GraphMMU: Memory Management Unit for Sparse Graph Accelerators

Nachiket Kapre[1], Han Jianglei[1], Andrew Bean[2], Pradeep Moorthy[1], and Siddhartha[1]

[1]School of Computer Engineering, Nanyang Technological University, nachiket@ieee.org
[2]Department of Electrical and Electronic Engineering, Imperial College London, andrew.bean06@imperial.ac.uk

***Abstract*— Memory management units that use low-level AXI descriptor chains to hold irregular graph-oriented access sequences can help improve DRAM memory throughput of graph algorithms by almost an order of magnitude. For the Xilinx Zedboard, we explore and compare the memory throughputs achievable when using (1) cache-enabled CPUs with an OS, (2) cache-enabled CPUs running bare metal code, (2) CPU-based control of FPGA-based AXI DMAs, and finally (3) local FPGA-based control of AXI DMA transfers. For short-burst irregular traffic generated from sparse graph access patterns, we observe a performance penalty of almost 10× due to DRAM row activations when compared to cache-friendly sequential access. When using an AXI DMA engine configured in FPGA logic and programmed in AXI register mode from the CPU, we can improve DRAM performance by as much as 2.4× over naïve random access on the CPU. In this mode, we use the host CPU to trigger DMA transfer by writing appropriate control information in the internal register of the DMA engine. We also encode the sparse graph access patterns as locally-stored BRAM-hosted AXI descriptor chains to drive the AXI DMA engines with minimal CPU involvement under Scatter Gather mode. In this configuration, we deliver an additional 3× speedup, for a cumulative throughput improvement of 7× over a CPU-based approach using caches while running an OS to manage irregular access.**

I. Introduction

Important graph problems in scientific computing and engineering, such as sparse matrix factorization, learning on contextual knowledge-bases, pagerank-style indexing, social network analytics, and neural network simulations are challenging problems for modern computing architectures. A common pattern in these applications involves repetitive, irregular access to large, sparsely distributed graph structures. For small graphs that fit the on-chip memory capacity of modern processors, we get fast random access even for irregular address sequences. However, larger graph structures that do not fit the cache must be stored off-chip. Accessing these structures results in high cache miss rates when trying to support such irregular accesses thereby lowering performance. This is inevitable as the access patterns have neither spatial nor temporal locality. However, we know that the underlying DRAM interfaces are capable of supporting much faster data rates. For example, on the Xilinx Zedboard, when accessing a graph with 32M nodes and edges, the observed random-access DRAM throughput can drop to as low at 50 MB/s as opposed to the sustained sequential-access throughput of ≈600 MB/s that is possible. While several projects have explored customizing hardware accelerators for graph algorithms, few have attempted to directly optimize the memory access aspect of this problem. Careful scheduling and low-level control of the DRAM controls can help us enhance the observed throughputs even for irregular access.

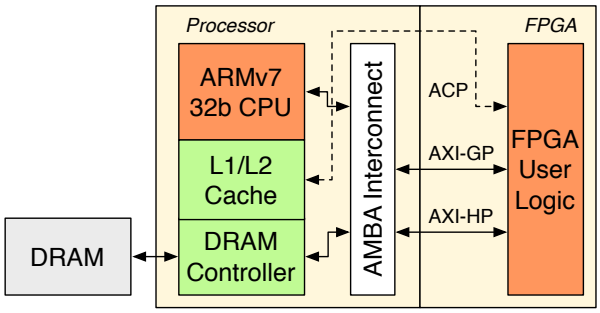


Fig. 1: An Abstract High-Level View of the Zedboard communication infrastructure showing the AMBA interconnect and AXI-HP, AXI-GP and ACP ports

Modern FPGA-based platforms, such as the Xilinx Zedboard, often provide hard-IP blocks for DRAM controllers. This is useful as it helps ensure fast low-latency access that is otherwise trickier to manage when having to embed the memory controllers directly in spatial FPGA logic. The Zedboard platform also supports DMA IPs that can directly co-ordinate and manage access to the DRAMs from FPGA logic. On the Zedboard, we also have ARM co-processors that can help manage control-oriented components of the algorithm while being directly connected to the FPGA and the off-chip DRAM. Data transfer from the ARM subsystem and the FPGA is orchestrated using AXI (ARM bus standard) compatible ports for high-throughput access. While it is possible to provide cache-coherent access between the FPGA and the ARM processor over the ACP port, there are direct AXI links that bypass the ARM memory subsystem entirely when feeding data to the FPGA logic. We hypothesize that we can manage low-level AXI controllers that help support irregular access patterns more efficiently than the alternatives. Hardened DRAM controllers, fast ARMv7 CPU support and specialized AXI-compatible DMA engines can be organized together to provide a competent memory management solution for sparse graph access. We first separate the graph data structure into two components (1) addressing, and (2) data. By sequencing

the addressing through the ARMv7 CPU, the AXI DMA engines can be directed to effectively get data into the FPGA logic for processing.

The contributions in this paper can be summarized as follows:

1) The design and engineering of an `AXI_DMA`-based memory management unit (MMU) for supporting irregular sparse access patterns.
2) Quantification of performance of our design and comparison with traditional cache-enabled approach for managing memory access.
3) Low-level performance optimization of the DMA control by directly constructing a chain of AXI descriptors that capture the required access patterns.

II. Background

In this section, we provide a brief overview of the Zedboard platform, the graph memory storage format and a description of the nature of memory access patterns on sparse graphs.

A. The Zedboard Platform

The Zynq Z7020 SoC [10] is an heterogeneous computing system that combines an ARMv7 32b CPU with an Artix-7 series FPGA accelerator on the same chip. It is a unique platform ideally suited for embedded systems with power and form factor restrictions. Zynq simplifies the FPGA development flow by providing a high degree of IO integration such as USB, Ethernet, HDMI and DRAM controllers by directly implementing the interface IPs as hard blocks. Of particular interest to this study are the embedded DRAM controllers and a variety of fast AXI interfaces for data transfer between the ARM CPUs and the FPGA logic regions. We tabulate the data rates and capabilities of the various interfaces available [10], [7] in Table I and represent the high-level block diagram in Figure 1. For graph problems, we need to support fast, low-latency access to the sparsely distributed memory structures. Ideally, we want to fit all available data fully on-chip. However, on-chip memory capacity is limited to 560KB of BlockRAMs (140×36Kb BRAMs) for the Zedboard Z7020 chip. This is barely enough for most interesting graph problems and even if we consider larger FPGAs, the peak on-chip storage is still limited to a paltry few MBs of state. In these circumstances, it is necessary to store the larger graph structures off-chip in the DRAM. Compared to the on-chip BRAM bandwidth of 0.3 TB/s, the off-chip DRAM bandwidth is a miserly 4.2 GB/s (two-three orders of magnitude less). Hence, it becomes important to consider possible low-level DRAM optimizations that use the already-constrained bandwidth as effectively as possible.

B. Graph Data Storage Format

We can formally define a graph (G) as a collection of vertices (V) connected by edges (E), *i.e.* $G = (V, E)$. When each vertex in the graph is only connected to one or a few neighbouring vertices, the graph is classified as being *sparse*. Figure 2 shows a sparse graph with labeled nodes and edges (label is state on the node or edge). To store the sparse graph structure in memory we can use an adjacency-list based mechanism that only stores a list of edges that are connected. For an optimized storage format, we typically use the compressed sparse row (CSR) format [5] when the physical structure is going to remain stable after construction of the graph. In the CSR format, we store the graph in four arrays as shown in Figure 3. The first array $edge\_offset$ holds the accumulated edge count for all nodes i (number of input edges of node i is $edge\_offset[i+1]$ - $edge\_offset[i]$). We use $j = edge\_offset[i]$ as an offset to locate state belonging to edges of node i in the array $edge\_state$. We can also locate the data stored at the input nodes of node i through $node\_index[j]$ dereference. This format is commonly used for sparse matrix

Interface description	Ports	Bandwidth (GB/s)	
		Total	Per-Port
AXI Accelerator Coherency Port (ACP)	1	2.4	2.4
AXI General Purpose (AXI-GP)	4	4.8	1.2
AXI High Performance Ports (AXI-HP)	4	9.6	2.4
External DDR memory	1	4.2	4.2
On-chip memory (OCM)	1	3.6	3.6

TABLE I: Theoretical Zedboard Interface Bandwidths

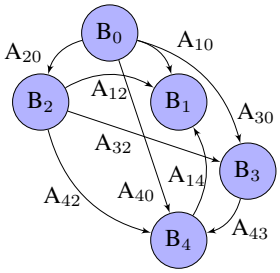


Fig. 2: A representative sparse graph with five nodes and nine edges, nodes labeled with node weights B_i, edges labeled with edge weights A_{ij}

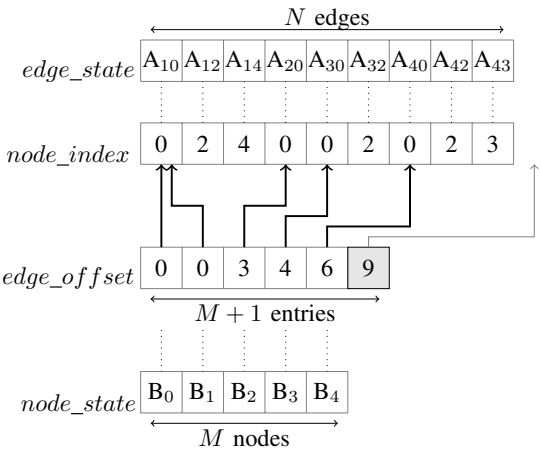


Fig. 3: Memory layout of the sparse graph in Figure 2

Function bsp(node_state, edge_state)

```
   /* multiple BSP iterations                                          */
1  foreach k = iterations do
       /* process all nodes                                           */
2      foreach i = nodes in graph do
           /* evalute all input edges                                 */
3          num_edges = edge_index[i+1]-edge_index[i];
4          foreach j = input edges of node i do
5              /* compute read addresses                              */
6              node = i*sizeof(node_state);
7              edge = (edge_offset[node]+j)*sizeof(edge_state);
               /* compute on node/edge data                           */
8              f(node_state[node],edge_state[edge]);
       /* implicit BSP barrier                                        */
```

vector multiplication algorithms and can be repurposed with little adaptation for other static graph structures.

C. Graph Algorithms

A key feature of most graph algorithms is the need to access nodes and edges in an ordered manner. For graph algorithms that obey the Bulk Synchronous Parallel (BSP) paradigm [8], we iterate over the entire graph structure multiple times. Each node and its neighbours are accessed in a loop-oriented fashion shown in Function bsp. We know how to design spatial datapaths for accelerating these kinds of graph algorithms [1], [4], [6]. The underlying principle is to implement the node and edge calculation logic as pipelined, streaming datapaths. As these datapaths operate at high throughputs and can usually be tiled across the FPGA fabric, we can process multiple node and edge calculations per clock cycle yielding high processing throughput. The key challenge in these spatial implementations is our inability to either hold all active graph data on-chip or to get data at high rates from off-chip DRAMs to keep the hardware blocks busy with useful work.

For these scenarios, we can pre-compute the access sequences once and reuse them across all iterations. In this paper, we focus on such algorithms where the access sequences may be irregular and scattered. A cache-enabled CPU that implements these loops will suffer high cache miss rates due to the inability to fully predict the needed $edge\_state[j]$ locations and prefetch them in the cache ahead of time. In contrast, an alternative approach that directs the DRAMs to load the specified order of addresses will do much better.

The peak rated throughputs shown earlier in Table I do not capture sustained behavior in the real world. We wrote a simple microbenchmark modeled on the BSP code sketch shown in Function bsp and compiled it using `gcc-4.8.3` with the `-O3` optimization. The goal was to stress the off-chip DRAM memory access routines to quantify performance of the Zedboard CPU. We show the sustained sequential access bandwidth throughput possible on the Zynq CPU running an embedded Linux OS when compared to scattered irregular access (generated pseudo-randomly) in Figure 4. It is clear that there is a performance gap of as much as 10× when accessing memory in an irregular manner. Caching works well at predicting sequential accesses resulting in fairly high throughputs across access counts. Even random access rates match the sequential access rates when the number of accesses is in the low 1000s. While we do not expect to completely close this 10× performance gap, we can certainly bypass the cache to support enhanced operation by accessing the DRAM controller in other ways.

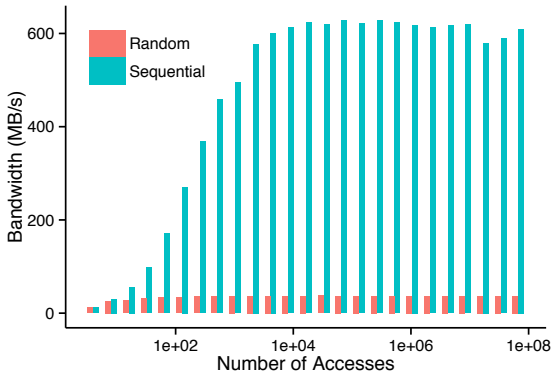


Fig. 4: Comparing Random and Sequential Access Bandwidth on the Zedboard ARMv7 CPUs running Xillinux

III. Design of the Memory Management Unit

We design our MMU for graph operations by (1) using a sparse graph storage encoding that separates structure from data, (2) co-designing the manager using the `AXI_DMA` block supported by a software driver running on the CPU, and (3) optimizing the accesses through careful tuning of low-level AXI operations. The MMU translates virtual graph node and edge indices into appropriate low-level DMA operations.

A. Graph Storage

We organize the sparse graph storage along the lines of the CSR format described earlier, but separate physical structure from the data. As before we need to store $edge\_offset$ and $node\_index$ to identify where the list of edges for each node

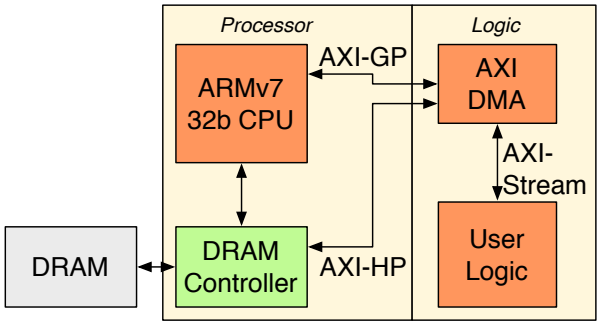


Fig. 5: AXI_DMA with direct control from CPU

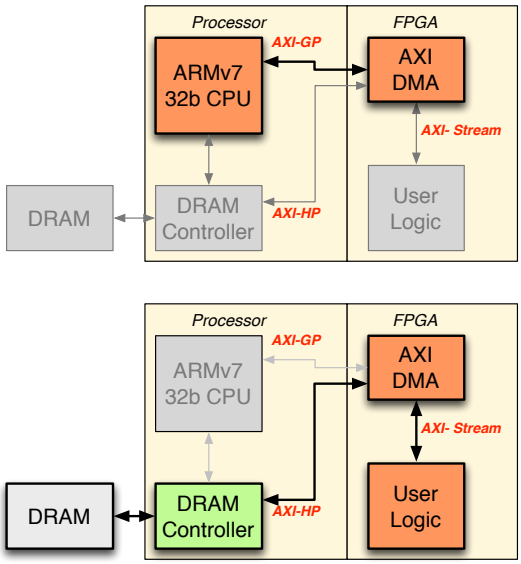


Fig. 6: AXI DMA flow for Register-mode operation

is located. This is useful for accessing *node_state* values corresponding to the neighborhood of a given node. Such accesses are useful when implementing sparse graph algorithms such as shortest-path search that performs summarization operations on the inputs to a node. In this case, we need to get fixed-size items of data from a series of scattered addresses. Another pattern of memory access involves accessing a contiguous set of items that may be edge properties stored along an edge to a node. For example, this information is stored in a sequence as an *edge_state* non-zero vector for the sparse matrix vector multiplication algorithm. We encode the physical address of the node and edge structures as `base_addr` and the size of the data fetched from that address as `length`. For our requirements, we consider a set of fixed-`length` transfers from randomly distributed `base_addr` addresses. While other DRAM-friendly forms are possible with larger variable-`length` transfers, we focus on the harder shorter fixed-`length` scenario.

B. *AXI_DMA Manager*

A common technique to maximize DRAM memory bandwidth is to use DMA (Direct Memory Access) protocol. Xilinx provides a few variations of DMA for different applications *e.g.* VDMA for video traffic, and CDMA for shuffling data between memory-mapped locations. For our graph access scenario, we cannot use these off-the-shelf DMA engines. Instead, we use the basic `AXI_DMA` IP core [9] and specialize it for our purpose. Besides the capability of stream and memory mapped data conversion, it also provides a high level control and configuration over the DMA operation.

We use the `AXI_DMA` IP block, shown as organized in Figure 5, as the basis of our memory throughput optimization study. This IP block interfaces directly with the hardened DRAM controller in the Zynq platform through the AXI-HP interface. The IP can be customized to support a variable

```
XScuGic InterruptController;

void InterruptHandler ( void ) {

  // clear interrupt
  int tmp = Xil_In32(DMAREG_ADDR + 0x04);
  tmp = tmpValue | 0x1000;
  Xil_Out32(DMAREG_ADDR + 0x04, tmp);

  // queue a new DMA operation
}

int main()
{
  // Initialize AXI DMA
  u32 tmp = Xil_In32(DMAREG_ADDR);
  tmp = tmp | 0x1001;
  Xil_Out32(DMAREG_ADDR, tmp);

  // Initialize Interrupt
  InitializeInterruptSystem();

  // Perform DMA
  for(i=0;i<GRAPH_ACCESS;i++) {
    Xil_Out32(DMAREG_ADDR + 0x18,
      base_addr[i]);
    Xil_Out32(DMAREG_ADDR + 0x28,
      length[i]);
    // wait for interrupt
  }
}
```

Fig. 7: Register-Mode `AXI_DMA` device driver

number of burst sizes to enable efficient use of the DRAM bandwidth. For our irregular, short-burst access case, we support burst sizes between 2–16. We can also configure the irregular memory accesses as a sequence of bursts. The IP supports programming through (1) register mode, and (2) scatter-gather mode which we now discuss in more detail.

C. *Register-Mode DMA*

Register-mode control of the `AXI_DMA` block is initiated by the ARMv7 CPU writing DMA-specific metadata to appropriate internal registers of the DMA engine over the AXI-GP control ports shown in Figure 6. The DMA engine interprets this register state to extract the operational details of the DMA task that we intend to perform. Each DMA operation involves access from `base_addr` up to a sequence of `length` bytes. The `AXI_DMA` block executes the DMA command over the AXI-HP port that directly interfaces with the embedded DRAM controller. Once the operation it competed, the engine informs the CPU by signaling an interrupt. The host CPU can the proceed to safely initiate another transfer. For a larger series of accesses, the CPU supplies the appropriate set of register commands to the DMA engine in the desired order.

In Figure 6, we show the two-step sequence that is iteratively evaluated for irregular access. For a graph traversal, we show the pseudocode for the device driver in Figure 7. In the first step, the host CPU sends the appropriate low-level AXI commands to the `AXI_DMA` engine to kickstart the transfer. In the second stage, the engine directly gets the requested data

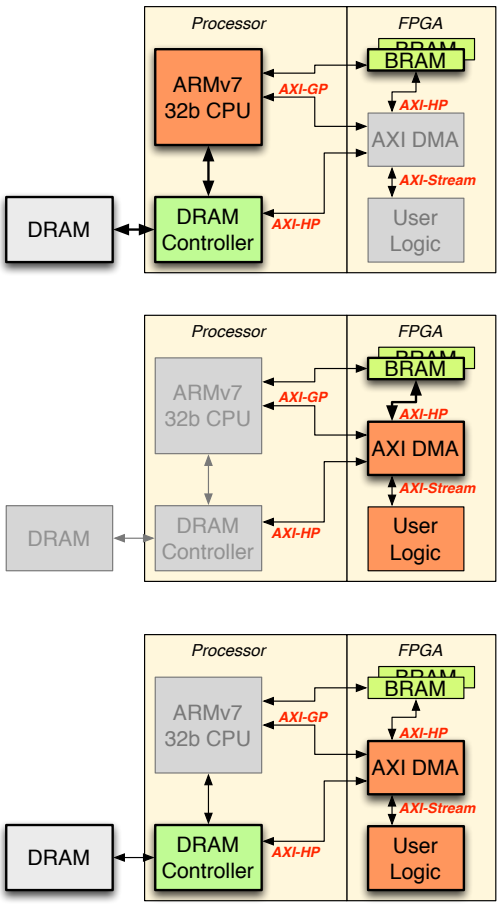


Fig. 8: AXI DMA flow for Scatter-Gather operation

from the off-chip DRAM through the AXI-HP port completely bypassing the host CPU on-chip memories and delivers it to the FPGA processing logic. This ability to avoid the caches and memory subsystem of the CPU, while still using the CPU to drive complex control-heavy series of DMA commands is a unique strength of this execution mode. In this mode, the need to wait for interrupts after each completed DMA transaction can become a performance bottleneck. While the register-mode flow is simple and straightforward, the need for interrupt-locked progress can impose a performance penalty on the accesses. We can remedy this using the scatter-gather mode of operation instead.

D. Scatter-Gather DMA

Scatter-Gather DMA mode allows the `AXI_DMA` engine to avoid requiring frequent assistance from the CPU and enables somewhat independent operation. In this mode, instead of programming the internal registers for each DMA transfer, the CPU only needs to construct a one-off linked list of AXI descriptor commands for the complete series of transfers. This can be done once at the start and reused repeatedly for iterative BSP-like graph algorithms. The descriptor chain is

```
XScuGic InterruptController;

struct axi_desc_t {
  u32 next;
  u32 base_addr;
  u32 control;
  u32 status;
};

void InitializeDescriptors() {

  struct axi_desc_t axi_desc[GRAPH_ACCESSES];

  for (i=0; i<GRAPH_ACCESSES; i++)  {

    // create an entry in linked list
    axi_desc[i].base_addr = base_addr[i];
    axi_desc[i].control = length[i];

    Xil_Out32(BRAM_ADDR +
      i*ALIGN + NXTDESC ,
      axi_desc[i].next );
    // copy other fields to BRAM
  }
}

int main()
{
  InitializeDescriptors ( );

  // Initialize DMA
  Xil_Out32(DMAREG_ADDR +
    MM2S_CURDESC, BRAM_ADDR );
  Xil_Out32(DMAREG_ADDR +
    MM2S_DMASR, 0x0000000);
  Xil_Out32(DMAREG_ADDR +
    MM2S_DMACR, 0x5001);

  // Perform DMA
  Xil_Out32(DMAREG_ADDR
    + MM2S_TAILDESC, BRAM_ADDR +
    (GRAPH_ACCESSES-1) * ALIGN);
}
```

Fig. 9: Scatter-Gather-Mode `AXI_DMA` device driver

stored locally on the FPGA fabric in BlockRAMs and coupled to the `AXI_DMA` engines over an AXI-HP interface.

In Figure 8, we show the slightly more complex three-step configuration flow for the Scatter-Gather DMA mode. As before, we still need to instruct the DMA engine to get `length` bytes starting from `base_addr` location. Instead of forcing an interrupt after each transfer, we are able to perform a set of back-to-back transfers directly without interrupting the host until after the full sequence has been transferred. This ability to avoid frequent CPU interrupts coupled with FPGA-based storage of AXI descriptor chain provides low-latency turnaround times between consecutive DMA transactions. In scatter-gather mode, we represent the irregular list of accesses as a linked list of `<base_addr>,<length>` tuples stored in local on-chip FPGA BlockRAM. This is loaded once at the start over AXI-GP ports from the CPU. We represent this in Figure 9 in the `InitializeDescriptors` function. The address of the next descriptor is specified in each descriptor.

The head and tail descriptors are provided to the DMA engine and it will process one descriptor after another.

IV. Experimental Setup

We use the Xilinx Zedboard with the Z7020 Zynq SoC for our DMA experiments. For most of our experiments, we are operating the ARMv7 CPU in bare-metal mode without any operating system and use the `AXI_DMA` engine under various configurations. We use the ARMv7 CPU to exercise complete low-level control of the various system components through appropriate AXI transactions while the DMA engine provides a high-performance link to the hardened DRAM controller. Our experiments consider an AXI-compatible spatial hardware accelerator reading and writing data within a graph algorithm to the $node\_state$ and $edge\_state$ arrays. Thus, the hardware has exclusive read/write access to the $node\_state$ and $edge\_state$ data items while the software has exclusive read access to the $edge\_offset$ and $edge\_index$ structures which are needed to build the `base_addr` and `length` fields. We also compare the performance of the memory interface when using the Xillinux 1.3 OS running on the ARM CPU. In this scenario, we measure the performance of irregular access from DRAM to the CPU in one case and via Xillybus FIFOs to the FPGA logic in another. Thus, we are able to quantify the effects of various individual components on the performance of the memory subsystem.

We develop two software libraries that interface with the AXI DMA engine configured on the FPGA to control access to the sparse access sequences. These are developed in C and compiled with `arm-gcc` with the `-O3` optimization. For scatter-gather mode, we currently configure the DMA engine to access the descriptor chains stored in a single BRAM. For larger problems, we can either allocate more on-chip BRAMs to hold the chains locally or use double-buffered memory loading to scale to larger graphs. We measure runtime of the various system tasks using a hardware timer in the Zynq SoC when running in bare metal mode and `get_clock_usec` API from `time.h` when running Xillinux OS. For all our experimental measurements we repeat the experiment 100s of times and report averaged runtime to eliminate any unpredictable noise effects during measurement.

Name	**LUTs**	**FFs**	**BRAMs** (36KB)	**Clock** (ns)
`AXI_DMA` Register Mode	3363	4138	0	4.3
(% of Zedboard)	6%	4%	0%	-
`AXI_DMA` Scatter-Gather Mode	6149	7738	4.5	4
(% of Zedboard)	12%	7%	3%	-

TABLE II: Resource Utilization of the MMUs

We use Xilinx Vivado 2013.4 to synthesize the hardware design along with the SDK for configuring the processor. We use the AXI DMA v7.1 IP core [9] generated using the Xilinx Core Generator and also use the BRAM builder to synthesize the BlockRAMs used to hold the AXI descriptor chains. We summarize the resource utilization of the system elements in Table II.

V. Experiments

In this section, we describe the results of our experiments on observed bandwidth for sparse graph access. We first quantify the overheads of an operating system for short-burst transfers on CPUs as well our initial DMA calibration experiments. Next, we investigate the performance tuning results for register-mode DMA operation. We finally show results for optimized scatter-gather operations.

A. Overheads of an OS and Cache

Caching effects and operating system overheads will limit the achievable throughput for sparse random accesses in graph algorithms. In Figure 10, we quantify the overhead of using an operating-system, and enabling caches on ARMv7 host CPU when conducting sequential and random read accesses. For sequential accesses shown in Figure 10a, there is a significant overhead when using an OS for short transfers, but for larger transfers the OS throughputs are faster. There is a clear need for enabling caches due to spatial locality of sequential access. For sparse accesses, we see a similar gap between OS-based and cache-enabled bare metal access for short transfers, but a very small one at larger transfers. Again, disabling caches results in particularly poor access times for longer accesses as data can still be prefetched/cached within one access. For sparse graph problems, we expect length per access to be a low 16–64 bytes per access. The data clearly shows that for short bursty accesses that are typical of sparse graph operations, there is significant overhead when using an OS+cache. Hence, our AXI optimizations are run in cache-enabled bare metal model for maximum performance.

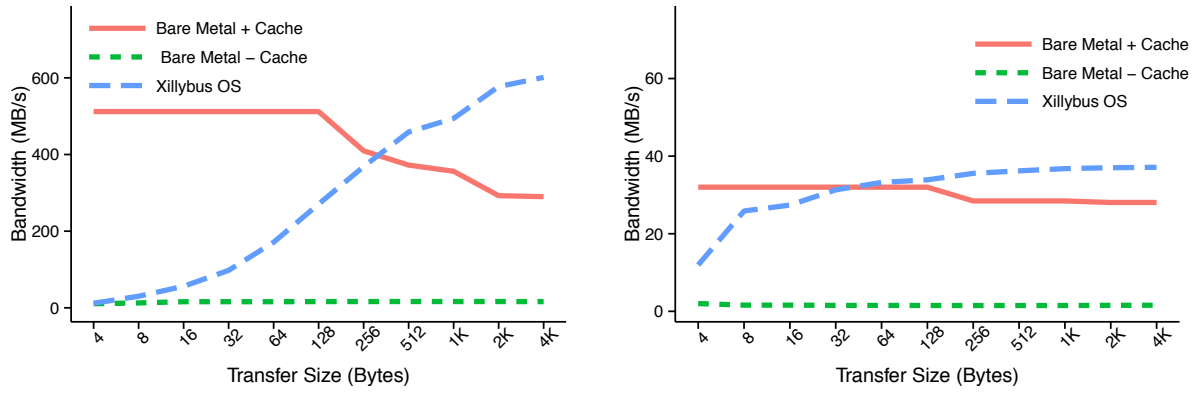
B. `AXI_DMA` Calibration

Next, we look at the impact of `burst_size` on DMA read performance for the `AXI_DMA` block as we vary the `length` of the data fetched in that access (Figure 11). As we would expect, longer bursts provide better results, but we observe performance saturation above 128. Furthermore, for sparse short-burst accesses, we are primarily interested in short `burst_size` results as well as short `length` accesses in the 16–64 byte range. In these situations, a `burst_size` of 8–32 perform well.

C. Comparing `AXI_DMA` Register and Scatter-Gather Mode Operation

In Figure 12, we show the impact of varying the number of accesses (aggregate size represented) on runtime of the computation. We see that longer 16-byte bursts tend to deliver superior runtimes compared to the shorter 2-byte bursts as expected. Most graph algorithms' metadata lies in the 8–32 byte range. The bandwidths measured (size/time) are roughly 25–250 MB/s with the lowest bandwidth for a burst size of 2 and the largest bandwidth for the 32 byte burst designs. For small graphs with few accesses, the slope (bandwidth) is lower across all designs.

In Figure 13, we compare the observed throughputs of the Register and Scatter-Gather modes of operation of the



(a) Sequential Access (Read)

(b) Random Access (Read)

Fig. 10: Comparing the DRAM Read Throughputs of Operating System vs. Bare Metal, Caches vs. No Caches.

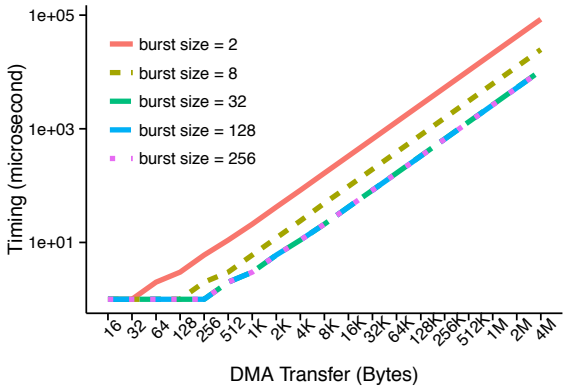


Fig. 11: Impact of Burst Size on memory performance

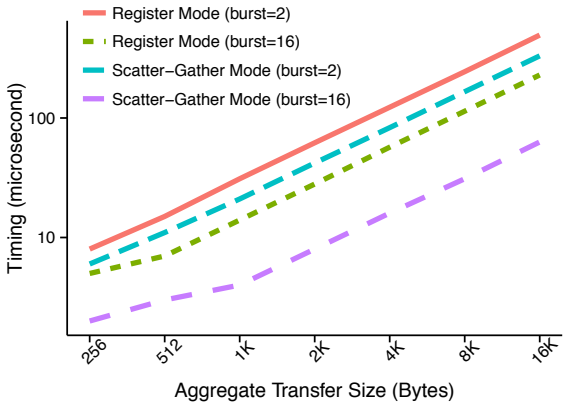


Fig. 12: Time taken by register mode transfers vs. scatter-gather mode transfer (memory read)

`AXI_DMA` block at a `burst_size` of 16 for identical access sequences. It is clear that there is a performance gap of 2–3× in favor of Scatter-Gather mode. We expect this due to the limited interrupt penalty and localized storage of the descriptor chains.

Finally, we summarize the best-case observed bandwidths on the DRAM under various access patterns and operating modes in Table III. As we can see, sequential access patterns deliver the highest observed throughputs from the DRAM interface. However, for random access, when the access is performed from the host CPU with caches and OS involvement, the peak data rates possible drop to merely 34–37 MB/s. We are able to deliver substantial 2–3× improvements in random access bandwidth by using Register mode DMA operation with a burst size of 16. The best improvements of up to 5× are possible when we carefully construct AXI descriptor chains for use with Scatter-Gather mode of the AXI DMA engine for our randomized sparse graph access. While this is still 2.5× less than the peak sequential throughput possible (as expected),

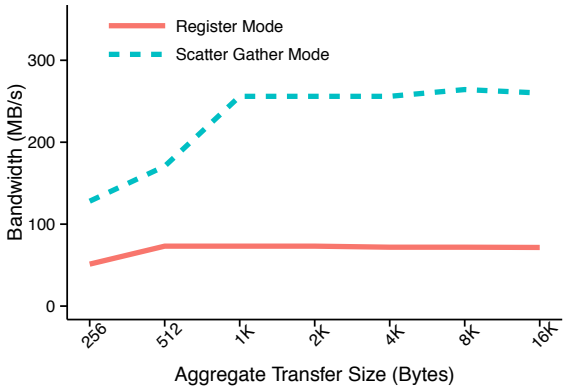


Fig. 13: Throughput comparison for random access using Register and Scatter-Gather DMA modes

our throughputs are still superior to cache-based and naïve register-mode DMA operation.

Mode	Bandwidth (MB/s)	Ratio
OS Sequential	610	-
Bare Metal Sequential	510	-
OS Random	37	1×
Bare Metal Random	34	0.9 ×
Register Mode Random	90	2.4×
Scatter Gather Mode Random	270	7.2×

TABLE III: Best-case observed bandwidths

D. Related Work

The design of specialized accelerators for sparse graph problems have been studied in the past. Some of them use graph data that is fully cached inside the chip thereby limiting the largest size of the problem that can be considered while others just fetch data from the DRAM without optimization.

[1] proposes a re-configurable graph processing architecture to address the latency differences between on-chip memory and off-chip memory with an on-chip memory interconnect between the logic and on-chip memory. However, this design is only applicable for a narrow class of graph problems that do not store weights or states along edges and simply perform a completely local summarization operation. The GraphStep system architecture [4] is designed to take advantage of the larger on-chip memory bandwidth available to store and concurrently process sparse graphs. Scaling to larger graph sizes was achieved across multiple FPGAs interconnected by a custom networking fabric. This may be impractical in most real-world scenarios as FPGAs are expensive devices. However, if cost is no concern, this architecture offers the highest performance reported by keeping all graph data fully inside the chip. GraphGen [6] is another graph-centric accelerator framework for FPGAs that relies purely on off-chip DRAM-based graph storage and streaming parallelism to deliver speedups. However, the memory controller in this design is not particularly optimized to handle graph-oriented accesses and performance is ultimately limited by DRAM bandwidth rather than FPGA parallelism.

CoRAM [3] is a FPGA memory abstraction designed to support memory accesses from within the programmable logic through a set of commonly-used memory organization patterns. While this does provide a convenient set of primitives to compose accelerators and load/unload on-chip memories with ease, there is no specific support for managing large, irregular data structures.

Convey Scatter-Gather DIMMs [2] allow fast random access from the FPGA accelerator organized as 8-byte operations to overcome the wasted cache bandwidth due to 64-byte cache lines on x86 systems. While this is an improvement, it is still a fixed-size access which may not match all graph-oriented access patterns and only operates with Convey boards and memory controllers.

Instead of forcing all data to be held on-chip or ignoring the DRAM bandwidth gap, we specifically focus on developing a memory management unit that can efficiently fetch data from the off-chip DRAM for irregular access sequences. Our GraphMMU is compatible with any AXI-supported board and IP block (accelerator portion) and is not locked to any specific platform. We consider a variety of AXI optimizations and quantify the impact of these optimizations on overall performance. While we demonstrate this design on a small Xilinx Zedboard, we can build larger-scale graph accelerators by splitting computations across multiple Zedboards or by upgrading the design to a denser FPGA.

VI. Conclusions

We show how to improve off-chip memory throughput for sparse irregular access by as much as 7× when compared to cache-based access on embedded CPUs on the Xilinx Zedboard. We use low-level optimizations of the AXI DMA engines by constructing descriptor chain sequences that capture the sparse irregular access. We are able to improve performance by eliminating caches and host CPU for sequencing the access for the AXI engines. In our AXI optimizations, we observe improved scatter-gather DMA throughput by as much as 3× when compared to register-mode DMA. We expect to build larger graph accelerators by composing multiple Zynq SoCs together and use the Zynq SoCs as intelligent scatter-gather engines for distributed graph data.

VII. Acknowledgements and Followup

We wish to thank Mohammad S. Sadri for his Zynq tutorials and extensive assistance in supporting us during the preliminary stages of the project. You may download code under a BSD license from https://bitbucket.org/nachiketkapre/graph_mmu with commit hash 08a3c4b.

References

[1] B. Betkaoui, D. Thomas, W. Luk, and N. Przulj. A framework for FPGA acceleration of large graph problems: Graphlet counting case study. In *Field-Programmable Technology (FPT), 2011 International Conference on*, pages 1–8, 2011.

[2] T. M. Brewer. Instruction Set Innovations for the Convey HC-1 Computer. *Micro, IEEE*, 30(2):70–79, 2010.

[3] E. S. Chung, J. C. Hoe, and K. Mai. CoRAM: An In-Fabric Memory Architecture for FPGA-based Computing. *Multiple values selected*, Jan. 2011.

[4] M. deLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. E. Uribe, T. F. J. Knight, and A. DeHon. GraphStep: A system architecture for sparse-graph algorithms. In *Field-Programmable Custom Computing Machines, 2006. FCCM'06. 14th Annual IEEE Symposium on*. IEEE, IEEE Computer Society, 2006.

[5] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press.

[6] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez, and C. Guestrin. GraphGen: An FPGA Framework for Vertex-Centric Graph Computation. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, 2014.

[7] M. Sadri, C. Weis, N. Wehn, and L. Benini. Energy and performance exploration of accelerator coherency port using Xilinx ZYNQ. In *FPGAworld '13: Proceedings of the 10th FPGAworld Conference*. ACM Request Permissions, Sept. 2013.

[8] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), Aug. 1990.

[9] Xilinx, Inc. *LogiCORE IP AXI DMA v7.1*, Mar. 2014.

[10] Xilinx, Inc. *Zynq-7000All Programmable SoC*, Nov. 2014.