A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing

Junwhan Ahn Sungpack Hong[§] Sungjoo Yoo Onur Mutlu[†] Kiyoung Choi
junwhan@snu.ac.kr, sungpack.hong@oracle.com, sungjoo.yoo@gmail.com, onur@cmu.edu, kchoi@snu.ac.kr
Seoul National University [§]Oracle Labs [†]Carnegie Mellon University

Abstract

The explosion of digital data and the ever-growing need for fast data analysis have made in-memory big-data processing in computer systems increasingly important. In particular, large-scale graph processing is gaining attention due to its broad applicability from social science to machine learning. However, scalable hardware design that can efficiently process large graphs in main memory is still an open problem. Ideally, cost-effective and scalable graph processing systems can be realized by building a system whose performance increases proportionally with the sizes of graphs that can be stored in the system, which is extremely challenging in conventional systems due to severe memory bandwidth limitations.

In this work, we argue that the conventional concept of processing-in-memory (PIM) can be a viable solution to achieve such an objective. The key modern enabler for PIM is the recent advancement of the 3D integration technology that facilitates stacking logic and memory dies in a single package, which was not available when the PIM concept was originally examined. In order to take advantage of such a new technology to enable memory-capacity-proportional performance, we design a programmable PIM accelerator for large-scale graph processing called Tesseract. Tesseract is composed of (1) a new hardware architecture that fully utilizes the available memory bandwidth, (2) an efficient method of communication between different memory partitions, and (3) a programming interface that reflects and exploits the unique hardware design. It also includes two hardware prefetchers specialized for memory access patterns of graph processing, which operate based on the hints provided by our programming model. Our comprehensive evaluations using five state-of-the-art graph processing workloads with large real-world graphs show that the proposed architecture improves average system performance by a factor of ten and achieves 87% average energy reduction over conventional systems.

1. Introduction

With the advent of the *big-data* era, which consists of increasingly data-intensive workloads and continuous supply and demand for more data and their analyses, the design of computer systems for efficiently processing large amounts of data has drawn great attention. From the data storage perspective, the current realization of big-data processing is based mostly on secondary storage such as hard disk drives and solid-state drives. However, the continuous effort on improving cost and density of DRAM opens up the possibility of *in-memory* big-data processing. Storing data in main memory achieves orders of magnitude speedup in accessing data compared to conventional disk-based systems, while providing up to terabytes of memory capacity per server. The potential of such an approach in data analytics has been confirmed by both academic and industrial projects, including RAMCloud [46], Pregel [39], GraphLab [37], Oracle TimesTen [44], and SAP HANA [52].

While the software stack for in-memory big-data processing has evolved, developing a hardware system that efficiently handles a large amount of data in main memory still remains as an open question. There are two key challenges determining the performance of such systems: (1) how fast they can process each item and request the next item from memory, and (2) how fast the massive amount of data can be delivered from memory to computation units. Unfortunately, traditional computer architectures composed of heavy-weight cores and large on-chip caches are tailored for neither of these two challenges, thereby experiencing severe underutilization of existing hardware resources [10].

In order to tackle the first challenge, recent studies have proposed specialized on-chip accelerators for a limited set of operations [13, 30, 34, 59]. Such accelerators mainly focus on improving core efficiency, thereby achieving better performance and energy efficiency compared to general-purpose cores, at the cost of generality. For example, Widx [30] is an on-chip accelerator for hash index lookups in main memory databases, which can be configured to accelerate either hash computation, index traversal, or output generation. Multiple Widx units can be used to exploit memory-level parallelism without the limitation of instruction window size, unlike conventional out-of-order processors [43].

Although specialized on-chip accelerators provide the benefit of computation efficiency, they impose a more fundamental challenge: *system performance does not scale well with the increase in the amount of data per server (or main memory capacity per server).* This is because putting more accelerators provides speedup as long as the memory bandwidth is sufficient to feed them all. Unfortunately, memory bandwidth remains almost constant irrespective of memory capacity due to the pin count limitation per chip. For instance, Kocberber *et al.* [30] observe that using more than four index traversal units

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ISCA'15, June 13–17, 2015, Portland, OR, USA
© 2015 ACM. ISBN 978-1-4503-3402-0/15/06$15.00
DOI: http://dx.doi.org/10.1145/2749469.2750386

in Widx may not provide additional speedup due to off-chip bandwidth limitations. This implies that, in order to process twice the amount of data with the same performance, one needs to double the number of servers (which keeps memory bandwidth per unit data constant by limiting the amount of data in a server), rather than simply adding more memory modules to store data. Consequently, such approaches limit the memory capacity per server (or the amount of data handled by a single server) to achieve target performance, thereby leading to a relatively cost-ineffective and likely less scalable design as opposed to one that can enable increasing of memory bandwidth in a node along with more data in a node.

This scalability problem caused by the memory bandwidth bottleneck is expected to be greatly aggravated with the emergence of increasingly memory-intensive big-data workloads. One of the representative examples of this is large-scale graph analysis [12, 16, 17, 37, 39, 51, 58], which has recently been studied as an alternative to relational database based analysis for applications in, for example, social science, computational biology, and machine learning. Graph analysis workloads are known to put more pressure on memory bandwidth due to (1) large amounts of random memory accesses across large memory regions (leading to very limited cache efficiency) and (2) very small amounts of computation per item (leading to very limited ability to hide long memory latencies). These two characteristics make it very challenging to scale up such workloads despite their inherent parallelism, especially with conventional architectures based on large on-chip caches and scarce off-chip memory bandwidth.

In this paper, we show that the processing-in-memory (PIM) can be a key enabler to realize *memory-capacity-proportional* performance in large-scale graph processing under the current pin count limitation. By putting computation units inside main memory, total memory bandwidth for the computation units scales well with the increase in memory capacity (and so does the computational power). Importantly, latency and energy overheads of moving data between computation units and main memory can be reduced as well. And, fortunately, such benefits can be realized in a cost-effective manner today through the 3D integration technology, which effectively combines logic and memory dies, as opposed to the PIM architectures in 1990s, which suffered from the lack of an appropriate technology that could tightly couple logic and memory.

The key contributions of this paper are as follows:

- We study an important domain of in-memory big-data processing workloads, large-scale graph processing, from the computer architecture perspective and show that memory bandwidth is the main bottleneck of such workloads.
- We provide the design and the programming interface of a new programmable accelerator for in-memory graph processing that can effectively utilize PIM using 3D-stacked memory technologies. Our new design is called Tesseract.[1]
- We develop an efficient mechanism for communication between different Tesseract cores based on message passing. This mechanism (1) enables effective hiding of long remote access latencies via the use of non-blocking message passing and (2) guarantees atomic memory updates without requiring software synchronization primitives.
- We introduce two new types of specialized hardware prefetchers that can fully utilize the available memory bandwidth with simple cores. These new designs take advantage of (1) the hints given by our new programming interface and (2) memory access characteristics of graph processing.
- We provide case studies of how five graph processing workloads can be mapped to our architecture and how they can benefit from it. Our evaluations show that Tesseract achieves 10x average performance improvement and 87% average reduction in energy consumption over a conventional high-performance baseline (a four-socket system with 32 out-of-order cores, having 640 GB/s of memory bandwidth), across five different graph processing workloads, including *average teenage follower* [20], *conductance* [17,20], *PageRank* [5,17,20,39], *single-source shortest path* [20,39], and *vertex cover* [17]. Our evaluations use three large input graphs having four to seven million vertices, which are collected from real-world social networks and internet domains.

[1] Tesseract means a four-dimensional hypercube. We named our architecture Tesseract because in-memory computation adds a new dimension to 3D-stacked memory technologies.

2. Background and Motivation

2.1. Large-Scale Graph Processing

A graph is a fundamental representation of relationship between objects. Examples of representative real-world graphs include social graphs, web graphs, transportation graphs, and citation graphs. These graphs often have millions to billions of vertices with even larger numbers of edges, thereby making them difficult to be analyzed at high performance.

In order to tackle this problem, there exist several frameworks for large-scale graph processing by exploiting data parallelism [12, 16, 17, 37, 39, 51, 58]. Most of these frameworks focus on executing computation for different vertices in parallel while hiding synchronization from programmers to ease programmability. For example, the PageRank computation shown in Figure 1 can be accelerated by parallelizing the vertex loops [17] (lines 1–4, 8–13, and 14–18) since computation for each vertex is almost independent of each other. In this style of parallelization, synchronization is necessary to guarantee atomic updates of shared data (`w.next_pagerank` and `diff`) and no overlap between different vertex loops, which are automatically handled by the graph processing frameworks. Such an approach exhibits a high degree of parallelism, which is effective in processing graphs with billions of vertices.

Although graph processing algorithms can be parallelized through such frameworks, there are several issues that make efficient graph processing very challenging. First, graph processing incurs a large number of random memory accesses during neighbor traversal (e.g., line 11 of Figure 1). Second, graph algorithms show poor locality of memory access since

```
1  for (v: graph.vertices) {
2    v.pagerank = 1 / graph.num_vertices;
3    v.next_pagerank = 0.15 / graph.num_vertices;
4  }
5  count = 0;
6  do {
7    diff = 0;
8    for (v: graph.vertices) {
9      value = 0.85 * v.pagerank / v.out_degree;
10     for (w: v.successors) {
11       w.next_pagerank += value;
12     }
13   }
14   for (v: graph.vertices) {
15     diff += abs(v.next_pagerank - v.pagerank);
16     v.pagerank = v.next_pagerank;
17     v.next_pagerank = 0.15 / graph.num_vertices;
18   }
19 } while (diff > e && ++count < max_iteration);
```

Figure 1: Pseudocode of PageRank computation.

many of them access the entire set of vertices in a graph for each iteration. Third, memory access latency cannot be easily overlapped with computation because of the small amount of computation per vertex [39]. These aspects should be carefully considered when designing a system that can efficiently perform large-scale graph processing.

2.2. Graph Processing on Conventional Systems

Despite its importance, graph processing is a challenging task for conventional systems, especially when scaling to larger amounts of data (i.e., larger graphs). Figure 2 shows a scenario where one intends to improve graph processing performance of a server node equipped with out-of-order cores and DDR3-based main memory by adding more cores. We evaluate the performance of five workloads with 32 or 128 cores and with different memory interfaces (see Section 4 for our detailed evaluation methodology and the description of our systems). As the figure shows, simply increasing the number of cores is ineffective in improving performance significantly. Adopting a high-bandwidth alternative to DDR3-based main memory based on 3D-stacked DRAM, called Hybrid Memory Cube (HMC) [22], helps this situation to some extent, however, the speedups provided by using HMCs are far below the expected speedup from quadrupling the number of cores.

However, if we assume that cores can use the internal memory bandwidth of HMCs[2] ideally, i.e., without traversing the off-chip links, we can provide much higher performance by taking advantage of the larger number of cores. This is shown in the rightmost bars of Figure 3. The problem is that such high performance requires a massive amount of memory bandwidth (near 500 GB/s) as shown in Figure 2b. This is beyond the level of what conventional systems can provide under the current pin count limitations. What is worse, such a high amount of memory bandwidth is mainly consumed by random memory accesses over a large memory region, as explained in

[2]The term *internal memory bandwidth* indicates aggregate memory bandwidth provided by 3D-stacked DRAM. In our system composed of 16 HMCs, the internal memory bandwidth is 12.8 times higher than the off-chip memory bandwidth (see Section 4 for details).

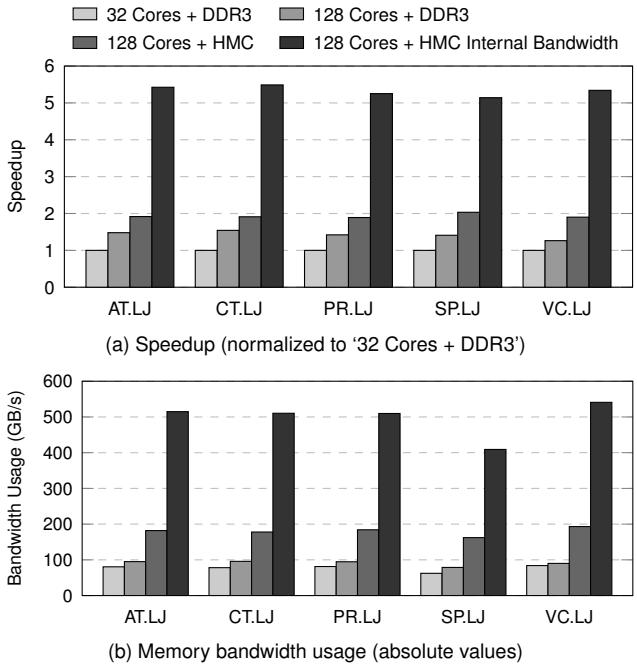

Figure 2: Performance of large-scale graph processing in conventional systems versus with ideal use of the HMC internal memory bandwidth.

Section 2.1, which cannot be efficiently handled by the current memory hierarchies that are based on and optimized for data locality (i.e., large on-chip caches). This leads to the key question that we intend to answer in this paper: *how can we provide such large amounts of memory bandwidth and utilize it for scalable and efficient graph processing in memory?*

2.3. Processing-in-Memory

To satisfy the high bandwidth requirement of large-scale graph processing workloads, we consider moving computation inside the memory, or *processing-in-memory*. The key objective of adopting PIM is not solely to provide high memory bandwidth, but especially to achieve *memory-capacity-proportional* bandwidth. Let us take the Hybrid Memory Cube [24] as a viable baseline platform for PIM. According to the HMC 1.0 specification [22], a single HMC provides up to 320 GB/s of *external* memory bandwidth through eight high-speed serial links. On the other hand, a 64-bit vertical interface for each DRAM partition (or *vault*, see Section 3.1 for details), 32 vaults per cube, and 2 Gb/s of TSV signaling rate [24] together achieve an *internal* memory bandwidth of 512 GB/s per cube. Moreover, this gap between external and internal memory bandwidth becomes much wider as the memory capacity increases with the use of more HMCs. Considering a system composed of 16 8 GB HMCs as an example, conventional processors are still limited to 320 GB/s of memory bandwidth assuming that the CPU chip has the same number of off-chip links as that of an HMC. In contrast, PIM exposes 8 TB/s ($= 16 \times 512\,\mathrm{GB/s}$) of aggregate internal bandwidth to the in-memory computation units. This memory-capacity-proportional bandwidth facili-

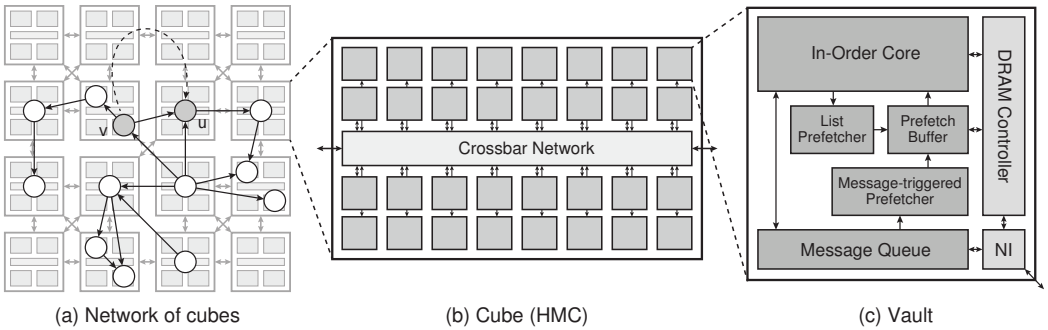

Figure 3: Tesseract architecture (the figure is not to scale).

tates scaling the system performance with increasing amount of data in a cost-effective way, which is a key concern in graph processing systems.

However, introducing a new processing paradigm brings a set of new challenges in designing a whole system. Throughout this paper, we will answer three critical questions in designing a PIM system for graph processing: (1) How to design an architecture that can fully utilize internal memory bandwidth in an energy-efficient way, (2) how to communicate between different memory partitions (i.e., vaults) with a minimal impact on performance, and (3) how to design an expressive programming interface that reflects the hardware design.

3. Tesseract Architecture

3.1. Overview

Organization. Figure 3 shows a conceptual diagram of the proposed architecture. Although Tesseract does not rely on a particular memory organization, we choose the hybrid memory cube having eight 8 Gb DRAM layers (the largest device available in the current HMC specification [22]) as our baseline. An HMC, shown conceptually in Figure 3b is composed of 32 vertical slices (called *vaults*), eight 40 GB/s high-speed serial links as the off-chip interface, and a crossbar network that connects them. Each vault, shown in Figure 3c, is composed of a 16-bank DRAM partition and a dedicated memory controller.[3] In order to perform computation inside memory, a single-issue in-order core is placed at the logic die of each vault (32 cores per cube). In terms of area, a Tesseract core fits well into a vault due to the small size of an in-order core. For example, the area of 32 ARM Cortex-A5 processors including an FPU (0.68 mm^2 for each core [1]) corresponds to only 9.6% of the area of an 8 Gb DRAM die area (e.g., 226 mm^2 [54]).

Host-Tesseract Interface. In the proposed system, host processors have their own main memory (without PIM capability) and Tesseract acts like an accelerator that is memory-mapped to part of a noncacheable memory region of the host processors. This eliminates the need for managing cache coherence between caches of the host processors and the 3D-stacked memory of Tesseract. Also, since in-memory big-data workloads usually do not require many features provided by virtual memory (along with the non-trivial performance overhead of supporting virtual memory) [3], Tesseract does not support virtual memory to avoid the need for address translation *inside* memory. Nevertheless, host processors can still use virtual addressing in their main memory since they use separate DRAM devices (apart from the DRAM of Tesseract) as their own main memory.[4]

Since host processors have access to the entire memory space of Tesseract, it is up to the host processors to distribute input graphs across HMC vaults. For this purpose, the host processors use a customized `malloc` call, which allocates an object (in this case, a vertex or a list of edges) to a specific vault. For example, `numa_alloc_onnode` in Linux (which allocates memory on a given NUMA node) can be extended to allocate memory on a designated vault. This information is exposed to applications since they use a single physical address space over all HMCs. An example of distributing an input graph to vaults is shown in Figure 3a. Algorithms to achieve a balanced distribution of vertices and edges to vaults are beyond the scope of this paper. However, we analyze the impact of better graph distribution on the performance of Tesseract in Section 5.7.

Message Passing (Section 3.2). Unlike host processors that have access to the *entire* address space of the HMCs, each Tesseract core is restricted to access its own *local* DRAM partition only. Thus, a low-cost message passing mechanism is employed for communication between Tesseract cores. For example, vertex `v` in Figure 3a can remotely update a property of vertex `u` by sending a message that contains the target vertex id and the computation that will be done in the remote core (dotted line in Figure 3a). We choose message passing to communicate between Tesseract cores in order to: (1) avoid cache coherence issues among L1 data caches of Tesseract cores, (2) eliminate the need for locks to guarantee atomic updates of shared data, and (3) facilitate the hiding of remote access latencies through asynchronous message communication.

Prefetching (Section 3.3). Although putting a core beneath memory exposes unprecedented memory bandwidth to the

[3] Due to the existence of built-in DRAM controllers, HMCs use a packet-based protocol for communication through the inter-/intra-HMC network instead of low-level DRAM commands as in DDRx protocols.

[4] For this purpose, Tesseract may adopt the direct segment approach [3] and interface its memory as a primary region. Supporting direct segment translation inside memory can be done simply by adding a small direct segment hardware for each Tesseract core and broadcasting the base, limit, and offset values from the host at the beginning of Tesseract execution.

core, a single-issue in-order core design is far from the best way of utilizing this ample memory bandwidth. This is because such a core has to stall on each L1 cache miss. To enable better exploitation of the large amount of memory bandwidth while keeping the core simple, we design two types of simple hardware prefetchers: a list prefetcher and a message-triggered prefetcher. These are carefully tailored to the memory access patterns of graph processing workloads.

Programming Interface (Section 3.4). Importantly, we define a new programming interface that enables the use of our system. Our programming interface is easy to use, yet general enough to express many different graph algorithms.

3.2. Remote Function Call via Message Passing

Tesseract moves computation to the target core that contains the data to be processed, instead of allowing remote memory accesses. For simplicity and generality, we implement computation movement as a remote function call [4, 57]. In this section, we propose two different message passing mechanisms, both of which are supported by Tesseract: blocking remote function call and non-blocking remote function call.

Blocking Remote Function Call. A blocking remote function call is the most intuitive way of accessing remote data. In this mechanism, a local core requests a remote core to (1) execute a specific function remotely and (2) send the return value back to the local core. The exact sequence of performing a blocking remote function call is as follows:

1. The local core sends a packet containing the function address[5] and function arguments[6] to the remote core and waits for its response.
2. Once the packet arrives at the remote vault, the network interface stores function arguments to the special registers visible from the core and emits an interrupt for the core.
3. The remote core executes the function in *interrupt mode*, writes the return value to a special register, and switches back to the normal execution mode.
4. The remote core sends the return value back to the local core.

Note that the execution of a remote function call is *not* preempted by another remote function call in order to guarantee atomicity. Also, cores may temporarily disable interrupt execution to modify data that might be accessed by blocking remote function calls.

This style of remote data access is useful for global state checks. For example, checking the condition '`diff > e`' in line 19 of Figure 1 can be done using this mechanism. However, it may not be the performance-optimal way of accessing remote data because (1) local cores are blocked until responses arrive from remote cores and (2) each remote function call emits an interrupt, incurring the latency overhead of context switching. This motivates the need for another mechanism for remote data access, a *non-blocking* remote function call.

Non-Blocking Remote Function Call. A non-blocking remote function call is semantically similar to its blocking counterpart, except that it cannot have return values. This simple restriction greatly helps to optimize the performance of remote function calls in two ways.

First, a local core can continue its execution after invoking a non-blocking remote function call since the core does not have to wait for the termination of the function. In other words, it allows hiding remote access latency because sender cores can perform their own work while messages are being transferred and processed. However, this makes it impossible to figure out whether or not the remote function call is finished. To simplify this problem, we ensure that all non-blocking remote function calls do not cross synchronization barriers. In other words, results of remote function calls are guaranteed to be visible after the execution of a barrier. Similar consistency models can be found in other parallelization frameworks such as OpenMP [8].

Second, since the execution of non-blocking remote function calls can be delayed, batch execution of such functions is possible by buffering them and executing all of them with a single interrupt. For this purpose, we add a *message queue* to each vault that stores messages for non-blocking remote function calls. Functions in this queue are executed once either the queue is full or a barrier is reached. Batching the execution of remote function calls helps to avoid the latency overhead of context switching incurred by frequent interrupts.

Non-blocking remote function calls are mainly used for updating remote data. For example, updating PageRank values of remote vertices in line 11 of Figure 1 can be implemented using this mechanism. Note that, unlike the original implementation where locks are required to guarantee atomic updates of `w.next_pagerank`, our mechanism eliminates the need for locks or other synchronization primitives since it guarantees that (1) only the local core of vertex `w` can access and modify its property and (2) remote function call execution is not preempted by other remote function calls.

3.3. Prefetching

We develop two prefetching mechanisms to enable each Tesseract core to exploit the high available memory bandwidth.

List Prefetching. One of the most common memory access patterns is sequential accesses with a constant stride. Such access patterns are found in graph processing as well. For example, most graph algorithms frequently traverse the list of vertices and the list of edges for each vertex (e.g., the **`for`** loops in Figure 1), resulting in strided access patterns.

Memory access latency of such a simple access pattern can be easily hidden by employing a stride prefetcher. In this paper, we use a stride prefetcher based on a reference prediction table (RPT) [6] that prefetches multiple cache blocks ahead to utilize the high memory bandwidth. In addition, we modify

[5] We assume that all Tesseract cores store the same code into the same location of their local memory so that function addresses are compatible across different Tesseract cores.

[6] In this paper, we restrict the maximum size of arguments to be 32 bytes, which should be sufficient for general use. We also provide an API to transfer data larger than 32 bytes in Section 3.4.

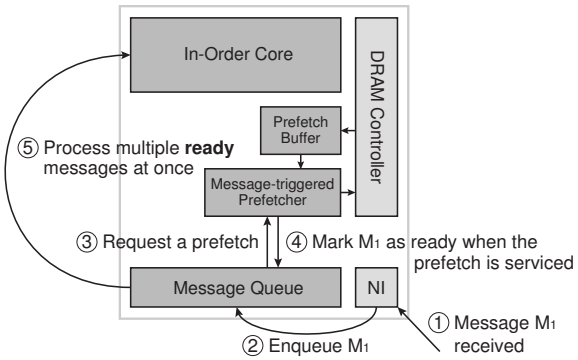

Figure 4: Message-triggered prefetching mechanism.

the prefetcher to accept information about the start address, the size, and the stride of each list from the application software. Such information is recorded in the four-entry list table at the beginning of a loop and is removed from it at the end of the loop. Inside the loop, the prefetcher keeps track of only the memory regions registered in the list table and installs an RPT entry if the observed stride conforms to the hint. An RPT entry is removed once it reaches the end of the memory region.

Message-triggered Prefetching. Although stride prefetchers can cover frequent sequential accesses, graph processing often involves a large amount of random access patterns. This is because, in graph processing, information flows through the edges, which requires pointer chasing over edges toward randomly-located target vertices. Such memory access patterns cannot be easily predicted by stride prefetchers.

Interestingly, most of the random memory accesses in graph processing happen on remote accesses (i.e., neighbor traversal). This motivates the second type of prefetching we devise, called *message-triggered prefetching*, shown in Figure 4. The key idea is to prefetch data that will be accessed during a non-blocking remote function call *before* the execution of the function call. For this purpose, we add an optional field for each non-blocking remote function call packet, indicating a memory address to be prefetched. As soon as a request containing the prefetch hint is inserted into the message queue, the message-triggered prefetcher issues a prefetch request based on the hint and marks the message as ready when the prefetch is serviced. When more than a predetermined number (M_{th}) of messages in the message queue are ready, the message queue issues an interrupt to the core to process the *ready* messages.[7]

Message-triggered prefetching is unique in two aspects. First, it can eliminate processor stalls due to memory accesses inside remote function call execution by processing only ready messages. This is achieved by exploiting the time slack between the arrival of a non-blocking remote function call message and the time when the core starts servicing the message. Second, it can be *exact*, unlike many other prefetching techniques, since graph algorithms use non-blocking remote function calls to send updates over edges, which contain the *exact* memory addresses of the target vertices. For example, a non-blocking remote function call for line 11 of Figure 1 can provide the address of `w.next_pagerank` as a prefetch hint, which is exact information on the address instead of a prediction that can be incorrect.

Prefetch Buffer. The two prefetch mechanisms store prefetched blocks into prefetch buffers [25] instead of L1 caches. This is to prevent the situation where prefetched blocks are evicted from the L1 cache before they are referenced due to the long interval between prefetch requests and their demand accesses. For instance, a cache block loaded by message-triggered prefetching has to wait to be accessed until at least M_{th} messages are ready. Meanwhile, other loads inside the normal execution mode may evict the block according to the replacement policy of the L1 cache. A similar effect can be observed when loop execution with list prefetching is preempted by a series of remote function call executions.

3.4. Programming Interface

In order to utilize the new Tesseract design, we provide the following primitives for programming in Tesseract. We introduce several major API calls for Tesseract: `get`, `put`, `disable_interrupt`, `enable_interrupt`, `copy`, `list_begin`, `list_end`, and `barrier`. Hereafter, we use **`A`** and **`S`** to indicate the memory address type (e.g., `void*` in C) and the size type (e.g., `size_t` in C), respectively.

```
get(id, A func, A arg, S arg_size, A ret, S ret_size)
put(id, A func, A arg, S arg_size, A prefetch_addr)
```

`get` and `put` calls represent blocking and non-blocking remote function calls, respectively. The id of the target remote core is specified by the `id` argument.[8] The start address and the size of the function argument is given by `arg` and `arg_size`, respectively, and the return value (in the case of `get`) is written to the address `ret`. In the case of `put`, an optional argument `prefetch_addr` can be used to specify the address to be prefetched by the message-triggered prefetcher.

```
disable_interrupt()
enable_interrupt()
```

`disable_interrupt` and `enable_interrupt` calls guarantee that the execution of instructions enclosed by them are not preempted by interrupts from remote function calls. This prevents data races between normal execution mode and interrupt mode as explained in Section 3.2.

```
copy(id, A local, A remote, S size)
```

The `copy` call implements copying a local memory region to a remote memory region. It is used instead of `get` or `put` commands if the size of transfer exceeds the maximum size of arguments. This command is guaranteed to take effect before the nearest barrier synchronization (similar to the `put` call).

```
list_begin(A address, S size, S stride)
list_end(A address, S size, S stride)
```

[7] If the message queue becomes full or a barrier is reached before M_{th} messages are ready, *all* messages are processed regardless of their readiness.

[8] If a core issues a `put` command with its own id, it can either be replaced by a simple function call or use the same message queue mechanism as in remote messages. In this paper, we insert local messages to the message queue only if message-triggered prefetching (Section 3.3) is available so that the prefetching can be applied to local messages as well.

list_begin and list_end calls are used to update the list table, which contains hints for list prefetching. Programmers can specify the start address of a list, the size of the list, and the size of an item in the list (i.e., stride) to initiate list prefetching.

```
barrier()
```

The barrier call implements a synchronization barrier across all Tesseract cores. One of the cores in the system (predetermined by designers or by the system software) works as a master core to collect the synchronization status of each core.

3.5. Application Mapping

Figure 5 shows the PageRank computation using our programming interface (recall that the original version was shown in Figure 1). We only show the transformation for lines 8–13 of Figure 1, which contain the main computation. list_for is used as an abbreviation of a **for** loop surrounded by list_begin and list_end calls.

```
1  ...
2  count = 0;
3  do {
4    ...
5    list_for (v: graph.vertices) {
6      value = 0.85 * v.pagerank / v.out_degree;
7      list_for (w: v.successors) {
8        arg = (w, value);
9        put(w.id, function(w, value) {
10         w.next_pagerank += value;
11       }, &arg, sizeof(arg), &w.next_pagerank);
12     }
13   }
14   barrier();
15   ...
16 } while (diff > e && ++count < max_iteration);
```

Figure 5: PageRank computation in Tesseract (corresponding to lines 8–13 in Figure 1).

Most notably, remote memory accesses for updating the next_pagerank field are transformed into put calls. Consequently, unlike the original implementation where every L1 cache miss or lock contention for w.next_pagerank stalls the core, our implementation facilitates cores to (1) continuously issue put commands without being blocked by cache misses or lock acquisition and (2) promptly update PageRank values without stalls due to L1 cache misses through message-triggered prefetching. List prefetching also helps to achieve the former objective by prefetching pointers to the successor vertices (i.e., the list of outgoing edges).

We believe that such transformation is simple enough to be easily integrated into existing graph processing frameworks [12, 16, 37, 39, 51, 58] or DSL compilers for graph processing [17, 20]. This is a part of our future work.

4. Evaluation Methodology

4.1. Simulation Configuration

We evaluate our architecture using an in-house cycle-accurate x86-64 simulator whose frontend is Pin [38]. The simulator has a cycle-level model of many microarchitectural components, including in-order/out-of-order cores considering register/structural dependencies, multi-bank caches with limited numbers of MSHRs, MESI cache coherence, DDR3 controllers, and HMC links. Our simulator runs multithreaded applications by inspecting pthread APIs for threads and synchronization primitives. For Tesseract, it also models remote function calls by intercepting get/put commands (manually inserted into software) and injecting messages into the timing model accordingly. The rest of this subsection briefly describes the system configuration used for our evaluations.

DDR3-Based System. We model a high-performance conventional DDR3-based system with 32 4 GHz four-wide out-of-order cores, each with a 128-entry instruction window and a 64-entry load-store queue (denoted as *DDR3-OoO*). Each socket contains eight cores and all four sockets are fully connected with each other by high-speed serial links, providing 40 GB/s of bandwidth per link. Each core has 32 KB L1 instruction/data caches and a 256 KB L2 cache, and eight cores in a socket share an 8 MB L3 cache. All three levels of caches are non-blocking, having 16 (L1), 16 (L2), and 64 (L3) MSHRs [32]. Each L3 cache is equipped with a feedback-directed prefetcher with 32 streams [56]. The main memory has 128 GB of memory capacity and is organized as two channels per CPU socket, four ranks per channel, eight banks per rank, and 8 KB rows with timing parameters of DDR3-1600 11-11-11 devices [41], yielding 102.4 GB/s of memory bandwidth exploitable by cores.

DDR3-OoO resembles modern commodity servers composed of multi-socket, high-end CPUs backed by DDR3 main memory. Thus, we choose it as the baseline of our evaluations.

HMC-Based System. We use two different types of cores for the HMC-based system: *HMC-OoO*, which consists of the same cores used in DDR3-OoO, and *HMC-MC*, which is comprised of 512 2 GHz single-issue in-order cores (128 cores per socket), each with 32 KB L1 instruction/data caches and no L2 cache. For the main memory, we use 16 8 GB HMCs (128 GB in total, 32 vaults per cube, 16 banks per vault [22], and 256 B pages) connected with the processor-centric topology proposed by Kim *et al.* [29]. The total memory bandwidth exploitable by the cores is 640 GB/s.

HMC-OoO and HMC-MC represent future server designs based on emerging memory technologies. They come with two flavors, one with few high-performance cores and the other with many low-power cores, in order to reflect recent trends in commercial server design.

Tesseract System. Our evaluated version of the Tesseract paradigm consists of 512 2 GHz single-issue in-order cores, each with 32 KB L1 instruction/data caches and a 32-entry message queue (1.5 KB), one for each vault of the HMCs. We conservatively assume that entering or exiting the interrupt mode takes 50 processor cycles (or 25 ns). We use the same number of HMCs (128 GB of main memory capacity) as that of the HMC-based system and connect the HMCs with the Dragonfly topology as suggested by previous work [29]. Each vault provides 16 GB/s of internal memory bandwidth to the

Tesseract core, thereby reaching 8 TB/s of total memory bandwidth exploitable by Tesseract cores. We do not model the host processors as computation is done entirely inside HMCs without intervention from host processors.

For our prefetching schemes, we use a 4 KB 16-way set-associative prefetch buffer for each vault. The message-triggered prefetcher handles up to 16 prefetches and triggers the message queue to start processing messages when more than 16 ($= M_{\text{th}}$) messages are ready. The list prefetcher is composed of a four-entry list table and a 16-entry reference prediction table (0.48 KB) and is set to prefetch up to 16 cache blocks ahead. M_{th} and the prefetch distance of the list prefetcher are determined based on our experiments on a limited set of configurations. Note that comparison of our schemes against other software prefetching approaches is hard to achieve because Tesseract is a message-passing architecture (i.e., each core can access its local DRAM partition only), and thus, existing mechanisms require significant modifications to be applied to Tesseract to prefetch data stored in remote memory.

4.2. Workloads

We implemented five graph algorithms in C++. Average Teenager Follower (AT) computes the average number of teenage followers of users over k years old [20]. Conductance (CT) counts the number of edges crossing a given partition X and its complement X^c [17, 20]. PageRank (PR) is an algorithm that evaluates the importance of web pages [5, 17, 20, 39]. Single-Source Shortest Path (SP) finds the shortest path from the given source to each vertex [20, 39]. Vertex Cover (VC) is an approximation algorithm for the minimum vertex cover problem [17]. Due to the long simulation times, we simulate only one iteration of PR, four iterations of SP, and one iteration of VC. Other algorithms are simulated to the end.

Since runtime characteristics of graph processing algorithms could depend on the shapes of input graphs, we use three real-world graphs as inputs of each algorithm: *ljournal-2008* from the LiveJournal social site (LJ, $|V| = 5.3$ M, $|E| = 79$ M), *enwiki-2013* from the English Wikipedia (WK, $|V| = 4.2$ M, $|E| = 101$ M), and *indochina-2004* from the country domains of Indochina (IC, $|V| = 7.4$ M, $|E| = 194$ M) [33]. These inputs yield 3–5 GB of memory footprint, which is much larger than the total cache capacity of any system in our evaluations. Although larger datasets cannot be used due to the long simulation times, our evaluation with relatively smaller memory footprints is conservative as it penalizes Tesseract because conventional systems in our evaluations have much larger caches (41 MB in HMC-OoO) than the Tesseract system (16 MB). The input graphs used in this paper are known to share similar characteristics with large real-world graphs in terms of their small diameters and power-law degree distributions [42].[9]

[9] We conducted a limited set of experiments with even larger graphs (*it-2004*, *arabic-2005*, and *uk-2002* [33], $|V| = 41$ M/23 M/19 M, $|E| = 1151$ M/640 M/298 M, 32 GB/18 GB/10 GB of memory footprints, respectively) and observed similar trends in performance and energy efficiency.

5. Evaluation Results

5.1. Performance

Figure 6 compares the performance of the proposed Tesseract system against that of conventional systems (DDR3-OoO, HMC-OoO, and HMC-MC). In this figure, LP and MTP indicate the use of list prefetching and message-triggered prefetching, respectively. The last set of bars, labeled as GM, indicates geometric mean across all workloads.

Our evaluation results show that Tesseract outperforms the DDR3-based conventional architecture (DDR3-OoO) by 9x even without prefetching techniques. Replacing the DDR3-based main memory with HMCs (HMC-OoO) and using many in-order cores instead of out-of-order cores (HMC-MC) bring only marginal performance improvements over the conventional systems.

Our prefetching mechanisms, when employed together, enable Tesseract to achieve a 14x average performance improvement over the DDR3-based conventional system, while minimizing the storage overhead to less than 5 KB per core (see Section 4.1). Message-triggered prefetching is particularly effective in graph algorithms with large numbers of neighbor accesses (e.g., CT, PR, and SP), which are difficult to handle efficiently in conventional architectures.

The reason why conventional systems fall behind Tesseract is that they are limited by the low off-chip link bandwidth (102.4 GB/s in DDR3-OoO or 640 GB/s in HMC-OoO/-MC) whereas our system utilizes the large internal memory bandwidth of HMCs (8 TB/s).[10] Perhaps more importantly, such bandwidth discrepancy becomes even more pronounced as the main memory capacity per server gets larger. For example, doubling the memory capacity linearly increases the memory bandwidth in our system, while the memory bandwidth of the conventional systems remains the same.

To provide more insight into the performance improvement of Tesseract, Figure 7 shows memory bandwidth usage and average memory access latency of each system (we omit results for workloads with WK and IC datasets for brevity). As the figure shows, the amount of memory bandwidth utilized by Tesseract is in the order of several TB/s, which is clearly beyond the level of what conventional architectures can reach even with advanced memory technologies. This, in turn, greatly affects the average memory access latency, leading to a 96% lower memory access latency in our architecture compared to the DDR3-based system. This explains the main source of the large speedup achieved by our system.

Figure 7a also provides support for our decision to have one-to-one mapping between cores and vaults. Since the total memory bandwidth usage does not reach its limit (8 TB/s),

[10] Although Tesseract also uses off-chip links for remote accesses, moving computation to where data reside (i.e., using the remote function calls in Tesseract) consumes much less bandwidth than fetching data to computation units. For example, the minimum memory access granularity of conventional systems is one cache block (typically 64 bytes), whereas each message in Tesseract consists of a function pointer and small-sized arguments (up to 32 bytes). Sections 5.5 and 5.6 discuss the impact of off-chip link bandwidth on Tesseract performance.

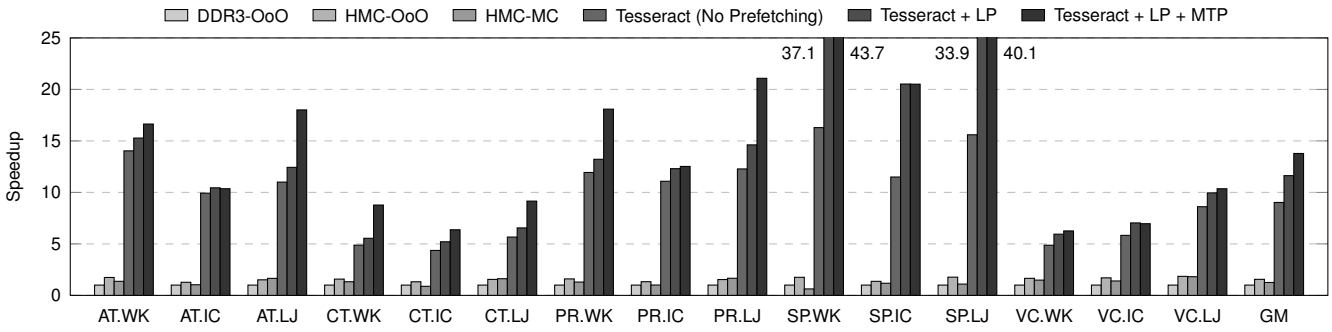

Figure 6: Performance comparison between conventional architectures and Tesseract (normalized to DDR3-OoO).

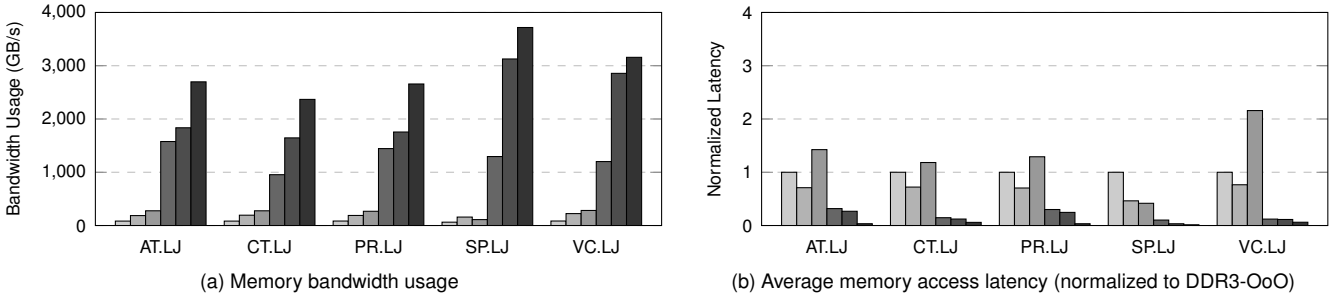

Figure 7: Memory characteristics of graph processing workloads in conventional architectures and Tesseract.

allocating multiple vaults to a single core could cause further imbalance between computation power and memory bandwidth. Also, putting more than one core per vault complicates the system design in terms of higher thermal density, degraded quality of service due to sharing of one memory controller between multiple cores, and potentially more sensitivity to placement of data. For these reasons, we choose to employ one core per vault.

5.2. Iso-Bandwidth Comparison of Tesseract and Conventional Architectures

In order to dissect the performance impact of increased memory bandwidth and our architecture design, we perform idealized limit studies of two new configurations: (1) HMC-MC utilizing the internal memory bandwidth of HMCs *without* off-chip bandwidth limitations (called *HMC-MC + PIM BW*) and (2) Tesseract, implemented on the host side, leading to severely constrained by off-chip link bandwidth (called *Tesseract + Conventional BW*). The first configuration shows the ideal performance of conventional architectures without any limitation due to off-chip bandwidth. The second configuration shows the performance of Tesseract if it were limited by conventional off-chip bandwidth. Note that HMC-MC has the same core and cache configuration as that of Tesseract. For fair comparison, prefetching mechanisms of Tesseract are disabled. We also show the performance of regular HMC-MC and Tesseract (the leftmost and the rightmost bars in Figure 8).

As shown in Figure 8, simply increasing the memory bandwidth of conventional architectures is not sufficient for them to reach the performance of Tesseract. Even if the memory bandwidth of HMC-MC is artificially provisioned to the level of Tesseract, Tesseract still outperforms HMC-MC by 2.2x even without prefetching. Considering that HMC-MC has the same number of cores and the same cache capacity as those of Tesseract, we found that this improvement comes from our programming model that can overlap long memory access latency with computation through non-blocking remote function calls. The performance benefit of our new programming model is also confirmed when we compare the performance of *Tesseract + Conventional BW* with that of HMC-MC. We observed that, under the conventional bandwidth limitation, Tesseract provides 2.3x the performance of HMC-MC, which is 2.8x less speedup compared to its PIM version. This implies that the use of PIM and our new programming model are roughly of equal importance in achieving the performance of Tesseract.

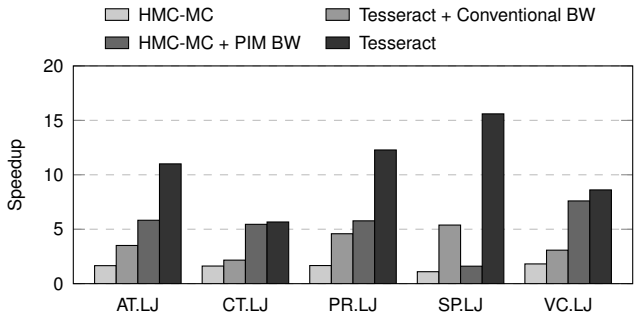

Figure 8: HMC-MC and Tesseract under the same bandwidth.

5.3. Execution Time Breakdown

Figure 9 shows execution time broken down into each operation in Tesseract (with prefetching mechanisms), averaged over all cores in the system. In many workloads, execution in normal execution mode and interrupt mode dominates the

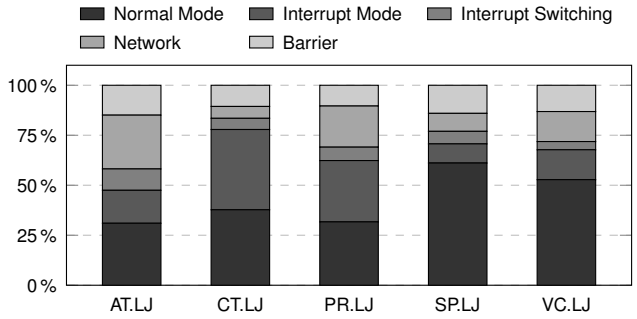

Figure 9: Execution time breakdown of our architecture.

total execution time. However, in some applications, up to 26% of execution time is spent on waiting for network due to a significant amount of off-chip communication caused by neighbor traversal. Since neighbor traversal uses *non-blocking* remote function calls, the time spent is essentially due to the network backpressure incurred as a result of limited network bandwidth. In Sections 5.6 and 5.7, we show how this problem can be mitigated by either increased off-chip bandwidth or better graph distribution schemes.

In addition, some workloads spend notable execution time waiting for barrier synchronization. This is due to workload imbalance across cores in the system. This problem can be alleviated by employing better data mapping (e.g., graph partitioning based vertex distribution, etc.), which is orthogonal to our proposed system.

5.4. Prefetch Efficiency

Figure 10 shows two metrics to evaluate the efficiency of our prefetching mechanisms. First, to evaluate prefetch timeliness, it compares our scheme against an ideal one where all prefetches are serviced instantly (in zero cycles) without incurring DRAM contention. Second, it depicts prefetch coverage, i.e., ratio of prefetch buffer hits over all L1 cache misses.

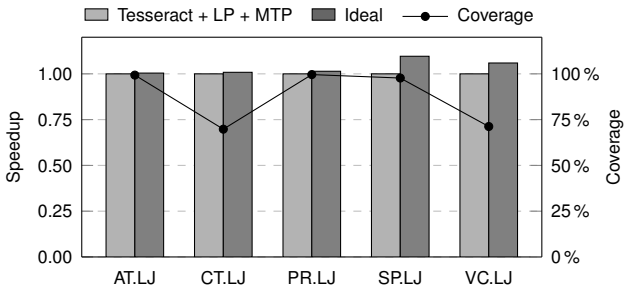

Figure 10: Efficiency of our prefetching mechanisms.

We observed that our prefetching schemes perform within 1.8% of their ideal implementation with perfect timeliness and no bandwidth contention. This is very promising, especially considering that pointer chasing in graph processing is not a prefetch-friendly access pattern. The reason why our message-triggered prefetching shows such good timeliness is that it utilizes the slack between message arrival time and message processing time. Thus, as long as there is enough slack, our proposed schemes can fully hide the DRAM access latency. Our experimental results indicate that, on average, each message stays in the message queue for 1400 Tesseract core cycles (i.e., 700 ns) before they get processed (not shown), which is much longer than the DRAM access latency in most cases.

Figure 10 also shows that our prefetching schemes cover 87% of L1 cache misses, on average. The coverage is high because our prefetchers tackle two major sources of memory accesses in graph processing, namely vertex/edge list traversal and neighbor traversal, with *exact* information from domain-specific knowledge provided as software hints.

We conclude that the new prefetching mechanisms can be very effective in our Tesseract design for graph processing workloads.

5.5. Scalability

Figure 11 evaluates the scalability of Tesseract by measuring the performance of 32/128/512-core systems (i.e., systems with 8/32/128 GB of main memory in total), normalized to the performance of the 32-core Tesseract system. Tesseract provides nearly ideal scaling of performance when the main memory capacity is increased from 8 GB to 32 GB. On the contrary, further quadrupling the main memory capacity to 128 GB shows less optimal performance compared to ideal scaling. The cause of this is that, as more cubes are added into our architecture, off-chip communication overhead becomes more dominant due to remote function calls. For example, as the number of Tesseract cores increases from 128 to 512, the average bandwidth consumption of the busiest off-chip link in Tesseract increases from 8.5 GB/s to 17.2 GB/s (i.e., bandwidth utilization of the busiest link increases from 43% to 86%) in the case of AT.LJ. However, it should be noticed that, despite this sublinear performance scaling, increasing the main memory capacity widens the performance gap between conventional architectures and ours even beyond 128 GB since conventional architectures do not scale well with the increasing memory capacity. We believe that optimizing the off-chip network and data mapping will further improve scalability of our architecture. We discuss these in the next two sections.

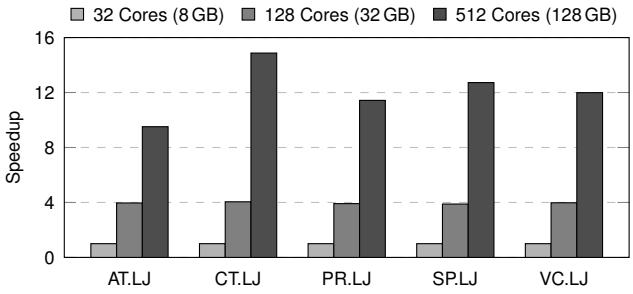

Figure 11: Performance scalability of Tesseract.

5.6. Effect of Higher Off-Chip Network Bandwidth

The recent HMC 2.0 specification boosts the off-chip memory bandwidth from 320 GB/s to 480 GB/s [23]. In order to evaluate the impact of such an increased off-chip bandwidth on both conventional systems and Tesseract, we evaluate HMC-OoO and Tesseract with 50% higher off-chip bandwidth. As shown

in Figure 12, such improvement in off-chip bandwidth widens the gap between HMC-OoO and Tesseract in graph processing workloads which intensively use the off-chip network for neighbor traversal. This is because the 1.5x off-chip link bandwidth is still *far below* the memory bandwidth required by large-scale graph processing workloads in conventional architectures (see Figure 7a). However, 1.5x off-chip bandwidth greatly helps to reduce network-induced stalls in Tesseract, enabling even more efficient utilization of internal memory bandwidth. We observed that, with this increase in off-chip link bandwidth, graph processing in Tesseract scales better to 512 cores (not shown: 14.9x speedup resulting from 16x more cores, going from 32 to 512 cores).

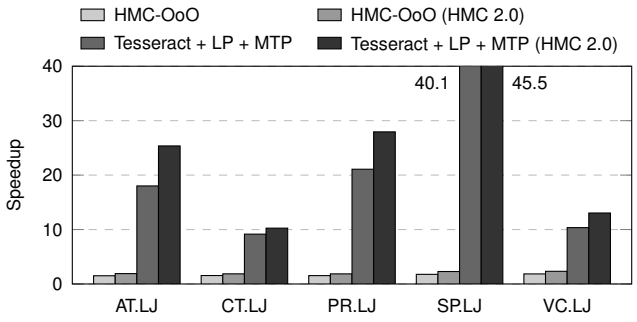

Figure 12: System performance under HMC 2.0 specification.

5.7. Effect of Better Graph Distribution

Another way to improve off-chip transfer efficiency is to employ better data partitioning schemes that can minimize communication between different vaults. In order to analyze the effect of data partitioning on system performance, Figure 13 shows the performance improvement of Tesseract when the input graphs are distributed across vaults based on graph partitioning algorithms. For this purpose, we use METIS [27] to perform 512-way multi-constraint partitioning to balance the number of vertices, outgoing edges, and incoming edges of each partition, as done in a recent previous work [51]. The evaluation results do not include the execution time of the partitioning algorithm to clearly show the impact of graph distribution on graph analysis performance.

Employing better graph distribution can further improve the performance of Tesseract. This is because graph partitioning minimizes the number of edges crossing between different partitions (53% fewer edge cuts compared to random partitioning in LJ), and thus, reduces off-chip network traffic for remote function calls. For example, in AT.LJ, the partitioning scheme eliminates 53% of non-blocking remote function calls compared to random partitioning (which is our baseline).

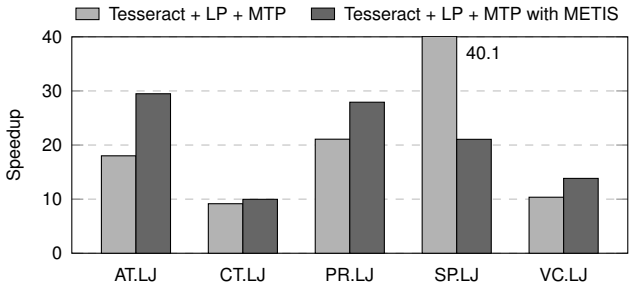

Figure 13: Performance improvement after graph partitioning.

However, in some workloads, graph partitioning shows only small performance improvement (CT.LJ) or even degrades performance (SP.LJ) over random partitioning. This is because graph partitioning algorithms are unaware of the amount of work per vertex, especially when it changes over time. As a result, they can exacerbate the workload imbalance across vaults. A representative example of this is the shortest path algorithm (SP.LJ), which skips computation for vertices whose distances did not change during the last iteration. This algorithm experiences severe imbalance at the beginning of execution, where vertex updates happen mostly within a single partition. This is confirmed by the observation that Tesseract with METIS spends 59% of execution time waiting for synchronization barriers. This problem can be alleviated with migration-based schemes, which will be explored in our future work.

5.8. Energy/Power Consumption and Thermal Analysis

Figure 14 shows the normalized energy consumption of HMCs in HMC-based systems including Tesseract. We model the power consumption of logic/memory layers and Tesseract cores by leveraging previous work [48], which is based on Micron's disclosure, and scaling the numbers as appropriate for our configuration. Tesseract consumes 87% less average energy compared to conventional HMC-based systems with out-of-order cores, mainly due to its shorter execution time. The dominant portion of the total energy consumption is from the SerDes circuits for off-chip links in both HMC-based systems and Tesseract (62% and 45%, respectively), while Tesseract cores contribute 15% of the total energy consumption.

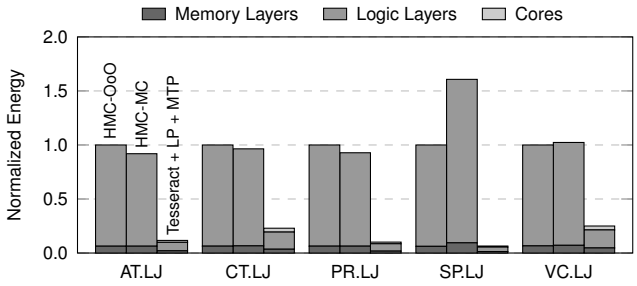

Figure 14: Normalized energy consumption of HMCs.

Tesseract increases the average power consumption (not shown) by 40% compared to HMC-OoO mainly due to the in-order cores inside it and the higher DRAM utilization. Although the increased power consumption may have a negative impact on device temperature, the power consumption is expected to be still within the power budget according to a recent industrial research on thermal feasibility of 3D-stacked PIM [9]. Specifically, assuming that a logic die of the HMC has the same area as an 8 Gb DRAM die (e.g., 226 mm^2 [54]), the highest power density of the logic die across all workloads in our experiments is 94 mW/mm^2 in Tesseract, which remains below the maximum power density that does not re-

quire faster DRAM refresh using a passive heat sink (i.e., 133 mW/mm^2 [9]).

We conclude that Tesseract is thermally feasible and leads to greatly reduced energy consumption on state-of-the-art graph processing workloads.

6. Related Work

To our knowledge, this paper provides the first comprehensive accelerator proposal for large-scale graph processing using the concept of processing-in-memory. We provide a new programming model, system design, and prefetching mechanisms for graph processing workloads, along with extensive evaluations of our proposed techniques. This section briefly discusses related work in PIM, 3D stacking, and architectures for data-intensive workloads.

Processing-in-Memory. Back in the 1990s, several researchers proposed to put computation units inside memory to overcome the memory wall [11, 14, 26, 31, 45, 47]. At the time, the industry moved toward increasing the off-chip memory bandwidth instead of adopting the PIM concept due to costly integration of computation units inside memory dies. Our architecture takes advantage of a much more realizable and cost-effective integration of processing and memory based on 3D stacking (e.g., the hybrid memory cube).

No prior work on processing-in-memory examined large-scale graph processing, which is not only commercially important but also extremely desirable for processing-in-memory as we have shown throughout this paper.

Other than performing computation inside memory, a few prior works examined the possibility of placing prefetchers near memory [21, 55, 60]. Our two prefetching mechanisms, which are completely in memory, are different from such approaches in that (1) prior works are still limited by the off-chip memory bandwidth, especially when prefetched data are sent to host processors and (2) our message-triggered prefetching enables *exact* prefetching through tight integration with our programming interface.

PIM based on 3D Stacking. With the advancement of 3D integration technologies, the PIM concept is regaining attention as it becomes more realizable [2, 36]. In this context, it is critical to examine specialized PIM systems for important domains of applications [2, 48, 53, 62, 63].

Pugsley *et al.* [48] evaluated the PIM concept with MapReduce workloads. Since their architecture does not support communication between PIM cores, only the *map* phase is handled inside memory while the *reduce* phase is executed on host processors. Due to this reason, it is not possible to execute graph processing workloads, which involve a significant amount of communication between PIM cores, with their architecture. On the contrary, Tesseract is able to handle MapReduce workloads since our programming interface provides sufficient flexibility for describing them.

Zhang *et al.* [61] proposed to integrate GPGPUs with 3D-stacked DRAM for in-memory computing. However, their approach lacks a communication mechanism between multiple PIM devices, which is important for graph processing, as we showed in Section 5. Moreover, specialized in-order cores are more desirable in designing a PIM architecture for large-scale graph processing over high-end processors or GPGPUs. This is because such workloads require stacked DRAM capacity to be maximized under a stringent chip thermal constraint for cost-effectiveness, which in turn necessitates minimizing the power consumption of in-memory computation units.

Zhu *et al.* [62, 63] developed a 3D-stacked logic-in-memory architecture for data-intensive workloads. In particular, they accelerated sparse matrix multiplication and mapped graph processing onto their architecture by formulating several graph algorithms using matrix operations. Apart from the fact that sparse matrix operations may not be the most efficient way of expressing graph algorithms, we believe that our architecture can also employ a programming model like theirs, if needed, due to the generality of our programming interface.

Architectures for Big-Data Processing. Specialized accelerators for database systems [7, 30, 59], key-value stores [34], and stream processing [49] have also been developed. Several studies have proposed 3D-stacked system designs targeting memory-intensive server workloads [13, 28, 35, 50]. Tesseract, in contrast, targets large-scale graph processing. We develop an efficient programming model for scalable graph processing and design two prefetchers specialized for graph processing by leveraging our programming interface.

Some works use GPGPUs to accelerate graph processing [15, 18, 19, 40]. While a GPU implementation provides a performance advantage over CPU-based systems, the memory capacity of a commodity GPGPU may not be enough to store real-world graphs with billions of vertices. Although the use of multiple GPGPUs alleviates this problem to some extent, relatively low bandwidth and high latency of PCIe-based interconnect may not be sufficient for fast graph processing, which generates a massive amount of random memory accesses across the entire graph [40].

7. Conclusion and Future Work

In this paper, we revisit the processing-in-memory concept in the completely new context of (1) cost-effective integration of logic and memory through 3D stacking and (2) emerging large-scale graph processing workloads that require an unprecedented amount of memory bandwidth. To this end, we introduce a programmable PIM accelerator for large-scale graph processing, called Tesseract. Our new system features (1) many in-order cores inside a memory chip, (2) a new message passing mechanism that can hide remote access latency within our PIM design, (3) new hardware prefetchers specialized for graph processing, and (4) a programming interface that exploits our new hardware design. We showed that Tesseract greatly outperforms conventional high-performance systems in terms of both performance and energy efficiency. Perhaps more importantly, Tesseract achieves *memory-capacity-proportional* performance, which is the key to handling increasing amounts of data in a cost-effective manner. We con-

clude that our new design can be an efficient and scalable substrate to execute emerging data-intensive applications with intense memory bandwidth demands.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work was supported in large part by the National Research Foundation of Korea (NRF) grants funded by the Korean government (MEST) (No. 2012R1A2A2A06047297) and the IT R&D program of MKE/KEIT (No. 10041608, Embedded System Software for New Memory-based Smart Devices). Onur Mutlu also acknowledges support from the Intel Science and Technology Center for Cloud Computing, Samsung, Intel, and NSF grants 0953246, 1065112, 1212962, and 1320531.

References

[1] ARM Cortex-A5 Processor. Available: http://www.arm.com/products/processors/cortex-a/cortex-a5.php
[2] R. Balasubramonian *et al.*, "Near-data processing: Insights from a MICRO-46 workshop," *IEEE Micro*, vol. 34, no. 4, pp. 36–42, 2014.
[3] A. Basu *et al.*, "Efficient virtual memory for big memory servers," in *Proc. ISCA*, 2013.
[4] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Trans. Comput. Syst.*, vol. 2, no. 1, pp. 39–59, 1984.
[5] S. Brin and L. Page, "The anatomy of a large-scale hypertextual Web search engine," in *Proc. WWW*, 1998.
[6] T.-F. Chen and J.-L. Baer, "Effective hardware-based data prefetching for high-performance processors," *IEEE Trans. Comput.*, vol. 44, no. 5, pp. 609–623, 1995.
[7] E. S. Chung *et al.*, "LINQits: Big data on little clients," in *Proc. ISCA*, 2013.
[8] L. Dagum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming," *IEEE Comput. Sci. & Eng.*, vol. 5, no. 1, pp. 46–55, 1998.
[9] Y. Eckert *et al.*, "Thermal feasibility of die-stacked processing in memory," in *WoNDP*, 2014.
[10] M. Ferdman *et al.*, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Proc. ASPLOS*, 2012.
[11] M. Gokhale *et al.*, "Processing in memory: The Terasys massively parallel PIM array," *IEEE Comput.*, vol. 28, no. 4, pp. 23–31, 1995.
[12] J. E. Gonzalez *et al.*, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. OSDI*, 2012.
[13] A. Gutierrez *et al.*, "Integrated 3D-stacked server designs for increasing physical density of key-value stores," in *Proc. ASPLOS*, 2014.
[14] M. Hall *et al.*, "Mapping irregular applications to DIVA, a PIM-based data-intensive architecture," in *Proc. SC*, 1999.
[15] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *Proc. HiPC*, 2007.
[16] Harshvardhan *et al.*, "KLA: A new algorithmic paradigm for parallel graph computations," in *Proc. PACT*, 2014.
[17] S. Hong *et al.*, "Green-Marl: A DSL for easy and efficient graph analysis," in *Proc. ASPLOS*, 2012.
[18] S. Hong *et al.*, "Accelerating CUDA graph algorithms at maximum warp," in *Proc. PPoPP*, 2011.
[19] S. Hong *et al.*, "Efficient parallel graph exploration on multi-core CPU and GPU," in *Proc. PACT*, 2011.
[20] S. Hong *et al.*, "Simplifying scalable graph processing with a domain-specific language," in *Proc. CGO*, 2014.
[21] C. J. Hughes and S. V. Adve, "Memory-side prefetching for linked data structures for processor-in-memory systems," *J. Parallel Distrib. Comput.*, vol. 65, no. 4, pp. 448–463, 2005.
[22] "Hybrid memory cube specification 1.0," Hybrid Memory Cube Consortium, Tech. Rep., Jan. 2013.
[23] "Hybrid memory cube specification 2.0," Hybrid Memory Cube Consortium, Tech. Rep., Nov. 2014.
[24] J. Jeddeloh and B. Keeth, "Hybrid memory cube new DRAM architecture increases density and performance," in *Proc. VLSIT*, 2012.
[25] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proc. ISCA*, 1990.
[26] Y. Kang *et al.*, "FlexRAM: Toward an advanced intelligent memory system," in *Proc. ICCD*, 1999.
[27] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, 1998.
[28] T. Kgil *et al.*, "PicoServer: Using 3D stacking technology to enable a compact energy efficient chip multiprocessor," in *Proc. ASPLOS*, 2006.
[29] G. Kim *et al.*, "Memory-centric system interconnect design with hybrid memory cubes," in *Proc. PACT*, 2013.
[30] O. Kocberber *et al.*, "Meet the walkers: Accelerating index traversals for in-memory databases," in *Proc. MICRO*, 2013.
[31] P. M. Kogge, "EXECUBE-a new architecture for scaleable MPPs," in *Proc. ICPP*, 1994.
[32] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proc. ISCA*, 1981.
[33] Laboratory for Web Algorithmics. Available: http://law.di.unimi.it/datasets.php
[34] K. Lim *et al.*, "Thin servers with smart pipes: Designing SoC accelerators for memcached," in *Proc. ISCA*, 2013.
[35] G. H. Loh, "3D-stacked memory architectures for multi-core processors," in *Proc. ISCA*, 2008.
[36] G. H. Loh *et al.*, "A processing-in-memory taxonomy and a case for studying fixed-function PIM," in *WoNDP*, 2013.
[37] Y. Low *et al.*, "Distributed GraphLab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, 2012.
[38] C.-K. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. PLDI*, 2005.
[39] G. Malewicz *et al.*, "Pregel: A system for large-scale graph processing," in *Proc. SIGMOD*, 2010.
[40] D. Merrill *et al.*, "Scalable GPU graph traversal," in *Proc. PPoPP*, 2012.
[41] *2Gb: x4, x8, x16 DDR3 SDRAM*, Micron Technology, 2006.
[42] A. Mislove *et al.*, "Measurement and analysis of online social networks," in *Proc. IMC*, 2007.
[43] O. Mutlu *et al.*, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *Proc. HPCA*, 2003.
[44] Oracle TimesTen in-memory database. Available: http://www.oracle.com/technetwork/database/timesten/
[45] M. Oskin *et al.*, "Active pages: A computation model for intelligent memory," in *Proc. ISCA*, 1998.
[46] J. Ousterhout *et al.*, "The case for RAMClouds: Scalable high-performance storage entirely in DRAM," *ACM SIGOPS Oper. Syst. Rev.*, vol. 43, no. 4, pp. 92–105, 2010.
[47] D. Patterson *et al.*, "Intelligent RAM (IRAM): Chips that remember and compute," in *ISSCC Dig. Tech. Pap.*, 1997.
[48] S. Pugsley *et al.*, "NDC: Analyzing the impact of 3D-stacked memory+logic devices on MapReduce workloads," in *Proc. ISPASS*, 2014.
[49] W. Qadeer *et al.*, "Convolution engine: Balancing efficiency & flexibility in specialized computing," in *Proc. ISCA*, 2013.
[50] P. Ranganathan, "From microprocessors to Nanostores: Rethinking data-centric systems," *IEEE Comput.*, vol. 44, no. 1, pp. 39–48, 2011.
[51] S. Salihoglu and J. Widom, "GPS: A graph processing system," in *Proc. SSDBM*, 2013.
[52] SAP HANA. Available: http://www.saphana.com/
[53] V. Seshadri *et al.*, "RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization," in *Proc. MICRO*, 2013.
[54] M. Shevgoor *et al.*, "Quantifying the relationship between the power delivery network and architectural policies in a 3D-stacked memory device," in *Proc. MICRO*, 2013.
[55] Y. Solihin *et al.*, "Using a user-level memory thread for correlation prefetching," in *Proc. ISCA*, 2002.
[56] S. Srinath *et al.*, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *Proc. HPCA*, 2007.
[57] M. A. Suleman *et al.*, "Accelerating critical section execution with asymmetric multi-core architectures," in *Proc. ASPLOS*, 2009.
[58] Y. Tian *et al.*, "From "think like a vertex" to "think like a graph"," *Proc. VLDB Endow.*, vol. 7, no. 3, pp. 193–204, 2013.
[59] L. Wu *et al.*, "Navigating big data with high-throughput, energy-efficient data partitioning," in *Proc. ISCA*, 2013.
[60] C.-L. Yang and A. R. Lebeck, "Push vs. pull: Data movement for linked data structures," in *Proc. ICS*, 2000.
[61] D. P. Zhang *et al.*, "TOP-PIM: Throughput-oriented programmable processing in memory," in *Proc. HPDC*, 2014.
[62] Q. Zhu *et al.*, "A 3D-stacked logic-in-memory accelerator for application-specific data intensive computing," in *Proc. 3DIC*, 2013.
[63] Q. Zhu *et al.*, "Accelerating sparse matrix-matrix multiplication with 3D-stacked logic-in-memory hardware," in *Proc. HPEC*, 2013.