# Design and Exploration of a Processing-in-Memory Architecture for the Smart Memory Cube

Erfan Azarkhish (erfan.azarkhish@unibo.it)

October 12, 2016

Partners: Davide Rossi (davide.rossi@unibo.it)
Igor Loi (igor.loi@unibo.it)
Supervisor: Professor Luca Benini (luca.benini@iis.ee.ethz.ch)

## Abstract

3D integration of solid-state memories and logic, as demonstrated by the Hybrid Memory Cube, offers major opportunities for revisiting near-memory computation and gives new hope to mitigate the power and performance losses caused by the "memory wall". In this report we present the first exploration steps towards the definition of the Smart Memory Cube, a new Processor-in-Memory (PIM) architecture that greatly enhances the capabilities of the logic-base die in a hybrid memory cube. Furthermore, the required software stack for such architecture is presented and benchmark applications are provided to help quantify strengths and weaknesses of the PIM architecture. Finally, results and insights gained in the first year of the project are reported and summarized. A software package is released along with this report.

# Contents

# 1 Introduction

An overview of the Smart Memory Cube (SMC) is illustrated in Figure1.left. In this project, two SMC models have been developed. The first one is a cycle-accurate model using SystemVerilog language. The second is a high-level gem5-based model of SMC, suitable for full-system simulations. Moreover, a closed-loop simulation infrastructure has been developed providing the possibility to instantiate the cycle-accurate SMC model inside gem5 (See Figure1.right).

The organization of this report is as follows: First, an introduction and overview of the SMC simulation environment is presented, then in Section 3 design of the PIM architecture is described. In Section 4 the developed software stack is presented, and next calibration of the simulation environment based on the data from the state of the art is described. Finally, benchmark applications are presented in Section 6, and observations and conclusions are described in Sections 7 and 8.

# 2 A Brief Review of the SMC Simulator

A full system simulation environment for the SMC has been developed in [Erfan Azarkhish, 2015]. As shown in Figure1, this environment aims to model a single Hybrid Memory Cube, augmented with a Processor-in-Memory (PIM), at two levels of abstraction: cycle-accurate and event driven. The main purpose of this environment has been full system simulation and exploration of a generic PIM located in the logic-base of the Hybrid Memory Cube (HMC) along with all overheads such as actual task and binary code offloading through a standard and realistic software stack, as well as system-level effects such as memory management, cache coherence, and the dynamic behaviours of the operating system.

In order to calibrate our simulation environment and guarantee consistency of the reported results, we also developed a cycle-accurate (CA) model [Erfan Azarkhish, 2014] of the HMC internals, and provided the facilities for closed-loop co-simulation between the gem5 host and CA memory model. In [Erfan Azarkhish, 2015] we had shown that the high-level gem5-based model delivers reasonable accuracy in terms of execution time and reported bandwidth. In this report, we will show how this model is calibrated based on the publicly available data, and we try to evaluate its performance using different benchmarks. Figure2 illustrates a snapshot of an ARM host with four processing cores connected to the Smart Memory Cube modeled in gem5. A PIM subsystem has been attached to the main interconnect located on the Logic-base layer of SMC.

In the next section we will describe the newly added features to PIM and the design choices in its architecture. Next, the software stack is presented and after benchmarking applications, conclusions are reported.

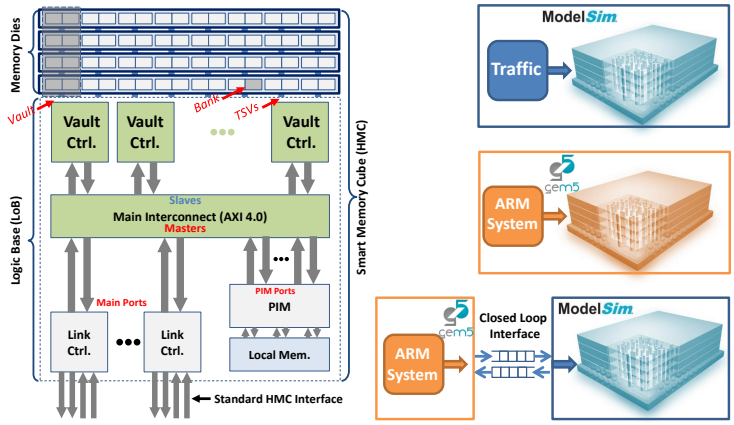


Figure 1: Overview of the Smart Memory Cube (SMC), as well as the three methodologies devised by ETHZ to study SMC: 1) ModelSim standalone simulation, 2) gem5 standalone simulation, and 3) gem5-ModelSim closed-loop simulation

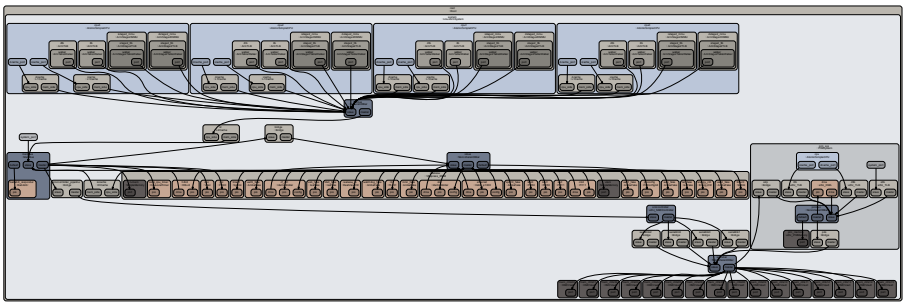

Figure 2: A snapshot of the SMC simulation environment composed by the ARM host, the HMC, and the PIM subsystem

# 3 Design of the Processor-in-Memory

As we had discussed previously in [Erfan Azarkhish, 2014], it is more beneficial to put a processing element behind the main interconnect of the SMC located on the Logic-Base (LoB) (See Figure1.left). Plus, we have shown that delivering a residual bandwidth to the Processor-in-Memory does not disrupt the bandwidth or latency delivered to the main serial links. We chose a simple ARM core as the baseline for our PIM, because it offers a mature software stack, well understood system bridges (AXI) and it is an energy-efficient architecture. Nevertheless, the architecture is not limited to ARM and will be enhanced later. As shown in Figure4, PIM is attached to the main interconnect on the LoB of the SMC through its own local interconnect, and is able to access all vaults through

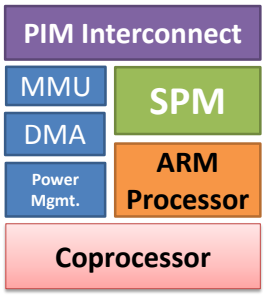


Figure 3: A block-diagram of the functionalities modelled in PIM

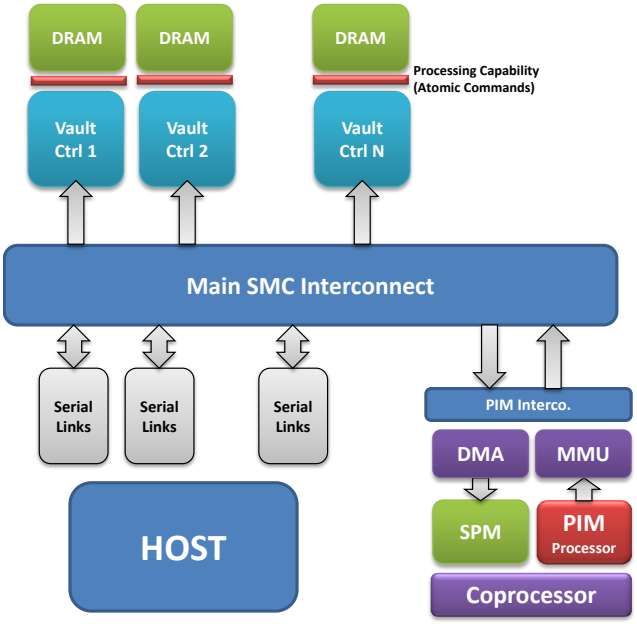


Figure 4: An overview of the PIM architecture inside the Smart memory Cube

the main interconnect. Figure3 depicts a high level block-diagram of the functionalities implemented in PIM. We will describe the newly added components throughout this report. PIM has been modelled as a different *subsystem* in gem5 and is composed by a simple ARM core, a scratchpad memory (PIMMemory), bridges to allow communication with the host system, three TLB components (iTLB, dTLB, and sTLB), and a DMA engine. Figure5 shows an schematic view of the PIM in gem5.

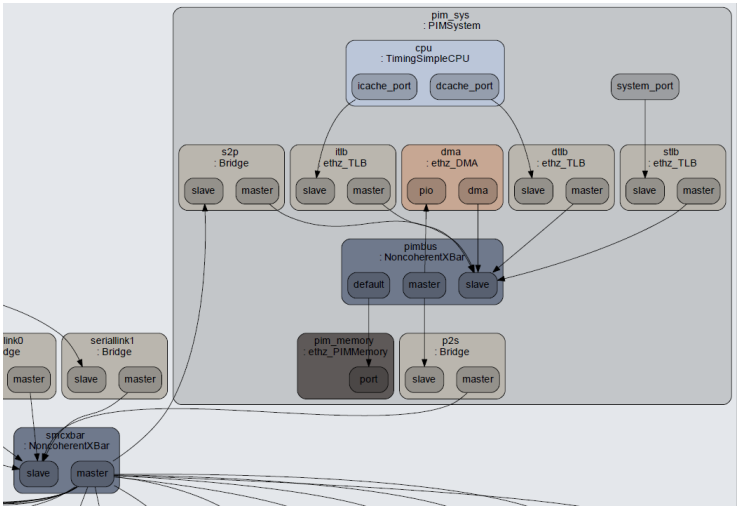


Figure 5: A snapshot of the PIM subsystem in gem5

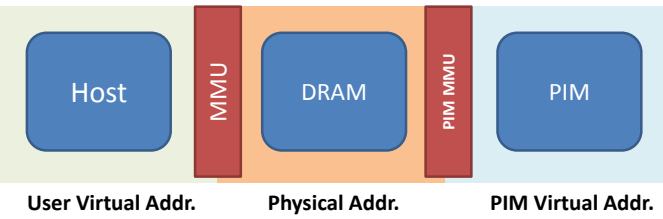


Figure 6: PIM's memory management unit to translate PIM's virtual address to physical address

## 3.1 PIM's Memory Model

PIM has been designed to access user-space virtual memory and to directly operate on user's data structures. A glance at Figure5 also reveals that PIM does not access physical memory locations directly, and a Translation Look-aside Buffer (TLB) mediates its accesses to memory. Apart from memory protection benefits, this replaces memory-copy from user's memory to PIM with a simple virtual pointer passing. Scalability and programmability are improved in this way, and offloading overheads are reduced to a great extent. Figure6 illustrates an overview of virtual memory in the SMC system.

Another important point is that user's memory is paged in conventional architectures. Therefore, PIM should support this, as well. However, to add more flexibility we introduce the concept of slices in the next subsection.

### 3.1.1 Slices

We define a *Slice* as a region of memory which satisfies the following properties:

- It should be composed by 1 or more memory-pages, there fore a slice is always page-aligned, and its size is a multiple of system's page-size.
- It should be contiguous both virtual and physical memory spaces.

With this definition, contiguous memory pages which map to contiguous page-frames in the underlying physical memory can be merged to build larger slices, and the size of different slices is not necessarily equal. This adds more flexibility in comparison with the memory pages. Plus, building large contiguous slices at the kernel offload time can reduce memory translation overheads and thus improve PIM's performance. Next subsection describes how these slices are used in PIM.

### 3.1.2 PIM's TLB

PIM works with linear virtual addresss starting from the address Zero. These virtual addresses are mapped to the physical address space in the granularity of slices. The first slice is devoted to PIM's scratchpad memory. As an example, if PIM has 1MB of SPM mapped starting from 0x70000000 in physical space. The region for the first slice will be [0x00000000, 0x000FFFFF], and the rule associated with this slice in the TLBs will be the following:

[0x00000000, 0x000FFFFF] ⇒ [0x70000000, 0x700FFFFF]

The rest of the memory space starts immediately after this region, at the granularity of slices, with the rules defined in the slice-table (described in the next subsection). Since the main target for our PIM is execution of medium sized computation kernels with a few hundreds or thousands of instructions repeating in nested loops, we can limit PIM's instruction fetch to its own SPM only. This can improve predictability and performance of the executed code, because the access time of PIM's SPM is very small compared to the main DRAM and there will be no need for TLB refills for instruction accesses. To achieve this, we pre-program the iTLB component shown in Figure5 with only one fixed rule (the one mentioned above). Also, sTLB component (Figure5) is connected to gem5's *system-port*, which is responsible for debugging and simulation related facilities. However, dTLB is a full-featured TLB with the ability to dynamically map new slices based on the rules defined in the slice-table. Figure7 depicts an example TLB with a maximum of four translation rules. The functionality of these TLBs is defined by the ethz_TLB class in the following files:

./GEM5/gem5/src/mem/ethz_TLB.py

./GEM5/gem5/src/mem/ethz_tlb.cc

| Rule | Source Region | Destination Region | Size | Read/Write Flag |
|---|---|---|---|---|
| Enabled | [0x00000000, 0x000FFFFF] | [0x70000000, 0x7000FFFFF] | 1MB | R/W |
| Enabled | [0x01100000, 0x0110FFFF] | [0x81000000, 0x81000FFFF] | 64KB | R/W |
| Disabled | | | | |
| Disabled | | | | |

Figure 7: Structure of PIM's TLB with four elements and two currently active rules

./GEM5/gem5/src/mem/ethz_tlb.hh

The request packets initiated by the PIM processor arrive at the slave port of the TLB components. TLB tries to remap the address of these packets to physical address using the rule already present in it. If not possible a TLB-miss will occur, and the processor will be stalled until the rule is loaded from the slice-table. The replacement policy currently implemented in these TLBs follows ideal Least Recently Used (LRU) algorithm, associating a time-stamp to each rule to keep track of the oldest rule present in it. This behavior has been modeled in all three transaction levels of gem5 (timing, functional, and atomic), and its parameters can be altered in the simulation scripts.

Upon a TLB miss, the new rule should be loaded from the slice-table. Most host-side accelerators rely on the host processor to perform this operation. The host OS consults system's page-table to reprogram the IO-MMU of the device, and they wakes up the accelerator to continue its operation. However, since PIM is located in the main memory and it is far from the host processors, asking them for a refill upon every miss results in a large performance overhead. As an alternative, our TLB contains a simple controller which is responsible for fetching the required rule from the slice-table.

### 3.1.3 Slice-Table

Slice-table is a data structure containing all translation rules for the computation kernel currently executing on PIM. This data structure is allocated by the device-driver in the kernel-memory space and is built during the task offloading procedure. There is no theoretical limitation on the number of slices it can map, and this number is only limited by the availability of contiguous memory in the kernel space. The basic structure defining the slice-table and a simple example for its operation are shown in Figure8.

Since the slices can have arbitrary sizes, implementing the slice-table as a simple table of slices can complicate the lookup procedure. In order to minimize the number of DRAM access required to fetch rules from this table, in this table, we keep entries for the underlying pages rather than the slices. Therefore, for pages in the same slice, we store the same translation rule (See Figure9). This

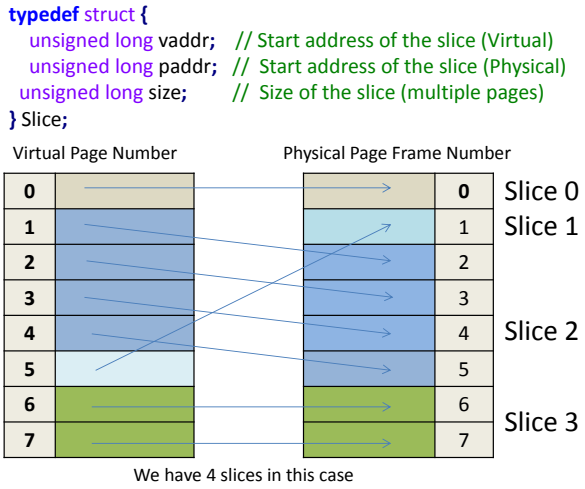


Figure 8: The basic structure defining the slice-table and a simple example

| PIM V.A | vaddr | paddr | Size (Pages) |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 1 | 1 | 2 | 4 |
| 2 | 1 | 2 | 4 |
| 3 | 1 | 2 | 4 |
| 4 | 1 | 2 | 4 |
| 5 | 5 | 1 | 1 |
| 6 | 6 | 6 | 2 |
| 7 | 6 | 6 | 2 |

Figure 9: An illustrative example for the slice-table

simplifies the lookup procedure at the cost of redundant rules in the slice-table. For example if PIM accesses the virtual address 3 (Figure8), using a single DRAM burst of 3 words (12Bytes for 32bit architectures) it is possible to fetch the rule completely for Slice 2. This way, there is no need to access the slice-table twice, regardless of the size of the slices. The structure for Slice is defined in:

`./SW/HOST/driver/pim.h`

More details are given in Section 4.4.2. Upon execution of any PIM-enabled simulation scenario, PIM's memory mapped will be reported on the screen.Figure10 shows an example.

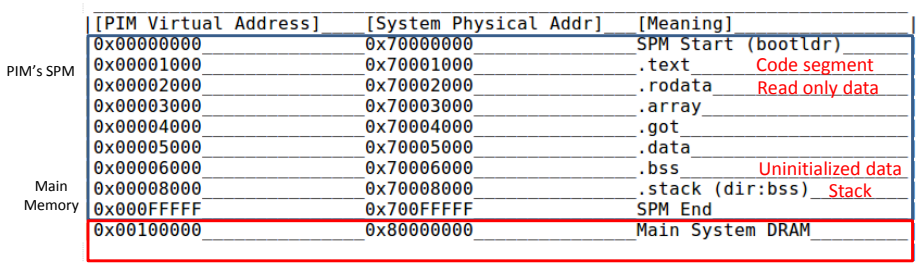


Figure 10: PIM's memory map, automatically reported in the beginning of the simulation

## 3.2 More Advanced Functionalities

The latency of the DRAM access by PIM is not negligible (See Section 5.1). To avoid this becoming the performance bottleneck, latency-hiding mechanisms should be employed. Caches and prefetchers provide a higher-level of abstraction without much control. This is desired when we are far from the memory and we need to be flexible to support many different main memory configurations. DMA engines, on the other hand, provide more control which makes more sense in a near-memory processor. For this reason we have augmented PIM with a DMA engine, and we use software's aid to manage it. A complementary way to address this problem is to move some very specific arithmetic operations directly to the DRAM dies and ask the vault controller to do them "atomically".

We have implemented both mechanisms in our architecture. For computations that need to gather information not fully localized to a single memory wall and have a high computational intensity, DMA can be more beneficial to gather the data in the SPM. On the other hand, highly localized computations with low computational intensity are better performed as close as possible to the memory dies. This section describes implementation of both mechanisms in the SMC Simulator.

### 3.2.1 Direct Memory Access (DMA)

As shown in Figure5, PIM features a DMA engine which is capable of bulk data transfers between the DRAM vaults and the PIM's SPM. It supports multiple outstanding transactions by having several DMA resources (See Figure11). DMA resources are used to identify the outstanding transactions and they are explicitly reserved by software and released when the transfer is finished. The software program running on PIM simply programs the registers devised for this purpose (See Figure12) to initiate the transactions. A macro called $DMA\_REQUEST$ is also provided to hide the low-level details. For example, to initiate a DMA transfer from the array B in DRAM to Bbar in the

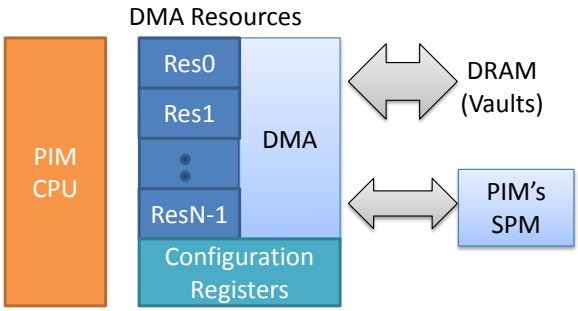


Figure 11: The DMA engine for PIM

```
/*
 DMA Access registers:
 PIM's DMA can transfer data between PIM's vector register file and a memory location:
*/
volatile ulong_t PIM_DMA_MEM_ADDR;  // (Virtual Address) of the main memory location
volatile ulong_t PIM_DMA_SPM_ADDR;  // (Physical Address) of the SPM location
volatile ulong_t PIM_DMA_NUMBYTES;  // Number of bytes to transfer
volatile uint8_t PIM_DMA_COMMAND;   // PIM DMA Command Register
volatile uint8_t PIM_DMA_STATUS;    // Status of the DMA Resources

#define DMA_REQUEST(MA,SA,NB,CM, RES) {\
    PIM_DMA_MEM_ADDR=(ulong_t)MA; \
    PIM_DMA_SPM_ADDR=(ulong_t)SA; \
    PIM_DMA_NUMBYTES=(ulong_t)NB; \
    PIM_DMA_COMMAND=((CM)|(RES)); }
```

Figure 12: A list of DMA's programmable configuration registers

SPM using **DMA Resource 1**, it is enough to write the following statement:

DMA_REQUEST(B, B_ping, XFER_SIZE, **PIM_DMA_READ**, DMA_RES1);

In current implementation, seven DMA resources are available (DMA_RES0 TO DMA_RES6) and each one can be used for a WRITE or READ independently. The key feature of this component is support for virtual address without any alignment or size restrictions. This means that the software running on PIM can pass any virtual pointer to the DMA and request transfers of any size (only limited by the size of the SPM). PIM's DMA breaks transactions into sub-transactions to fit in slices and then converts their addresses to physical address separately. ethz_DMA represents the name of the component which has been designed for this purpose and here are the related source codes for it:

./GEM5/gem5/src/dev/ethz_DMA.py

./GEM5/gem5/src/dev/ethz_dma.cc

./GEM5/gem5/src/dev/ethz_dma.hh

The sub-transactions are broken into smaller DRAM aligned transactions using the functionality inherited from gem5's standard DMA, and then injected towards the memory. Here are the source codes of this base class:

./GEM5/gem5/src/dev/Device.py

./GEM5/gem5/src/dev/dma_device.cc

./GEM5/gem5/src/dev/dma_device.hh

After all sub-transactions complete successfully, the status register of the ethz_DMA is set to ready, and an interrupt is sent to PIM to wake it up (only if it is asleep). The ethz_DMA component works based on a state machine and its clock frequency can be set independently from the PIM processor. Please refer to the Section 6 for example usage in the software codes. One subtle point to mention here is that currently, virtual to physical address translation for the DMA bursts is done using gem5's functional accesses and in zero time. Our observations show that due to the large size of the slices, the overhead of this operation is negligible and does not contribute to a significant change in the results.

### 3.2.2 Moving Part of the Computation to the DRAM Dies

Abstracted memory interface is an important feature in the Hybrid Memory Cube which allows for supporting commands beyond simple READ and WRITE operations. Therefore, it is possible to augment the vault controllers with simple arithmetic capabilities and expose these commands to the external world. Atomic HMC commands specified in the [HMC, 2014] are good examples but for a different purpose: to implement synchronization mechanisms for highly parallel applications. Other examples can be found as well, which benefit from in-memory operations but they are not necessarily parallel. Figure13 contains one such example. As can be seen, instead of fetching a location from DRAM, incrementing it, and writing it back, it seems more beneficial in terms of both performance and energy to directly do it in the memory itself. Similarly, operations on a single operand, or a small set of operands local to one DRAM row and with no dependency on the other banks/vaults, can be directly computed in the DRAM devices (or in the vault controllers). We have extended the Packet class in gem5 (which is the communication means in its standard memory system) to support a subset of HMC specific commands (e.g. Incrementing and Addition). This class can be found here:

./GEM5/gem5/src/mem/packet.hh

We have also implemented the functionality of these commands in the Ab-

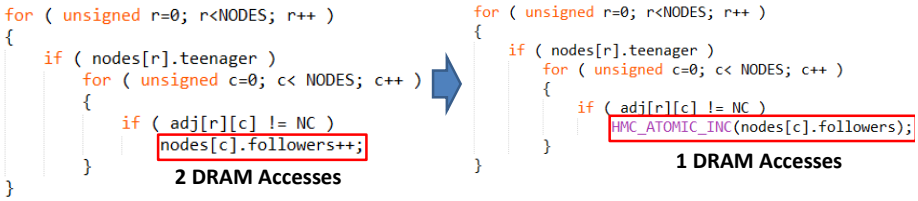


Figure 13: A use-case example for the atomic increment command

stractMemory class. We assume that these simple operations are implementable in the DRAM-dies and their latencies are hidden behind the timings of the DRAM timings. Therefore, we do not consider any additional overhead for execution of these atomic operations, and they are handled similar to normal WRITE operations from the timing point of view.

On the other hand, instead of modifying PIM's ISA to support these commands, we added specific memory mapped registers to the PIM processor (e.g. HMC_ATOMIC_INCR). As an example, to increment memory location X (virtual address) atomically, the user should assign the address X to the HMC_ATOMIC_INCR register, using the macro provided for this purpose:

```
#define HMC_ATOMIC___INCR(X) {HMC_ATOMIC_INCR = (ulong_t)&((X));}
```

This request will go to the dTLB component and the address X will be translated to physical address, and a packet containing this command will be sent to the main memory. For practical examples of these atomic commands, please refer to the Section 6.

### 3.2.3 Arithmetic Coprocessor

A simple arithmetic coprocessor has been added to PIM which is able to operate directly on PIM's scalar and vector registers. Currently, this coprocessor is used for debugging and observing the floating-point behaviour in PIM, but its functionality can be easily extended to support vector arithmetic. The source codes related to this module are present in:

```
./GEM5/gem5/src/mem/ethz_pim_memory.cc
```

Finally, a complete list of parameters related to PIM is available in:

```
UTILS/models/system/gem5_pim.sh
```

## 3.3 Supporting ARMv8 Architectures

The SMC Simulation environment supports building systems composed by both ARMv7 and ARMv8 based processors. The user simply passes 7/8 as an argu-

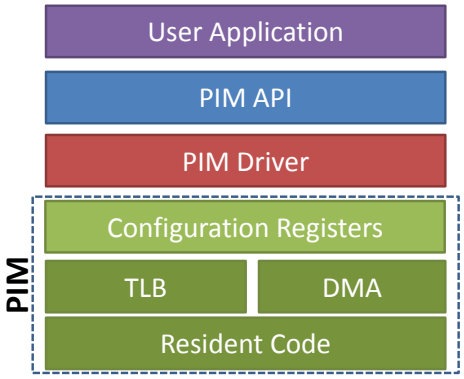


Figure 14: An overview of the software stack designed for communication with PIM

ment to the simulation script, and the scripts will automatically handle rest. For example, the following command will build and run an ARMv8 based system running linux, with no PIM:

```
./scenarios/c-gem5/a-tests/1-arm-hmc-linux.sh 8 -o
```

One important point to mention is that when an ARMv8 based host is chosen, PIM's model will automatically change to ARMv8, as well. This ensures that the pointers can be passed across host and PIM without any problems, and that the whole virtual address space of the user applications is visible by PIM. All software programs, drivers, APIs, and firmwares are scalable and are easily and automatically ported from ARMv7 to ARMv8.

# 4 Design of the Software Stack

An overview of a simple software stack for communication with PIM is illustrated in Figure14. The main goal in design of this software stack has been for any user level application to be able to communicate with PIM and offload tasks to it. In this section, we will take a look at different levels of this software stack starting from the lowest level: the codes running on the PIM processor.

## 4.1 PIM's Resident Code

The main structure of the resident program is the same as what is reported in [Erfan Azarkhish, 2015] in Section 5.3.1. The only difference is the newly added special registers to extend the functionalities:

```
./SW/PIM/resident/registers.h
```

PIM features two types of scalar and vector memory mapped regions which can be seen as tightly coupled processor registers in a real hardware implementation. In current simulation environment, however, they have been modelled using simple memory-mapped locations, rather than extending PIM's ISA. The scalar registers have a width of 32-bits for ARMv7 and 64-bits for ARMv8-based PIM. The number of these registers is defined by PIM_SREG_COUNT which can be modified in the simulation scripts (It is set to 8 by default). Scalar registers can be used for pointer and parameter passing from host, status checking, internal computation and communication with the coprocessor.

```
volatile ulong_t PIM_SREG[PIM_SREG_COUNT];
```

The vector register file, on the other hand, provides a contiguous scratchpad area composed of PIM_VREG_SIZE bytes (4096 Bytes by default):

```
volatile uint8_t PIM_VREG[PIM_VREG_SIZE];
```

Any location in the vector register-file can a source or destination for DMA transfers. This provides a high flexibility in support of different DMA transfer models (e.g. ping-pong or triple buffering) with different transfer sizes. Furthermore, PIM's coprocessor can directly operate on these registers and use them as vector operands (This feature has not been implemented yet).

PIM also keeps pointers to the slice-table (described in Section 3.1.3). These pointers are updated during the offload procedure and are used by dTLB to handle TLB misses. It should be noted that the kernel code running on PIM should not modify the slice-table anyway as the resulting behaviour is unexpected. Finally, a set of special registers are available to facilitate communication with gem5's infrastructure for the sake of debugging, gathering statistics, or simulation hacks:

```
volatile uint8_t PIM_M5_REG;

volatile ulong_t PIM_M5_D1_REG;

volatile ulong_t PIM_M5_D2_REG;
```

The values which can be written to all PIM's registers are defined in:

```
./SW/PIM/resident/definitions.h
```

For example writing PIM_EXIT_GEM5 to the PIM_M5_REG will cause the simulator to exit immediately, and writing PIM_TIME_STAMP will record a time-stamp and reset all statistics. User applications can use these facilities to gather statistics for different sections of the executed code.

The boot-loader, link scripts, and the resident code have been updated since the last report to support the newly added functionalities. Here is a list of the modified files: Boot-loaders for ARMv7 and ARMv7:

```
./SW/PIM/resident/boot_arm7.S
```

```
./SW/PIM/resident/boot_arm8.S
```

The link script for both ARMv7 and ARMv8:

`./UTILS/models/system/pim_bare_metal_link.sh`

## 4.2 Device Driver for PIM

The device-driver code for PIM has been adopted from the standard Mali GPU Driver [MALI, ], and it has been designed carefully to be compatible with standard accelerators as well as parallel programming APIs (e.g. OpenCL). This light-weight device-driver provides a low-overhead and high-performance communication mechanism between the APIs and PIM. The main codes related to the device driver are available in:

`./SW/HOST/driver`

Device driver is responsible for the following functionalities:

- Sending specific commands to PIM
- Pinning the user pages to the main memory
- Slice-table allocation and sending its information to PIM

Direct memory-mapped communication, on the other hand, is handled by the user-level API to avoid context switching overhead. The three main structures for the PIM device (See Figure15)are defined in:

`./SW/HOST/driver/pim.h`

PIMDevice is the main device structure which contains the required information to communicate with PIM; Pages is the list of pages pinned for use by the PIM device, along with their physical addresses; and Slice is the main structure used for building the slice-table.

Upon installation, the **pim_init()** method will initialize PIM as a character device and allocate major and minor numbers for it. Also, it remaps the physical addresses range associated with PIM's SPM to the kernel virtual memory map. This memory map can be later used by the API for direct memory-mapped communication with PIM. The **ioctl()** systemcall is responsible for issuing device specific commands to PIM. The implementation for this system call is available in:

`./SW/HOST/driver/pim.c`

The PIM_IOCTL_ALLOCATE_DATA command to **ioctl** is responsible for offloading user-level data structures to the PIM device (described in Section 4.4.2). PIM_IOCTL_RELEASE_ALL_DATA releases all pinned memory pages, and PIM_IOCTL_REPORT_STATS reports driver-level statistics. One point to mention here is that, the current device-driver for PIM does not support recurrence and concurrent execution, therefore, it is not possible for multiple

```
// PIM device structure
typedef struct {
   dev_t dev;                        // Device number
   struct file_operations *fops;     // File operations
   struct cdev cdev;                 // Character device structure
   int min;                          // Minor number
   int maj;                          // Major number
   void *mem;                        // memory mapped region (kernel virtual address)
} PIMDevice;

/*
List of pages pinned for use by the PIM device, along with their physical address
*/
typedef struct {
   struct page** list;          // Pages pinned for use by the PIM device
   unsigned count;              // Number of pages pinned
   ulong_t *physical_addr;      // Physical address of each page
} Pages;

/*
Struct slice
*/
typedef struct {
   ulong_t vaddr;           // Start address of the slice (Virtual Address)
   ulong_t paddr;           // Start address of the slice (Physical Address)
   ulong_t  size;           // Size of the slice (multiple pages)
} Slice;
```

Figure 15: The tree main structs inside PIM's device-driver

user-level applications to work with it in parallel. Only one instance of user-level API is supported for now.

## 4.3 Application Programming Interface (API) for PIM

An object-oriented user-level API has been designed to abstract the low-level details of the device driver and facilitate user's communication with PIM device. The source codes related to this API are available in:

`./SW/HOST/api/pim_api.hh`

`./SW/HOST/api/pim_api.cc`

All offloading facilities and coordinating the computation on PIM are provided by this API. In addition to communication with the device-driver, it provides direct read and write to PIM's vector and scalar registers, as well as, giving simulation related commands to the gem5 simulator. The **open_device()** method opens the PIM device as a file and **mmap_device()** maps PIM's addressable memory to user level virtual address. The details of the offloading mechanisms are described in the next sections. If we take a look at one of the simulation scripts related to PIM:

`./scenarios/c-gem5/a-tests/6-arm-pim-manual-offload.sh`

We can see that, one important step in all of them is to build the required

firmware and software codes. For example for the device driver, the following set of commands are executed:

```
clonedir $HOST_SW_DIR/driver
```

```
run ./build.sh
```

```
returntopwd
```

These set of commands will create a copy of the **driver/** directory into the $SCENARIO_CASE_DIR directory, and execute the build script for it. A similar procedure should be followed for the resident code running on PIM ($PIM_SW_DIR/resident), the API ($HOST_SW_DIR/api), and the user-level application ($HOST_SW_DIR/app/offload_matrix). Finally, all the required files should be copied to the extra disk image dedicated for this purpose to be able to be used by the simulated system:

```
copy_to_extra_image driver/pim.ko driver/ins.sh ...
```

All these steps are necessary before starting the actual simulation.

## 4.4 Offloading Mechanisms

This section describes the implemented offloading mechanisms.

### 4.4.1 Computation Kernel Offloading

When the user level application calls the **offload_kernel()** method in the API and passes the name of the computation kernel to it, the API offloads the computation kernel to PIM's SPM byte-by-byte (See Figure16). This procedure results in dynamic binary modification for the PIM processor and allows for actually changing the executable code of the PIM processor during runtime (rather than relying on simulation hacks and tricks). For this procedure to be able to work, some preprocessing steps are required which we will describe here. The source code of all available computation kernels to be offloaded to PIM are available in:

```
./SW/PIM/kernels/
```

If we take another look at the previous simulation script:

```
./scenarios/c-gem5/a-tests/6-arm-pim-manual-offload.sh
```

We can see that the **matrix_add** kernel has been introduced as the computation kernel to offload to PIM using the following variable:

```
export OFFLOADED_KERNEL_NAME=matrix_add
```

Moreover, the build.sh script for the resident code is executed twice:

clonedir $PIM_SW_DIR/resident

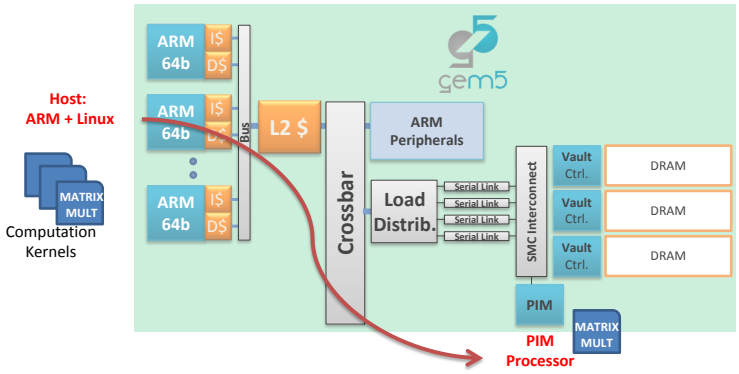


Figure 16: Offloading computation kernels from host to PIM

run ./build.sh $1

run ./build.sh $1 ${OFFLOADED_KERNEL_NAME}

returntopwd

The first call to the build.sh script, build the main resident code which should be always running on PIM. But, the second call build the kernel code (in this case matrix_add), and extracts the required binary sections from the ELF file. Taking a look at the build.sh reveals how this procedure is handled:

```
./SW/PIM/resident/build.sh
```

In the second call to build.sh, the source code for the **execute_kernel()** method will be replaced by the one provided by the matrix_add computation kernel, and then the source codes are compiled and linked the same as before. However, this time, the required sections are extracted from the resident.elf and stored in a special format (matrix_add.hex). The key point in this procedure is to store the modified sections of the binary ELF file in *.hex format and later ask the API to offload only these modified sections to PIM's SPM and replace the existing ones. Since the resident code running on PIM does not feature dynamic memory allocation or object oriented codes with polymorphic methods, the only modified sections are .text and .rodata which store the executable code and ready-only data, respectively (See Figure17). The two following commands in the build.sh script will do the job and append the extracted section to matrix_add.hex:

```
hexdump_section resident.elf "text" $2.hex
```

```
hexdump_section resident.elf "rodata" $2.hex
```

The resulting matrix_add.hex is stored in text format and is partly shown in Figure18. As can be seen, 3168 bytes will be offloaded to PIM's SPM in this example. One point to mention is that, it is possible to further reduce

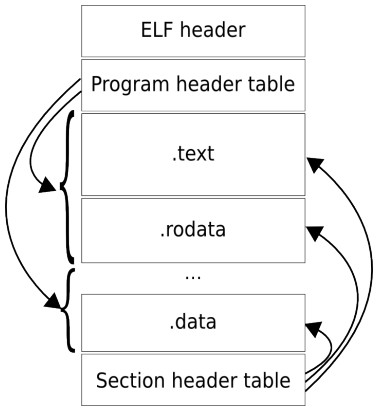


Figure 17: Different sections of the ELF binary format

```
@ADDR 1536          Load Address                    (.text section)
@SIZE 2436          Load Size
07 00 00 ea 06 00 00 ea 05 00 00 ea 04 00 00 ea 03 00 00 ea 02 00 00 ea 01 00 00 ea 00 00 00 ea ff ff ff ea 50 0f 11 ee 03 06 80
e3 03 05 80 e3 50 0f 01 ee 01 01 a0 e3 10 0a e8 ee 00 00 40 e0 2c 30 9f e5 2c d0 9f e5 b0 8f 10 ee ff 84 d8 e3 13 ff 2f 01 01 80
a0 e3 00 80 00 00 60 13 00 00 @END

@ADDR 8192                                          (.rodata section)
@SIZE 732
28 45 72 72 6f 72 29 3a 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 42 00 00 00 43 00 00 00 4d 61 74 72 69 78 20 61 64 64 69 74 69 6f 6e 20 66 69 6e 69 73
68 65 64 20 2e 2e 2e 00 00 00 00 @END
```

Figure 18: The *.hex format, for the kernel to offload to PIM

this number by only offloading the modified function (i.e. **execute_kernel()**), nevertheless, in current release, the whole .text and .rodata sections will be offloaded.

Taking a look at the user application for matrix addition:

`./SW/HOST/app/offload_matrix/main.cc`

We can see that, when the user application starts, after instantiating an object of PIMAPI, it offloads the binary kernel code to PIM by calling:

```
pim->offload_kernel((char*)FILE_NAME);
```

This API function simply parses the matrix_add.hex file and writes the bytes to the intended addresses in PIM's SPM (previously mapped to user space by a call to **mmap_device()**). Again, it is possible to reduce this offload latency by using system's DMA or employing compression techniques, however, in current release, kernel offloading is done byte-by-byte by a call to **write_byte()**. After this step, and after offloading the task (See the next section), an interrupt is sent to PIM to wake it up from sleep. PIM will check its commands register, and will start to execute the newly offloaded kernel. Needless to say, it is possible

```
// Data to be offloaded to PIM (virtual address)
struct PIMData
{
        void*       v_addr;     // Pointer to data (user virtual address)
        size_t      size;       // Size of the data (bytes)
        unsigned    reg;        // Index of PIM SREG to point to hold a virtual ptr to this data (0, 1
};

/*****************************************************/
// Task to be offloaded to PIM
class PIMTask
{
public:
        void addData(void* addr, size_t size, unsigned reg);
        void updatePages();
        // The data sets allocated by this task (Sorted List)
        vector<PIMData*>        data;

        // The virtual pages occupied by this task (Sorted List)
        vector<PIMPage*>        vpages;
};
```

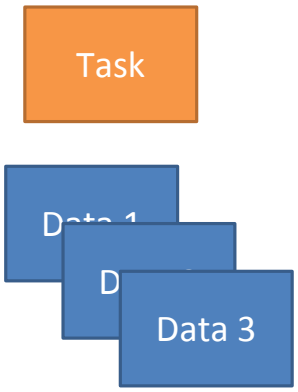


Figure 19: The PIMTask and PIMData structures for virtual pointer passing to PIM

to repeat these procedure several times (for different computation kernels or different tasks using the same kernel).

### 4.4.2 Task Offloading

Assume that three matrices (A, B, and C) are previously allocated in the user level memory. Our goal is to pass the pointer (virtual address) of this matrices to PIM and ask it to perform (C = A + B). In the previous section, we studied how to offload the actual computation kernel (matrix_add.hex), and now we will see the required steps for offloading a task to PIM. Each **PIMTask** is defined as a set of PIMData structures which are meant to be offloaded to PIM (See Figure19). It is the responsibility of the user application to create a task and add its data sets (the matrices) to this task. Here is an example of how this should be done for matrix addition:

```
PIMTask* task = new PIMTask("my matrix addition");
```

```
task->addData(A, (SIZE*SIZE)*sizeof(TYPE), 0);
```

```
task->addData(B, (SIZE*SIZE)*sizeof(TYPE), 1);
```

```
task->addData(C, (SIZE*SIZE)*sizeof(TYPE), 2);
```

User level data sets should be previously allocated in any section of the

user program (e.g. stack, heap, bss) using any memory allocation mechanism (**malloc** or **new**). However, for correct operation, they should obey from two rules:

First: The start address of the data structures must be aligned to the burst size of the system (cache-line-size). Otherwise, cache flush operation won't work correctly and obtained results may be incorrect. For example gcc's align attribute can be used for this purpose:

```
__attribute__((aligned(256))) int A[SIZE][SIZE];
```

Second: Current version of the API does not support overlaps between different ranges. Therefore, for any two data structures A and B it is important to ensure that the ranges [**start_address(A), end_address(A)**] and [**start_address(B), end_address(B)**] do not overlap. For this reason, multiple calls to the C++ **new** operator to build dynamic structures such as trees may violate this condition.

In order to avoid these problems, we have provided a simple memory allocation facility called **allocate_region()** which can be used by including the following header file:

```
./SW/HOST/app/app_utils.hh
```

Using this utility for the data structures to be shared with PIM has the following advantages:

- It ensures cache aligned memory allocation.
- It is guaranteed the different data sets will allocate non-overlapping regions.
- Since all required memory is allocated in the BSS section and during the load time of the user application, it is more probable that the underlying physical pages are contiguous. This results in larger memory **slices** and lower address translation overhead in PIM.
- Due to the overheads of page-locking and cache flushing it is important to ensure that the number of partially used memory pages is minimum. This mechanism guarantees this behaviour while in other allocation schemes, there is no such guarantee.

The **task->addData()** method creates a list of the memory regions sorted in ascending order of the starting address. The goal is to obtain virtual page numbers occupied by these regions, by a later call to PIMTask::updatePages(). The second argument to **task->addData()** is a number indicating the PIM's scalar register to hold the pointer to this data structure (described later). After adding all data sets to the task, user should offload this task by calling:

```
pim->offload_task(task);
```

In this method, at first task->updatePages() is called, and then using the **ioctl** system-call, pointer to the virtual pages is sent to the device-driver. We should remind here that, currently only one contiguous virtual region is supported, therefore, the use of the allocate_region utility is encouraged. The **ioctl** system-call inside the device-driver gets the virtual page pointers and calls the pim_get_user_pages() utility. This method creates a list of the pages using **kmalloc** and using the get_user_pages() system-call pins the user pages into the DRAM. After a call to this system-call, it is guaranteed that all required pages are present in the main memory, even if they have been previously swapped out the secondary storage. After a successful call to pim_get_user_pages(), the pim_create_slice_table() function is called to build the slice table as shown in Figure8. The slice-table itself is allocated using __get_free_pages() system-call provided by the Linux kernel, and its pointer and size are passed to PIM to be used by the dTLB:

```
pim_write_ulong_t(PIM_SLICETABLE, slicetable_paddr);
```

```
pim_write_ulong_t(PIM_SLICEVSTART, slicetable[0].vaddr);
```

Next, in order to make sure that non-stale data is available to PIM, the driver calls the pim_cache_flush() method. This method should call __cpuc_flush_dcache_area() and outer_clean_range() utilities to flush the L1 and L2 caches for the required memory ranges. However since cache flush instructions are not implemented in the current version of gem5-stable, we used a simulation hack to flush the caches instantly by communicating with gem5's simulation infrastructure. The cache-flush works correctly, however, it is done in zero-time for now.

For the final step, the virtual pointer to the user-level data structures (the matrices) will be written to PIM's scalar registers. This is automatically done by the API based on the second argument previously passed to the task->addData() method. Later, the PIM's kernel code can access these matrices by simply dereferencing these scalar registers:

./SW/PIM/kernels/matrix_add.c

```
A = (ulong_t*) PIM_SREG[0];
B = (ulong_t*) PIM_SREG[1];
C = (ulong_t*) PIM_SREG[2];
S = PIM_SREG[3];
```

After these steps, the task is ready to be executed on PIM. To do so, the user program can simply call the following functions:

```
pim->start_computation(PIMAPI::CMD_RUN_KERNEL);
pim->wait_for_completion();
```

There is one subtle point which we would like to mention here: In general, PIM's virtual address and user's virtual address are not necessarily identical

even though they are mapped to the same physical location. This can cause issues for pointer-based data structures such as trees and graphs. The source of this problem is that user's virtual address starts from 0x0, but PIM's virtual address starts from 0x00100000 in case the size of PIM's SPM is 1MB (please refer to Section 3.1.2 for explanation). To avoid this problem we can simply reserve the first 1MB of the user-virtual space and avoid sharing it with PIM. This way we will ensure that all user virtual addresses to be shared with PIM are above 0x00100000, and thus understood and translated correctly by PIM. Needless to say, 0x00100000 is just an example and in general the PIM_ADDRESS_SIZE defines this limit. The **allocate_region()** utility which we introduced before takes care of this issue automatically.

# 5 Calibration of the Simulation Environment

In this report we described the latest features added to the SMC simulation environment with the main focus of near-memory-computation. A calibration step is required, before this tool can be reliably used for measurements. Previously, we had shown that the obtained results from our high-level models correlate well with the cycle-accurate SMC model [Erfan Azarkhish, 2015]. Now, in the next subsection, we provide a zero-load estimate and calibration for our developed models.

## 5.1 Zero-load Latency Calibration

Based on the public data available on HMC ( [HMC, 2014], [Rosenfeld, ], [Jeddeloh and Keeth, 2012], and some recent measurements on a real Hybrid Memory Cube device [Gokhale, 2014], we have performed a zero-load estimation of both host and PIM accessing the HMC device. The results are summarized in Figure20. For the host processor, the access represents a cache refill for a cache line size of 256 Bytes, while for PIM the access is a single word fetch from the memory. From this figure it can be seen that the memory access latency of the host is almost 2.5X larger than the PIM.

We have calibrated our models in gem5 based on the numbers obtained from Figure20. gem5 does not models transactions on a packet basis rather than FLITS, and its components such as XBar only try to estimate the FLIT timings. Nevertheless, wormhole routing of packets is correctly modelled in gem5, as opposed to the traditional store-and-forward mechanism. We have considered the same rules in modelling the serial-links: they don't wait to receive the whole packet and serialize each FLIT as soon as they receive it. This is feasible in real world because if an error occurs they can invalidate the packet in its tail FLIT, and a retransmission can occur later (errors are not modelled in current release).

One last point to mention is that gem5's execution time and bandwidth match cycle-accurate (CA) simulation quite well. However, its reported MAT

HOST: Latency of a cache refill (after the L2 cache port) - Size = 256Bytes

| | | |
|---|---|---|
| Membus | 1Cycle@2GHz | (FlitSize=64b) |
| SMCController | 8Cycles@2GHz | (Pipeline Latency) |
| SERDES | 1.6ns | (SER=1.6, DES=1.6) |
| Packet Transfer | 13.6ns | (For 256Bytes packet +128b overhead, 16 Lanes @ 10Gbits/s) |
| PCB Trace Latency | 3.2ns + 3.2ns | (Round Trip) |
| SMCXBar | 1Cycle@1GHz | (FlitSize=256b) |
| VaultCtrl.frontend | 4Cycles@1.2GHz | |
| tRCD | 13.75ns | Activate |
| tCL | 13.75ns | Issue Read Command |
| tBURST | 25.6ns | (For 256Bytes packet) Burst |
| VaultCtrl.backend | 4Cycles@1.2GHz | |
| SERDES | 1.6ns | (SER=1.6, DES=1.6) |
| Packet Transfer | 13.6ns | (For 256Bytes packet +128b overhead, 16 Lanes @ 10Gbits/s) |
| SMCController | 1Cycles@2GHz | (Pipeline Latency) |
| Membus | 1Cycle@2GHz | (FlitSize=64b) |

Total Latency = 102.3ns

PIM: Latency of a 1-word access (No caches) - Size = 4Bytes

| | | |
|---|---|---|
| PimBus | 1Cycle@1GHz | (FlitSize=32b) |
| SMCXBar | 1Cycle@1GHz | (FlitSize=256b) |
| VaultCtrl.frontend | 4Cycles@1.2GHz | |
| tRCD | 13.75ns | |
| tCL | 13.75ns | |
| tBURST | 3.2ns | (1 Beat only) |
| VaultCtrl.backend | 4Cycles@1.2GHz | |
| PimBus | 1Cycle@1GHz | |

Total Latency = 39.1ns

Figure 20: A zero-load memory access estimate for both PIM and the host

does not correlate very well with CA simulation, partly due to the reason mentioned above. Also, XBars have zero latency to avoid deadlock in the coherence protocols. To conclude, the overall execution time in gem5 is reported with acceptable accuracy, yet, the reported MAT is less than expected, and should be dealt with care.

## 5.2 State of the Art

Figure21 presents a summary of the recent papers on near data processing along with their baseline configurations and reported results. From this figure, firstly, it can be obtained that trace based simulations cannot be trusted for system level evaluation (e.g. total execution time), even if traces are applied to a cycle-accurate model. This is why the results obtained in [Pugsley et al., 2014a] and [Zhu et al., 2013] do not seem realistic. Also, improvements achieved from a single HMC device seem to be less than 2X ( [Farmahini-Farahani et al., 2015], [Islam et al., 2014], [Zhang et al., 2014], and [Ahn et al., 2015b]). Further performance improvements (up to 5X) seem to be achievable by using location-

| Paper | PIM Model | Platform | PIM Location | Cach on PIM | Cache Coherence | Memory Management | Application | Energy Saving | Perf. Gain |
|---|---|---|---|---|---|---|---|---|---|
| [Farmahini-Farahani et al., 2015] | CGRA | gem5 | Logic Die | NO | Uncacheable | Segmentation with no paging | Big Data (MapReduce) | 46% | 1.6X |
| [Islam et al., 2014] | Cortex-A5 x 64 | gem5 | Logic Die | L1I, L1D, L2 | Software flush | Contiguous preallocated | Big Data (MapReduce) | 23% | 1.1X |
| [Pugsley et al., 2014a] | Cortex-A5 x 1000 | SIMICS (Trace-based) | Logic Die | L1I, L1D | - | - | Big Data (MapReduce) | 18X | 15X |
| [Stelle et al., 2014] | Light GP Cores x 16 | SST (Instruction Trace) | Logic Die | L1I, L1D, L2 | Yes - Hardware | Preallocation | Scientific: Sparse Linear Algebra, … | - | ~2X |
| [Sura et al., 2015] | Vector Processors | Mambo simulator | Logic Die | No | Yes - Hardware | Full Virtual Memory + Allocation | Dense Matrix Operations | - | Up to 5X |
| [Zhu et al., 2013] | App. Specific | Sniper (Trace-based) + CA | Logic Die | No | ? | Preallocation | Dense Matrix, FFT, BLAS | >100X | >100X |
| [Zhang et al., 2014] | PIM= CPU+GPU | Analytical | Logic Die | L1, L2 | Gathered Statistics | Preallocated | Graph, HPC, GPGPU | 85% | 7% |
| [Ahn et al., 2015b] | Low-level operations | In-house | Logic Die | No | Yes - Hardware | Full Virtual Memory | PageRank | 1.6X | 20% |
| [Ahn et al., 2015a] | In-order cores x 512 | In-house | Logic Die | L1 + 2 Prefetchers | Uncacheable | Segmentation | PageRank | 87% | 10X *** |

*** [9] uses a processor-centric network of 16 HMCs each one having 32 PIM cores

Figure 21: A comparison of the state of the art in near-memory-computation

aware memory allocation and preallocating vaults to the processors in their vicinity ( [Stelle et al., 2014] and [Sura et al., 2015]). This can be done by the aid from the device-driver the OS kernel. Also, looking at the baseline configuration of almost all these references reveals that, a latency hiding mechanism is mandatory, otherwise PIM cannot compete with powerful hosts with multiple levels of caches. This justifies our the design of our DMA engine. One last point to notice is the 10X performance improvement reported in [Ahn et al., 2015a]. This interesting paper suggests that the only way to achieve a performance proportional to the size of the memory is to connect several memory devices to each other (e.g. network of HMC devices), and place clusters of PIM processors inside them. This way, it should be possible to scale the performance achieved from the overall system to large values, however, we need to focus first on maximizing efficiency in a single cube configuration.

# 6 Benchmarking

The user-level applications developed to evaluate PIM's capabilities are accessible in:

`./SW/HOST/app/`

Each simulation scenario chooses one of these applications, along with a related computation kernel. Different types of applications have been developed

a) MATRIX ADD:

```
for ( ulong_t r=0; r< SIZE; r++ )
        for ( ulong_t c=0; c< SIZE; c++ )
                Z[r][c] = X[r][c] + Y[r][c];
}
```

b) MATRIX MULTIPLY:

```
for ( ulong_t r=0; r< SIZE; r++ )
        for ( ulong_t c=0; c< SIZE; c++ )
        {
                TYPE Sum = 0;
                for (ulong_t k=0;k<SIZE; k++)
                        Sum += X[r][k] * Y[k][c];
                Z[r][c] = Sum;
        }
```


c) TREE SEARCH:

```
for ( int i=0; i<NUM_OPERATIONS; i++ )
{
        node* curr = tree;
        while (curr)
        {
          if ( curr-> key == search_list[i] )
            break;
          else
          if ( curr-> key > search_list[i] )
            curr = curr->left;
          else
            curr = curr-> right;
        }
}
```

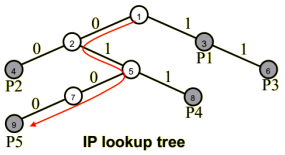


Figure 22: a) matrix addition, b) matrix multiplication, c) traversal of binary search tree

to evaluate PIM. We have also extracted the main computing kernels of the most famous benchmarks in the PIM community and implemented them both on host and PIM. In this section we present these benchmarks and explain how we have accelerated them on PIM.

## 6.1 Matrix Operations

Arithmetic operations on matrices are the first group of benchmarks which we studied. We have implemented golden models for matrix addition and multiplication as two representatives of these group, and they are accessible in:

`./SW/HOST/app/offload_matrix`

This application offloads the pointers of three matrices to PIM, and asks it to perform an operation on it. After PIM completes execution, the user application executes the golden model and compares their outputs to ensure correctness. Figure22.a,b illustrate simple implementations for matrix addition and multiplication. We used PIM's DMA engine in a ping-pong fashion to hide the DRAM access latency. The resulting kernel is shown in Figure23, and is available in:

`./SW/PIM/kernels/matrix_add.c`

```
ping = &PIM_VREG[0];
pong = &PIM_VREG[0] + XFER_SIZE*3;

DMA_REQUEST(A, A_ping, XFER_SIZE, PIM_DMA_READ, DMA_RES0 );
DMA_REQUEST(B, B_ping, XFER_SIZE, PIM_DMA_READ, DMA_RES1 );
while(PIM_DMA_STATUS & DMA_RES0);
while(PIM_DMA_STATUS & DMA_RES1);

num_bursts = SS*sizeof(ulong_t)/XFER_SIZE;
for (x=0; x<num_bursts; x++)
{
  // Fill Pong (Request)
  if ( x+1 < num_bursts ) // Boundary
  {
    DMA_REQUEST(A+(x+1)*XFER_SIZE, A_pong, XFER_SIZE, PIM_DMA_READ, DMA_RES2 );
    DMA_REQUEST(B+(x+1)*XFER_SIZE, B_pong, XFER_SIZE, PIM_DMA_READ, DMA_RES3 );
  }

  // Work on Ping's data
  for ( j=0; j< XFER_SIZE/sizeof(ulong_t); j++ )
    ((ulong_t*)C_ping)[j] = ((ulong_t*)A_ping)[j] + ((ulong_t*)B_ping)[j];

  // Write back the result of Ping
  while(PIM_DMA_STATUS & DMA_RES4);
  DMA_REQUEST(C+x*XFER_SIZE, C_ping, XFER_SIZE, PIM_DMA_WRITE, DMA_RES4 );

  // Wait for Pong to finish
  while(PIM_DMA_STATUS & DMA_RES2);
  while(PIM_DMA_STATUS & DMA_RES3);

  // Swap ping and pong
  swap = ping;
  ping = pong;
  pong = swap;
}
while(PIM_DMA_STATUS & DMA_RES4);
```

Figure 23: Matrix addition kernel for PIM using ping-pong DMA mechanism

Due to the simplicity of matrix addition, implementation of ping-pong buffering was not difficult and we could easily hide the DRAM access latency. By scaling the clock frequency of PIM we obtained that, addition of a vector co-processor to PIM to improve its IPC can easily enhance its performance. We are currently comparing energy consumption, and also looking at applications with random memory access patterns.

## 6.2 Tree Search

The second group of data structures we looked at are Binary Search Trees (BST). BST has been implemented using pointer-based structures and the golden model for the search operation is shown in Figure22.c available in:

`./SW/HOST/app/offload_tree`

Unlike matrix operations which are bandwidth-sensitive and can easily benefit from DMA, tree based structures are more latency-sensitive due to their low computational complexity and irregular access memory access patterns,

and DMA cannot be used for them easily. We are currently working on these groups, and we will present our results shortly.

## 6.3 Graph Traversal

Graph traversal benchmarks have been used for PIM evaluation in recent publications (including [Ahn et al., 2015a] and [Ahn et al., 2015b]). But, graph algorithms highly rely on the underlying implementation of the graph itself, and their behaviours highly depends on the topology of the graph (i.e. sparse/dense, number of nodes, etc). Graphs can be stored in different ways, but among them, adjacency matrix is more suitable for storing dense graphs where the number of edges is comparable to the number of nodes, and the number of nodes is not large (<10000).

Sparse graphs, on the other hand, are better represented by one of the compression techniques introduced in [Wikipedia, ]. Otherwise, the adjacency matrix will explode. To give an example, a typical social graphs contains 100000 to millions of nodes with maximum degree of less than 100 outgoing from each node. Compressed Sparse Row (CSR) implementation has been used in [Ahn et al., 2015b] and provides one of the highest compression factors, but it is not easy to parallelize among several processing nodes. List of Lists (LIL), is another famous implementation method used in [Ahn et al., 2015a] and [Salihoglu and Widom, 2013] with slightly higher storage requirement in comparsion with CSR, but well suited for extension to parallel versions of PIM. In our models, we have implemented both Adjacency Matrix (for dense graphs) and the LIL method (for sparse graphs), however, for the social graph algorithms presented in this section, we focus on the LIL implementation. The user applications for the dense and sparse graph representations are available in: `./SW/HOST/app/offload_dgraph`

`./SW/HOST/app/offload_sgraph`

The LIL implementation is available in defs.hh and utils.hh of the **sgraph** user application.

### 6.3.1 Average Teenage Follower (ATF)

Average Teenage Follower is an example kernel of social network analysis. It counts for each vertex the number of its teenage followers by iterating over all teenager vertices and incrementing the "follower" counters of their successor vertices (See Figure24.a). This generates a very large amount of random memory accesses over the entire graph. We accelerated this kernel on PIM based on two observations: First, due to the sparsity of social graphs, it does not seem very beneficial to initiate DMA transactions for the successors list of each node in ping-pong mode. For this reason we use multiple DMA resources to fetch the successors list for multiple nodes (only the teenager ones) and then iterate over all of them to increment the **followers** counters (See Figure25). Second,

a) Average Teenage Follower (ATF)

```
for ( ulong_t r=0; r<NODES; r++ )
    {
      if ( nodes[r].teenager )
        for ( ulong_t c=0; c<nodes[r].out_degree ; c++ )
        {
           node*succ = nodes[r].successors[c];
           succ->followers++;
        }
    }
```

b) Pagerank

```
for ( ulong_t i=0; i<NODES; i++ )
    {
      nodes[i].page_rank = 1.0 / NODES;
      nodes[i].next_rank = 0.15 / NODES;
    }
    ulong_t count = 0;
    float diff = 0.0;
    do {
      for ( ulong_t i=0; i<NODES; i++ )
      {
        float delta = 0.85 * nodes[i].page_rank / nodes[i].out_degree;
        for ( ulong_t j=0; j<nodes[i].out_degree; j++ ) // for node.successors
            nodes[i].successors[j]->next_rank += delta;
      }
      diff = 0.0;
      for ( ulong_t i=0; i<NODES; i++ )
      {
        diff += fabsf(nodes[i].next_rank - nodes[i].page_rank);
        nodes[i].page_rank = nodes[i].next_rank;
        nodes[i].next_rank = 0.15 / NODES;
      }
    } while (++count < PAGERANK_MAX_ITERATIONS && diff > PAGERANK_MAX_ERROR);
```

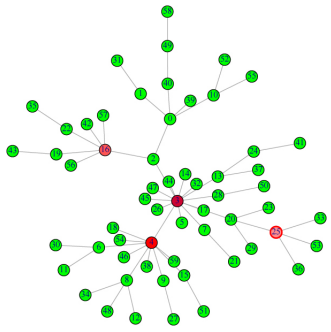

Figure 24: a) Average Teenage Follower, b) Google's Pagerank

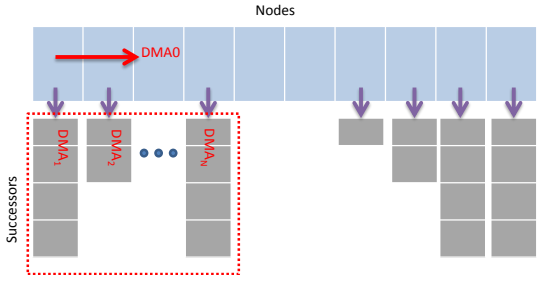


Figure 25: Traversing the List of Lists (LIL) using multiple DMA resources

incrementing the **followers** counters looks like a perfect example for use of the atomic HMC commands (discussed in Section 3.2.2). Applying these two observations to the original kernel code results in the code illustrated in Figure26. The performance achieved by this code was modestly more than the host, and currently, we are doing more experiments, and also trying to measure the energy efficiency achieved by PIM's execution.

```
DMA_REQUEST(nodes, nodes_pong, nodes_chunk, PIM_DMA_READ, DMA_RES0 );
total_followers = 0;
rr = nodes_count;
d = 0;
for ( r=0; r<NODES; r++ )
{
  if ( rr == nodes_count )
  {
    rr = 0;
    while(PIM_DMA_STATUS & DMA_RES0);
    nodes_swap = nodes_ping; nodes_ping = nodes_pong; nodes_pong = nodes_swap; // Swap Ping and Pong
    DMA_REQUEST(&nodes[r+nodes_count], nodes_pong, nodes_chunk, PIM_DMA_READ, DMA_RES0 );
  }
  if ( nodes_ping[rr].teenager && nodes_ping[rr].out_degree )
  {
    fetch_count[d] = nodes_ping[rr].out_degree;
    DMA_REQUEST(nodes_ping[rr].successors, succ_list[d], nodes_ping[rr].out_degree*sizeof(node*),
      PIM_DMA_READ, DMA_RES[d+1] );
    d++;
  }
  if ( d >= MAX_DMA_RESOURCES_TO_USE ) // We use N DMA resources
  {
    for ( dd = 0; dd < d; dd++ )
    {
      while(PIM_DMA_STATUS & DMA_RES[dd+1]);
      for ( c=0; c< fetch_count[dd]; c++ )
      {
        node* succ = succ_list[dd][c];
        #ifdef USE_HMC_ATOMIC_CMD
        HMC_ATOMIC_INC(succ->followers);
        #else
        succ->followers++;
        #endif
      }
    }
    d = 0;
  }
  rr++;
}
if ( d != 0 ) /* If any pending data is remaining in the DMA buffers */
{
  for ( dd = 0; dd < d; dd++ )
  {
    while(PIM_DMA_STATUS & DMA_RES[dd+1]);
    for ( c=0; c< fetch_count[dd]; c++ )
    {
      node*succ = succ_list[dd][c];
      #ifdef USE_HMC_ATOMIC_CMD
      HMC_ATOMIC_INC(succ->followers);
      #else
      succ->followers++;
      #endif
    }
  }
  d = 0;
}
```

Figure 26: Average Teenage Follower accelerated by DMA and HMC's atomic increment

### 6.3.2 Google's Pagerank

Pagerank algorithm is widely used for network benchmarking and the PIM community, as well. As shown in Figure24.b, this algorithm consists of two

main loops and requires floating point arithmetic. The first loop can be easily optimized similar to the ATF benchmark using DMAs and an atomic HMC command. For this reason we implemented an Atomic Floating-point Addition command, which asks HMC to perform floating point addition on a memory location and an immediate value. The hardware costs associated with this operation seem reasonable as this command does not require communication with any other banks or vaults, and floating-point addition can be implemented simply, as well. For the second loop however, we implemented a triple-buffering mechanism using the flexible DMA engine available in PIM. While one buffer is being filled from memory, and another one be being processed, a third buffer is being written back with the previously generated results. The resulting code is available in:

`./SW/PIM/kernels/sgraph_page_rank.c`

Further optimizations on this benchmark and performance and energy impacts are under investigation.

### 6.3.3 Breadth First Search (BFS)

Breadth-first-search is a graph traversal algorithm, which visits vertices closer to a given source vertex first. A reference implementation is presented in Figure27.a. This kernel can also benefit from both atomic commands and DMA techniques, and its kernel code is available in:

`./SW/PIM/kernels/sgraph_bfs.c`

### 6.3.4 Bellman Ford's Shortest Path

The last benchmark currently implemented is the Bellman-ford's shorted path algorithm (See Figure27.b). This algorithm requires weighted graphs, therefore, we have added a **weights** list to the sparse implementation provided in:

`./SW/HOST/app/offload_sgraph/utils.hh`

We are currently measuring the excution performance and energy of these benchmarks and comparing it with the host.

# 7 Observations

This section provides a list of observations made during the first year of this project:

**Phase 1: Cycle-accurate trace-based simulation:**

- Single-cycle uniform logarithmic interconnect can easily meet the required

```
a)  Breadth First Search (BFS)
ulong_t total_distance = 0;
    while ( !(queue_empty) )
    {
      ulong_t v = queue_top;
      queue_pop;
      for ( ulong_t c=0; c<nodes[v].out_degree ; c++ )
      {
        node*succ = nodes[v].successors[c];
        if (succ->distance == NC) // Infinite
        {
          succ->distance = nodes[v].distance + 1;
          total_distance += succ->distance;
          queue_push(succ->ID);
        }
      }
    }
b) Bellman-Ford
ulong_t total_distance = 0;
    for ( unsigned r=0; r<NODES; r++ )
      for ( ulong_t c=0; c<nodes[r].out_degree; c++ ) // for node.successors
      {
        node*u = &nodes[r];
        node*v = nodes[r].successors[c];
        ulong_t w = nodes[r].weights[c];
        if ( u->distance != NC && v->distance > u->distance + w )
          v->distance = u->distance + w;
      }
```

Figure 27: a) Breadth-first Search, b) Bellman-ford's all pairs shortest path

bandwidth and latency demands of the current HMC model, and can easily scale to the future projections of HMC.

- To achieve higher bandwidth per link, it is more suitable to increase the flit-size, rather than the clock frequency. This is because any improvement in clock frequency requires more synthesis effort and results in higher power consumption and temperatures increase in the Logic Base.

- Address mapping was found to have a very important effect on the delivered bandwidth and total execution time. Therefore, a configurable and programmable addressing scheme at the interconnect ports should be employed.

- Unlike existing DDR memories, HMC utilizes Closed-Page policy and its DRAM devices have been redesigned to have shorter rows (256 Bytes matching the maximum burst size of serial links, rather than 8-16KB in a typical DDR3 device) [Jeddeloh and Keeth, 2012], focusing on workloads which typically exhibit little or no locality. The reduced row length helps save power by alleviating the over-fetch problem, however, reduces the row buffer hit probability, which makes open page mode impractical. In addition, open page policy exhibits additional overheads for little locality workloads, due to delaying the precharge between accesses to different rows [Rosenfeld, ] [Pugsley et al., 2014b]. Open page row buffer models also impose a logic cost as the scheduling hardware is typically more com-

plex [Rosenfeld, ]. As a result, with the large number of banks in HMC, it is more efficient to utilize memory-level parallelism to achieve high performance rather than relying on locality which may or may not be present in a given memory access stream.

- It is safe and suitable in terms of traffic interference to attach a PIM device to the main interconnect, allowing for global visibility to the whole address space. This PIM device can access the main memory periodically (e.g. using a double buffering DMA engine), carry-out computations, and store back the results.

**Phase 2: Event-driven full-system simulation:**

- Trace-based simulation results cannot be trusted for system-level evaluation (e.g. total execution time), even if the traces are applied to highly accurate models. On the other hand, high-level simulation models need to be calibrated before reasoning about their results. In this project, we developed a cycle-accurate model to reason about local measurements such as bandwidth and latency. Then we moved to a higher level model for system-level performance analysis in presence of all dynamic effects in the system. We used the cycle-accurate model to verify the correctness of the high-level gem5 model, and used the available data from the literature to calibrate these models. At this point, the performance results obtained from the SMC Simulator can be trusted to a great extent.

- Smart memory allocation based on vicinity and availability, and keeping an eye on the underlying organization of data in DRAM rows can lead to significant improvements and may results in a total of 5X performance improvement in comparison with placement and data structure oblivious algorithms implemented on the host.

- We observed that the memory access latency observed by PIM is not small enough in comparsion with the host, and therefore, latency hiding techniques are necessary. Even so, employing caches inside the PIM architecture does not seem necessary, and multi-channel DMA engines with aid from software can lead to more optimized implementations at lower costs.

- Using HMC atomic operations for moving small parts of computation directly to the memory storage can be very beneficial specially for energy saving. This is another direction which we would like to work on and perform extended explorations.

- From the application point of view, we observed different requirements in different benchmark applications: some are bandwidth sensitive while others are more sensitive to memory access latency. Some require large burst transfers while others exhibit random word-based memory access

patterns. The computational complexity also varies across different benchmarks. We are trying to optimize these benchmarks taking advantage of the different features provided in PIM: the vector coprocessor, the support for atomic HMC commands, and the DMA engine. We will present our results shortly.

- We believe that adding more intelligence to the vault controllers in terms of transaction scheduling or some computation capabilities can be helpful, as well. In general, having a hierarchy of PIM processors starting from the Logic-based to the vault controllers and then to the DRAM dies can provide more flexibility in solving different kinds of problems and allow for further energy reduction and performance gains.

# 8 Conclusions and Future Directions

In the first year of this project we focused on design of a low-power and energy efficient PIM to be used within a single HMC device. We targeted computations with low arithmetic intensity (e.g. graph traversal and pointer chasing). We provided zero-copy virtual memory support for our PIM, as well as, low-overhead kernel and task offloading. We augmented our PIM processor with a multi-channel DMA engine supporting zero-copy virtual memory, and implemented limited computation capabilities in the DRAM dies of the vaults. Our estimates showed modest performance improvements, but better energy savings. The main focus of the proposed architecture was latency and energy reduction, even for applications which are not able to utilize the huge bandwidth provided by the Hybrid Memory Cube. As a short term plan, we will extend our experiments, obtain more performance numbers, and estimate energy efficiency for current benchmarks, and would like to publish a paper in DATE 2015 conference with the obtained results.

For the next year of this project, we propose to look more in depth on pushing computation towards the memory and how to orchestrate PIM with vault-level or DRAM-level computation and manage the amount of required resources (i.e. vector processing blocks) to be put in the PIM. We believe that having a hierarchy of PIM processors starting from the Logic-based to the vault controllers and then to the DRAM dies can provide more flexibility in solving different kinds of problems and allow for further energy reduction and performance gains. Furthermore, designing a cluster of parallel PIM processors can lead to even more interesting results and requires dealing with new challenges such as more advanced virtual memory management, design of MMU for the cluster, and DMA engines to support transaction coalescing. In our opinion, it is important design an optimized SMC device before moving towards a network of multiple SMCs.

# References


[HMC, 2014] (2014). Hybrid memory cube specification 1.1.

[Ahn et al., 2015a] Ahn, J., Hong, S., Yoo, S., Mutlu, O., and Choi, K. (2015a). A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 105–117, New York, NY, USA. ACM.

[Ahn et al., 2015b] Ahn, J., Yoo, S., Mutlu, O., and Choi, K. (2015b). Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 336–348, New York, NY, USA. ACM.

[Erfan Azarkhish, 2014] Erfan Azarkhish, Davide Rossi, I. L. L. B. (2014). A cycle accurate simulation environment for the smart memory cube. Technical Report 1, ETHZ University.

[Erfan Azarkhish, 2015] Erfan Azarkhish, Davide Rossi, I. L. L. B. (2015). A full-system simulation environment for the smart memory cube. Technical Report 2, ETHZ University.

[Farmahini-Farahani et al., 2015] Farmahini-Farahani, A., Ahn, J. H., Morrow, K., and Kim, N. S. (2015). Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 283–295.

[Gokhale, 2014] Gokhale, M. (2014). Near data processing: Are we there yet?

[Islam et al., 2014] Islam, M., Scrbak, M., Kavi, K., Ignatowski, M., and Jayasena, N. (2014). Improving node-level mapreduce performance using processing-in-memory technologies. In Lopes, L., Zilinskas, J., Costan, A., Cascella, R., Kecskemeti, G., Jeannot, E., Cannataro, M., Ricci, L., Benkner, S., Petit, S., Scarano, V., Gracia, J., Hunold, S., Scott, S., Lankes, S., Lengauer, C., Carretero, J., Breitbart, J., and Alexander, M., editors, *Euro-Par 2014: Parallel Processing Workshops*, volume 8806 of *Lecture Notes in Computer Science*, pages 425–437. Springer International Publishing.

[Jeddeloh and Keeth, 2012] Jeddeloh, J. and Keeth, B. (2012). Hybrid memory cube new DRAM architecture increases density and performance. In *VLSI Technology (VLSIT), 2012 Symposium on*, pages 87–88.

[MALI, ] MALI. Open source mali-400/450 gpu kernel device drivers.

[Pugsley et al., 2014a] Pugsley, S., Jestes, J., Zhang, H., Balasubramonian, R., Srinivasan, V., Buyuktosunoglu, A., Davis, A., and Li, F. (2014a). Ndc: Analyzing the impact of 3d-stacked memory+logic devices on mapreduce work-

loads. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pages 190–200.

[Pugsley et al., 2014b] Pugsley, S. H., Jestes, J., Balasubramonian, R., et al. (2014b). Comparing implementations of near-data computing with in-memory mapreduce workloads. *Micro, IEEE*, 34(4):44–52.

[Rosenfeld, ] Rosenfeld, P. *Performance Exploration of the Hybrid Memory Cube*. PhD thesis. Univ. of Maryland, 2014.

[Salihoglu and Widom, 2013] Salihoglu, S. and Widom, J. (2013). Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pages 22:1–22:12, New York, NY, USA. ACM.

[Stelle et al., 2014] Stelle, G., Olivier, S. L., Stark, D., Rodrigues, A. F., and Hemmert, K. S. (2014). Using a complementary emulation-simulation co-design approach to assess application readiness for processing-in-memory systems. In *Proceedings of the 1st International Workshop on Hardware-Software Co-Design for High Performance Computing*, Co-HPC '14, pages 64–71, Piscataway, NJ, USA. IEEE Press.

[Sura et al., 2015] Sura, Z., Jacob, A., Chen, T., Rosenburg, B., Sallenave, O., Bertolli, C., Antao, S., Brunheroto, J., Park, Y., O'Brien, K., and Nair, R. (2015). Data access optimization in a processing-in-memory system. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, pages 6:1–6:8, New York, NY, USA. ACM.

[Wikipedia, ] Wikipedia. Sparse matrix.

[Zhang et al., 2014] Zhang, D., Jayasena, N., Lyashevsky, A., et al. (2014). TOP-PIM: Throughput-oriented programmable processing in memory. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 85–98, New York, NY, USA. ACM.

[Zhu et al., 2013] Zhu, Q., Akin, B., Sumbul, H., et al. (2013). A 3D-stacked logic-in-memory accelerator for application-specific data intensive computing. In *3D Systems Integration Conference (3DIC), 2013 IEEE International*, pages 1–7.