Memory Processing Units

By

Thiruvengadam Vijayaraghavan

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

2017

Date of final oral examination: 07/17/2017

To my parents:
For their unconditional love and support

Acknowledgments

I started graduate school in Fall 2011. It's been 6 years now. Yet, I still remember my first semester vividly. Time flew by.

I have learnt and grown a lot over these years, both personally and professionally. I attribute this learning, first and foremost, to my own perseverance and the desire to keep growing and improving in life. Several people and experiences in life have helped shape this attitude and I am thankful to all of them. Professionally, I am very thankful to my advisor, Professor Karthikeyan Sankaralingam, whose hands-on approach to advising has helped instill several important lessons and taught me how to do research. Personally, I am thankful to my parents without whose support I could possibly never have reached where I am today. I am thankful to my brother Venkatesh who has always been a constant source of inspiration, my sister Shilpa, my friends Tony, Newsha, Siddharth, Nayana, Arul, Kalyani, Vineet, Dev and Vaidy for their tremendous understanding and moral support through difficult phases during this journey.

I have learnt a lot from all of my colleagues on the CS 6th floor, especially from Tony Nowatzki (who was my officemate throughout my PhD), Newsha Ardalani, Vinay Gangadhar, Jason Lowe-Power, Gagan Gupta, Hongil Yoon, Emily Blem, Venkataraman Govindarajan, Raghuraman Balasubramanium and Jaikrishnan Menon. I am thankful for all the insight discussions during the weekly architecture reading group, which I believe is one of the most important learning tools outside of conventional classes.

Finally, I want to thank my better half, my wife, Anu Narayanan for her support during the final leg of my PhD. Though we have only known each other for less than an year now, most of which was long distance, her presence in my life made all my challenges easier to handle.

Table of Contents

List of Tables

List of Figures

Abstract

In this work, we observe that 3D die-stacking of logic and DRAM provides a unique opportunity to revisit the ideas of processing-in-memory. Compared to conventional DRAMs, 3D die-stacked DRAM (embodied by standards like HMC and HBM), have almost an order of magnitude improvement in bandwidth between logic and memory, significant power reductions, and modest latency reduction as well. We make the case for Memory Processing Units, a co-processor architecture, which build on logic+DRAM stacking technology embodying three principles: high performance through massive concurrency, low-energy computation using cores that are efficient at idling, and generality through a flexible programming model of remote procedure calls. On evaluation across diverse domains spanning text analytics, end-to-end database analytics, and graph analytics, MPU provides integer factors better performance and energy savings compared to conventional CPUs, and matches specialized PIM architectures. Our work demonstrates the following qualitative architectural insight: a brute-force approach consisting of a large array of low-power cores suffices to achieve the goals of accelerating computation while reducing energy for many workload domains, without having to sacrifice generality.

1 | Introduction

Processing-in-memory (PIM), pioneered in the early 90s to address the memory-wall [1, 2, 3, 4, 5, 6, 7, 8, 9], is a computational paradigm which consists of placing computation physically close to memory. Then hampered by technology limitations, PIM is now being revisited in various novel ways due to the recent emergence of cost-effective 3D stacking of memory and logic. However, most current approaches suffer from narrow specialization, either to specific applications [10, 11, 12, 13] or specific workload characteristics [14, 15]. The solutions that do not incur this pitfall require disruptive changes to the main processor architecture, and provide limited benefits both in terms of performance and energy savings [16]. The question of whether it is possible to build a general PIM architecture, that can bring integer-factors energy and performance benefits while supporting a wide variety of workloads, has thus remained open.

In this work, we answer this question in the affirmative by proposing the MPU (Memory Processing Unit) processing-in-memory architecture. Our architecture applies to a wide range of practically relevant workloads, leverages an easy-to-use, general programming model, require no processor modifications and provides significant performance and energy benefits.

The fundamental challenge for PIM systems is workload heterogeneity: algorithms from different domains present a variety of memory layouts and access patterns, and involve computations with different degrees of parallelism and complexity. Even within a single workload, regions that are essentially sequential—which can most efficiently be executed on a conventional CPU—may alternate with regions with abundant thread-level parallelism and limited required synchronization, which are ideal targets for PIM engines. Because of this variety, traditional wisdom is that PIM systems can only achieve performance and energy improvements by specializing the hardware to specific classes of workloads (see chapter 3). However, in this work we show that it is indeed possible to realize a general PIM architecture by adhering to the following principles:

1. *Performance through massive concurrency:* the MPU architecture includes large arrays of cores allowing tens of concurrent threads
2. *Energy savings through low-energy computation:* the MPU architecture uses low-power cores, that remain efficient when idling (unused or waiting for memory I/O)
3. *Generality through flexible programming model:* the MPU uses a general approach to PIM offload, based on low-overhead remote procedure calls. Also, the MPU shares a unified address space with the CPU.

The MPU design is built around these principles, and consists of a series of extensions to the Micron HMC 3D-stacking memory. Concurrency (principle 1) is provided by augmenting each HMC cube with 128 general-purpose cores (Xtensa LX3 in our design). The cores are lightweight, consisting of a simple 3-stage pipeline, and run at only 500MHz. Therefore, they compute and idle efficiently (principle 2). The MPU programming model aims at generality and is based on memory procedure calls (MPC): arbitrary programmer-defined function, which the application posts to the MPU using an asynchronous queue. This mechanism is generic (principle 3) and enables the host CPU to quickly offload a batch of MPCs to the MPU and then wait for results. Internally, non-blocking offload is implemented by a dedicated MPU controller, sitting on the HMC logic layer.

In terms of data layout, our programming model abstracts the organization of HMC memory into 256-MB DRAM banks (vaults) by exposing memory as a set of 256-MB pages (to simplify interactions, CPU and MPU share the same address space). The 128 cores in each HMC cube are also partitioned into 16 tiles with 8 cores each; each tile is directly connected to one vault. Each core

Context	Performance		Energy	
	1-MPU	**4-MPU**	**1-MPU**	**4-MPU**
Database Analytics	1.2×	1.1×	3.1×	1.3×
Graph Analytics	5.9×	15.3×	14.4×	14.5×
Text Analytics	2.1×	6.9×	2.7×	3.8×

Table 1.1: MPU improvements over 4-core WSM + DDR3

can efficiently access memory within its home vault; accessing other vaults is transparent to the programmer, but incurs an energy and latency cost.

Achieving efficiency within this architecture requires that the workloads are memory intensive (i.e. they present a high memory access to arithmetic instruction ratio) and can be sharded, i.e. partitioned into many independent tasks that access separate memory regions. In addition, MPU particularly shines for workloads exhibiting high concurrency and irregular memory access pattern. Our evaluation shows that workloads across many domains incorporate large regions meeting many of these properties. It should be noted that the hardware-managed offloading and the unified virtual memory space make it easy for the programmer to alternate MPU and CPU phases in a program: program regions that do not present enough parallelism for the MPU can be simply executed on the host processor.

Using detailed evaluation comparing our PIM architecture both with specific, previously proposed PIM architectures and a conventional architecture on a variety of workloads, we demonstrate significant performance and energy gains. The results in Table 1.1 show geomean performance and energy gains of MPU systems with 1 and 4 HMC cubes over a 4-core OoO Skylake system.

1.1 Dissertation Contribution

PIM Design Principles We present a set of principles for the design of PIM architectures. We note that the individual conceptual components of our approach—arrays of low-frequency, low-power cores, concurrency through massive parallelism, offloading via remote procedural calls—have been individually proposed in the past. *The novelty of our work is NOT in any novel mechanisms over* *existing state-of-art, but rather, the novelty lies in showing that a composition of such elements suffices to achieve general and efficient in-memory processing*.

Hardware Architecture Design We present a detailed hardware architecture built around mechanisms that implement the principles we have identified.

Programming Model Design We present an intuitive, easy-to-use programming model to enable programmers to port programs to MPU and offload computation from CPU to MPU. This programming model is built around a remote procedure call like mechanism, that we call *memory procedure calls*.

System Level Design To enable non-intrusive integration of MPU to conventional processors, we present a system architecture design that does not require any hardware modifications or operating system modifications on the host processor to which MPU is integrated.

Experimental Infrastructure Development - Emulator, Workloads, Simulator We ported several workloads (18 total) to the MPU programming model, built a multi-threaded emulator to enable functional testing of these ported workloads on a conventional processor and built a performance simulator and power model to quantitatively evaluate the potential of our architecture.

Quantitative Analysis of MPU Speedup and Energy Reduction over CPU We quantitatively evaluate several kernels and several workloads from three different domains - database analytics, graph analytics and text analytics. With analysis of kernels, we derive the main sources of performance benefits, based on the workload behavior and micro architecture, while keeping relatively complex phenomenon like load balancing and concurrency out of the picture. With analysis of workloads from graph and database domains, we evaluate impact of these two phenomenon also. In addition, evaluation across these three important domains helps us gauge the potential of the MPU architecture for practical, real workloads.

Quantitative Comparison against Related Work We quantitatively compare MPU against two recent works that cover different ends of the performance-generality spectrum.

1.2 Dissertation Organization

The dissertation is organized as follows:

Chapter 2 present an overview of MPU, Chapter 3 discusses related work, Chapter 4 describes the MPU programming model, Chapter 5 describes the MPU architecture, Chapter 6 describes the system level design, Chapter 7 details our experimental infrastructure and workloads, Chapter 8 compares MPU to a CPU-based (Skylake) architecture, Chapter 9 compares MPU to two other PIM architectures.

2 | MPU Overview

As we base our work around Micron's Hybrid Memory Cube product line, we first describe the HMC (v1.0) architecture, summarized in Figure 2.1. One HMC chip provides 4GB of storage comprising 8 stacked DRAM dies, one logic die, and SERDES links for I/O. The memory is organized into 16 partitions or vaults. Each vault is a vertical column of DRAM dies, with an associated vault-controller in the logic layer which handles DRAM command sequencing. Within each die, two banks belong to a vault, for a total of 16 banks per vault. There are 32 data TSVs per vault that connect the logic die to the DRAM dies and some number of additional command TSVs. Abstracting away implementation details, HMC provides two capabilities - **Extremely high-bandwidth** communication between logic and memory, enabled by the 32 TSVs within each vault, and **Low energy communication** enabled by the physical proximity of logic to memory, avoiding the need for on-chip and off-chip wire energy and multilevel caches. In particular, we observe that the effective way to utilize this bandwidth and low access energy is to logically/physically couple cores to DRAM banks.

MPU modification to the HMC design reflect the goals of achieving concurrency, low energy

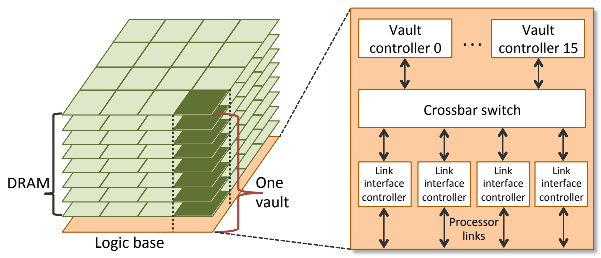
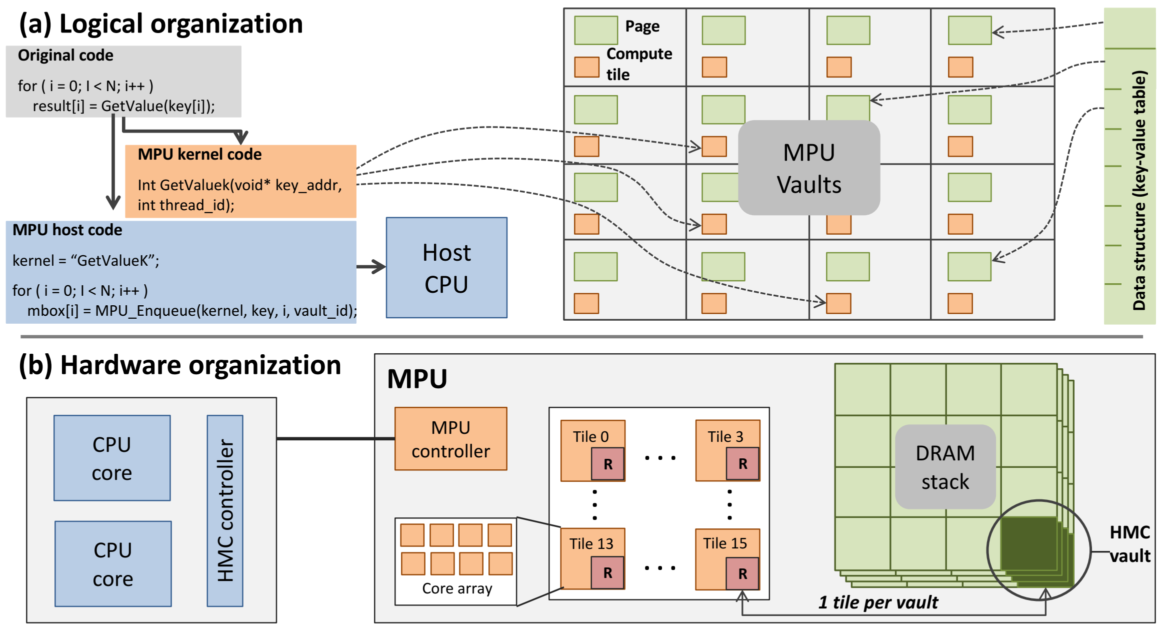

DRAM
Logic base
One vault
Vault controller 0
...
Vault controller 15
Crossbar switch
Link interface controller
Link interface controller
Link interface controller
Link interface controller
Processor links

Figure 2.1: Micron Hybrid Memory Cube



```
for ( i = 0; I < N; i++ )
    result[i] = GetValue(key[i]);
```

```
Int GetValuek(void* key_addr,
int thread_id);
```

```
kernel = "GetValueK";

for ( i = 0; I < N; i++ )
    mbox[i] = MPU_Enqueue(kernel, key, i, vault_id);
```

Figure 2.2: MPU hardware and programming model organization

consumption and generality on PIM workload, while ensuring the design remains practical to deploy on today's architecture.

Principle 1: Concurrency. MPU augments each HMC vault with an array of 8 lightweight processing cores and some coordination logic. As there are 16 vaults in the standard HMC design, the MPU is a 128-core architecture. The location of cores in the logic layer of each vault is depicted in Figure 2.2(b). The processing cores can access vault-local memory quickly and remote vaults slowly and can execute arbitrary computations on behalf of the host CPU. Each core has a dedicated 16KB instruction and data-cache. In total, MPU die area is about 15% for the cores and 85% for SRAMs and totals 33mm^2 at 45nm process.

Principle 2: Low-energy computation. The large number of cores in the MPU's computational substrate enables the architecture to take advantage of the abundant parallelism available in PIM-oriented workloads while letting cores run at low frequency (500MHz), thus reducing their dynamic energy. Per-core caches act as a further *energy filter*, with the additional benefit of reducing latency. Furthermore, our cores of choice (Tensilica Xtensa-LX3) exhibit extremely low static power (1.4 mW per core).

The advantage of this approach is the simplicity of the design: provisioning a large number of cores while allowing a fraction of them to be idling at any given point in time enables us to do away with workload-specific schedulers, prefetchers and in general any form of architectural specialization aimed at optimizing core usage. Contrary to popular belief, this brute-force approach to parallel computation has little impact on performance and energy compared to specialized architectures. In other words, *our work shows that the cost of achieving generality with PIM architecture is small, and can be achieved by following simple principles: high parallelism and lightweight computational elements*.

Principle 3: Generality. The programming model makes only one assumption about workloads: that computation can be partitioned in independent tasks, each mostly accessing only a specific partition of the application dataset.

Figure 2.2(a) represents this model, including organization of the computation and data layout, through a simple key-value store application (Figure 2.2(b) illustrates how each conceptual component maps to the hardware). The MPU paradigm splits applications ("Original code" in the figure) in two parts. One part, running on the host ("MPU host code"), loads data, performs initialization, initiates computation, and retrieves results. The other part consists of one or more programmer-defined functions, called *MPU kernels* which run on the MPU and execute the memory-intensive part of the workload.

Conceptually, memory within each HMC cube is partitioned into sixteen segments/vaults. All computation is associated to a "home" vault, but can access data from any vault (access to memory outside the home vault incurs a latency penalty). This is reflected in the hardware organization by the fact that each HMC vault is associated to one of 16 computing tiles. The programmer is tasked with partitioning the dataset across vaults and associate computation with each vault. The example application statically distributes the key-value store across all vaults, and partitions the workload among them.

Communication between the host and the MPU-side uses a mechanism of remote procedure calls, that allow the host CPU to trigger computation on the MPU, and retrieve results. These calls are triggered by using a host-side MPU API that internally sends commands to the MPU controller. The controller examines the address targeted by each request, from which it can infer the vault and route it there to trigger execution of MPU kernels on the vault's embedded cores.

3 | Related Work

3.1 Comparison to works published prior to emergence of cost effective 3D die stacking

Early PIM studies include several works [1, 2, 3, 4, 5, 6, 7, 8, 9, 17], but were hampered by technological limitations. Below, we discuss and highlight this dissertation's contribution over these prior works (mostly from the 1990's).

The IRAM article [1] was one of the foundational publications in the PIM space. The authors bring into question the practice of fabricating memory and logic as two separate chips and lay out all the problems that arise due to this practice. They propose fabricating memory and logic into one chip using DRAM fabrication technology, explain why that might be a good solution and the challenges involved in doing so. Finally, they propose that a Vector processor [2] is more suitable to be integrated with memory compared to conventional OoO processors. In the MPU work, we chose not to limit the architecture to single instruction multiple data (SIMD) style programs with regular memory access pattern (best suited for Vector IRAM proposed in [2] and Computational RAM proposed in [3]).

Of all the related work in this era, FlexRAM [4, 5, 6] is the most closely related. Though FlexRAM is designed with similar principles as MPU, the specific architecture and programming model are different. They have a single in-order management core placed near memory, offloading computation to several *P-Arrays* connected to DRAM cells. These *P-arrays* are RISC pipelined computing units that share resources with other P-Arrays. FlexRAM supports page tables and TLBs for virtual address translation. Finally, the design adds some hardware logic to achieve integration of memory and logic as 3D stacking was still an unsolved problem at the time. HMC possibly solves these issues with lower complexity. We cannot make direct quantitative comparison between MPU and FlexRAM due to different 3D stacking technologies. In terms of programming model, FlexRAM uses an OpenMP-style model called *CFlex*. Unlike OpenMP which is suited to express loop parallel code, *CFlex* is more flexible as it can express MIMD style execution and programs with indirect pointer access. We believe that MPU programming model is equally, if not more, intuitive and flexible as *CFlex*. The main contribution of this thesis over FlexRAM is our detailed analysis over a wide spectrum of workloads that shows that a simple and general PIM architecture with very low hardware overhead can come close to the efficiency of recently proposed specialized PIM architectures and match or exceed efficiency of state-of-art OoO processors. In fact, this is the primary contribution of our work over all of the other related work in the PIM space.

Active Pages [7, 8] integrates FPGAs with DRAM. Programmer can define *functions* that are mapped into FPGA. They provide a programming API to bind these functions to certain memory blocks. It is not easy to port existing applications to it due to the complexity of programming FPGAs.

DIVA [9] proposes a novel architecture that can support virtual memory and concurrent accesses between host and PIM. The basic mechanisms of using simple in-order, low power cores remain the same as MPU. Their programming model is message-passing-like. They use *parcels* to move computation from one PIM unit to another, and also from host to PIM. This model is arguably not suited for workload classes that can most benefit from a PIM architecture (graph analytics, for example). MPU's contribution over DIVA remains the same as that over FlexRAM.

Centip3De [17] demonstrates the physical feasibility of a 3D design, and studies circuit-level design issues, 3D floor planing and clock skew introduced by the TSVs. Though it studies the benefits of a cluster-based CMP architecture with 64 Cortex-M3 cores with L1/L2 caches stacked on DRAM, it remains mainly a circuit-level project.

3.2 Comparison to works published post emergence of cost effective 3D die stacking

PIM is now being revisited in various novel ways due to the recent emergence of cost-effective 3D stacking of memory and logic. However, most approaches either suffer from narrow specialization or lose out on performance and energy efficiency while trying to stay general.

NDC architecture proposed by Pugsley et al [15] is closely related to our work. It uses similar mechanisms as MPU to propose an architecture with simple in-order cores stacked near memory. They propose use of a map-reduce programming model and evaluate on five database kernels. The main contribution of MPU over NDC is threefold. First, we undertake detailed analysis across 17 workloads spanning different domains, including end-to-end database queries, which enables us to make the claim that a general architecture like MPU can be competitive or more efficient than OoO processors for workloads with good locality, are significantly more efficient for low locality workloads, and come close to the efficiency of proposed specialized solutions in the literature. Second, MPU work includes a detailed proposal of a more flexible programming model, hardware architecture and system architecture that can be non-intrusively deployed on state-of-art OoO processors. Finally, specifically comparing the programming models, the map-reduce programming model is less flexible. For instance, it is non-intuitive to write graph analytics applications in map-reduce that often require atomic/shared accesses to data by multiple threads.

Picoserver [13] also uses similar mechanisms and principles as MPU. However, it is primarily suited for front end web servers where work is sent over the network along with data required for that work - there is no shared memory between host and the PIM units. Hence, unlike MPU, it is unsuitable for workloads with mix of serial and parallel regions.

Each of the studies [10], [11] and [12], target a specific application domain by specializing their system design. They also specialize their programming model which leads to unintuitive porting of workloads outside their domain. Gutierrez et al [10] study the potential of 3D die stacking for backend key-value stores. They specialize the APIs in the proposed programming model to support key-value store functionality (GET and PUT requests). Ahn et al [11] propose an architecture and programming model specialized for graph processing that combines the PIM idea, message-passing based programming model and hardware prefetching techniques in a novel fashion (discussed and quantitatively compared against MPU in Chapter 9). Ham et al [12] propose a highly specialized architecture for graph processing that specializes computation as well as memory accesses using custom hardware blocks.

There have been recent work like the one proposed by Zhang et al [14] and Vijayaraghavan et al [18] that utilize GPU as the compute substrate stacked with memory in a 3D fashion. Such architectures have much to gain from high bandwidth access, critical to GPU performance. Having said that, GPUs are best suited for workloads that exhibit high data parallelism and regular memory access pattern, and thus can provide excellent performance and energy efficiency for such workloads. However, this efficiency cannot be sustained in the presence of irregular memory access pattern (graph analytics workloads, for instance), as the GPU micro architecture include components like memory coalescer that burns power without being able to coalesce accesses and boost performance. MPU, fundamentally, takes a very different approach than GPU to gain energy efficiency and performance. While GPU invests significant hardware resources to enable massive multithreading and fast memory accesses for data parallel workloads with regular access pattern, MPU primarily targets workloads with irregular access pattern, utilizes large number of ultra-low power cores to gain performance via concurrency, and achieves energy efficiency by *waiting efficiently* on main memory accesses.

Conceptually, our goals are similar to Ahn et al [16] who propose a novel, low overhead technique for exploiting PIM benefits without modifications to the cache coherence protocols, virtual memory support while requiring minimal changes to the programming model. They provide hardware support on the host CPU chip to identify and offload low-locality memory accesses to the PIM substrate to take advantage of the high bandwidth availability, while keeping all the host CPUs ON to manage the computation. MPU keeps the CPU chip unmodified, but embraces disruptive change to the memory chip. It needs only 1 CPU ON (static power savings) to manage computation and offloads all computation to the massively concurrent, and ultra-low power hardware substrate. Consequently, it provides much better performance and energy savings as shown in Chapter 9. The GraphPIM work by Nai et al [19] is similar in concept to this work by Ahn et al, with the added advantage of not having the programmer annotate variables that would have potentially low-locality access.

HRL work [20] explores reconfigurable logic (combining ideas of CGRA and FPGA) computation near memory. They show that their compute units are more efficient/lower area than FPGA, CGRA and other types of compute units and almost as efficient as custom FUs for the programs they run. So, they can fully fit these in the PIM and more effectively utilize all the available bandwidth. However, FPGAs are still hard to program. MPU is much easier to program given the use of programmable cores and remote procedure call based programming model.

State-of-art processors like Xeon Phi Knights Landing and Knights Corner [21] are related to our work to the extent that they utilize several cores and die stacking (2.5D) to gain performance and efficiency. Xeon Phi has the advantage of supporting Linux and thus many existing applications. However, given that it still retains mechanisms like powerful out-of-order cores and cache coherence, it is not well suited, in terms of energy efficiency, for applications that have limited data sharing, irregular access pattern and low compute-to-memory ratio (all of which are exhibited by graph analytics, for example).

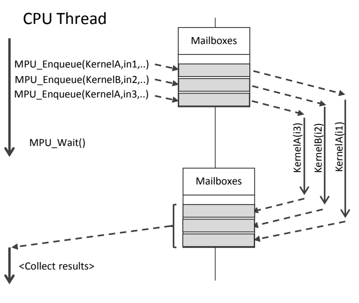
4 | Programming Model

In this section, we present the MPU programming model by describing its computation abstraction (Section 4.1) and data abstraction (Section 4.2). In Section 4.3, we discuss the generality of our model.

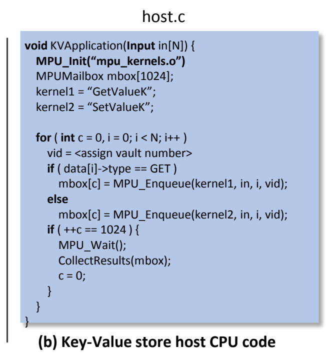
Note that, beyond the fundamental mechanisms of memory procedure calls and sharded view of memory, all of the specific design choices/APIs presented in this section were driven by one main factor: simplicity. Our objective in this thesis is not to propose the most-optimized programming model, but to present a design that can be further optimized and improved.

4.1 Computation Abstraction

The fundamental mechanism through which we offload computation is that of remote procedure calls to offload arbitrary granularity of work to memory. We call these *memory procedure calls* (MPC



(a) Execution flow for MPU applications

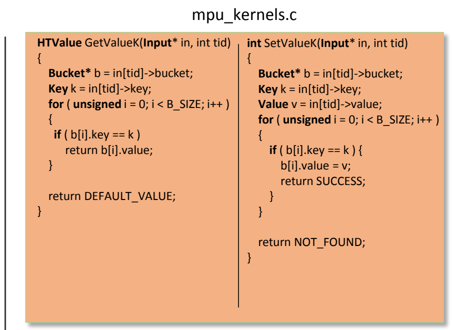


host.c

```
void KVApplication(Input in[N]) {
  MPU_Init("mpu_kernels.o")
  MPUMailbox mbox[1024];
  kernel1 = "GetValueK";
  kernel2 = "SetValueK";

  for ( int c = 0, i = 0; i < N; i++ )
    vid = <assign vault number>
    if ( data[i]->type == GET )
      mbox[c] = MPU_Enqueue(kernel1, in, i, vid);
    else
      mbox[c] = MPU_Enqueue(kernel2, in, i, vid);
    if ( ++c == 1024 ) {
      MPU_Wait();
      CollectResults(mbox);
      c = 0;
    }
  }
}
```

(b) Key-Value store host CPU code



mpu_kernels.c

```
HTValue GetValueK(Input* in, int tid)
{
  Bucket* b = in[tid]->bucket;
  Key k = in[tid]->key;
  for ( unsigned i = 0; i < B_SIZE; i++ )
  {
   if ( b[i].key == k )
      return b[i].value;
  }

  return DEFAULT_VALUE;
}
```

```
int SetValueK(Input* in, int tid)
{
  Bucket* b = in[tid]->bucket;
  Key k = in[tid]->key;
  Value v = in[tid]->value;
  for ( unsigned i = 0; i < B_SIZE; i++ )
  {
    if ( b[i].key == k ) {
      b[i].value = v;
      return SUCCESS;
    }
  }

  return NOT_FOUND;
}
```

(c) Key-Value store application kernels

Figure 4.1: Example application

in the following). A MPC is an invocation, on behalf of the host CPU, of a computational kernel. This kernel typically performs the memory-intensive part of the workload being executed, with the remaining part being run on the host CPU. In order to run code on the MPU, applications leverage primitives offered by our MPU API. The API consists of 10 calls - 2 related to initialization (`MPU_Init()`, `MPU_Close()`), 2 related to memory management (`MPU_Malloc()`, `MPU_Free()`), 2 related to computation management (`MPU_Enqueue()`, `MPU_Wait()`) and others for synchronization and global variable access. In this section, we focus only on the two core API calls used to manage the computation, `MPU_Enqueue()` and `MPU_Wait()` and leave out the rest for brevity's sake. We explain the model and API using a running example.

As an example, in this section we consider a slightly more complicated version of our key-value store application from Figure 2.2. The application, given in Figure 4.1(b) defines two kernels (Figure 4.1(c)) that respectively retrieve (`GetValueK`) and update (`SetValueK`) an entry. It then processes a set of input commands, each of which calls either kernel.

The host CPU interfaces to the MPC mechanism via a *queue abstraction* (Figure 4.1(a)). Each MPC request is enqueued via a call to `MPU_Enqueue()`. The runtime then allocates a *mailbox* for the request. A mailbox is a temporary memory buffer used to store inputs and outputs of a MPC request; mailboxes are the main interface between the the host CPU application and the MPCs.

The mechanism is asynchronous: each call to `MPU_Enqueue()` allocates a mailbox, stores input data in it, forwards the command to the MPU logic, and returns immediately. This favors a batch processing model (Figure 4.1(a)) where the host CPU can schedule many commands in parallel (by enqueuing then) without waiting for each command to terminate. Each `MPU_Enqueue()` in the example receives 4 parameters: (1) a string describing which kernel to run, (2) the data argument address (shared by all launched threads), (3) a thread ID and (4) a vault ID on which to Enqueue this kernel. The thread ID is used by the Enqueue'ed kernel to index into the argument data structure (which is simple a C structure that can hold multiple datatypes/variables). The MPU runtime and hardware controller direct the command to the correct vault based on the vault ID argument.

Internally, the MPU kernels (Figure 4.1(c)) manipulate the data structure. If a result is returned, it

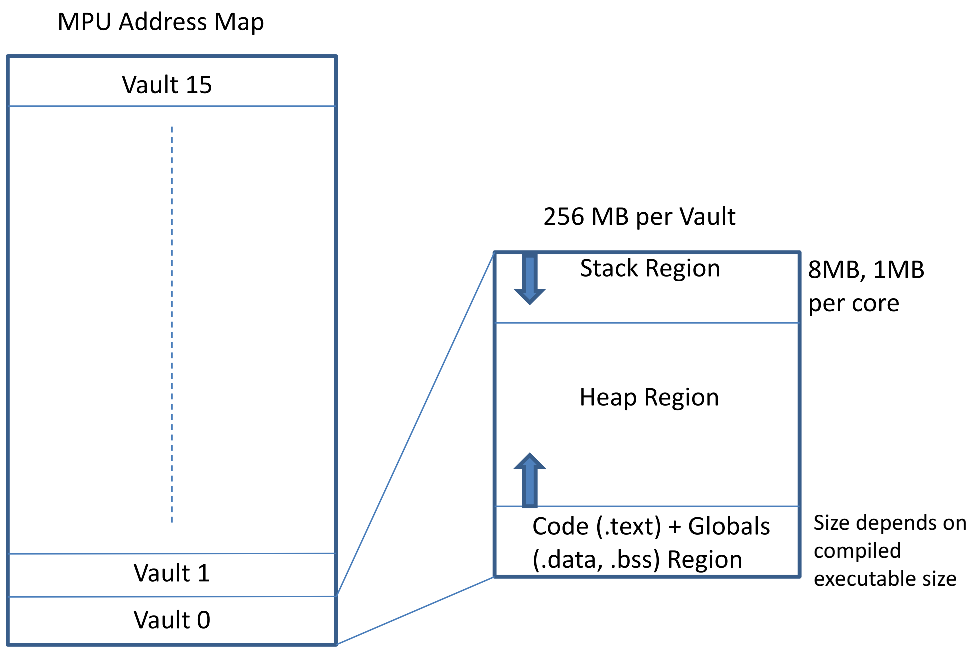


Figure 4.2: MPU Address Map

is stored in the mailbox associated with the command. Note that the mailbox can only store a single 64-bit value, so any results that exceed this limit must be returned via memory (programmer would need to allocate memory for this return data before launching the kernels). It is also possible to return all data through memory without using the mailbox. In order to wait for all enqueued threads to complete and safely retrieve the results, the host CPU must call `MPU_Wait()`, which stalls until all the pending MPCs have completed. Once `MPU_Wait()` returns, the results of issued commands can be retrieved by the host CPU from the respective mailboxes (and memory, if required).

The goal of the queue-based mechanism is to allow a single host CPU thread to issue a large number of MPU commands without blocking. This enables massively parallel workloads on the MPU with only a single host thread managing them, with consequent power/energy savings.

MPU provides a relaxed memory consistency model. Memory operations issued within execution of a single Enqueue of a kernel commit in-order, but there is no ordering guarantee between memory accesses issued by different Enqueues. Programmers can use an `MPU_Wait()` to create a fence as necessary.

4.2 Data Abstraction

The MPU does away with the notion of a flat memory space, exposing to the the programmer the abstraction of 16 memory segments, matching the number of vaults in our underlying hardware. In our current design, as shown in Figure 4.2, each vault consists of code+globals region, heap region and stack region. 8MB in each vault is reserved for the stack used by each core (1MB per core). The size of the code and global variables region is determined by the size of the *mpu_kernels* executable (all MPU kernels need to be defined in a separate *mpu_kernels.c* file and compiled separately for the target LX3 ISA). The remaining memory is allocated to heap region. Note that, in our current design, all code and global variables reside in vault 0. In the remaining vaults, this region is unused, which is reasonable given that this region is expected to take a small portion of the vault.

Each instance of an MPU kernel execution is tuned by the programmer to primarily access data within a single vault, to maximize performance. However, note that the architecture or programming model imposes no constraint on data placement. A kernel running on one vault can access data in another vault the same way it accesses data in the same vault, though it would take a latency hit due to inter-vault access.

Global variables can be used within the MPU kernels in a fashion similar to conventional global variable usage. However, to allow the host CPU code to access these variables, an *MPU_GetGlobal* API needs to be used with the variable name string as argument. The API extracts the corresponding symbol from the executable to get the symbol address and returns the address as a pointer to the global variable.

To deploy the key-value store application in Figure 4.1, the programmer would allocate the memory required to store the application dataset while passing a vault ID number with each allocation request (*MPU_Malloc(mem_size,vault_id)*). A pointer in the host CPU virtual address space is returned. The programmer would then instantiate a hash table in the allocated memory in each vault. As certain applications may still need to access data across vaults, MPU uses an interconnection network in the logic layer to access memory from a remote vault. The same underlying mechanism is extended for multi-MPU chips.

4.3 Generality of the programming model

Types of code: Since kernels are off-loaded like remote procedure calls, they can be arbitrary pieces of code, including pointers, control-flow, recursion, and calls to standard library functions. They must however avoid any system calls. For the purpose of this work, we do a *bare metal* compilation of the *mpu_kernels.c* file, including only the compiler provided headers (no GLIBC). We use our own custom implementation of required standard library functions like *strlen*, *strcmp* among others. As mentioned before, all the code resides in vault 0 code section. Some workloads (especially interpreted code like php, ruby etc.) are known to have an extremely large static code footprint and are unsuitable for MPU without modifications to our design. The workloads we consider are all C or C++ and the offloaded portion is user code without system calls or I/O.

Locks, coherence, and races: MPU kernels can also include various forms of synchronization within/between kernels. We extend the ISA with load-reserved/store-conditional instructions (LR/SC) to support atomic operations like locks, atomic fetch-and-add, among others, in the MPU programming API. Since we expect locks and other atomic operations to be used sparingly, and memory access time is small, any atomic/lock operation is not cached (our proposed LR/SC instruction implementation ensures this). This avoids the need for hardware coherence protocols.

Comparison with other programming models: The model closest to MPU is OpenCL. The main conceptual difference between OpenCL and the MPU model is that the former chiefly focuses on organizing computation, directly supporting SIMD-style operations and hierarchies of threads collaborating and communicating at different scales. Conversely, the MPU model focuses on data-layout and irregular code allowing multiple, independent computations. From a syntax standpoint, MPU applications can be written in OpenCL, and a MPU backend is straight-forward to implement. Similarly, one could implement a MapReduce backend that targets the MPU - since our underlying mechanisms can support Map/Sort/Reduce.

Compiler and Runtime: In order to support application deployment on the MPU, the MPU part (marked using custom annotation) must be compiled and fed to a modified linker, that must ensure ABI conventions are matched between the host CPU and the core used on MPU. While deploying on a real system, the ABI (application binary interface) of the core's ISA used on MPU should ideally match the ABI of the host ISA as much as possible to ease integration. Note that we chose Xtensa LX3, despite it being a VLIW core, throughout this document and analysis primarily because it represents a micro-controller class ultra low power core with easy to access specifications for our analysis. Setting up this compiler infrastructure is future work. For simulation studies, we use the x86 instruction trace [1]. Communication between the host and the MPU-based application components is mediated by the MPU runtime, which implements the MPU API in the form of a shared library.

[1]Blem et al [22] show that both ARM and x86 ISAs have similar instruction mix characteristics

5 | Architecture

Figure 5.1 shows the detailed organization of the MPU. We leave the DRAM dies unmodified and augment the logic die. The colored components are the modifications and clear white boxes are components that are part of standard HMC design. We add one compute tile associated with each vault and physically tightly integrated with the vault controller. We also add an interconnection network (called the MPU NoC) between the compute tiles to allow a core to access memory in other vaults. A centralized MPU controller interfaces with the host CPU to offload requests to MPU compute tiles.

All the design choices presented here are driven by one over-arching objective: choose the simplest design that matches what is required for the workloads that MPU is best suited for - highly concurrent, memory intensive workloads that present low cache locality and minimal synchronization among the concurrent tasks. Also, note that though the design described here is a

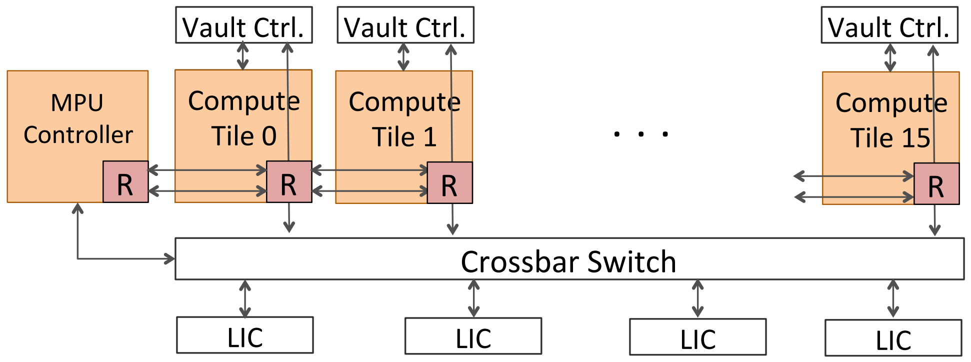


Figure 5.1: MPU High Level Organization

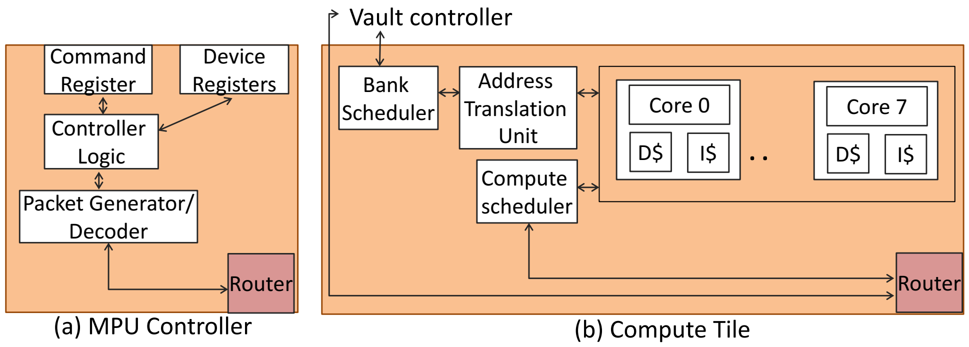


Figure 5.2: Controller and Tile Organization

detailed proposal, it still needs to be refined further for an RTL implementation.

5.1 MPU Controller

The MPU controller interfaces the MPU API to the MPU compute tiles, the HMC vault controller and implements the mailbox interface. It uses memory-mapped IO to communicate to the host CPU and receive MPU API's calls converted into loads and stores. It converts these into MPU messages delivered on the MPU NoC.

As shown in Figure 5.2(a), the MPU controller consists of controller logic, a command register, device registers SRAM that serves as a mailbox buffer and store of other necessary registers, packet generator and router. The mailboxes (within device registers block in the Figure) are addressable by the host CPU using memory-mapped I/O and x86 strong uncacheable loads and stores. There are a total of 1024 mailboxes.

Each mailbox entry is 24 bytes long to include a 64-bit kernel address, a 64-bit argument data address or return data, vault ID and thread ID. These comprise 3 of 7 64-bit device registers. The other registers are used during initialization to communicate the following information - base address of virtual memory region on the host CPU to which MPU memory is mapped (explained in Section 6), total size of MPU memory, stack size per core and stack base offset from start of vault memory. The command register (64-bit) is used to communicate the MPU API operation that was last sent by the host. There are four fields in the command register - command ID, 10-bit mailbox ID, 1 bit indicating that the last Enqueue was received (set by host CPU on *MPU_Wait*) and 1 bit indicating completion (set by controller logic on completion of all Enqueues). This requires a reasonably small SRAM of 24KB for mailboxes and 40 bytes for remaining registers and command register.

`MPU_Enqueue()` commands are compiled down into 3x64-bit strong uncacheable stores that write into a mailbox register, followed by a write into the command field to indicate that an Enqueue was received and mailbox ID field of the command register to indicate which mailbox was populated. This triggers the controller logic to read the mailbox, invoke the packet generator to create a fully formatted NoC message and send it to the correct vault for scheduling the kernel. The controller logic resets the command register as soon as it has successfully read the previous command, which is an indication to the host CPU to send the next command (using polling). Note that the `MPU_Enqueue()` only needs to block until the controller reads the command, not until completion of the Enqueue.

Values received from the compute tiles are written into the corresponding mailbox entry. The controller logic keep track of Enqueued kernels and completed kernels. Once all kernels have completed and the "last enqueue" indicator bit is set, the "done" bit is set in the command register. Reading the "done" bit would again require polling by the host CPU in this design. Note that potentially better designs may forego the requirement of polling by the CPU or reduce the frequency of polling using some timer-based mechanism in the controller.

5.2 Compute tile

The compute tile organization shown in Figure 5.2(b) consists of four sub-components. It includes an array of eight cores that make up the *compute fabric*. Its *compute scheduler* receives enqueue commands, and assigns work to an available core. The cores execute instructions and they interface to the DRAM banks with the *bank scheduler* and block until values return. The *address translation unit* converts virtual address, post an L1 miss, into physical address before accessing the bank scheduler and main memory. Its described in more detail in Section 6.1.3.

The **compute scheduler** dequeues MPU requests in-order and schedules them to available cores. It keeps track of the status of all the cores and when one become free it *assigns* an MPU request to that free core. The process of assigning a request entails delivering the input arguments and code address for the request (i.e. the parameters for the MPU kernel code) to the compute fabric and the specific core. Once assigned, the core executes the kernel code and on-termination notifies its completion status to the compute scheduler. Return values are delivered to the MPU controller through the MPU NoC, and from there to the host CPU.

The **compute fabric** comprises eight simple micro-controller class cores. The choice of core count is driven by the goal of building enough compute capability to maximize the utilization of the available bandwidth in the vault. Each vault consists of 16 banks. In our initial design, we provisioned 16 cores in the vault to sustain concurrent accesses to each bank. However, upon studying the actual access pattern and performance achieved by our workload set with 16 and 8 cores, we settled on 8 cores per vault. Further design space exploration is required to optimize the core count and cache sizes. In this specific paper, we consider a design with Xtensa LX3 cores since it allows us to estimate power and area[1]. We assume the ISA is extended with uncacheable ld/store instructions and LR/SC instructions to implement locks and other atomic operations like fetch-and-add. The core pipeline is short and blocks on memory access. The cores also include private small (16KB) level-1 data and instruction caches. The primary purpose of the caches is for stack access and to act as an energy filter - accessing a 128-bit line from SRAM is almost 10$\times$ less energy than a DRAM bank access, from models like CACTI [23] and other estimates [24, 25]. The cores access memory through a **bank scheduler** which keeps track of the status of all 8 banks and schedules accesses to these banks while adhering to DRAM timing requirements. The compute scheduler allows the cores to “complete” out of order since this has no impact on correctness and the programming model semantics expose this to the programmer. It is important to note here that the cores are really small in area and eschew many sophisticated features that make general-purpose cores large. According the detailed area estimates, at 45nm place-and-route area of one core is

[1]We ignore its VLIW issue capability and treat it as single issue in-order for performance modeling

$0.044mm^2$ [26], including the L1 caches its area is $0.257mm^2$; scaled to 22nm this is $0.064mm^2$. In contrast one core (excluding L2 and L3) of the Intel Ivybridge processor at 22nm, based on its die photos [27], has area of $8.57mm^2$, making it 133× larger. Excluding L1-caches for both, it is roughly $600\times$ larger. Compared to the ND Cores in [15], our core is 2.8× smaller, and $11\times$ lower power.

Our architecture and system design enables consistent pointers between the CPU and MPU, using a simplified mechanism, without need for additional page table support, TLBs or cache coherence. The address space that MPU operates on is the regular virtual address space of the CPU. Any CPU code can access data that belongs to the MPU after a cache flush of all the level-1 data caches, triggered in HW after all Enqueues complete execution. We discuss our system design in detail in chapter 6.

The MPU design provides hardware support for atomic operations like locks, fetch-and-add, among others, by extending the XTensa ISA with load-reserved (LR) and store-conditional (SC) instructions. These instructions are implemented by making appropriate modifications to the LX3 micro architecture and additional hardware at the vault controller. These two instructions form the building block for implementing any atomic operation in the software.

Cache coherence (within MPU) issues are avoided using following mechanisms. Cache coherence between CPU and MPU is discussed in Section 6.

1. *Uncacheable accesses to Global Variable Segment:* Any access to global variables (including lock variables) is ensured to be coherent by bypassing the L1 cache. This is achieved using the following mechanism: All global variables are placed in the .data and .bss segments of code+global region in vault 0 as shown in Figure 4.2. The start and end address of this region is placed in 2 registers in each MPU core. On each load and store, the target address is compared against these registers to see if it is a global memory region access. If so, the core bypasses the L1 cache and sends the request directly to vault controller. For LR/SC instructions, coherence is ensured by having these instruction implementations bypass the L1 cache.

2. *Cache flush and invalidation of any data touched within a critical section:* To maintain coherence for data shared in a read-write fashion, but allocated in the heap region of the vault(s), we use the L1 cache controller to record all cache lines accessed between *MPU_Lock* and *MPU_Unlock*. On *MPU_Unlock*, the core writes back and invalidates dirtied lines, and also invalidates lines that were only read. It is necessary to do the later (invalidate lines that were read but not written) as they might get written by another core (say core#2) that enters the critical section after core#1, and core#1 might read the line again in a future entry to the critical section. We assume good programmer behavior where these racy accesses are enclosed within a critical section (using *MPU_Lock*, *MPU_Unlock*). To reduce hardware overhead in the cache controller, a more coarse grained recording of the cache lines may be more appropriate (for instance, recording address of one cache line might represent that the adjacent cache line was also accessed, even though it wasn't - false positive).

5.3 Implementation

The compute fabric can be built using off-the-shelf embedded processor cores like the Xtensa LX3 (which is highly configurable and hence our choice for this paper) and existing interrupt and memory-mapped mechanisms can be used for interfacing to other components. Based on product sheets [26], the Xtensa-LX3 has power of 7.1mW at 500 MHz and 0.044mm^2 at 45nm (excluding cache areas). From CACTI the area of two 16KB caches built with LOP transistors (to minimize leakage power) totals to 0.213 mm^2 with an access time of 1.5ns. We expect our custom logic added to the core and L1 cache to have minimal area and power overhead. Considering 8 such cores running at 500 MHz gives 57 mW power and 2.06mm^2 area for the compute fabric. In our evaluation section, we account for the cache access energy which varies based on the workload. For the interconnection network, we use a lightweight network mesh network commonly used in spatial architectures [28, 29]. The router area is 0.012mm^2. Considering the entire MPU which consists of 16 such tiles, and a 64 KB SRAM, its total area is 33.4 mm^2. We use low power cores, since thermal containment in 3D integration is important. Our logic die power at 45nm (including SRAMs = 2.4 watts on average) is similar in power to two HMC links (3.5 W). Similar to others [15], we propose keeping only one SERDES link ON during MPU execution (computation by LX3 cores).

6 | System Level Design

The MPU system design adheres to two primary guiding principles:

1. *Consistent Pointers:* In order to enable high programmer productivity for new applications and ease of porting existing ones, the design must support consistent pointers between CPU and MPU code.

2. *Unmodified Host CPU Chip:* In order to enable plug-and-play integration to existing CPU SoCs, the design must not require any hardware modification to the CPU die, and require minimal modifications to the operating system.

This section details our design approach that adheres to these principles. We first present an overview describing the address map, how we maintain coherence between CPU and MPU, hardware support required on MPU for address translation, the challenges associated with this approach and how we resolve them. We then present an example pseudo-code of an MPU program that ties together the system architecture and the programming model described in Chapter 4.

Note that the design presented here is a proposal and might need further refinement for actual implementation.

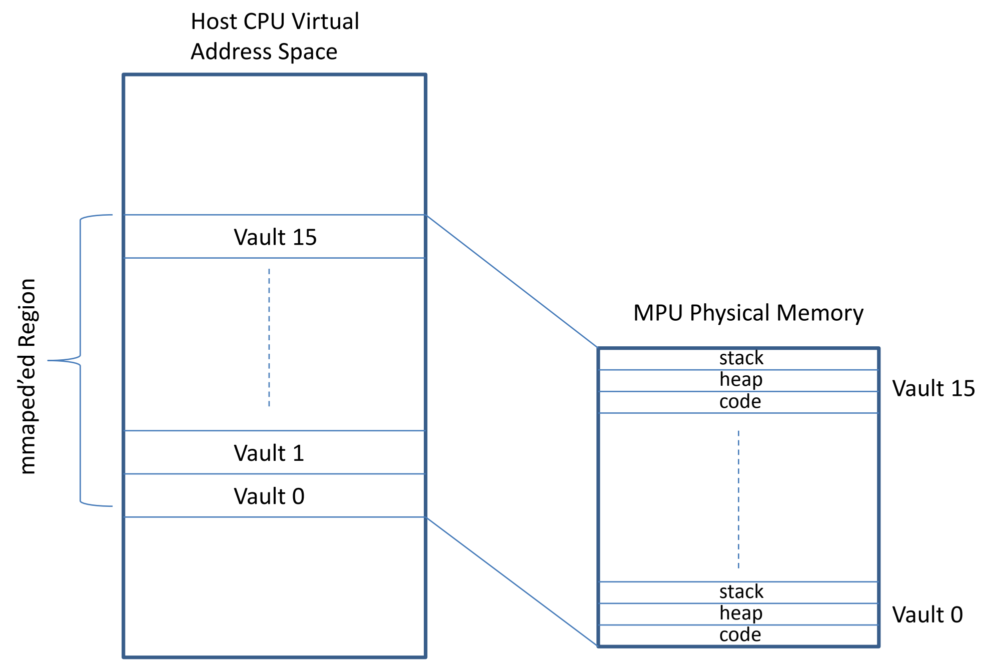


Figure 6.1: CPU-to-MPU Address Mapping

6.1 Overview

6.1.1 CPU Virtual to MPU Physical Address Map

Figure 4.2 lays out the address map of the MPU physical memory. The code, heap and stack regions for each vault are contiguously located. These regions are allocated and managed at a megabyte (MB) granularity by the MPU software library running on the CPU. Each vault is sized 256 MB.

To enable consistent pointers between CPU and MPU code, we provision a simple address translation mechanism without any additional page table and TLB support. Figure 6.1 shows how MPU physical memory is mapped into the host CPU virtual address space. The entire physical memory on MPU is mapped into the CPU virtual address space using the *mmap* system call at the beginning of the program. Essentially, this can be thought of as mapping the entire MPU memory with a single large page, thus required only a single base address for address translation (refer Figure 6.2. The programming model provides APIs for allocation (`MPU_Malloc`) and deallocation (`MPU_Free`)

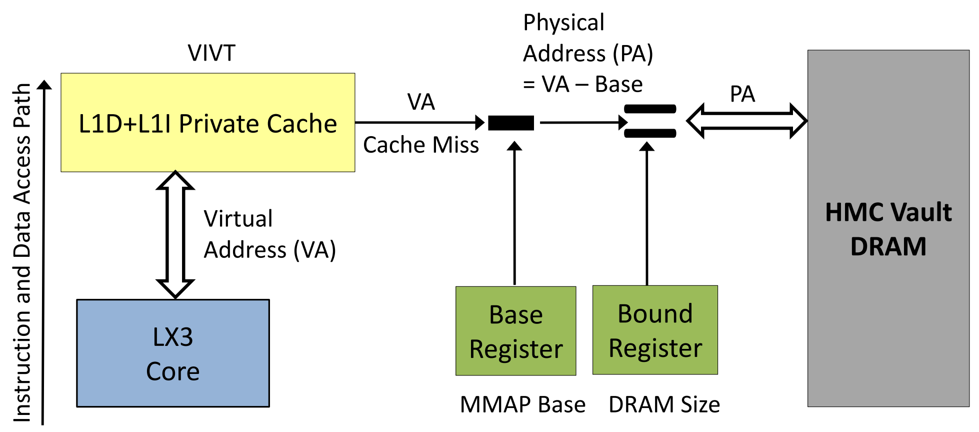


Figure 6.2: Code/Data Access Mechanism

of memory into a particular vault in this *mmap*'ed region. The MPU software library on host CPU manages memory allocation and deallocation using a modified *malloc* library implementation.

6.1.2 CPU and MPU Coherence

Since the *mmap* memory region mapped to the HMC can be accessed by both CPU and MPU code, maintaining coherence between the CPU and MPU caches is a concern. We do not support hardware cache coherence, and instead make a simplifying requirement of good programmer behavior where the CPU and MPU cores are not accessing this memory at the same time. We believe this is a reasonable choice for the type of workloads that we target, where the CPU hands off work to the MPU and does not touch the data associated with that work until MPU has finished the assigned work. We ensure that the caches of CPU or MPU are flushed and invalidated before transfer of execution from CPU to MPU or vice-versa.

6.1.3 Code/Data Access Mechanism

Each of the three sections (code, stack and heap) are accessed using a common translation mechanism as shown in Figure 6.2. As mentioned earlier, the MPU kernel code uses virtual addresses, both for code and data. The L1 data and instruction cache are virtually indexed, virtually tagged, so no translation is required for cache accesses. On an L1 miss, the virtual address is translated to MPU physical address using a simple base-and-bound mechanism. The base and bound registers are initialized during initialization phase of the program along with the *mmap* call (base register to the mmap base and bound register to the size of MPU physical memory). This mechanism is built into an address translation unit located in each compute tile.

Each MPU core uses a fixed 1MB stack page to store local variables. To ensure that stack pointer increments do not exceed its allocated 1MB space, each increment to the stack pointer will generate a check against the stack bound register (not shown in Figure 6.2). This register and checking mechanism is built into the core. The stack pointer and bound registers are also initialized along with the base and bound registers during *MPU_Init()*.

6.1.4 Challenges

There are four main challenges to this approach from the system software perspective, as listed below:

1. Ensuring cache flush and invalidation of dirty CPU/MPU cache lines while transferring execution between CPU and MPU
2. Ensuring that the *mmap*'ed virtual address space is kept contiguous in physical address space, as shown in Figure 4.2
3. Ensuring that the *mmap*'ed virtual memory region is mapped to the HMC physical memory ONLY, instead of some other physical memory in the system
4. Ensuring that the contiguous physical addresses are mapped to a single vault, followed by the next vault and so on.

Challenges 1 and 2 would require use of new/existing *kernel functions*, which would in turn require corresponding new system calls so that these kernel functions can invoked by the user-space MPU memory manager. The other two challenges can be resolved more easily without OS modifications. Below, we briefly describe a proposed solution each challenge. Note that these are only preliminary solution proposals, and need to refined further.

Cache Flush On execution transfer from CPU to MPU, we would need a a kernel function similar to `dma_cache_wback_inv()` to flush and invalidate potentially dirty cache lines on the CPU (across all levels of cache). This particular function was designed for DMA purposes, and hence cannot be directly used for our design. This flushing would have to be invoked by the programmer using a cache flush invocation API. For the other transfer direction (MPU to CPU), on *MPU_Wait*, hardware support for flushing all the L1 data caches would be required. We expect the latency of walking the L1 caches on MPU and generating the flushes to not have much overhead (few hundreds of cycles) given the total cache size of only 2 MB. Walking the CPU cache and flushing is expected to have a much larger overhead (around tens of thousands of cycles - assuming 10 MB cache flush, 80 GB/sec bandwidth). This should be acceptable as we expect the MPU programs to execute for a much longer duration. The workloads that we have evaluated consume several hundred thousand to millions of cycles.

Contiguous Physical Memory To ensure contiguous physical memory for the *mmap*'ed region, a kernel function similar to `kmalloc` [30] would be required. `kmalloc` is meant for contiguous physical allocation of relatively smaller request sizes (up to around 128 KB).

Using HMC Memory for *mmap'ed* Virtual Region To ensure that HMC memory chip is used to back the *mmap*'ed virtual address region, a system call similar to `mbind()` [31] would be required. This call binds the specified virtual address range to a specific memory node. It is meant for non-uniform memory access (NUMA) machines, and can be readily used for our design.

Physical Address to Device-Specific Location Mapping The mapping of physical address to physical locations of the memory device is handled by the particular memory controller, not the operating system. The HMC memory controller allows configuration of how the addresses are mapped to vaults and to banks within a particular vault. This can be achieved by setting the "Address Mapping Mode" register as described in the HMC specification [32].

6.2 Example MPU Program Pseudo-Code

This example MPU program ties together the MPU programming model and the proposed address space management technique. Each MPU thread in the program takes two arrays, adds the corresponding elements and writes it to an output array. As each thread completes it work, it atomically increments a global variable.

To aid understanding, the code has been heavily commented. The code is presented in two parts - host.c for the host CPU setup code and mpu_kernels.c for the MPU kernel code.

```
//*****************************************//
//*************host.c code****************//
//*****************************************//

//Argument structure (to be passed to the MPU kernel)
typedef struct {
  int*  input1_array;
  int*  input2_array;
  int*  output_array;
  int   array_length;
} args_t;

int main(void) {

  //Allocate 4 GB worth of virtual address space on the host
  //Initialize base, bound, stack base and stack bound registers on MPU
  MPU_Init();
```

```
//Allocate memory on MPU physical memory heap for input and output data
//for each thread
int length = 100;
int vault_id = 0;
for(int i=0; i<NUM_THREADS; i++ ) {
  input1_ptr[i] = MPU_Malloc(length, vault_id);
  input2_ptr[i] = MPU_Malloc(length, vault_id);
  output_ptr[i] = MPU_Malloc(length, vault_id);

  populate(input1_ptr[i], length, ..., ...);
  populate(input2_ptr[i], length, ..., ...);
  vault_id = (vault_id + 1) % MAX_VAULTS;
}

//Allocate memory on MPU physical memory heap for arguments
args = MPU_Malloc(sizeof(args_t), 0); //allocate in vault 0
args->input1_array = input1_ptr;
args->input2_array = input2_ptr;
args->output_array = output_ptr;
args->array_length = length;

kernel = "compute_ct";

//Schedule the kernel for execution on the MPU compute tiles
vault_id = 0;
for(int i=0; i<NUM_THREADS; i++) {
  //Use of mailbox is optional, used only if there is a return value
  MPU_Enqueue(kernel, args, i, vault_id);
  vault_id = vault_id % NUM_VAULTS;
  }

  //Block until all scheduled kernels finish execution
  MPU_Wait();

  //Print output data
  for(int i=0; i<NUM_THREADS; i++) {
    for(int j=0; j < length; j++) {
      print(output_array[i][j]);
    }
  }

  //Print the global variable value
  int* globalVal = MPU_GetGlobal("global_var");
  print(*globalVal);

  //Free up MPU memory not required
  for(int i=0; i<NUM_THREADS; i++ ) {
    MPU_Free(input1_ptr[i]);
    MPU_Free(input2_ptr[i]);
    MPU_Free(output_ptr[i]);
  }
  MPU_Free(args);

  MPU_Close();
}

//*****************************************//
//**********mpu_kernels.c code************//
//*****************************************//
```

```
unsigned global_var = 0;

int compute_something(void* args, int thread_id) {
  args_t *arguments = (args_t *)args;

  int* input1 = arguments->input1_array[thread_id];
  int* input2 = arguments->input2_array[thread_id];
  int* output = arguments->output_array[thread_id];
  int length = arguments->array_length;

  for(int i=0; i <length; i++)
    output[i] = input1[i] + input2[i];

  MPU_FETCH_AND_ADD(&global_var, 1);

  return 0;
}
```

7 | Experimental Infrastructure

7.1 Simulation Methodology

The goal of our evaluation is to understand the effectiveness of our principles in achieving goals of efficiency and generality. In order to do so, we use simulation to determine performance and energy consumption of a range of workloads on single MPU (Sections 8.1 and 8.2) and multi MPU configuration (Section 8.3). We also use the same approach to evaluate across various DDR3 and HMC based baselines (Chapter 8). We use a 4-core Intel Skylake processor as the baseline. Additionally, we compare improvements achieved by MPU with those reported by competing PIM architectures (Chapter 9).

In addition to the MPU configuration and the baseline CPU, we also model two intermediate design points as shown in Figure 7.1. Note that, in *SKY+HMC* as well as *SKY+HMC+128Tiny*, 4 SERDES links connect the host CPU chip to the MPU while 3 of these links are turned off for the

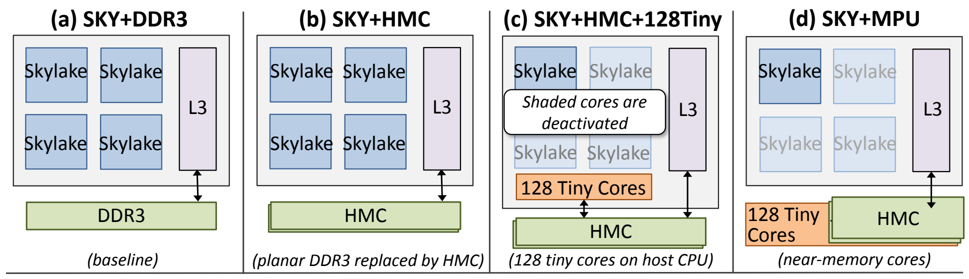


Figure 7.1: Organization of Baseline Designs vs. MPU

SKY+MPU configuration as only 1 CPU core needs access to the bandwidth. This approach allows us to determine the individual impact of the three most important aspects of the MPU architecture: replacement of planar memory with 3D-stacked HMC, array of lightweight cores, and near-memory placement of cores.

We customized the fast, multi-core ZSim [33] simulator to model the MPU performance. Specifically, we use ZSim to model a system that consists of a single Westmere core + multiple single-issue in-order cores with a memory system as described in Chapter 5. We customize ZSim to model out-of-vault accesses and cache flushes while transfering control from MPU to CPU and vice-versa. Our canonical baseline CPU is a Skylake system (details in Appendix), for which we obtain measurements from a desktop CPU (i7-6700K). We chose this baseline as Skylake arguably represents the current state-of-art processor and is widely used in the desktop and server markets.

Note that we use Westmere as the host core for MPU simulation instead of Skylake as ZSim cannot model a Skylake core faithfully. It has been heavily tuned to model Westmere core. We believe this is a reasonable infrastructure choice as the MPU performance will be more conservative with Westmere instead of Skylake as the host core. In addition, most workloads execute relatively small regions of code on the host CPU compared to MPU.

In order to measure the power and energy of both the baseline and MPU system, we built a McPAT-like power model and integrated it with ZSim. We used datasheets on DDR3, HMC, CPU-power, and Xtensa LX3, and CACTI to build this model that leverages access counts from ZSim and energy-per-access for various structures from literature (Tables A.1, A.2 in Appendix).

Note that we are able to simulate end-to-end programs that include the serial code (modeled on Westmere) and multiple parallel phases (modeled on in-order cores). This capability allows us to faithfully model the effect of load-balancing.

Workload	Description	Dataset
StringMatch	Search file for an encrypted word	50MiB
Kmeans	Iterative data-points clustering	100000 points
Histogram	Histogram of each RGB component	100MiB image
Scan	Scan 1 column of a database table	6 million rows
Aggregate	Aggregate 1 column using 2-column key	6 million rows
HashJoin	Join two database tables	6 million rows
BuildHashTable	Build hash table to get row index from actual database item value	6 million rows

Table 7.1: Description of Kernels

7.2 Workloads Description

We have twin goals in our choice of workloads - representation of domain diversity and representation of "real-world" usage. Our first set of workloads are the commonly used kernels for evaluating the PIM paradigm. Table 7.1 describes them. Next, we look at graph-processing workloads whose properties are ideally suited for MPU/PIM. Next, we look at end-to-end database processing by implementing TPC-H queries end-to-end. The first two workloads-classes do not stress the intermingling of CPU phases with MPU phases. TPC-H queries stresses this aspect of MPU as well - a single query has up to 15 MPU phases with intermingled CPU phases. Finally, to push the limits on cache-friendliness and data-level parallelism, we look at state-of-art text analytics (which often has even specialized accelerators of its own).

Note that we implemented two versions of each workload - pthread API version and MPU API version. The pthread version is used for baseline simulation while the MPU version is used for MPU simulation.

The remainder of this subsection describes the workloads.

Graph processing We implement 5 workloads common to two recent PIM papers [11, 16].

1. *PageRank (PR):* ranks the relevance of a graph vertex based on the relevance of its predecessors, by performing multiple passes over the graph until vertex ranks stabilize.

2. *Single-Source Shortest Path (SSSP):* computes the minimum distance of each graph vertex from

an arbitrary root vertex.

3. *Average Teenage Follower (ATF)* is used to process social network graphs. It computes for each node, the average number of predecessors representing a 13 to 19-year old user.

4. *Conductance (CT):* measures the number of edges in a graph that go outside a specified partition versus the number of nodes within that partition.

5. *Vertex Cover (VC):* finds the minimum set of vertices such that each edge of the graph touches at least one of the vertices in the set.

For all five workloads, we use the same *ljournal-2008* dataset used in [11, 16], obtained from [34].

DB processing TPC-H is an important database workload class which even includes dedicated accelerators [35]. We used the MPU API and implemented the most commonly used operators, along with 6 queries (1, 3, 5, 6, 10, 14) that use them. These operators - *scan, aggregate, sort, hash join, build hash table* and *materialize* - exhibit varying degrees of memory regularity. The queries involve heavy intermingling of host CPU code (code that is not offloaded to MPU) with MPU code with both sharing the same address space, and multiple MPU operator calls per query. We use a 1GB dataset sharded for the single MPU evaluation.

Our C implementation avoids some loading overheads etc. and is slightly faster than databases like mysql, postgresql, etc. It provides a fair comparison to MPU, and gives us better control for making detailed measurements and one-to-one comparisons. We use the query plan from PostgreSQL.

Text processing We consider two types of finite-state-machine processing. *DFAGroup* implements intrusion detection: rules expressed as regular expressions (2612 regexps from the popular Snort IDS [36]), which are grouped into a set of discrete finite automata (32 DFAs total occupying 3.7GB), are matched against network packet trace byte-by-byte (Lincoln Lab DARPA set [37]). It is extensively parallelizable as each packet can be matched against each DFA in parallel, and different packets can be processed independently.

The second workload *HTMLTOK* is a state-of-art tokenization using the data-parallel finite state machine (DP-FSM) algorithm from [38] which is irregular and concurrent. The algorithm processes the input from every possible start state, simulating a set of FSMs running in parallel on the same input (each starting from a different state). This code is SIMDized and multi-threaded for the CPU. Our MPU implementation breaks the implementation in two phases (processing+tokenization) and deploys the same FSM on multiple vaults, using each vault to match a different chunk of the input. The 8 cores within each vault can be used as independent SIMD lanes, each advancing a different state of the FSM. We used a 2GB Wikipedia text dump as input.

The range of workloads in our evaluation (described in this section) shows that the MPU API and Programming model is sufficiently general to implement diverse applications.

8 | MPU vs CPU

We seek to understand the effectiveness of our principles in achieving efficiency and generality. To that end, in this section, we first compare MPU against CPU. We perform a narrow evaluation by considering a single MPU integrated to a 4-core Skylake class processor. This allows us to understand the source of MPU's benefits and isolate how the three principles help.

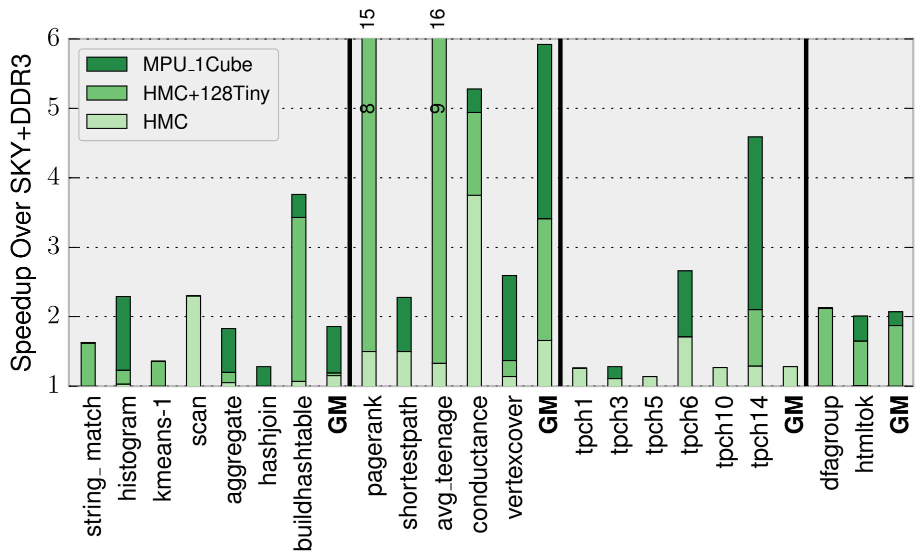


Figure 8.1: Performance Benefits of Single MPU

8.1 Performance

Figure 8.1 shows the performance of a single MPU across the workloads. MPU provides up to $16\times$ *speedup with geomean speedups of* $1.86\times$, $5.92\times$, $1.21\times$ *and* $2.01\times$ *across the four workload sets - kernels, graph analytics, database analytics and text analytics.* It also shows break-down of the three intermediate design points.

Fundamentally, the workloads that are suitable for MPU need concurrency. Sequential code regions are executed on the CPU (3 of 4 Skylake cores turned off), and concurrent code regions on the MPU cores, using the MPU's tightly-coupled offload model. Thus, considering only the concurrent MPU regions of code, MPU performance compared to a multi-core CPU can be expressed as: $CoreCount_{ratio} \times Frequency_{ratio} \times IPC_{ratio}$. Only the third term is workload-dependent and the product of the first two terms is *32 / 7 = 4.57*. Thus, (1 / IPC_{ratio}) needs to be greater than *4.57* for the baseline to outperform MPU on the concurrent region.

Based on this formula, the insight on why MPU can perform well is two-fold. First, on memory-intense workloads with irregular access pattern, the IPC on the OOO Skylake CPU is typically low due to high main memory latency. Second, since MPU runs at a much lower frequency (500 MHz), the load-to-use and memory access latency (in cycles) are only 1 and 20, respectively. Due to these reasons, MPU IPC is higher than CPU for some of these workloads (most graph workloads) and when it is not, the CPU-to-MPU IPC ratio is typically lower than *4.57* (geomean 1.47).

However, note that this simplistic formula only applies when the overall instruction count is same in baseline program and MPU program and load is equally distributed among all cores. As we show in the analysis to follow, this does not always hold true. Also, for workloads that are not 100% concurrent (TPCH), Amdahl's law effect plays its role in determining the speedup.

In our analysis, we also consider two intermediate design points - *SKY+HMC* and *SKY+HMC+128Tiny* - to analyze where the benefits of MPU are coming from. Considering the intermediate design point of adding only HMC (*SKY+HMC*), we should see performance improvements when the workload is bandwidth bound by DDR3 even by the small number of CPU cores. The *SKY+HMC+128Tiny*

Workload	Speedup	Description	Suff. Arch.
stringmatch	1.63	Good locality + Parallelism in updating different reference keys + High mem-to-total-inst ratio (50%) = Medium baseline IPC (1.63). Good locality = Sufficient BW in *HMC+128Tiny*. (1 / IPC_{ratio})=1.83	HMC+128Tiny
histogram	2.29	RAW dependence due to same bucket = Medium CPU IPC (1.86). High BW demand due to 128 cores = BW throttling in *HMC+128Tiny* due to low available BW. Sufficient BW in *MPU_1Cube*. (1 / IPC_{ratio})=2.35	MPU_1Cube
kmeans	1.36	Good locality + Parallelism in handling diff points together = High baseline IPC (3.91). Sufficient BW in *HMC+128Tiny*. (1 / IPC_{ratio})=3.95	HMC+128Tiny
scan	1.71	Medium L1D hit rate for baseline (94%) + no temporal locality = low-medium baseline IPC (0.91) due to high BW demand. With *HMC*, CPU IPC = 2.1 leading to 2.3× speedup, better than 1.71 × speedup with *MPU_1Cube*. (1 / IPC_{ratio})=2.6	HMC
aggregate	1.83	Serial dependency in aggregating hash table content = Medium CPU IPC (1.9). High BW demand due to 128 cores = BW throttling in *HMC+128Tiny* due to low available BW. Sufficient BW in *MPU_1Cube*. (1 / IPC_{ratio})=2.47	MPU_1Cube
hashjoin	1.28	Serial dependency b/w instrs + irregular access pattern leading to low cache hit rate (88%) = low baseline IPC (0.33). High BW demand due to 128 cores = BW throttling in HMC+ 128Tiny due to low available BW. Less BW throttling in *MPU_1Cube*. (1 / IPC_{ratio})=2.75	MPU_1Cube
buildhashtbl	3.76	Serial dependency b/w instrs + irregular access pattern leading to low cache hit rate (90%) = low baseline IPC (0.3). Sufficient BW in *HMC+128Tiny*. (1 / IPC_{ratio})=0.63	HMC+128Tiny

Table 8.1: Kernels Performance Analysis

configuration of adding the array of processing cores to the CPU die can match the performance of MPU if the workload does not become bandwidth hungry even when running with many tiny cores - this will happen when a workload has extremely good cache-behavior even on a small L1-cache. Finally, the *MPU_1Cube* configuration will outperform both intermediate designs when the workload has enough concurrency and irregularity to take advantage of the much higher bandwidth available internally in HMC than externally via SERDES.

In the following sub-sections, we analyze each class of workloads separately and summarize the main analysis takeaways. In each section, we use a tabular format to analyze each workload in detail.

Workload	Speedup	Description	Suff. Arch.
pagerank	14.63	High serial dependency + Very low locality (50% L1 hit rate) + = BW throttling in baseline + low baseline IPC (0.05). BW throttling in *HMC+128Tiny* also due to insufficient BW. Less BW throttling in *MPU_1Cube*. (1 / IPC_{ratio})=0.33. (slowest core load is 60% > perfect load distribution)	MPU_1Cube
shortestpath	2.28	Limited branch predictor behavior on CPU leads to fewer poor locality accesses in core loop = better CPU IPC (0.23). Similar MPU behavior and load imbalance as pagerank. (1 / IPC_{ratio})=1.64.	MPU_1Cube
avgteenage	15.94	Similar analysis findings as pagerank. Load imbalance. (1 / IPC_{ratio})=0.29.	MPU_1Cube
conductance	5.28	Inner loop not taken often, so good load balancing and poor locality accesses avoided = better CPU IPC (0.14). (1 / IPC_{ratio})=0.7. For *HMC+128Tiny* and MPU, no BW throttling + load balacing much better leading to speedup closer to (4.57/0.7)	HMC+128Tiny
vertexcover	2.59	High serial dependency + good locality (90% L1D hit rate) = Better baseline IPC (0.56) and MPU IPC (0.45) than other graph workloads. BW throttling in *HMC+128Tiny* due to insufficient BW. Less BW throttling in *MPU_1Cube*. (1 / IPC_{ratio})=1.24	MPU_1Cube

Table 8.2: Graph Analytics Performance Analysis

8.1.1 Kernels Analysis

Table 8.1 presents performance analysis of each kernel. Below, we summarize the main takeaways from our analysis.

Takeaway 1: Overall speedup is dependent on the IPC_{ratio}, which is affected by a variety of reasons affecting baseline CPU core and MPU LX3 core performance - cache locality, mem-to-total-inst ratio, instruction level parallelism.

Takeaway 2: Degree of parallelization and load balancing are other major factors dictating speedup, but doesn't affect the kernels evaluated in this section as they are almost 100% parallelizable and well load balanced. Effect of load balancing is observed for graph workloads (Section 8.1.2) while effect of parallelizability is observed for TPCH queries (Section 8.1.3).

8.1.2 Graph Workloads Analysis

Table 8.2 presents performance analysis of 5 important graph analytics workloads[1]. Below, we summarize the main takeaways.

Takeaway 3: All 5 graph analytics workloads have a very irregular access pattern leading to a low first-level cache hit rates (50%) and much lower hit rate on lower-level caches for both baseline and MPU.

Takeaway 4: These workloads all have high serial dependency, leading to low baseline OoO CPU IPC (0.05-0.56) when combined with the low cache hit rates. For 3 of the 5 workloads (pagerank, averageteenage, conductance), MPU IPC exceeds CPU IPC.

Takeaway 5: Load balancing is a significant contributor to speedup achieved with MPU for graph workloads. This is mainly because of the structure of the graph making it hard to statically do load balancing. We observe that for 4 of 5 graph workloads evaluated, the slowest core executes 60% more instructions than the ideal equally-distributed load scenario.

8.1.3 Database Workloads Analysis

Table 8.3 presents performance analysis of six database analytics queries from the TPCH benchmark suite. These queries cover a fairly wide behavior pattern. Below, we summarize the main takeaways[2].

Takeaway 6: Degree of parallelization of the program (and each operator) is the most significant contributor to final speedup. We observe highest speedup for queries with maximum parallelism (TPCH 6 and 14).

Takeaway 7: High degree of parallelization must be coupled with sufficient bandwidth to ensure high speedup. As seen for tpch 5, we observe low speedup despite high parallelism which is due to insufficient bandwidth.

[1]One of these workloads (pagerank) contains FP operations. However, LX3 core micro-architecture does not have an FPU. Pagerank data here reflects performance and power of a hypothetical LX3 core that includes single-precision FPU unit

[2]For some of these queries, we encounter a known simulation issue which under-estimates MPU performance. For instance, in tpch1, MPU parallel-icount to total-icount is reported by simulator as 96% while it should be close to 100% as per extensive human-expert analysis. So actual speedup should be higher than 0.54×

Workload	Speedup	Description	Suff. Arch.
tpch1	0.54	medium parallelism (96%) + medium locality = medium baseline IPC (1.56). For MPU program, high BW demand leading to significant BW throttling in *HMC+128Tiny* as well as *MPU_1Cube* (IPC = 0.51)	HMC
tpch3	1.28	low parallelism (86%) + low locality + serial dependence in hash table lookups = low baseline IPC (0.44). For MPU program, high BW demand leading to BW throttling in *HMC+128Tiny*. Less BW throttling in *MPU_1Cube*	MPU_1Cube
tpch5	0.86	Similar behavior as tpch3 but with higher parallelism (99%), leading to significant BW throttling in *HMC+128Tiny* as well as *MPU_1Cube* (IPC = 0.09). baseline IPC (0.4)	HMC
tpch6	2.66	High parallelism (99%) + low-to-medium locality = low-medium baseline IPC (0.65). For MPU program, high BW demand leading to BW throttling in *HMC+128Tiny*. Less BW throttling in *MPU_1Cube*	MPU_1Cube
tpch10	0.44	Similar behavior as tpch3 but with lower parallelism (73%), leading to very low speedups on *HMC+128Tiny* and *MPU_1Cube* due to lack of parallelism. baseline IPC (0.42)	HMC
tpch14	4.59	Very High parallelism (close to 100%) + low locality = low baseline IPC (0.35). For MPU program, high BW demand leading to BW throttling in *HMC+128Tiny*. Less BW throttling in *MPU_1Cube*	MPU_1Cube

Table 8.3: Database Analytics Performance Analysis

Workload	Speedup	Description	Suff. Arch.
dfagroup	2.13	Parallelism in processing input chunks in parallel + Good locality due to few hot state traversals + serial dependency in core loop = Medium baseline IPC (1.38), but close to max achievable IPC for MPU (0.9). Good locality = Sufficient BW in *HMC+128Tiny*. (1 / IPC_{ratio})=1.53	HMC+128Tiny
htmltok	2.01	SIMD compute in phase 1 to find start states for all input chunks. DFAGroup-like processing in next phase. Overall, similar baseline and MPU IPC as DFAGroup. However, more memory intensity causing BW throttling in *HMC+128Tiny* but sufficient BW in *MPU_1Cube*. (1 / IPC_{ratio})=1.61	MPU_1Cube

Table 8.4: Text Analytics Performance Analysis

Takeaway 8: Compared to graph workloads, it is fairly straight-forward to load-balance the threads in the TPCH queries. This can be attributed to inherit regular structuring of data in the database tables.

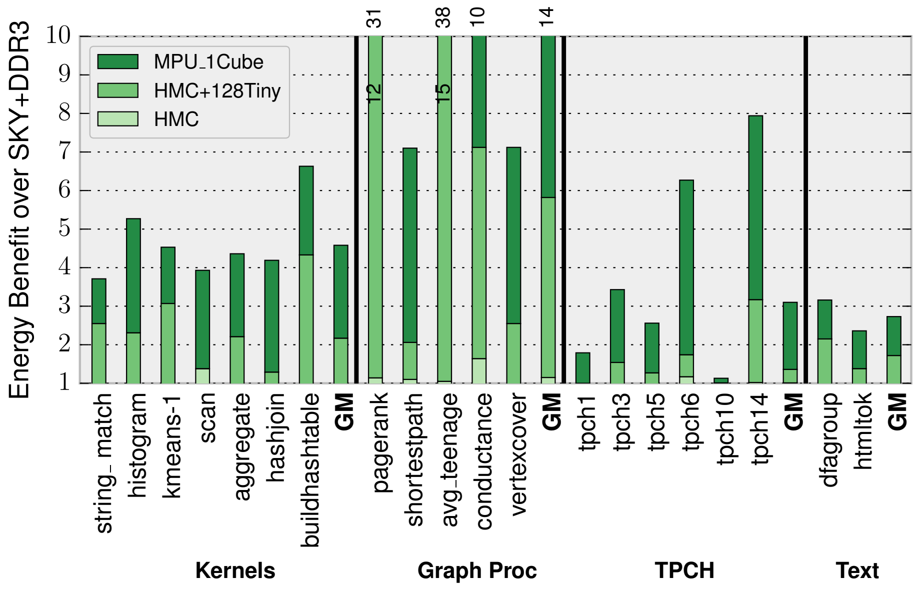


Figure 8.2: Energy Benefits of Single MPU

8.1.4 Text Workloads Analysis

Table 8.4 presents performance analysis of 2 two types of finite state machine processing workloads.

Takeaway 9: *DFAGroup* and *htmltok* both present good task level parallelism, good cache locality and serial dependence in the core loop of computation. For these reasons, an MPU-like architecture with high concurrency and efficient cores to execute low-ILP code regions is well-suited.

Summary Result: MPU proves most effective (in performance as well as energy efficiency) on workloads exhibiting low cache locality (graph analytics). On other workloads, MPU remains competitive in performance to state-of-art Skylake multi-core processor while consuming much less energy.

8.2 Energy

Figure 8.2 shows the energy improvement of MPU. Across all workloads, energy reduction ranges between $1.1\times$ *to* $38\times$. We analyze the sources of energy improvement by considering a single MPU attached to a CPU. We have four sources of power savings. First, we turn-off all but one host core, providing static-power reduction (in practice other applications may run on it, charging the power/energy to those). Second, 3 of the 4 HMC links are turned off since memory traffic from host core will be low, providing static-power reduction. Third, and a source of small dynamic power savings, is total memory access energy as we can save on the IO-energy of sending the values all the way to the processor. Finally, a large source of dynamic power savings is that the MPU cores are more energy-efficient than the 3 host cores that are turned off. Thus the total energy savings is speedup multiplied by static power savings combined with the modest dynamic power savings.

Below, we summarize the main takeaways from our power analysis. Note that we choose to discuss power instead of energy (for the most part except DRAM) to avoid the effect of execution time on the analysis.

Takeaway 10: *Baseline vs MPU_1Cube Configuration* For all workloads, the baseline spends most power (37-69%) on static compute power (core+SRAM) compared to other 3 power components. *MPU_1Cube* configuration achieves 2.9× power reduction on this power component, resulting in 1.3× to 3.1× power reduction overall. In other words, most power savings come from turning off 3 of the 4 OoO cores on the baseline.

Takeaway 11: *Baseline vs HMC Configuration* The advantage of *HMC* configuration over baseline lies in the more efficient DRAM access capability (lower dynamic access energy and higher bandwidth availability), which comes at a cost of 2.4× higher DRAM interface (SERDES) power. Hence, workloads that have good cache locality (thus fewer DRAM accesses) consume more energy with the *HMC* configuration than baseline. This is true for about half of the workloads evaluated.

Takeaway 12: *HMC vs HMC_128Tiny* Main source of power reduction going from *HMC* to *HMC_128Tiny* configuration is the reduction in static compute (core+SRAM) power (2.9×).

Takeaway 13: *HMC_128Tiny vs MPU Configuration* Main source of power reduction going from *HMC_128Tiny* to *MPU_1Cube* configuration is the 4x reduction in static DRAM power (due to fewer SERDES links).

Summary Result: *Static power savings from retaining only 1 OoO core and 1 SERDES link, combined*

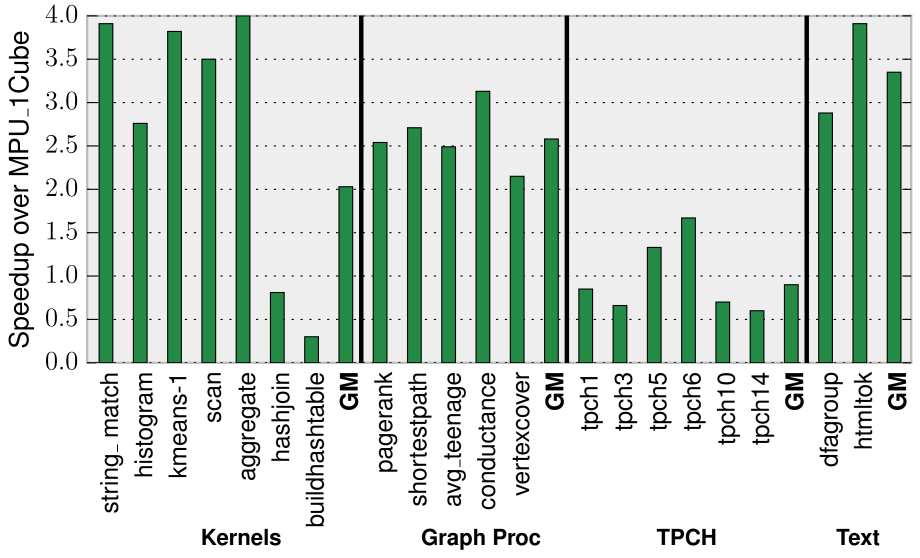


Figure 8.3: Performance Benefits of 4 MPU over Single MPU

with MPU's energy efficient cores and energy-efficient access from 3D-stacked memory integration, provide significant energy savings, which is amplified further with the performance improvements.

8.3 Multi-MPU System Analysis

In this section, we evaluate a multi-cube/multi-MPU system against a single MPU system to understand how the performance and energy reduction scales. In terms of hardware configuration, *MPU_4Cube* differs from the *MPU_1Cube* in the number of MPU cubes and the number of SERDES links between host CPU chip and the MPU(s). The 4 MPUs each connect to a single SERDES channel, thus driving up the total number of SERDES links to 4, compared to 1 in the *MPU_1Cube*.

Figure 8.3 and Figure 8.4 shows the performance and energy improvement of MPU_4Cube over MPU_1Cube. Across the workloads, the performance improvement ranges from $0.3\times$ *to* $3.91\times$ *while energy reduction ranges between* $0.18\times$ *to* $1.5\times$.

Below, we summarize the main takeaways from our multi-MPU analysis:

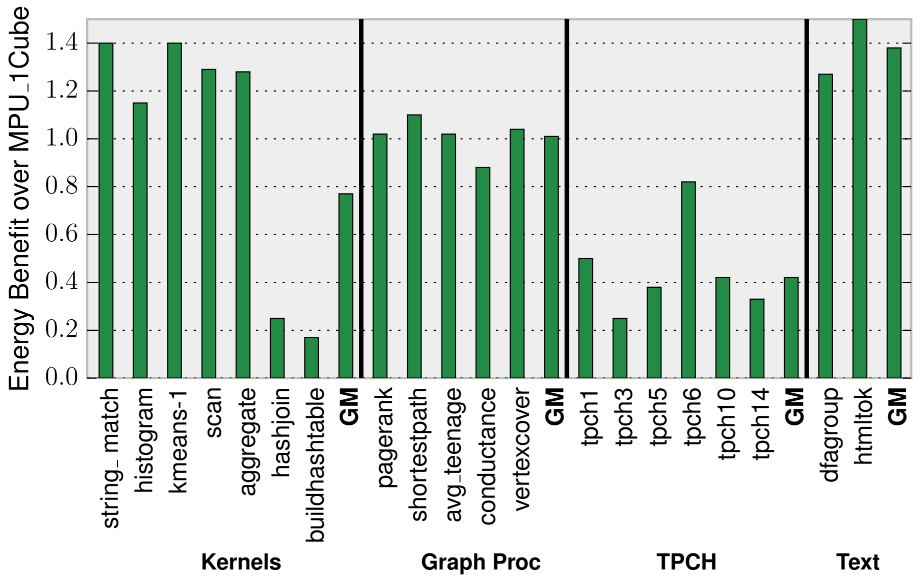


Figure 8.4: Energy Benefits of 4 MPU over Single MPU

Kernels Performance Analysis stringmatch, histogram and kmeans achieve an almost linear speedup primarily owing to close to 100% parallelizability and perfect load balancing. Histogram has 99% parallelizability, leading to lower speedup.

scan and aggregate kernels witness an improvement in IPC due to lower bandwidth throttling, owing to 4× bandwidth compared to MPU_1Cube. This leads to >4× speedup in parallel phase. However, a small serial phase lowers overall speedup.

hashjoin and buildhashtable witness a slowdown due to significantly higher total instruction count across all threads compared to MPU_1Cube. This is counter intuitive since the total number of keys being inserted/looked-up are the same. This is most likely a result of some algorithmic issue that results in more hash table collisions on the MPU_4Cube configuration compared to MPU_1Cube. Since these kernels are also used in several TPCH queries, it results in slowdown in some of those queries too (tpch5, tpch10, tpch14). Detailed investigation was hampered by debugging infrastructure capability.

Graph Analytics Performance Analysis Load balancing deteriorates compared to MPU_1Cube due to larger number of parallel threads. However, IPC improves owing to 4× bandwidth availability. Overall, this results in 2×-3× speedup over MPU_1Cube.

Database Analytics Performance Analysis Most TPCH queries see a slowdown or a 1.7× speedup at best over MPU_1Cube. This can be attributed to higher serial phase instruction count in the reduction phase of the workload and the higher parallel instruction count due to hash table collisions in the hashjoin and buildhashtable operators/kernels. This again highlights the importance of concurrency for performance scaling. For most queries, IPC per core improves substantially (2×-3×) due to lower bandwidth throttling enabled by 4× available bandwidth. However, the reasons mentioned above prevent this from increasing the overall speedup.

Text Analytics Performance Analysis Given abundant task level parallelism, the text workloads achieve close to 4× speedup. The *DFAGroup* has a small sequential code region which results in lower speedup.

Takeaway 15 (Performance Analysis): Concurrency and load balancing play a significant part in determining how much we can scale performance with a larger MPU system. With high concurrency and good load balancing, a larger MPU system can achieve significant performance improvement owing to the much higher internal- bandwidth (available to MPU cores) to external-bandwidth (available externally through SERDES) ratio compared to a single-MPU system.

Takeaway 16 (Energy Analysis): Across all workloads, the maximum energy savings going from MPU_1Cube to MPU_4Cube is 1.4×, despite much higher speedups. This is primarily due to increase in all power components, with dynamic DRAM power contributing the most. In other words, whatever power savings were achieved by MPU_1Cube over baseline by a 2.9× reduction in static core power gets counteracted by an increase in dynamic DRAM+SRAM+Core power and static DRAM power (due to more HMC links) in the MPU_4Cube configuration, with dynamic DRAM power contributing most.

9 | MPU vs Other Specialized PIM Architectures

This section compares MPU to two other PIM architectures covering different ends of the performance-generality spectrum. Tesseract [11] is a specialized PIM accelerator for graph processing with a specialized programming model. PIM-enabled instructions [16] trades performance for the ability to support a broader set of workloads while requiring modest changes to hardware and no changes to the programming model. In our evaluation, we consider all five graph processing workloads discussed in Chapter 8. All five have been evaluated in the Tesseract work and three of them evaluated in PIM-enabled (for the same dataset).

For comparison with each of the two works, we model and evaluate an MPU architecture that is similarly provisioned in terms of total capacity, number of HMC cubes and interconnect network among the cubes. Each HMC cube is essentially an MPU_1Cube. Also, we sought to model a similar baseline CPU as their work. Both papers are from the same author and description of the baseline core is similar in both, thus leading us to believe that both papers used the same baseline. The capability of their modeled core seems to be in the ballpark of Westmere core, based off of the instruction queue and load/store queue size information they provide in their paper. To be as close as possible to their baseline, we also chose to model the Westmere core in ZSim instead of using a real Skylake CPU machine as the baseline, as we did in Chapter 8.

	PR		SSSP		ATF		CT		VC	
Arch.	**Perf.**	**Energy**	**Perf.**	**Energy**	**Perf.**	**Energy**	**Perf.**	**Energy**	**Perf.**	**Energy**
MPU*	11.75×	9.04×	15.32×	14.4×	22.63×	20.89×	21.02×	10.92×	5.41×	6.6×
Tess.NoPft*	8×	-	10.3×	-	7.3×	-	3.5×	-	4.7×	-
Tess.*	14×	10×	26.7×	20×	12×	10×	5.6×	4×	6×	4×
MPU†	5.75×	7.66×	8.34×	13.62×	12.33×	19.76×	11.63×	10.51×	5.8×	12.31×
PIM-e.†	1.58×	<25%	1.51×	<25%	1.6×	<25%	-	-	-	-

***Baseline:** 32 4GHz 4-issue OoO + 128GB DDR3
†**Baseline:** 16 4GHz 4-issue OoO + 32GB HMC

Table 9.1: Comparison w/ Tesseract and PIM-enabled

Methodology Given the large scale of both Tesseract and PIM-enabled architectures (number of cores, number of HMC cubes, interconnect), we follow a different simulation methodology than used in Chapter 8. We built an analytical model on top of the zsim simulator to model the two MPU architectures and two baseline architectures resembling those used in the two works. We obtain simulation results from zsim for *HMC* configuration and *MPU_4Cube* (refer Figure 8.1 and 8.3), for baseline and MPU modeling, respectively. We then scale various performance and energy estimates like IPC, each component of power/energy, among others to reflect performance and energy on the four architectures modeled here (2 MPU architectures, 2 baseline architectures). Finally, we report Tesseract & PIM-enabled's performance/energy numbers from *their* papers.

9.1 Tesseract

Architecture: Tesseract [11] is a PIM accelerator for graph processing workloads. The system consists of 16 HMC 2.0 memory cubes with total 128 GB capacity and 512 cores. It augments each cube with one core (2 GHz single-issue) per vault and two hardware prefetchers - a *list prefetcher* that retrieves neighbors of vertexes being processed and a *message prefetcher* that retrieves vertex content on computation transfer to remote vault. Unlike MPU, processing of each graph vertex executes within the vault where the node is stored. Out-of-vault memory accesses are dis-allowed; in such cases, the Tesseract programming model requires computation to be moved on the external vault via a continuation-passing mechanism. The baseline CPU is 32 4GHz 4-wide OoO cores and 128 GB of HMC.

MPU organization: For the MPU evaluation, we integrate 32 MPU's (to get 128 GB capacity) to the multi-core CPU. As the MPU programming model allows out-of-vault accesses, we simplify the workload code and take advantage of that instead of generating tail calls as in the Tesseract implementation. This is a major programmatic advantage that MPU holds over Tesseract.

9.2 PIM-enabled instructions

Architecture: This architecture [16] offloads work at basic-block granularity of (dynamically determined by hardware predictor - PMU), which have loads/stores likely to generate a cache miss. The system consists of 16 HMC 1.0 memory cubes with total 64 GB capacity and 128 ALUs (called PCUs in their work). It augments each HMC 1.0 vault with one PCU. The baseline used in [16] consists of 16 4GHz 4-wide OoO cores and 8 4-GB memory cubes.

MPU organization: The MPU system integrates 16 HMC 1.0 MPUs to the same baseline host processor.

9.3 Quantitative comparison

Performance. Table 9.1 reports performance and energy results. For **Tesseract** comparison, we report data for Tesseract with (*Tess.**) and without the prefetchers (*TessNoPft.**), from their paper. Given that MPU provisions 8 cores per vault at 0.5GHz and Tesseract provisions 1 core per vault running at 2GHz, the ideal speedup of MPU over *TessNoPft.** should be 2× ($8 * 0.5/2$) as this comparison removes the prefetching advantage of Tesseract. However, the data in the table is not consistent with our expectation. In some cases (ATF and CT), MPU speedup over Tesseract is well over 2× while in others, it is about 1.1× to 1.5×. It is not possible to investigate more about this discrepancy as further details about their simulation methodology and Tesseract architecture is unknown. Difference in load balancing of the graph across the threads between MPU and Tesseract might also be a potential reason. With MPU, we do random partitioning of graph. Load balancing assumptions in both Tesseract and PIM-enabled are unknown. Load balancing can be achieved using some static or dynamic partitioning of the graph, both of which are active research topics [39].

After including the prefetching advantage for Tesseract, MPU achieves superior performance on two of the graph workloads (ATF, CT) while coming within 15% of performance for two others (PR,VC). Overall, the takeaway here is that MPU comes close to the performance of Tesseract while not specializing either the hardware or the programming model for graph analytics.

Across all workloads, **PIM-enabled instructions** achieves modest performance improvements over the baseline, remaining well below the MPU. This can be explained by observing that this approach offloads small program regions with irregular access patterns, while the rest of the computation is done on the host CPU. Thus, concurrency is limited to the number of cores $\times$ issue width.

Energy. In the Tesseract experiments, MPU energy savings are significant but inferior to speedup, which implies that MPU's power consumption is higher than the baseline. This is due to the high number of concurrent memory accesses while executing the workloads, which drives memory power up. The same phenomenon affects Tesseract.

The PIM-enabled MPU configuration uses a significantly smaller number of memory cubes. Therefore memory does not dominate MPU power consumption, and overall MPU energy savings are higher than speedup. PIM-enabled offers limited energy savings; in this architecture the main cores remain active during memory phases to orchestrate computation, limiting the impact of offloading on overall energy.

Result: On two of five graph workloads, MPU outperforms a specialized PIM architecture, while remaining competitive for two other workloads. Over a general PIM architecture, MPU achieves significant speedup and energy efficiency. This shows that it is feasible to achieve generality in PIM while retaining significant performance and energy benefits.

10 | Conclusion

In this dissertation, we have presented a new class of processor called Memory Processing Units (MPU) that enables efficient processing-in-memory leveraging simple principles: massive concurrency to achieve performance, simple low frequency cores that idle efficiently, and a general programming model that covers a large, diverse range of workloads. We have proposed a detailed hardware architecture, system architecture and programming model that can be non intrusively deployed on today's commercial OoO processors. Finally, we have presented a detailed evaluation study across a wide range of workloads spanning three commercially important workload domains. This detailed evaluation enabled us to show that a simple, low overhead architecture like MPU can be competitive or more efficient than commercial state-of-art OoO processors for a wide array of workloads with a variety of behavior, and come close to efficiency of recently proposed specialized architectures.

We conclude this dissertation with other thoughts/lessons learnt.

1. **General Architecture is a better trade-off:** The primary objective of our work is to investigate whether a simple, general purpose architecture could prove to be a better trade-off between performance, energy efficiency and flexibility compared to more specialized solutions, for workloads that exhibit low cache locality, by coming close to the performance and efficiency of specialized solutions while remaining highly flexible/programmable. I believe a general architecture like MPU is indeed a better trade-off, given the following insights:

a) The main performance benefit of PIM architectures comes from exploiting as much of the available bandwidth as possible. Given the small area and low power characteristics of tiny micro-controller class cores, we found that the available bandwidth can be saturated by employing a large array of such cores placed near memory, thus resulting in high performance and energy efficiency.

b) While additional performance can be attained by specializing the data path and the memory sub-system (thus sacrificing flexibility), the gain in performance is unlikely to be much more than what is achieved with a general architecture like MPU (based on studying the speedup attained by Graphicionado [12] for pagerank and shortest-path workloads) due to low cache locality and low instruction level parallelism in the workloads of interest (graph analytics). The gain in energy efficiency is also unlikely to be much higher than the speedup as memory access power starts to dominate power consumption as one scales out the number of compute units, and it's hard to reduce this power consumption due to the hard-to-predict, irregular access pattern.

2. **MPU is NOT a sufficient architecture for all kinds of workloads:** We believe that MPU is a good *baseline* architecture upon which future work can add more capability to further improve efficiency. For instance, GPU can be more efficient than MPU for workloads with high concurrency and regular memory access pattern. One could imagine some kind of high throughput engine connected to the MPU cores that provides highly efficient execution of workloads that have traditionally done well on GPUs, with ideally similar or easier programmability.

3. **Small data cache is beneficial:** Though the tiny cores on MPU are placed near memory and run at a much lower frequency than conventional OoO cores, the penalty is still non-trivial (20 cycles) compared to 1 cycle latency of accessing a small data cache. Though the workloads best suited for MPU would have a very low cache hit rate for heap accesses, they exhibit much better cache locality for stack accesses. The stack stores frequently accessed local data

structures that do not fit in the register file. Hence, we found a small data cache (to the order of 16KB) to be beneficial for performance and energy efficiency.

In conclusion, we believe that MPU presents a promising direction to better understand the tradeoffs between generality and specialization in the processing-in-memory space.

A | Simulation Configuration

Tables A.1 and A.2 detail the baseline and MPU hardware configuration and power model paramemters, respectively.

Configuration	Value
SKY Config	4 cores, 4-issue, 3.5 GHz, 32KB 8-way L1D, 256 KB 8-way L2, 8MB 16-way shared L3
MPU Config	128 cores, in-order, 500 MHz, 16 KB private L1D & L1-I
MPU latencies	1 cycle hit, 20 cycle miss (40ns), 1 cycle non-memory insts,25-cycle out-of-vault latency
MPU power	0.0056 dynamic [26], 0.0014 static (per core),0.03 SRAM static power [23]

Table A.1: Hardware Configuration

Parameter	Value
SKY Power Factor	1.1; 1-core DynamicPower=1.1*IPC [40]
SKY Static Power(W)	11.88; 2.97W per core x 4 [40, 41]; assuming 35% uncore.
SRAM Static Power(W)	0.03 lstp devices assumed [23]
DDR3 Static Power(W)	2.5; static power at 12.8 GB/sec [42]
DDR3 Access Energy(nJ)	23.3 per 64-byte; 70 pJ/bit at 12.8 GB/sec & 2.5W static power
HMC Static Power(W)	6; all 4 HMC links ON [15]
HMC External Access Energy (nJ)	3.06; per 64-byte access at 5.98 pJ/bit
HMC Internal Access Energy (nJ)	1.95; per 64-byte access, 3.8 pJ/bit [15]
HMC Internal Rd b/w(GB/sec)	160 [24]
HMC External Rd b/w(GB/sec)	80; Assuming all 4 HMC links are ON [24]

Table A.2: Power Data

Bibliography

[1] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent ram," *Micro, IEEE*, vol. 17, pp. 34–44, Mar 1997.

[2] C. Kozyrakis, J. Gebis, D. Martin, S. Williams, I. Mavroidis, S. Pope, D. Jones, and D. Patterson, "Vector iram a media enhanced vector processor with embedded dram," in *Hotchips 12*, 2000.

[3] D. Elliott, W. Snelgrove, and M. Stumm, "Computational ram: A memory-simd hybrid and its application to dsp," in *Custom Integrated Circuits Conference, 1992., Proceedings of the IEEE 1992*, pp. 30.6.1–30.6.4, May 1992.

[4] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, "Flexram: toward an advanced intelligent memory system," in *Computer Design, 1999. (ICCD '99) International Conference on*, pp. 192–201, 1999.

[5] J. Torrellas, "Flexram: Toward an advanced intelligent memory system: A retrospective paper," *2012 IEEE 30th International Conference on Computer Design (ICCD)*, vol. 0, pp. 3–4, 2012.

[6] B. B. Fraguela, J. Renau, P. Feautrier, D. Padua, and J. Torrellas, "Programming the flexram parallel intelligent memory system," in *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, (New York, NY, USA), pp. 49–60, ACM, 2003.

[7] M. Oskin, F. T. Chong, and T. Sherwood, "Active pages: A computation model for intelligent memory," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ISCA '98, (Washington, DC, USA), pp. 192–203, IEEE Computer Society, 1998.

[8] M. Oskin, J. Hensley, D. Keen, F. Chong, M. Farrens, and A. Chopra, "Exploiting ilp in page-based intelligent memory," in *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*, pp. 208–218, 1999.

[9] J. T. Draper, J. Chame, M. W. Hall, C. S. Steele, T. Barrett, J. LaCoss, J. J. Granacki, J. Shin, C. Chen, C. W. Kang, and I. K. Gokhan, "The architecture of the diva processing-in-memory chip.," in *ICS*, pp. 14–25, 2002.

[10] A. Gutierrez, M. Cieslak, B. Giridhar, R. G. Dreslinski, L. Ceze, and T. Mudge, "Integrated 3d-stacked server designs for increasing physical density of key-value stores," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (New York, NY, USA), pp. 485–498, ACM, 2014.

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

[12] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13, Oct 2016.

[13] T. Kgil, S. D'Souza, A. Saidi, N. Binkert, R. Dreslinski, T. Mudge, S. Reinhardt, and K. Flautner, "Picoserver: Using 3d stacking technology to enable a compact energy efficient chip multiprocessor," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, (New York, NY, USA), pp. 117–128, ACM, 2006.

[14] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, “Top-pim: Throughput-oriented programmable processing in memory,” in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC ’14, (New York, NY, USA), pp. 85–98, ACM, 2014.

[15] S. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, “Ndc: Analyzing the impact of 3d-stacked memory+logic devices on mapreduce workloads,” in *Proceedings of ISPASS*, 2014.

[16] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, “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, (New York, NY, USA), pp. 336–348, ACM, 2015.

[17] R. Dreslinski, D. Fick, B. Giridhar, G. Kim, S. Seo, M. Fojtik, S. Satpathy, Y. Lee, D. Kim, N. Liu, M. Wieckowski, G. Chen, D. Sylvester, D. Blaauw, and T. Mudge, “Centip3de: A 64-core, 3d stacked near-threshold system,” *Micro, IEEE*, vol. 33, pp. 8–16, March 2013.

[18] T. Vijayaraghavan, Y. Eckert, G. H. Loh, M. J. Schulte, M. Ignatowski, B. M. Beckmann, W. C. Brantley, J. L. Greathouse, W. Huang, A. Karunanithi, O. Kayiran, M. Meswani, I. Paul, M. Poremba, S. Raasch, S. K. Reinhardt, G. Sadowski, and V. Sridharan, “Design and analysis of an apu for exascale computing,” in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 85–96, Feb 2017.

[19] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, “Graphpim: Enabling instruction-level pim offloading in graph computing frameworks,” in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 457–468, Feb 2017.

[20] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, “Tetris: Scalable and efficient neural network acceleration with 3d memory,” in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’17, (New York, NY, USA), pp. 751–764, ACM, 2017.

[21] "Intel xeon phi processors." https://www.intel.com/content/www/us/en/products/processors/xeon-phi/xeon-phi-processors.html.

[22] E. Blem, J. Menon, and K. Sankaralingam, "Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures," in *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2013.

[23] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP Laboratories*, 2009.

[24] T. Pawlowski, "Hybrid memory cube," in *Hotchips*, 2013.

[25] J. Jeddeloh and B. Keeth, "Hybrid memory cube new dram architecture increases density and performance," in *VLSI Technology (VLSIT), 2012 Symposium on*, pp. 87–88, June 2012.

[26] "Xtensa lx3 customizable dpu," *Cadence DataSheet. http://ip.cadence.com/uploads/pdf/LX3.pdf*.

[27] A. L. Shimpi and R. Smith, "The intel ivy bridge (core i7 3770k) review," *Anandtech, http://www.anandtech.com/show/5771/the-intel-ivy-bridge-core-i7-3770k-review/3*.

[28] P. Gratz, K. Sankaralingam, H. Hanson, P. Shivakumar, R. McDonald, S. Keckler, and D. Burger, "Implementation and Evaluation of a Dynamically Routed Processor Operand Network," in *Proceedings of the 1st ACM/IEEE International Symposium on Networks-on-Chip*, May 2007.

[29] A. Kahng, B. Li, L.-S. Peh, and K. Samadi, "Orion 2.0: A fast and accurate noc power and area model for early-stage design space exploration," in *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pp. 423–428, April 2009.

[30] "kmalloc manpage." http://www.makelinux.net/books/lkd2/ch11lev1sec4.

[31] "mbind - linux programmer's manual." http://man7.org/linux/man-pages/man2/mbind.2.html, 2016.

[32] H. M. C. Consortium, "Hybrid memory cube specification 1.0," *Published at http://hybridmemorycube.org/files/specification-download/*, 2013.

[33] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 475–486, ACM, 2013.

[34] "Laboratory for web algorithmics - datasets." http://law.di.unimi.it/datasets.php, Nov. 2015.

[35] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "Q100: The architecture and design of a database processing unit," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (New York, NY, USA), pp. 255–268, ACM, 2014.

[36] "Snort IDS." http://www.snort.org/, May 2015.

[37] R. Lippmann, J. W. Haines, D. J. Fried, J. Korba, and K. Das, "Analysis and results of the 1999 darpa off-line intrusion detection evaluation," in *Recent Advances in Intrusion Detection*, pp. 162–182, 2000.

[38] T. Mytkowicz, M. Musuvathi, and W. Schulte, "Data-parallel finite-state machines," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (New York, NY, USA), pp. 529–542, ACM, 2014.

[39] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, "Mizan: A system for dynamic load balancing in large-scale graph processing," in *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, (New York, NY, USA), pp. 169–182, ACM, 2013.

[40] K. Czechowski, V. W. Lee, E. Grochowski, R. Ronen, R. Singhal, R. Vuduc, and P. Dubey, “Improving the energy efficiency of big cores,” in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA ’14, (Piscataway, NJ, USA), pp. 493–504, IEEE Press, 2014.

[41] Anandtech, “Ivybridge review,” *Published at http://www.anandtech.com/show/5771/the-intel-ivy-bridge-core-i7-3770k-review*, 2014.

[42] K. T. Malladi, B. C. Lee, F. A. Nothaft, C. Kozyrakis, K. Periyathambi, and M. Horowitz, “Towards energy-proportional datacenter memory with mobile dram,” in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA ’12, (Washington, DC, USA), pp. 37–48, IEEE Computer Society, 2012.