMD *FastForward P12: High-level Language Extensions for Exascale Heterogeneous Processors Report*. Such pragmas may be mapped automatically by a compiler to the lower-level API described here. While the pragmas do not expose the full feature set of the PIM API, they are intended to support a subset of common use cases with reduced programmer effort. Other high-level abstractions may include domain-specific languages (DSL), library implementations of common operations and data structures, and load-balancers that encompass support for PIM capabilities in their underlying implementations. The run-time system components of such frameworks may incorporate static or dynamic placement of and migration of data and compute among PIM devices by calling the API described in this document. We also do not preclude the possibility of even lower level abstractions than what is described here for underpinning the development of alternate run-time systems in the longer term. However, the development of such a layered set of APIs is out of the scope of our current PIM research.

### Discovering Available PIM Resources

These calls enable applications to query the availability of PIM resources on the node as well as their static and dynamic properties. This enables the development of applications that are portable across a variety of node configurations. Such applications also enable the programmer to explore various PIM configurations without the need to individually tune applications for each configuration.

A specification for a device discovery framework is also under development as part of the HSA runtime system specification. We intend to reconcile the PIM device discovery API with that specification once it becomes available.**[\*]**

### Orchestrating Memory Data Layout

Many of the prior PIM research efforts deal with expressing in-memory computation on data already assumed to be in the co-located memory. However, realistic high-performance systems will contain multiple memory modules, possibly with some heterogeneity among them (e.g., the DRAM and NVRAM memory modules in our proposed baseline FastForwared node architecture). The utility, and perhaps even the viability, of PIM can be entirely dependent on how data are distributed among those multiple memory modules. Therefore, enabling control over layout of data among memory modules is a crucial component of our programmability-centric approach. Our API enables expressing memory layout preferences among the memories associated with the PIMs available on the node.

The API also enables migration of allocated data from memory associated with one PIM device to another while keeping the virtual addresses unchanged. This allows the migration to be transparent to the application as any pointers to or within the migrated data remains valid. Such a feature can have utility in enabling run-time systems, libraries, or sophisticated applications to manage locality over multiple phases of execution or to better tradeoff system resource utilization and locality.

### Dispatching PIM Computations

Once the data layout among memory stacks has been orchestrated, this aspect of the API deals with dispatching the compute routines to each stack. These API calls build on the mechanisms described in Section 3.2.1 for initiating operations on PIM.

### Providing Visibility of Intended Memory Usage

Exposing application-level knowledge of memory access and sharing patterns can provide key insights on the desired level of cache coherence among various processors within the node. Therefore we allow the expression of such knowledge via the API.

## API Specification

The following subsections describe the user-level calls that comprise the PIM API.

### pim\_get\_device\_ids

Query the number of PIM devices of a given type that are available within the node and retrieve their device IDs.

int pim\_get\_device\_ids( pim\_device\_type device\_type,

uint32\_t num\_entries,

pim\_device\_id \*devices,

uint32\_t \*num\_devices);

Parameters

*device\_type*

Denotes the particular class of PIMs that are to be returned. Supported device type specifiers are listed in Table 6. Multiple specifiers may be ORed together to query devices with multiple characteristics. For example, a query with a *device\_type* of (PIM\_CLASS\_GPU | PIM\_CLASS\_LEVEL\_0 | PIM\_CLASS\_DP\_RATIO\_1) returns only the PIM devices in the node with at least one GPU CU in the first level of the memory hierarchy and having a compute/bandwidth (peak) ratio ≤ 2. Any property not explicitly defined it treated as a “don’t care” (e.g., if neither PIM\_CLASS\_LEVEL\_0 nor PIM\_CLASS\_LEVEL\_GE1 are specified, PIM devices in all memory levels are returned).

Table 6: Supported PIM device type specifiers

|  |  |
| --- | --- |
| **Device Type** | **Description** |
| PIM\_CLASS\_0 | All PIM devices in the node |
| PIM\_CLASS\_CPU | PIM devices that have at least one CPU core |
| PIM\_CLASS\_GPU | PIM devices that have at least one GPU (with at least one CU per GPU) |
| PIM\_CLASS\_LEVEL\_0 | PIM devices on first level of main memory |
| PIM\_CLASS\_LEVEL\_GE1 | PIM devices on second or higher-numbered level(s) of memory in a multi-level main memory system |
| PIM\_CLASS\_SP\_RATIO\_*n*  (where 0 ≤ *n* ≤ 7) | PIM devices whose peak\_compute to peak\_bandwidth ratio is ≤ 2n, where peak\_compute is expressed in single-precision FLOPS and peak bandwidth is expressed in 32b words per second. For example, a PIM\_CLASS\_SP\_RATIO\_3 device is peak-bandwidth-limited only when there’s 8 (=23) or fewer single-precision FLOPS per 32b word accessed from memory per second. Only one PIM\_CLASS\_SP\_RATIO\_*n* specifier can be included in a *pim\_get\_device\_ids()* query (behavior is undefined if multiples are ORed together). |
| PIM\_CLASS\_DP\_RATIO\_*n*  (where 0 ≤ *n* ≤ 7) | PIM devices whose peak\_compute to peak\_bandwidth ratio is ≤ 2n, where peak\_compute is expressed in double-precision FLOPS and peak bandwidth is expressed in 64b words per second. For example, a PIM\_CLASS\_DP\_RATIO\_2 device is peak-bandwidth-limited only when there’s 4 (=22) or fewer double-precision FLOPS per 64b word accessed from memory per second. Only one PIM\_CLASS\_DP\_RATIO\_*n* specifier can be included in a *pim\_get\_device\_ids()* query (behavior is undefined if multiples are ORed together). |

*num\_entries*

The maximum number of devices that this function call should return in *devices*.

*devices*

Pointer to memory where the list of PIM devices of type *device\_type* found in the node will be stored. The memory should already be allocated prior to calling this function. Passing NULL will cause this function to not fill in any list. The number of devices returned is the minimum of *num\_entries* and the number of devices available in the system of the type *device\_type*.

*num\_devices*

After the call, contains the total number of devices (of type *device\_type*) found in the system. If *num\_devices* is NULL, this argument is ignored.

Return Value

PIM\_SUCCESS if no errors.

PIM\_INVALID\_DEVICE\_TYPE if *device\_type* is not a valid specifier.

PIM\_GENERAL\_ERROR on all other errors.

Notes

This can only be called from CPU code on the host processor.

This call’s semantics are independent of the underlying platform (e.g., POSIX threads, OpenCL) the API is layered on.

### pim\_get\_device\_info

Query detailed information about a specific PIM device.

int pim\_get\_device\_info( pim\_device\_id device,

pim\_device\_info param\_name,

size\_t param\_value\_size,

void \*param\_value,

size\_t \*param\_value\_size\_ret);

Parameters

*device*

The ID, as returned by *pim\_get\_device\_ids()*, of the device being queried.

*param\_name*

The parameter being queried. The current parameter space for PIM devices is listed in Table 7.[[17]](#footnote-17) Properties of the host may be queried by passing in PIM\_HOST\_ID as the *device*. The parameters supported for host queries are described in Table 8.

Table 7: Supported PIM device information query parameters

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| PIM\_MEM\_CAPACITY | Total amount of memory stacked with *device* (in megabytes) |
| PIM\_MEM\_FREE | Currently available (unallocated) memory on stack associated with *device* (in bytes) |
| PIM\_MEM\_BANDWIDTH | Peak bandwidth from *device* to its local memory stack (in megabytes/s) |
| PIM\_HOST\_BANDWIDTH | Peak bandwidth from host to memory stacked with *device* (in megabytes/s) |
| PIM\_INTERPIM\_BANDWIDTH | Peak bandwidth from another PIM of the same level of the memory hierarchy to the memory stacked with *device* (in megabytes/s) |
| PIM\_CPU\_CORES | Number of CPU cores in *device* |
| PIM\_GPU\_CORES | Number of distinct GPUs in *device* (note that this is not a count of the compute cores within the GPU). A typical value for this parameter is 1. |
| PIM\_GPU\_NUM\_CUS | Number of CUs in each GPU of *device* (all GPUs of *device* have this number of CUs) |
| PIM\_GPU\_FREQ | Maximum GPU frequency of *device* (in megahertz) |
| PIM\_CPU\_FREQ | Maximum CPU frequency of *device* (in megahertz) |
| PIM\_CPU\_ISA | Instruction set identifier of CPU cores on *device* |
| PIM\_GPU\_ISA | Instruction set identifier of GPU CUs on *device* |
| PIM\_MEM\_MBPS | Peak transfer rate of the interface to memory associated with *device* (in mega-transfers per second). Note that this is the bit-rate; e.g., in a double-data-rate interface what is returned here is 2x the memory clock frequency. |
| PIM\_MEM\_NUM\_CHANNELS | Number of channels in memory stack associated with *device* |
| PIM\_MEM\_LEVEL | Level of the main memory hierarchy *device* and its stacked memory belong to in a multi-level memory hierarchy (0 = first level, 1 = second level and so on) |
| PIM\_MEM\_TECHNOLOGY | Memory type identifier of memory stacked with *device* (currently supported types are elaborated in Table 9)[[18]](#footnote-18) |
| PIM\_HOST\_COHERENT | Returns non-zero if PIM device is cache coherent with host. Returns zero otherwise. |
| PIM\_PEER\_COHERENT | Returns non-zero if PIM device is cache coherent with other PIM devices in the node (at the same level of the memory hierarchy, if a multi-level memory hierarchy[[19]](#footnote-19)). Returns zero otherwise. |

Table 8: Supported host information query parameters

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| HOST\_CPU\_CORES | Number of CPU cores in host |
| HOST\_GPU\_CORES | Number of distinct GPUs in host |
| HOST\_GPU\_NUM\_CUS | Number of CUs in each GPU of host (all GPUs of host have this number of CUs) |
| HOST\_GPU\_FREQ | Maximum GPU frequency of host |
| HOST\_CPU\_FREQ | Maximum CPU frequency of host |
| HOST\_CPU\_ISA | Instruction set identifier of CPU cores on host |
| HOST\_GPU\_ISA | Instruction set identifier of GPU CUs on host |

Table 9: Supported memory types for PIM\_MEM\_TECHNOLOLOGY

|  |  |
| --- | --- |
| **Memory Type** | **Description** |
| PIM\_MEM\_TECH\_DRAM | DRAM memory |
| PIM\_MEM\_TECH\_NVM | One of the NVM technologies (e.g., PCM, STT-RAM)[[20]](#footnote-20) |
| PIM\_MEM\_TECH\_OTHER | Unknown or undefined memory type |

*param\_value\_size*

Size, in bytes, of the memory pointed to by *param\_value*.

*param\_value*

Pointer to memory location where the value of the parameter being queried is to be stored. Passing NULL will cause this function to not store a value.

*param\_value\_size\_ret*

Is updated by the call to reflect the total number of bytes returned in *param\_value* and is upper-bounded by *param\_value\_size*. If *param\_value\_size\_ret* is NULL, the argument is ignored.

Return Value

PIM\_SUCCESS if no errors.

PIM\_INVALID\_DEVICE\_ID if *device* does not specify a known device.

PIM\_INVALID\_PARAMETER if the parameter being queried is not valid for the type of the queried device.

PIM\_GENERAL\_ERROR on all other errors.

Notes

This can only be called from CPU code on the host processor.

This call’s semantics are independent of the underlying platform (e.g., POSIX threads, OpenCL) the API is layered on.

### pim\_malloc

Allocates memory on a memory stack associated with a specified PIM device.

void\* pim\_malloc( size\_t size,

pim\_device\_id device,

pim\_acc\_flags flags,

pim\_platform\_type platform);

Parameters

*size*

Size of memory to be allocated, in bytes.

*device*

ID of the PIM device whose associated memory stack the physical memory is to be allocated on.

*flags*

Flags describing access characteristics of the allocated memory. The supported values are described in Table 10. The read/write flags may be used as hints to optimize performance for the specified access types when possible. The alignment flags guarantee alignment at the specified granularity (if multiple alignment flags are specified, the coarsest granularity takes precedence). If no flags are specified, a default of (PIM\_MEM\_PIM\_RW | PIM\_MEM\_HOST\_RW | PIM\_MEM\_PEER\_RW) is assumed.

Table 10: *pim\_malloc()* flags

|  |  |
| --- | --- |
| **Flag** | **Description** |
| PIM\_MEM\_PIM\_READ | Allocated memory will be read by the PIM device specified by *device* |
| PIM\_MEM\_PIM\_WRITE | Allocated memory will be written by the PIM device specified by *device* |
| PIM\_MEM\_PIM\_RW | Allocated memory will be read and written by the PIM device specified by *device* |
| PIM\_MEM\_HOST\_READ | Allocated memory will be read by host |
| PIM\_MEM\_HOST\_WRITE | Allocated memory will be written by host |
| PIM\_MEM\_HOST\_RW | Allocated memory will be read and written by host |
| PIM\_MEM\_PEER\_READ | Allocated memory will be read by PIM devices other than *device* |
| PIM\_MEM\_PEER\_WRITE | Allocated memory will be written by PIM devices other than *device* |
| PIM\_MEM\_PEER\_RW | Allocated memory will be read and written by PIM devices other than *device* |
| PIM\_MEM\_ALIGN\_WORD | Align allocation to a 64-bit word boundary |
| PIM\_MEM\_ALIGN\_CACHELINE | Align allocation to a cache line boundary |
| PIM\_MEM\_ALIGN\_PAGE | Align allocation to an OS page boundary |

*platform*

Platform and device type expected to primarily access this memory. This argument determines the type of handle returned to the allocated memory. Supported platform types in the current revision of the API are described in Table 11.

Table 11: Supported PIM platform types

|  |  |
| --- | --- |
| **Platform** | **Description** |
| PIM\_PLATFORM\_PTHREAD\_CPU | POSIX thread running on the CPU of a PIM device. |
| PIM\_PLATFORM\_OPENCL\_GPU | OpenCL kernel running on the GPU of a PIM device. |

Return Value

A handle (cast as *void\**) to the allocated memory or buffer structure based on the specified *platform*. See Section 4.4 for more details. NULL on errors and sets *errno*. Errors are the same as seen with LIBC *malloc()* for POSIX thread platform type(s) or with *clCreateBuffer()* for OpenCL platform type(s) along with the following additional error types:

* PIM\_INVALID\_DEVICE\_ID if *device* does not specify a known device.
* PIM\_INVALID\_PLATFORM\_TYPE if *platform* does not specify a known platform type (note that *pim\_malloc()* does not require the platform type specified by *platform* to be supported by the device specified by *device*).
* PIM\_INSUFFICIENT\_SPACE if requested capacity cannot be allocated on the memory stack associated with *device*.
* PIM\_GENERAL\_ERROR on all other errors.

Notes

This can only be called from CPU code on the host processor.

This call has platform-specific semantics. Please refer to Section 4.4 for more details.

In the current revision of the API, *device* must identify a specific PIM device. In future revisions, we may support special device IDs that specify a data placement policy rather than a specific device.**[\*]** For example, a device ID of PIM\_DEVICE\_FIRST\_TOUCH may allocate the memory (possibly at a page granularity) on the memory stack associated with the first PIM device to access the allocated data.

### pim\_reloc

Relocates memory previously allocated with *pim\_malloc()* on one memory stack to the memory stack associated with another PIM device. Only objects that were allocated with the PIM\_MEM\_ALIGN\_PAGE flag can be relocated. Virtual address of the memory remains unchanged after the relocation (i.e., pointers to the data need not be updated).

int pim\_reloc( void \*handle,

pim\_device\_id device,

pim\_acc\_flags flags);

Parameters

*handle*

The handle to the object to be relocated (as returned by *pim\_malloc()*) cast as *void\**. The object must have been allocated with the PIM\_MEM\_ALIGN\_PAGE flag.

*device*

ID of the PIM device whose associated memory stack the object is to be relocated to.

*flags*

Flags describing access characteristics of the relocated memory object. The supported values are described in Table 10. Only the read/write flags may be specified for *pim\_reloc()*. Any alignment flags are ignored. Specified flags may be used as hints to optimize performance for the specified access types when possible. If no flags are specified, a default of (PIM\_MEM\_PIM\_RW | PIM\_MEM\_HOST\_RW | PIM\_MEM\_PEER\_RW) is assumed.

Return Value

PIM\_SUCCESS if no errors.

PIM\_INVALID\_DEVICE\_ID if *device* does not specify a known device.

PIM\_INSUFFICIENT\_SPACE if requested capacity cannot be allocated on the memory stack associated with *device*.

PIM\_INVALID\_HANDLE if *handle* is not an object allocated by *pim\_malloc()* or was not allocated with the PIM\_MEM\_ALIGN\_PAGE flag.

PIM\_GENERAL\_ERROR on all other errors.

Notes

This can only be called from CPU code on the host processor.

This is a synchronous call – control returns to the calling thread only after the operation has completed. This call may take an arbitrary amount of time to complete as it typically involves a physical copying of the data from one memory to another. It may also have performance side effects on the host as well as PIM devices (e.g., may require cache flushes).

This call’s semantics are independent of the underlying platform (e.g., POSIX threads, OpenCL) the API is layered on.

### pim\_free

De-allocates memory allocated with *pim\_malloc()*. All memory allocated with *pim\_malloc()* must be de-allocated with *pim\_free()*.

void pim\_free(void \*handle);

Parameters

*handle*

The handle to the object to be deallocated (as returned by *pim\_malloc()*) cast as *void\**. If *handle* does not point to a block of memory allocated by *pim\_malloc()*, the behavior is undefined.

Return Value

None.

Notes

This can only be called from CPU code on the host processor.

This call’s semantics are independent of the underlying platform (e.g., POSIX threads, OpenCL) the API is layered on.

### pim\_map

Translates a handle returned by *pim\_malloc()* to a native pointer that can be dereferenced by standard C code on the host processor. This call may also be used as a hint for cache coherence optimizations and, in some implementations, a *pim\_map()* call may be required before a memory allocated via *pim\_malloc()* can be accessed by any part of the host processor (even for platforms that directly return a native pointer from memory allocation, such as POSIX threads).

void\* pim\_map( void \*handle,

pim\_platform\_type platform);

Parameters

*handle*

Handle (returned by *pim\_malloc()*) to be mapped for host processor access. All outstanding operations that access data pointed to by *handle* must have completed before *pim\_map()* is called.

*platform*

Platform type that *handle* was originally allocated for. Supported platform types are described in Table 11.

Return Value

A native C pointer to the data pointed to by *handle*.

NULL and sets *errno* as follows if errors are encountered:

* PIM\_INVALID\_PLATFORM\_TYPE if *platform* does not specify a known platform type.
* PIM\_INVALID\_HANDLE if *handle* is not a known allocation for the platform specified by *platform*.
* PIM\_GENERAL\_ERROR on all other errors.

Notes

This can only be called from CPU code on the host processor.

This is a synchronous call – control returns to the calling thread only after the operation has completed. Depending on the level of PIM cache coherence support in the underlying hardware, this call may take an arbitrary amount of time to complete and may have performance side effects on the host as well as PIM devices (e.g., may require cache flushes). Please refer to Section 3.2.9 for a discussion on PIM cache coherence.

This call is has platform-specific semantics. Please refer to Section 4.4 for more details.

### pim\_unmap

Signal end of main processor access to a region of memory whose handler that was previously translated by *pim\_map()*. This call may also be used as a hint for cache coherence optimizations and, in some implementations, a *pim\_unmap()* call may be required before memory who handler was previously translated using *pim\_map()* can be subsequently accessed on a PIM device.

int pim\_unmap( void \*ptr);

Parameters

*ptr*

Native C pointer returned by the corresponding *pim\_map()*. All outstanding operations that access the data pointed to by *ptr* must have completed before *pim\_unmap()* is called.

Return Value

PIM\_SUCCESS on success.

PIM\_INVALID\_HANDLE if *ptr* is not a pointer previously returned by *pim\_map()*.

PIM\_GENERAL\_ERROR on all other errors.

Notes

This can only be called from CPU code on the host processor.

This is a synchronous call – control returns to the calling thread only after the operation has completed. Depending on the level of PIM cache coherence support in the underlying hardware, this call may take an arbitrary amount of time to complete and may have performance side effects on the host as well as PIM devices (e.g., may require cache flushes). Please refer to Section 3.2.9 for a discussion on PIM cache coherence.

This call’s semantics are independent of the underlying platform (e.g., POSIX threads, OpenCL) the API is layered on.

### pim\_spawn

Dispatch the execution of an arbitrary piece of (user) code to a specified PIM device.

void\* pim\_spawn( pim\_f to\_launch,

void \*\*args,

size\_t \*arg\_sizes,

size\_t nargs,

pim\_device\_id device,

pim\_platform\_type platform\_type);

Parameters

*to\_launch*

Identifies routine to start executing on a PIM device. The semantics of this argument depends on *platform\_type* (e.g., a function pointer in POSIX threads; a pointer to a string that is the kernel name in OpenCL). Please refer to Section 4.4 for more information.

*args*

An array of pointers to arguments relevant to spawning the new task, including arguments to the routine identified by *to\_launch*. The exact contents and format of this list of arguments depends on *platform\_type*. Please refer to Section 4.4 for more information.

*arg\_sizes*

An array describing the length of each of the arguments pointed to in *args*.

*nargs*

The number of arguments pointed to by *args* and *arg\_sizes*. This defines the number of elements in the first dimension of *args* (which is also the number of elements in *arg\_sizes*). Please refer to Section 4.4 for more information.

*device*

ID of PIM device that this computation will run on.

*platform\_type*

Platform and device type of the task being spawned. Supported types are described in Table 11.

Return Value

A pointer to a *platform\_type*-specific structure on success (cast as *void\**).

NULL on any type of failure, with *errno* set as follows depending on the error:

* PIM\_INVALID\_DEVICE\_ID if *device* does not point to a legal PIM device.
* PIM\_INVALID\_PLATFORM\_TYPE if *platform\_type* does not specify a supported platform type.
* PIM\_SPAWN\_ERROR\_DEVICE\_TYPE\_MISMATCH if the PIM device specified by *device* does not support the platform specified by *platform\_type*.
* PIM\_INVALID\_PARAMETER if *args* does not contain or does not match required parameters for the platform identified by *platform\_type*.
* PIM\_GENERAL\_ERROR on all other errors.

Notes

This can only be called from CPU code on the host processor.

This call is has platform-specific semantics. Please refer to Section 4.4 for more details.

### pim\_get\_id

Get the device ID of the PIM device the calling code is running on.

int pim\_get\_id();

Parameters

None.

Return Value

The ID of the PIM device the calling code is running on.

PIM\_HOST\_ID if called from the host processor.

Notes

This can be called from CPU code on the host processor or a PIM.

This call’s semantics are independent of the underlying platform (e.g., POSIX threads, OpenCL) the API is layered on.

## Execution Model and Synchronization

As discussed in Section 4.1, the application-level PIM API is intended to layer on top of existing C-based programming frameworks. As such, the API does not impose a specific execution model but rather inherits the parallel execution models inherent in the underlying frameworks.

In the case of POSIX threads, the API conforms to the task-parallel execution model. As with a normal POSIX thread creation, a *pim\_spawn()* creates a new thread running on the specified PIM device and returns a unique thread ID. The thread ID may be used to implement a barrier with that thread at a later point. Further, the presence of a single, shared virtual address space among the host and the PIM processors allows the use of memory-based synchronization primitives (e.g., locks) for the various threads to coordinate.

In the case of OpenCL, the API adopts the grid-based kernel dispatch model. The *pim\_spawn()* call implements a dispatch model analogous to an asynchronous dispatch in OpenCL. However, *pim\_spawn()* eliminates the need for the user to set up contexts and command queues (as is needed in native OpenCL) for kernels dispatched to PIM devices. Further, as the host and PIM devices share a single physical memory space visible through a unified virtual address space, it is anticipated that explicit data copies common in today’s GPU-based OpenCL computations may be avoided through careful data placement. The PIM API passes OpenCL event objects back and forth to enable synchronization with and among kernels. As is the case with a normal OpenCL kernel dispatch, a *pim\_spawn()* can take a list of OpenCL events that must be completed before the kernel is executed. These form a set of preconditions that must be satisfied prior to the execution of that kernel. Similarly, each kernel dispatched via *pim\_spawn()* may have a completion event associated with it (again, as is the case with a normal OpenCL kernel dispatch) that can be used to wait for kernel completion or to be used as a precondition in subsequent dispatches.

The API also adopts the dispatch semantics of the underlying framework. For example, for OpenCL, *pim\_spawn()* corresponds to an asynchronous queued dispatch (i.e., the dispatch is placed as a task on a queue and control returns immediately to the caller).

The API currently does not implement any additional synchronization primitives aside from those inherited from the supported native frameworks. As a broader set of applications and other classes of architectures beyond the baseline are evaluated during the second year of the project, we may revisit the utility of such primitives and associated extensions.**[\*]**

## Platform-Specific Semantics

Semantics of the PIM API can vary based on the underlying platform being tunneled through the API. The following sections describe the mapping of the API to currently supported platforms.

It is legal for the API to tunnel multiple platforms concurrently. For example, a single program may interleave *pim\_spawn()* calls to launch POSIX threads and OpenCL kernels to the same or multiple PIM devices.

### POSIX Threads

When layering the PIM API over POSIX threads, the PIM API is only used to spawn threads from the host on to PIM CPUs. Spawning new threads within the host must use the standard POSIX threads API. Similarly, a thread running on a PIM CPU may use the standard POSIX thread API to launch additional threads on the same PIM.

Spawning threads from a PIM to another PIM or back to the host is currently unsupported.

The following user-level PIM API calls have platform-dependent semantics. Their usage for POSIX threads (i.e., when the platform specified for the call is PIM\_PLATFORM\_PTHREAD\_CPU)is described below. Please refer to appropriate documentation (e.g., Linux POSIX threads manual page [26]) for information on POSIX threads.

pim\_malloc

The return value is a native C pointer. Returned pointers may be directly dereferenced by PIM CPUs to access the allocated memory. However, calling *pim\_map()* is highly recommended before dereferencing the pointers on the host as some implementations of the API may rely on *pim\_map()* and *pim\_unmap()* calls to optimize cache coherence.

pim\_map

The input argument *handle* must be a native C pointer obtained by a *pim\_malloc()* call with a *platform* value of PIM\_PLATFORM\_PTHREAD\_CPU. The returned value is also a native C pointer.

pim\_spawn

The input arguments *to\_launch* and *args* must specify the inputs necessary to launch the new POSIX thread on the PIM identified by the argument *device*:

* *to\_launch* must specify the function to start executing in the newly launched thread. This is defined by assigning a function pointer to *to\_launch.func\_ptr*. This is equivalent to the *start\_routine* argument of standard *pthread\_create()*.
* a*rgs* must contain the following entries in this order:
  + A pointer to a *pthread\_t* thread structure. This is equivalent to the *thread* argument of standard *pthread\_create()*.
  + A pointer to a *pthread\_attr\_t* thread attributes structure (or NULL for default attributes). This is equivalent to the *attr* argument of standard *pthread\_create()*.
  + A pointer to a structure that is passed as the sole argument to the function pointed to by *to\_launch.func\_ptr*. This is equivalent to the *arg* argument of standard *pthread\_create()*.
* *arg\_sizes* must specify the sizes of each of the entries in *args.*
* *nargs* must be 3.

The return value is a pointer to a status output identical to that of the standard POSIX thread call *pthread\_create()*.

### OpenCL

When layering the PIM API over OpenCL, the PIM API is only used to dispatch kernels from the host onto PIM GPUs. Dispatching kernels from a host’s CPU core to the host’s GPU must use the standard OpenCL API. Similarly, a thread running on a PIM CPU may use the standard OpenCL API to dispatch kernels to the GPU on the same PIM.

Dispatching kernels from a PIM to another PIM or back to the host is currently unsupported.

Note that the PIM API abstracts the creation of device contexts and command queues as well as explicit kernel compilation when dispatching to PIM devices. This eliminates the need for the user application to perform these tasks for operations issued through the PIM API. However, operations that use the standard OpenCL API (e.g., intra-host or intra-PIM kernel dispatch) still require these infrastructure setup steps.

The following user-level PIM API calls have platform-dependent semantics. Their usage for OpenCL (i.e., when the platform specified for the call is PIM\_PLATFORM\_OPENCL\_GPU)is described below. Please refer to appropriate documentation (e.g., OpenCL 1.2 Reference Pages [27]) for more information on OpenCL.

pim\_malloc

The return value is a pointer to an OpenCL buffer handle (*cl\_mem*) cast as *void\**. This handle may be cast to *cl\_mem* and directly passed to OpenCL commands. Non-OpenCL access to the data region of the buffer requires calling *pim\_map()*, which returns a native C pointer. Some implementations of the API may rely on *pim\_map()* and *pim\_unmap()* calls to optimize cache coherence.

pim\_map

The input argument *handle* must be a pointer to an OpenCL buffer (*cl\_mem*) obtained by a *pim\_malloc()* call with a *platform* value of PIM\_PLATFORM\_OPENCL\_GPU. The returned value is a native C pointer to the data region of the OpenCL buffer.

pim\_spawn

Queuing behavior of kernel dispatch via the PIM API is that of asynchronous dispatch via an in-order queue per PIM GPU device. The *pim\_spawn()* call returns immediately after enqueuing the kernel dispatch.

The input arguments *to\_launch* and *args* must specify the inputs necessary to launch the kernel on the PIM identified by the argument *device*:

* *to\_launch* must identify the OpenCL kernel to execute. This is defined by setting *to\_launch\_func\_name* to point to a null-terminated character array containing the name of the kernel.
* a*rgs* must contain the following entries in this order:
  + A pointer to a null-terminated character array holding the name of the kernel source file. All kernels in the source file are built and cached during the first call to any kernel in that file.
  + A pointer to a null-terminated character array specifying OpenCL build flags. This is equivalent to the *options* argument of the OpenCL *clBuildprpgram()* call. Please refer to OpenCL documentation for a list of supported flags.
  + A pointer to an integer specifying the number of dimensions used to specify global work item domain and work group sizes. This is equivalent to the *work\_dim* argument of the OpenCL *clEnqueueNDRangeKernel()* call.
  + A pointer to an array of type *size\_t* entries specifying the global work item domain size. The number of entries in the array must match the number of dimensions specified above. This is equivalent to the *global\_work\_size* argument of the OpenCL *clEnqueueNDRangeKernel()* call.
  + A pointer to an array of type *size\_t* entries specifying the work group size in terms of work items. The number of entries in the array must match the number of dimensions specified above. This is equivalent to the *local\_work\_size* argument of the OpenCL *clEnqueueNDRangeKernel()* call.
  + A pointer to an integer specifying the number of OpenCL events to wait for before launching the kernel. This is equivalent to the *num\_events\_in\_wait\_list* argument of the OpenCL *clEnqueueNDRangeKernel()* call.
  + A pointer to an array of *cl\_event* entries that must complete before this kernel can execute. The number of entries in the array must match the number specified by the previous argument. This is equivalent to the *event\_wait\_list* argument of the OpenCL *clEnqueueNDRangeKernel()* call.
  + A *cl\_event* that, after the *pim\_spawn()* call returns, identifies this kernel execution. This is equivalent to the *event* argument of the OpenCL *clEnqueueNDRangeKernel()* call.
  + Kernel arguments. The number of these arguments must match the number of arguments in the kernel definition.
* *arg\_sizes* must specify the sizes of each of the entries in *args.*
* *nargs* must be the total number of arguments in *args* (i.e., eight required arguments plus the number of kernel arguments).

# System Integration

The following sub-sections discuss key system software and system integration implications of PIM.

## System Software Considerations

With the PIM being a fully programmable APU that can share page tables with the host, a range of OS scheduling models may be possible.  We briefly explore three existing OS models below, explore ways to extend them to support PIM, examine their suitability for our baseline architecture, and consider paths for extension or future research.  The first is a conventional OS for symmetric multiprocessing (SMP).  The second is an OS with support for heterogeneous CPU core characteristics.  The third is a conventional OS that uses a device driver model to interface with accelerators.  Each of these models is discussed below in more detail. Given the host and PIM baselines, we consider the device driver model to be the most approachable in the near term, while other models require further research to fully account for asymmetries of host and PIM. However, in the long term, evolutions of other models or combinations of them may be more desirable.

Independent of the OS/runtime model, placement of tasks and memory are of special concern.  Care should be taken to locate pages and page-tables on the memory stack of the PIM device that uses them.  This ensures the benefit of high local memory bandwidth.  The *pim\_malloc()*API  is intended to help facilitate placement of data near the PIM that uses it, while the *pim\_spawn()*API is intended to help facilitate job affinity on a given PIM.

Symmetric Multiprocessing (SMP)

An SMP system has two or more identical processors that are connected to a shared memory, and are scheduled by a single OS image, such as Linux.  While the processors are identical, memory access time may be non-uniform (NUMA).

Under an SMP model, the proposed PIM User APIs may be implemented using (or as extensions to) POSIX-thread and NUMA interfaces in Linux.  However, the SMP model is not an exact fit for scheduling PIM cores with a host, mainly because the processors may not be symmetric.  This leads to a number of task scheduling and resource allocation issues that are not concerns in a conventional SMP system.

One issue is that the host may have more compute resources than a PIM device, but with uniform access to all memory stacks.  For the PIM cores, the situation is reversed; a PIM device would have fewer compute resources and NUMA characteristics, as local memory bandwidth would be higher than bandwidth to a remote memory stack.  It follows that different types of tasks may have different affinity preferences.  Compute-bound tasks might be better suited to the host, while memory bandwidth-bound tasks might be better suited to PIM.  Another issue that arises in an asymmetric system is the potential for POSIX-like priority inversion, where lower-priority software executing on the host might out-contend higher-priority software running on a PIM.  There is also the possibility that the ISA of the PIM device may not match the ISA of the host; SIMD extensions are one possible area for ISA difference.  In this case, tasks may need to be migrated from one core type to another, regardless of priority, in order to continue efficient execution.  Extending a conventional SMP OS to schedule tasks for asymmetric processors may provide one path for future research. Enabling such true multi-processing OS capabilities among the host and PIM devices may be the most desirable solution in the long term.**[\*]**

Heterogeneous Cores

Another OS scheduling model available in Linux combines one big core with a smaller core, and then presents a system with a single logical core that has a very broad range of operating points.  The benefits of this approach are two-fold.  When lower compute latency is required, tasks are scheduled on the bigger core; when lower power is required, the big core goes to sleep while tasks are scheduled on the smaller core.

Under the heterogeneous core model, the OS dynamically switches between scheduling on core types based on extant conditions.  This may make implementation of proposed PIM user APIs more difficult, because PIM cores may come online only during periods of "low frequency" computation.  Moreover, the heterogeneous core model is not an exact match for PIM cores in our FastForward node architecture as the ratio of big (host) processors to little (PIM) processors is not 1:1, but rather 1:many.  Extending heterogeneous cores to allow gradations from one big core to groups of smaller cores may provide one path for future research that may make this approach more amenable to supporting PIM devices.**[\*]**

Device Driver

Our last example is a device driver model.  The host may run a full-function OS such as Linux, and interact with accelerators via device driver calls such as *ioctl()*.  As with HSA, the host and accelerator may share an address space, but tasks on the accelerator are scheduled independent of the host, possibly under control of micro-code or a light weight kernel (LWK) running on the accelerator.  One example of a LWK is Kitten from Sandia National Laboratories [28], which offers low-overhead dispatch and a familiar POSIX-like environment for targeted programs.

The device driver model is perhaps the most convenient way to expose PIM functions to a host OS in the context of today’s system software, and the proposed PIM user APIs might map to *ioctl()* or UMQ requests.  However, current implementations of the device driver model were conceived for devices attached via IO interfaces such as PCIe.  Extending this model to non-IO devices and enabling the delegation of more core OS responsibilities to LWK instances running on accelerators might provide a path for future studies.**[\*]**

Software running on a PIM accelerator might encounter exceptions or faults, including address translation faults.  Translation faults can be broadly categorized into *major faults*, requiring data for a page to be (re)acquired from backing storage such as a disk or the network, and *minor faults* such as page table miss and/or copy-on-write reprotect (COW).  It is conceivable that the LWK might directly update page tables, particularly for minor faults.  However the simplest policy might be for a LWK to forward page table update requests to the host, particularly in an environment where the host is run with hardware-level virtualization.  Classifying the frequency and types of page table faults in proxy applications, and ensuring that a LWK can efficiently hand off fault processing to the host might provide one path for future studies.**[\*]**

## RAS, Checkpoint Restart and Debug

A study of reliability, availability and serviceability (RAS) for the EHP node is under independent investigation for FastForward.  In this section, we briefly consider RAS topics specific to PIM, including low-level memory error detection, predictive failure analysis, application checkpointing, and preemptive context save/restore.

PIM includes both processor and memory components.  Many of the low-level RAS features for detecting/correcting errors for both components are considered a necessary part of the baseline architecture.  As these activities are performed automatically by hardware, as part of a periodic memory scrubbing for instance, these are typically outside the scope of application and even OS programming interfaces.

Predictive failure analysis attempts to identify pre-fault indicators and anticipate failures that are about to occur.   Being a fully-programmable APU, the PIM device would be equipped with its own set of counters, sensors, and other measurement mechanisms found to modern processors.  Standard application interfaces such as the Linux Performance API (PAPI) provide access to a wide variety of hardware performance counters.  Similarly, APIs for detecting thermal and power events are under independent study for FastForward. Further, Section 5.4 discusses PIM-specific instrumentation requirements. Characterizing errors in proxy applications running on a PIM stack, analyzing counter and thermal state at time of failure, and developing a predictive model could be a path for future studies.**[\*]**

Reactive fault tolerance includes techniques such as application checkpointing and task/memory migration in response to actual and predictive system health threats.  To create an application checkpoint, the application must receive all network messages and stop all parallel activity at a barrier, then proceed to write just the data needed to restart to backing storage.  The application is responsible for stopping activity, including PIM activity, and writing the data, including data in regions returned by *pim\_malloc()*.  Being fully programmable APUs, the PIM devices will participate in application-driven checkpoint and restore/restart as would any other processor in the system. Additionally, PIM devices could perform or aid in additional, enhanced checkpointing activities, including automatic checkpointing, task/memory migration, or compression.  Characterizing memory errors in proxy applications and exploring the feasibility of enhanced checkpointing using PIM could be a path for future studies.**[\*]**

The ability to perform preemptive context save/restore enables not only pre-emptive OS schedulers but also debuggers.  The HSA feature roadmap includes capability for GPU compute context switching. A fully programmable APU-in-memory PIM device would be capable of preemptive save/restore under direction of the host OS.  Save/restore of architected state is typically driver- or OS-dependent, but GPU state would include register, local data store and command FIFO state necessary for resuming execution of all threads in a wave-front at some point in the future.  Extending the baseline PIM user APIs to query/set context state for execution launched under *pim\_spawn()*, along lines of Linux *getcontext()*or *setcontext()*, could be a path for future extension.**[\*]**

## System Management

A number of system management issues need to be addressed for systems containing PIMs (or any accelerator).**[\*]** These include:

BIOS and system initiation activities:

* Power-up activities and sequencing
* Enumerate PIMs and their characteristics
* Memory test/failures
* PIM APU hardware test/failures
* Light weight PIM kernel load and boot

Other:

* OS/kernel software updates
* Security
* Monitoring of runtime activity
* Handling of hardware failures
* Access to hardware performance counters
* Power / thermal monitoring, and control over power saving modes

Further, PIM virtualization is not addressed in this report, but commercial systems exploiting PIMs may wish to implement virtualization features for the PIMs. Support will be needed in the hypervisor for such cases.**[\*]**

## Instrumentation

Since the PIM APU shares the same basic architecture as the host APU in our standard model, it will have substantially the same hardware instrumentation requirements and features. It is anticipated that the PIM instrumentation data can be obtained and analyzed with the same tools as the instrumentation data for the host APU. Candidates for possible additional hardware instrumentation on the PIM include:

* Cache misses to data not located on the local memory stack.
* Cache misses serviced from data located in the caches of the host and other PIMs
* Data for any special operations implemented as discussed in Section 3.4.

Standard instrumentation for any IOMMU located on a PIM to enable memory translation functions is expected to be adequate. The necessary software support for instrumentation access may need to be added to a light weight kernel if it is running on a PIM. System monitoring/management software will need to include PIM functions. It is not anticipated that any new hardware instrumentation functions will be needed on the host APU, except those for any special operations as discussed in Section 3.4.

# A Potential Path for PIM Adoption

The commercial viability of PIM solutions and the nature of such solutions is highly dependent on the corresponding evolution of host (CPU) processor architectures and interfaces to accommodate in-memory computation. Here we explore the relationship between various stages of such an evolution of host processors and the corresponding PIM capabilities enabled at each stage.

Near Term

The PIM architecture discussed in this document assumes a fully shared virtual and physical address space, where the host and PIM may access the same memory spaces. Such a solution naturally requires, at a minimum, a re-architecting of the host processor’s memory interface. However, more limited PIM capabilities may be possible in the near-term (e.g., 1-3 years) using existing interfaces.

One potential approach is to attach PIM devices to a host via existing NUMA-capable multi-socket interfaces (e.g., coherent HyperTransport). Such an organization would still allow communication with PIM using memory-mapped queues placed within the physical memory address region mapped to the PIM device. However, such an organization requires traditional memory attached to the host with no PIM capabilities for performance reasons, resulting in the presence of some memory regions where PIM capabilities are not present. Further, host access to PIM-attached memory is lower-performing than the non-PIM-capable memory. This introduces an additional degree of heterogeneity in the machine impeding programmability. However, a key advantage of this approach is that, from a hardware point of view, this is feasible with no significant modifications to today’s processors (as long as they incorporate coherent multi-socket interfaces).

Another alternative is to attach PIM-capable memory modules via existing DDR memory interfaces with fixed timing, but restrict the set of memory channels or banks that may be accessed by the host at any given point in time. This allows PIM memory accesses to take place to a subset of memory without interfering with fixed-timing expectations of host accesses. However, such ad-hoc methods lack the generality of a true architected PIM solution and introduce significantly greater programming model complexities.

Medium Term

The minimum host processor requirement for enabling non-trivial PIM capabilities within an architected PIM solution, such as the one described in this document, is the adoption of a split-transaction-level memory interface (i.e., an interface that issues load/store requests to memory and expects data returns or completions back after a variable delay instead of DDR-like fixed-timing interfaces). This allows PIM accesses to memory to be interleaved with host accesses and scheduled for optimal memory performance.

This also implies moving the detailed memory timing controller to the memory stack. Such an organization has a variety of desirable features independent of PIM and productization of memory modules for such systems seem imminent [3]. As a result, we believe this level of evolution in (at least some) host processors is likely in the foreseeable future. We also assume the presence of HSA-level support for accelerators in host processors which is already being adopted by multiple processor vendors (e.g., AMD, ARM) through the HSA foundation. As a result, this level of PIM capabilities may be viable in 2-5 years.

Key drawbacks at this level of host evolution include the lack of cache coherence support of any sort, requiring the flushing of host and PIM caches at synchronization and communication points. Further, communication back to the host requires polling or other such sub-optimal methods. Further discussions on these limitations can be found in Sections 3.2.9 and 3.2.1.

Long Term

We consider the above minimum requirements the first phase of host evolution to enabling architected PIM solutions. Further host evolution with an explicit expectation of computing within the memory system is necessary to enable the full set of PIM capabilities envisioned in this document. Naturally, this also implies the definition and standardization of new memory interface standards (or extension of existing or emerging ones) to accommodate PIM. Table 12 lists a few of the desirable capabilities for PIM and the host and memory interface evolution needed to enable them.

Table 12: Desirable PIM capabilities and host and memory interface evolution necessary to enable them

|  |  |  |
| --- | --- | --- |
| **PIM Capabilities** | **Host Requirements** | **Memory Interface Requirements** |
| Cache coherence between host and PIM, which reduces the need to flush host and PIM caches | Cache coherence interface for external accelerators; encode coherence requirements (e.g. coherent vs. non-coherent) in memory requests | Accommodate coherence traffic (may be a side channel depending on the nature of coherence implementation); support encoding of coherence requirements (e.g. coherent vs. non-coherent) with memory accesses |
| Inter-PIM interconnect implementation using a variant of the memory interface protocol[[21]](#footnote-21) | None | Accommodate dual-mode operation of interface for either host-PIM or inter-PIM communication; flexible allocation of interface width (or bandwidth) resources to accommodate different needs between the two usage types |
| Efficient, user-level communication of status and error signals from PIM to host | Sufficient numbers of user-level resources to monitor updates to user-defined memory addresses and/or coherence events to user-defined addresses and trigger actions in response | Accommodate access notifications on accesses to specified addresses; or no additional capabilities (beyond coherence traffic) if processor monitors coherence events |
| In-memory complex gather/scatter and arbitrary-depth pointer dereferencing | Support for API features to communicate complex memory operations from software to hardware and/or hardware features to detect such opportunities for such operations; ability to issue complex memory commands (beyond simple loads and stores) and their corresponding arguments; ability to store or load arbitrary numbers of data words associated with a single complex memory operation | Flexible command opcode encodings (beyond simple loads and stores); variable length argument encodings for commands (address ranges or complex access patterns instead of a single addresses) which may span multiple command packets; and arbitrary numbers of data transfer packets associated with a single complex opcode |
| Low-overhead enqueue/dequeue to/from UMQ | ISA extensions for atomic enque/dequeue operations; optionally, capabilities to monitor updates to head/tail pointers of multiple queues | None |
| Low-overhead issue of fine-grain in-memory operations (e.g., op-and-store) | ISA extensions to issue fine-grain, in-memory operations | Extended opcode space for issuing augmented load/store operations |
| Inter-PIM communication via memory switch on host (see Section 3.2.6) | Ability to receive read/write requests from memory interfaces and route them to other memory channels (and route responses back to requestors) | “Full-duplex” ability to send and receive memory requests and responses to/from either the host or memory stack (PIM) |

# Case Studies

In this section, we present two simple examples and one proxy app adapted for a PIM system.

While our API and evaluation tools are intended to support multi-level memories, that support is currently not validated and we limit our early application analysis to a single-level of PIM-attached (DRAM) memory.

## Simple Tests

As a preliminary study, we evaluated the proposed PIM architecture over two simple test cases: a multi-threaded implementation of a prefix sum computation and a data-parallel implementation of a weighted sum of vectors kernel. The former is primarily latency bound while the latter is bandwidth-limited. These examples are intended to illustrate the potential performance benefits of PIM using simple benchmarks and are not intended to be representative of the performance impact of PIM on any specific real application(s). Further, these test cases are run on scaled-down machines configurations and do not correlate directly to the proposed FastForward node architecture.

Table 13 shows the performance of the parallel prefix sum benchmark on four PIM processors (i.e., four memory stacks) with 1 CPU core each normalized to the execution on a 4-core host processor. As expected, performance improves for configurations with lower memory latency despite the reduced execution frequency of PIM processors. While the specific latency reduction in a PIM processor is implementation-dependent, factors such as the elimination of off-chip transfers, shorter wire delays, and simpler cache hierarchies can contribute to this reduction.

Table 13: Normalized parallel prefix sum performance on PIM

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Computation on Host** | **PIM Config. 1** | **PIM Config. 2** |
| Host CPU cores used | 4 | N/A | |
| PIM cores used | N/A | 4 | |
| Normalized core freq. | 1 | 0.5 | |
| Normalized memory latency | 1 | 0.7 | 0.5 |
| Normalized execution time | 1 | 0.87 | 0.77 |

Table 14 shows the performance of the weighted sum of vectors kernel running on four PIM processors with one GPU each normalized to the execution on a high-performance host GPU. To ensure a fair comparison, we assume the host also uses stacked DRAM mounted on an interposer with the processor to achieve high bandwidth. The PIM processors are still able to achieve higher intra-stack bandwidth. As was done throughout this document, we make conservative assumptions about intra-stack bandwidth that could still enable the use of commodity DRAM dies in a PIM-enabled memory stack. More aggressive assumptions about modifications to the DRAM dies will enable even greater performance gains.

Table 14: Normalized weighted sum of vectors kernel performance on PIM

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Computation on Host** | **PIM Config. 1** | **PIM Config. 2** |
| Normalized FLOPs used | 1 | 0.27 | |
| Normalized memory bandwidth | 1 | 2 | 4 |
| Normalized execution time | 1 | 0.51 | 0.26 |

While PIM-based execution can also yield energy benefits as well as performance improvements, our evaluation infrastructure currently does not incorporate energy models. We envision the inclusion of energy modeling capabilities in a future release of the tools.**[\*]**







# References

1. 3D-ICs. [http://www.jedec.org/category/technology-focus-area/3d-ics-0](http://www.jedec.org/category/technology-focus-area/3d-ics-0/).
2. JEDEC Wide I/O Single Data Rate Specification. <http://www.jedec.org/standards-documents/results/jesd229>.
3. Hybrid Memory Cube Consortium. <http://hybridmemorycube.org/>.
4. Heterogeneous System Architecture: A Technical Review. <http://developer.amd.com/wordpress/media/2012/10/hsa10.pdf>.
5. The HSA Foundation. <http://hsafoundation.com/>.
6. Stone, Harold S., "A Logic-in-Memory Computer," IEEE Transactions on Computers, vol.C-19, no.1, pp.73-78, Jan 1970.
7. Kogge, P.M.; Sunaga, T.; Miyataka, H.; Kitamura, K.; Retter, E., "Combined DRAM and logic chip for massively parallel systems," Proceedings of the Sixteenth Conference on Advanced Research in VLSI, pp. 4-16, Mar 1995.
8. Patterson, D.; Anderson, T.; Cardwell, N.; Fromm, R.; Keeton, K.; Kozyrakis, C.; Thomas, R.; Yelick, K., "Intelligent RAM (IRAM): chips that remember and compute," Digest of Technical Papers, IEEE International Solid-State Circuits Conference, pp. 224-225, Feb 1997.
9. Oskin, M.; Chong, F.T.; Sherwood, T., "Active Pages: a computation model for intelligent memory," Proceedings of the 25th Annual International Symposium on Computer Architecture, pp. 192-203, 27 Jun-1 Jul 1998.
10. Yi Kang; Wei Huang; Seung-Moon Yoo; Keen, D.; Zhenzhou Ge; Lam, V.; Pattnaik, P.; Torrellas, J., "FlexRAM: toward an advanced intelligent memory system," Proceeding of the International Conference on Computer Design, 1999, pp. 192-201, 1999
11. Hall, M.; Kogge, P.; Koller, J.; Diniz, P.; Chame, J.; Draper, J.; LaCoss, J.; Granacki, J.; Brockman, J.; Srivastava, A.; Athas, W.; Freeh, V.; Jaewook Shin; Joonseok Park, "Mapping Irregular Applications to DIVA, a PIM-based Data-Intensive Architecture," Proceeding of Supercomputing 1999, 13-18 Nov 1999.
12. Richard C. Murphy, Peter M. Kogge, and Arun Rodrigues. “The Characterization of Data Intensive Memory Workloads on Distributed PIM Systems”, In Revised Papers from the Second International Workshop on Intelligent Memory Systems (IMS '00), Frederic T. Chong, Christoforos E. Kozyrakis, and Mark Oskin (Eds.). Springer-Verlag, London, UK, pp. 85-103, 2000.
13. Arun Rodrigues, “The UHPC X-Caliber Project Architecture, Design Space, & Codesign,” Sandia National Labs.
14. Torrellas, J., "FlexRAM: Toward an advanced Intelligent Memory system: A retrospective paper," IEEE 30th International Conference on Computer Design, 30 Sep-3 Oct 2012.
15. Christos Kozyrakis, “Is it time for processing near memory?,” Panel Discussion, Stanford PPL Retreat, May 31st 2012.
16. J. Thomas Pawlowski, “Hybrid Memory Cube (HMC)”, Hot Chips 23, Augus, 2011.
17. AMD I/O Virtualization Technology (IOMMU) Specification, Revision 2.00. <http://support.amd.com/us/Processor_TechDocs/48882.pdf>
18. Exascale FastForward Heterogeneous Processor Research Report, AMD FastForward P2 Deliverable, Jun 2013.
19. Derek R. Hower, “Acoherent Shared Memory,” Ph.D. Thesis, 2012.
20. Cantin, J.F.; Smith, J.E.; Lipasti, M.H.; Moshovos, A.; Falsafi, B., "Coarse-Grain Coherence Tracking: RegionScout and Region Coherence Arrays," IEEE Micro, vol.26, no.1, pp. 70-79, Jan-Feb 2006.
21. Zebchuk, J.; Cain, H.W.; Tong, X.; Srinivasan, V.; Moshovos, A., “ReCaP: A Region-Based Cure for the Common Cold (Cache)”, Proceedings of the 19th International Symposium on High Performance Computer Architecture, Feb 2013.
22. Lebeck, A.R.; Wood, D.A., "Dynamic self-invalidation: reducing coherence overhead in shared-memory multiprocessors," Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp.48-59, 22-24 Jun 1995.
23. Singh, I.; Shriraman, A.; Fung, W.W.L; O'Connor, M.; Aamodt, T.M., "Cache Coherence for GPU Architectures," Proceedings of the 19th International Symposium on High-Performance Computer Architecture, pp. 578-590, Shenzhen, China, Feb 23-27, 2013.
24. Gupta, A.; Weber, W.-D.; Mowry. T., “Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes,” Proceedings of 1990 International Conference on Parallel Processing, pp. 312-321, Aug 1990.
25. ITRS 2011 Edition. <http://www.itrs.net/Links/2011ITRS/Home2011.htm>.
26. Linux Programmer's Manual, pthreads – POSIX threads.
27. OpenCL 1.2 Reference Pages. <http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/>.
28. Kitten Lightweight Kernel. <https://software.sandia.gov/trac/kitten>.

1. Please refer to Section 4 for work dispatch API descriptions. [↑](#footnote-ref-1)
2. While the baseline hardware/software architecture does not preclude the implementation of work-stealing and other load-balancing schemes on top of this queuing framework, exploration of such schemes is outside the scope of this milestone. [↑](#footnote-ref-2)
3. Enabling the OS components that update page tables and propagate those changes to all sharers of the tables to run on accelerators would require much more significant OS modifications than we assume for our baseline. [↑](#footnote-ref-3)
4. If a PIM device needs access to data in another level of the memory system, we assume the path for that access consists of a combination of communication within the PIM’s level of the memory hierarchy and the links that need to exist between memory levels regardless of PIM capabilities. [↑](#footnote-ref-4)
5. Please see Section 3.2.9 for a discussion of host-PIM cache coherence considerations. [↑](#footnote-ref-5)
6. Note that this organization likely increases memory switch area and energy due to higher bandwidth needs. However, that overhead is likely (at least in part) offset by unifying the switching among memory channels and L2 banks as well as the reduced size of cache bank accessed on each L2 access. [↑](#footnote-ref-6)
7. We include snooping schemes with probe filtering in the general classification of “directory-based” coherence schemes from a storage overhead point of view. [↑](#footnote-ref-7)
8. While this is not a strict requirement, matching the die sizes helps improve interposer area utilization for a given DRAM capacity or bandwidth. [↑](#footnote-ref-8)
9. CU area estimates are based on 14nm technology for which we have better visibility; further technology scaling may be available in the exascale timeframe that may allow additional capabilities. [↑](#footnote-ref-9)
10. We make conservative assumptions about in-stack bandwidth that assume DRAM die designs that are expected to be viable as commodity parts in 2020. More aggressive assumptions about freedom to customize DRAM die architectures can provide significantly greater in-stack bandwidth. [↑](#footnote-ref-10)
11. This assumes an in-stack logic die size of 8mm x 12mm, and average traversal of half of each dimension. Includes only the energy to drive the wires (i.e., excludes routing energy). [↑](#footnote-ref-11)
12. This assumes a host die size of 30mm x 30mm, which is comparable to the anticipated planar area of AMD’s EHP baseline, and average traversal of half of each dimension. Includes only the energy to drive the wires (i.e., excludes routing energy). [↑](#footnote-ref-12)
13. This assumes a host die size of 30mm x 30mm, which is comparable to the anticipated planar area of AMD’s EHP baseline, and average traversal of a quarter of each dimension as a result of data and compute placement to improve locality. Includes only the energy to drive the wires (i.e., excludes routing energy). [↑](#footnote-ref-13)
14. This assumes the same GPU CU cluster architecture as EHP, with a similar 2MB cache per cluster. [↑](#footnote-ref-14)
15. EHP incorporates 2MB L2 per CU cluster. Please refer to EHP documentation [18] for more details. [↑](#footnote-ref-15)
16. EHP incorporates 4MB LLC per *memory slice* (this estimate ignores the additional L3 on EHP for CPU accesses). Please refer to EHP documentation [18] for more details. [↑](#footnote-ref-16)
17. This list is based on our current needs and potential future use cases currently envisioned. Supported parameters may evolve in the future as our understanding of PIM and usage models mature. [↑](#footnote-ref-17)
18. Supported memory types may evolve in the future as our understanding of PIM and usage models mature. [↑](#footnote-ref-18)
19. The current version of the API assumes that PIM processors at different levels of a multi-level main memory hierarchy are not cache coherent with each other. [↑](#footnote-ref-19)
20. The current version of the API does not differentiate between specific NVRAM implementation technologies. It is expected that the application must query other memory and PIM parameters to determine the detailed capability profile. We envision extending these capabilities to fully reflect emerging NVRAM technology characteristics in future revisions of the API.**[\*]** [↑](#footnote-ref-20)
21. While inter-PIM interconnect may use dedicated interfaces and protocols, using a variant of the memory interface enables a variety of flexible design and implementation simplifications. [↑](#footnote-ref-21)