Hybrid Memory Cube performance characterization on data-centric workloads

Maya Gokhale
Lawrence Livermore National Laboratory, Livermore, CA
gokhale2@llnl.gov

Scott Lloyd
Lawrence Livermore National Laboratory, Livermore, CA
lloyd23@llnl.gov

Chris Macaraeg
Lawrence Livermore National Laboratory, Livermore, CA
macaraeg1@llnl.gov

ABSTRACT

The Hybrid Memory Cube is an early commercial product embodying attributes of future stacked DRAM architectures, namely large capacity, high bandwidth, on-package memory controller, and high speed serial interface. We study the performance and energy of a Gen2 HMC on data-centric workloads through a combination of emulation and execution on an HMC FPGA board. An in-house FPGA emulator has been used to obtain memory traces for a small collection of data-centric benchmarks. Our FPGA emulator is based on a 32-bit ARM processor and non-intrusively captures complete memory access traces at only 20X slowdown from real time. We have developed tools to run combined trace fragments from multiple benchmarks on the HMC board, giving a unique capability to characterize HMC performance and power usage under a data parallel workload. We find that the HMC's separate read and write channels are not well exploited by read-dominated data-centric workloads. Our benchmarks achieve between 66% – 80% of peak bandwidth (80 GB/s for 32-byte packets with 50-50 read/write mix) on the HMC, suggesting that combined read/write channels might show higher utilization on these access patterns. Bandwidth scales linearly up to saturation with increased demand on highly concurrent application workloads with many independent memory requests. There is a corresponding increase in latency, ranging from 80 ns on an extremely light load to 130 ns at high bandwidth.

1. INTRODUCTION

In recent years, memory has become a dominant concern in the design of future HPC architectures due to constraints of power, capacity, bandwidth, and latency. Compute cycles continue to increase with the emergence of many-core and GPU-integrated node architectures, leading to a widening gap between available FLOPS vs. memory with sufficient bandwidth, latency, and capacity to supply the data. In response, high performance memory systems have been developed to help alleviate the processor/memory bottleneck, including High Bandwidth Memory (HBM) [2], which is well suited to GPU access patterns; WideIO [3], targeted to mobile applications; and the Hybrid Memory Cube[8], targeting high throughput and capacity with low energy per bit for server/HPC applications.

In this work, we characterize the Gen2 HMC under memory access patterns derived from data-centric applications. The HMC differs from conventional memories by incorporating the memory controller in a base logic layer under the 3D memory stack, and to date is available only with connectivity to an FPGA board or specialized ASIC that communicate using the HMC link protocol. Thus it is not possible to run arbitrary software applications that read and write the HMC and measure performance and power directly. Hardware modules on the FPGA board can be developed to supply memory requests, but it is difficult to create modules that faithfully re-create memory access patterns of a running application.

In this work, we have captured timing accurate memory traces from individual runs of serial data-centric application benchmarks running in an emulator. We have then identified and extracted trace fragments representative of the benchmark's inner loop. We have developed tools to combine memory requests from multiple traces to simulate concurrent accesses generated by multiple cores. We have run the "folded" traces on the Arira Design [1] HMC board and used Arira Design measurement hardware to record bandwidth, latency, and power. The Arira Design board uses four FPGAs, each driving one of the HMC's four serial links. We have generated two forms of folded traces: HPC mode in which the same trace is folded on itself as would occur when a single multi-threaded program has exclusive access to the memory, and datacenter mode in which traces from different benchmarks are combined as would occur when a mixed workload is running on a server. Previous studies have explored HMC performance under simulation [10]. To our knowledge this is the first work to characterize the actual Gen2 HMC hardware under actual application memory access patterns, showing to what extent these irregular requests can benefit from such emerging high bandwidth memories.

Our contributions in this work as are follows:

- We have built an FPGA-based emulator that runs C/-C++ benchmarks at 20X real time and non-intrusively captures every memory reference generated by the running program. We then captured traces from a small collection of data-centric benchmarks.
- We have developed a set of tools to merge time stamped

IA^3 2015, November 15-20 2015, Austin, TX, USA

ACM ISBN 978-1-4503-4001-4/15/11…$15.00
DOI: http://dx.doi.org/10.1145/2833179.2833184

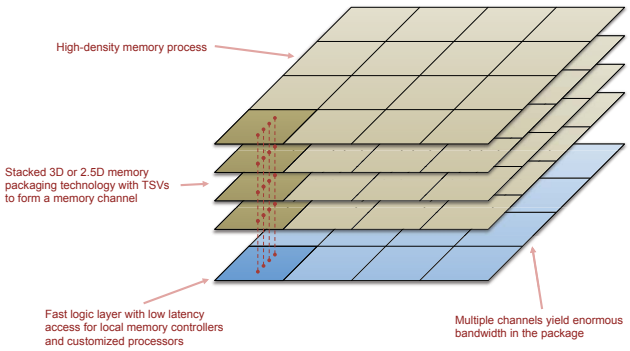


Figure 1: 3D memory with through silicon vias, multiple channels, and fast logic layer.

trace fragments (*fold*) and translate the traces into commands to the Arira Design pattern generator (*trace-2pg*). The latter tool, generates HMC read and write request packets, filling the packet stream with null packets as required so that the pattern generator can transmit a valid packet every clock cycle.

- We have analyzed applications' memory traces on the HMC board, characterizing the maximum available concurrency for different workloads, bandwidth and latency under varying degrees of concurrency, and the power usage.

We find that the Gen2 HMC can support memory requests from 36–84 concurrent threads depending on the workload. If there are fewer concurrent threads than the measured optimal, bandwidth utilization drops. If more concurrent threads than optimal are run, latency increases exponentially. Bandwidth for these read-mostly data-centric benchmarks ranges from 1 GB/s for serial programs to 65 GB/s for highly concurrent applications. Latency varies from 80 ns for a serial program to 130 ns for full concurrency. As expected power usage tracks bandwidth, ranging from 11 W at idle to almost 20 W at full concurrency and bandwidth.

2. THE HYBRID MEMORY CUBE

The Gen2 Hybrid Memory Cube used in this study is a 4 GB part with 4 high-speed serial links. The design goals of the HMC were to achieve significantly higher bandwidth and capacity while reducing energy per useful unit of work done and lower overall system latency.

2.1 Internal Organization

Internally, the DRAM die is divided into multiple partitions, and each partition includes multiple independent memory banks, as diagrammed in Figure 1. The partitions are vertically stacked, and each stack is called a vault. There are 16 vaults in the 4 GB part. Each vault is stacked 8 high with 2 banks on each layer for a total of 256 MB per vault and 16 MB per bank. The four HMC links connect to an interconnection network on the base logic layer that is also connected to the DRAM subsystem. Each link has a more direct connection to a quarter of the vaults and forwards requests for non-local addresses through a longer latency path in the crossbar switch. The memory control in the HMC reorders requests and delivers results in the most advantageous order from the memory point of view. The HMC communicates to external devices through a packet-based protocol. Several packet sizes are supported in the protocol [6] ranging from 16 to 128 bytes in 16-byte increments. There are separate channels for read and write, so that a 50/50 read/write mix is required to attain the highest bandwidth.

2.2 HMC board

The Arira Design HMC board contains four FPGAs, one for each link. Each FPGA has a memory interface unit to communicate with its HMC link using the HMC Spec 1.1 protocol [6]. Additionally the FPGA is configured with a proprietary programmable pattern generator. Programs are written in an assembly-language-like format in which each line either specifies a request packet to transmit or a control instruction such as a loop. Pattern generator programs are translated into instructions up to a limit of 2K as determined by the pattern buffer size.

The board has been instrumented to measure power, and the FPGA design tracks bandwidth, latency, and the token count. In the HMC protocol, the slave link controller issues tokens to indicate free slots in the input queue. Sending a packet uses several tokens depending on its size. Enough free slots must be available before a packet can be sent, otherwise the link interface will stall. As slots become available again, the slave link controller issues more tokens.

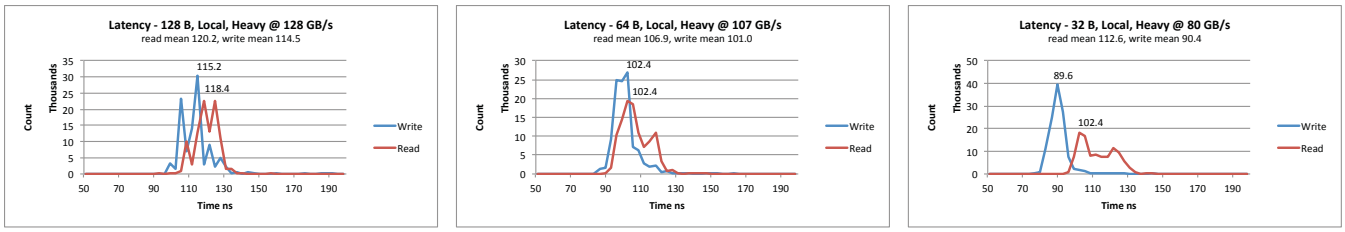
Arira Design provides a suite of test patterns that demonstrate the wide range of performance and power observed on the HMC. Figure 2 shows plots of latency with three different packet sizes corresponding to CPU-cache-line sizes. In these tests, each link accesses data in vaults local to the link. As the plots show, read and write latency may be different, and latency increases with packet size.

Figure 3 shows the corresponding measured bandwidth for the three packet sizes. Highest bandwidth of 128 GB/s is reached with the largest packet size of 128 bytes.

Given the ranges of latency and bandwidth on the optimal read/write mix, it is not obvious what performance to expect on real application workloads. To help gain insight into this question for irregular applications, we have developed an extensive infrastructure to drive the HMC with fragments of actual application traces, and measure latency, bandwidth, and power.

3. OBTAINING WORKLOAD TRACES

Our evaluation process uses memory traces captured from the regions of interest in application benchmarks. Memory traces can be obtained through a variety of methods. Computer architecture simulators such as gem5 [4] perform detailed simulation of CPU and memory hierarchy and can emit many statistics and traces, including memory traces of arbitrary full system workloads. Architecture simulators can provide models of existing or proposed CPUs, and by including cache models, can generate timestamped memory traces of requests that go to the memory bus. Due to the detailed software simulation of CPU microarchitecture, these simulators are very slow. Other simulators work on a per-application basis through binary instrumentation such as the Valgrind [9] tool suite or Pin [7]. The latter are faster than full architecture simulators, but emit memory traces at the load/store level and thus include requests that are satisfied in cache and don't actually to go the memory. Our approach employs emulation in programmable hardware.



(a) Latency with 128-byte packets. (b) Latency with 64-byte packets. (c) Latency with 32-byte packets.

Figure 2: Latency profiles on HMC with different packet sizes and a 50/50 read/write mix.

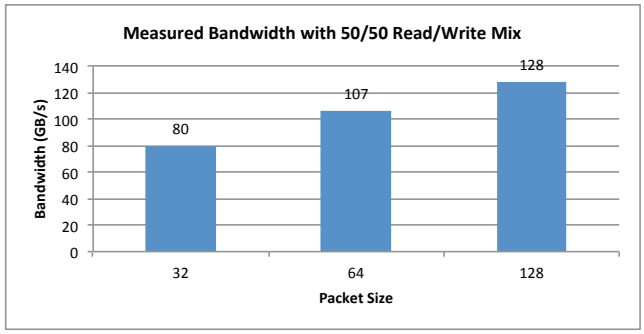


Figure 3: Best-case bandwidth measured at different packet sizes

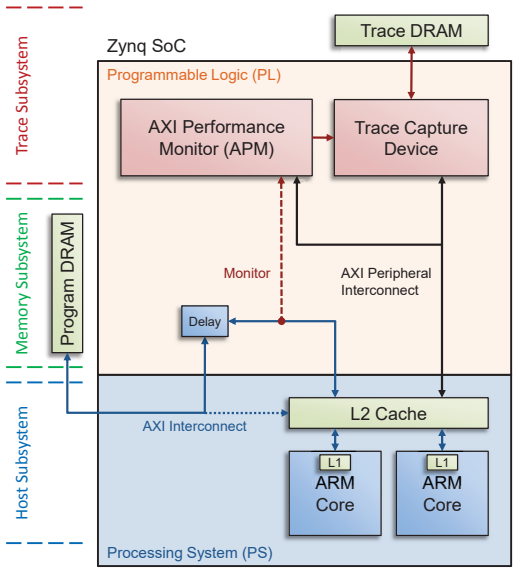


Figure 4: Zynq SoC with emulation framework

3.1 Emulator

We have built an FPGA emulator that records memory transactions generated by software applications without affecting cache, memory traffic, or emulated speed of the application. The emulator, shown in Figure 4, uses a Xilinx Zynq 7000 System-on-Chip with a dual ARM CPU, onboard memory, and programmable logic.

3.1.1 Host

The Host subsystem (the Processing System section of the diagram) includes the two ARM A9 processor cores with separate L1 and shared L2 caches. The Programmable Logic section includes the Memory Subsystem and Trace Subsystem. The Memory Subsystem is a 1 GB 32-bit wide DDR3 memory used to hold application code and data. In our experiments, the Host subsystem software runs "bare metal" without an Operating System, allowing measurement of the application in isolation and as importantly, giving the application nearly the entire 1 GB memory.

3.1.2 Trace subsystem

The top portion of the diagram shows the Trace Subsystem, composed of the AXI Performance Monitor (a Xilinx soft IP module), the Trace Capture Device (soft IP we built), and the Trace DRAM, a 64-bit wide 1 GB DDR3 memory. The Trace Subsystem captures memory requests as they appear on the AXI bus. When a software program runs on an ARM core and trace is enabled, memory transactions are forwarded to the Trace Capture Device, which writes each transaction along with a time stamp to the Trace Memory. This process occurs concurrently with memory requests being looped back onto the AXI bus and going to the on-chip memory controller and subsequently to Program DRAM. Trace capture is completely independent of on-chip caches or ARM memory operations. A Xilinx library is used to retrieve event counter values from the APM in programmable logic and from the Performance Monitoring Unit (PMU) that is part of the Processing System. The library also provides the capability to inject commands telling the APM to start and stop tracing, enabling selective tracing of regions of code.

Statistics collected by the emulator include emulated run time and memory read and write requests of a serial program running on a single ARM core. The program typically turns on trace capture when it enters a Region of Interest (RoI) to record steady state memory traffic of the benchmark kernel.

A list of pre-existing FPGA logic modules we have used to build the emulator's trace capture functionality is shown in Table 1.

3.1.3 Timing-accurate emulation

Rather than a full software-based simulation of CPU, cache hierarchy, and memory, the emulator uses a combination of fixed and programmable hardware. In this emulation framework our focus is on memory requests, and therefore we run the application in the fixed logic host subsystem, capturing application memory requests as they appear on the AXI interconnect. However, for meaningful time stamps on mem-

IP	Description	Usage
AXI Interconnect	Connects master and slave devices	Attaches ARM to memory and peripherals
AXI Performance Monitor	Monitors activity on AXI Interconnect	Creates a trace and provides event counters
FIFO Generator	Configures FIFOs in programmable logic	Local buffers for Trace Capture Device
Memory Interface Generator	Configures AXI interface to DDR3 memory	Main storage for Trace Capture Device

Table 1: Pre-existing soft IP used in the emulator

Component	Actual	Emulated
CPU clock	128.6 MHz	2.57 GHz
CPU BW	257 MB/s	5.1 GB/s
Mem BW	1.6 GB/s	32 GB/s (3 vaults)
Latency	*180 ns*	*9 ns (too low)*
Delayed Lat.	180 ns	$9 + \mathbf{91} = 100$ ns

Table 2: Scaling the emulator by a factor of 20.

ory transactions, we must carefully manage Zynq clocks so that the fixed function logic, programmable logic, and DDR3 memory system present a timing-consistent system architecture.

The clock system on the Zynq platform supports many configurable clocks that can span a wide range of frequencies. The A9 cores on the Zynq 7000 can run up to 800 MHz and down to under 1 MHz. The programmable logic clock frequency depends on the specific design placed on the FPGA but is typically around 200 MHz. The DDR3 program memories run at 1066 MT/s. Since the trace capture modules are soft IP in programmable logic, they need a slower clock than the CPU. Therefore, the CPU is slowed to run at a comparable frequency. To maintain full memory bandwidth that the Zynq can sustain, the DRAM clocks are not slowed; therefore, all memory requests are routed through a set of programmable delay units (labeled Delay in Figure 4) to emulate scaled memory latencies. All memory requests go through the delay units and concurrently the trace subsystem decides whether or not to forward the transaction to the trace memory. The delay units are parameterized to allow emulation of a wide range of memory latencies encompassing current and future technologies.

Table 2 shows an illustrative correspondence between actual and emulated specifications for memory and a single CPU core. In this example of a 2.57 GHz CPU, a 91 ns delay must be added to each memory transaction to keep the emulated memory latency consistent with other architecture parameters.

Our use of an SoC to emulate a system offers efficiency and challenges. Using fixed IP modules such as the ARM cores, caches, and memory controllers saves FPGA logic and development time. However, these fixed components also limit host design space exploration and require coordination of multiple clocks to accurately model the desired system.

3.1.4 Emulated Architecture

In the evaluation of Section 4, the emulated CPU is a 32-bit A9 ARM core running at 2.57 GHz. It has a 32 KB instruction and 32 KB data L1 cache, and shared 512 KB L2 cache. The cache line size is 32 bytes. Memory bandwidth is 5 GB/s. During emulation, the ARM is running at 128.6 MHz and the trace capture logic runs at 200 MHz. Our methodology to tune HMC memory read and write latencies to application access patterns is as follows:

1. measure latencies on a completely unloaded link on the HMC board using the best possible test pattern (one link to read/write its local vaults),
2. run the benchmarks on the emulator, using measured latencies to program the delay units, and collect memory traces
3. replay trace segments on the HMC board and measure latencies,
4. repeat the benchmark runs with revised latencies

Steps 2 and 3 are repeated until near convergence. Using this methodology on a random access workload, memory delay in emulation was set to 84 ns read and 103 ns write latencies for the evaluation.

3.2 Translating traces to packets

We have built a set of tools to translate the time-stamped memory traces into a sequence of HMC request packets that maintain the original timing. Traces capture a window of memory activity up to about a second in emulated time. Each trace transaction records an event with a quarter nanosecond resolution time stamp. Traces typically range in size from a few hundred megabytes to a gigabyte. However, the Arira Design pattern generator (a custom logic design on the HMC board's FPGAs) can only replay a short segment of HMC activity from its buffers, ranging from about 6 us to 45 us depending on the load factor. Therefore, only a small fraction of the trace can be replayed on the HMC board.

The memory traces collected by the emulator record the memory requests of a single core. However, a realistic scenario on a multi-core CPU would generate requests from many cores concurrently. We have developed a tool *fold* that merges multiple traces into a single trace, maintaining the relative timing between dependent requests in individual traces. This behavior is similar to a memory controller receiving requests from multiple cores and merging them into a request stream to the memory.

The fold tool can be applied either to a single trace with multiple offsets into the trace file or to multiple trace files at arbitrary offsets into each file. Representative segments within the RoI are manually selected for replay. Given a collection of trace segments, the tool merges transactions from each stream in round robin fashion into a single trace while maintaining the relative timing between the memory requests. As the traces are ingested, the lowest time stamp relative to a segment's start time is selected for output. The folder tool merges traces without regard to actual memory channel capacity. Merged events that occur at the same time have the same time stamp even though they are serialized in the output trace. The output of the tool drives a single

Single Trace	Folded Trace
0, R, 0x50E9BE60, 32, 0-30, 3992	0, R, 0x3413ADD8, 32, 7-24, 4194
0, R, 0x75E1C8F8, 32, 0-24, 4253	0, R, 0x7423FAA0, 32, 6-31, 4200
0, R, 0x75E1C900, 32, 0-29, 4300	0, R, 0x3413ADE0, 32, 7-29, 4200
0, R, 0x74F3CF48, 32, 0-24, 4334	0, R, 0x73FCD528, 32, 2-25, 4210
0, R, 0x74EBECA0, 32, 0-26, 4377	0, R, 0x74A7F750, 32, 8-25, 4212
0, R, 0x74F3CF60, 32, 0-28, 4636	0, R, 0x74BC52F0, 32, 4-25, 4215
0, R, 0x7407E158, 32, 0-26, 4679	0, R, 0x3542DA28, 32, 9-24, 4219
0, R, 0x7407E160, 32, 0-31, 4720	0, R, 0x1C576D00, 32, 3-28, 4234
0, R, 0x74FA9170, 32, 0-26, 4760	0, R, 0x75E1C8F8, 32, 0-24, 4253

Figure 5: Memory trace fragments

HMC link. An example of a single and folded trace segment is shown in Figure 5.

Each line describes a memory event. The comma-separated items indicate

- the trace event source,
- whether it is a read or write request,
- the address,
- the size of the request in bytes,
- the AXI bus ID,
- the timestamp (cycle number).

The AXI ID is shown as fields "a–b." Field "b" is the actual AXI bus ID. Field "a" is the index of the trace file. For example, the first line with AXI ID 7–24 refers to a transaction with AXI bus ID 24 issued by trace file 7.

The folded traces are next translated into commands specific to the Arira Design FPGA pattern generator by another tool we have developed, *trace2pg*. HMC links transfer data in 16-byte blocks called flits (flow control digits). To maintain high-speed serial-link integrity, data continually flows on an HMC link. When no requests are being sent, null flits are sent instead. On a 10 Gb/s link, a flit is sent every 0.8 ns. The trace2pg tool, after converting trace events into HMC request packets, schedules them in available flit slots. If necessary, null flits are inserted in between memory requests to maintain relative timing. When a trace has more requests than can fit on a link, they are queued until flits slots become available. Once the (folded) trace segment is translated, loop code is inserted to repeat the pattern indefinitely. Output from trace2pg is compiled to pattern generator machine code and loaded into the pattern buffer for each link. Each pattern generator loops over its program, recording latency, bandwidth and power.

The trace2pg tool has an option to add an address offset to all memory requests. This is useful in preparing workloads for multiple links and have them access different address ranges. In our evaluation, we have used the same trace segment for all links, but apply a different address offset to each link so that the HMC sees a more diverse memory request pattern.

4. EVALUATION

Given the variance in bandwidths and latencies observed on the HMC with an optimal 50/50 read/write mix, it is hard to predict from test patterns how effectively applications, particularly data-centric, irregular applications, can use the HMC. To characterize bandwidth, latency, and power on application workloads, we have run folded trace fragments from a set of data-centric benchmarks on the HMC board.

4.1 Benchmarks

The benchmarks are as follows:

- PageRank, a well-known benchmark to rank web pages in popularity. PageRank is typically run on scale free graphs characteristic of social networks. A synthetic scale 22 RMAT graph was used (2^{22} nodes, with on average 16 edges per node).
- Kmeans clustering, a data intensive benchmark to cluster a data set into k groups. A synthetic data set of 50 000 30-dimensional vectors was clustered into four classes.
- Image Difference. This benchmark computes the pixel-wise difference of two images, optionally decimating each 512 MB image to compare reduced resolution images. Two versions of the benchmark were measured, at full resolution and at 16x decimation.
- Sparse MatVec, multiplying a sparse matrix with a dense vector. The benchmark was adapted from Berkeley's BEBOP benchmark set [5] using a 2M $\times$ 2M sparse matrix with 34 non-zero double-precision elements per row.
- RandomAccess ("gups"), a well-known benchmark that reads and updates random locations in a table. The table is almost 1 GB.

Each benchmark is configured to use up to the maximum 1 GB memory. The benchmarks are serial and run on one ARM core. During execution, traces of the Region of Interest are written to Trace Memory. Representative trace fragments are manually extracted, and the fragments are run through *fold* multiple times with successively higher degrees of folding, up to the maximum theoretical bandwidth for a link (13.3 GB/s for 32-byte packets). Folding the same serial trace simulates an HPC workload in which multiple data parallel tasks perform the same computation in SPMD (Same Program Multiple Data) mode on different parts of the data set. To simulate data center mode in which a server runs multiple applications, we run the folder over the entire collection of application trace fragments.

4.2 Benchmark profiles

Table 3 profiles each application by its trace fragment's read/write mix and requested bandwidth of a single thread (i.e. the original serial code). It is evident that these data-centric benchmarks are read-dominated. In fact, SparseMatVec has only 0.3% writes. In this benchmark, the only data being written is the dense output vector. The highest write requests come from RandomAccess, whose main loop does a read-modify-write of a random table element. However, full utilization of the write channel is not possible, even on RandomAccess.

The bandwidth requested by one serial thread (third column) is very low. KMeans needs only 0.64 GB/s bandwidth. A single RandomAccess thread requires the highest bandwidth of 1.71 GB/s. ImageDiff at full resolution needs less bandwidth than at 16x reduced resolution. The thread's bandwidth load is reported by the *fold* tool.

Trace	Read/Write Mix	BW (GB/s/thr.)
PageRank	99/1	1.32
KMeans	96/4	0.64
ImageDiff x16	98/2	1.33
ImageDiff full	92/8	1.02
SparseMatVec	99.7/0.3	1.31
RandomAccess	66/34	1.71
mixed	90/10	1.33

Table 3: Benchmark Read/Write Mix

Trace	Max Thr. Pred.	BW Pred.	Max Thr. Act.	BW Act.	BW % Pred.
PageRank	40	52.8	40	50.6	96%
KMeans	88	56.3	84	53.0	94%
ImageDiff x16	40	53.2	40	53.7	101%
ImageDiff full	56	57.1	56	54.3	95%
SparseMatVec	40	52.4	40	51.5	98%
RandomAccess	44	75.2	36	64.1	85%
mixed	44	58.5	44	55.0	94%

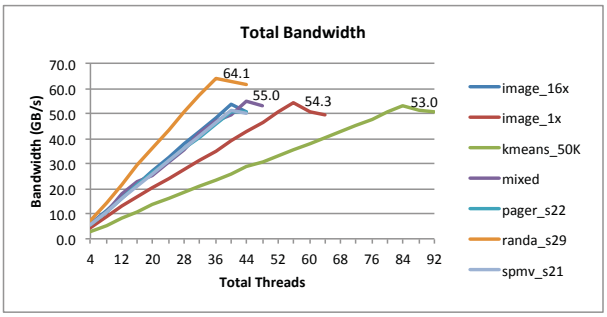
Table 4: Benchmark predicted and actual thread count and bandwidth.

Table 4 is divided into sections. Column 2 (maximum predicted thread count) shows the highest number of concurrent threads of the benchmark that it is possible to run before saturating link bandwidth. The predicted bandwidth in GB/s at saturation is shown in the third column. This is simply the product of the bandwidth required by a single thread and the maximum number of threads. This value is different for each benchmark. It depends on the benchmark's cache occupancy (portion of cache lines actually used by the application), compute-to-memory-access ratio, and dependency from one memory operation to the next.

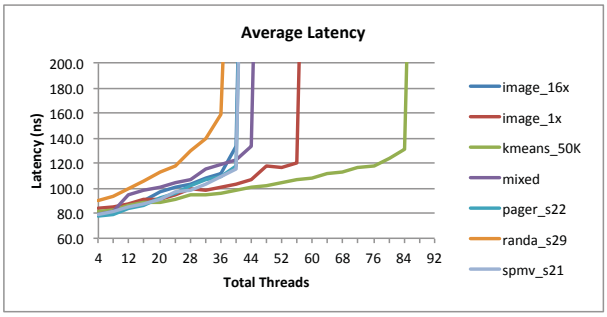
The second part of Table 4 shows the actual maximum thread count and measured bandwidth when running the maximum threads. The final column gives the ratio of actual bandwidth to predicted. It can be seen that measured bandwidth is very close to predicted, indicating excellent scaling of bandwidth by the HMC up to the point that the benchmark's concurrency level saturates internal HMC resources.

4.3 Concurrency tests

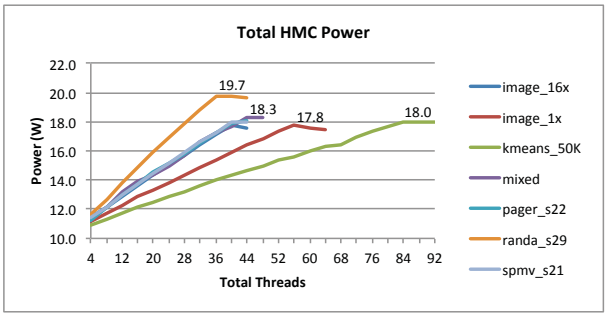
The HMC's scalability as bandwidth demand increases is apparent in Figure 6a, which shows that as the thread count increases, bandwidth scales linearly up to saturation. If the thread count is increased beyond the capacity of the link, bandwidth drops. The differing concurrency levels of the benchmarks is highlighted in this figure and even more dramatically in Figure 6b, which is a latency plot. Average latency increases from under 80 ns as threads are added. At saturation, latency increases steeply. The y-axis goes to 200 ns; however, the measured latency at post-saturation concurrency climbs to 600 ns or more. It is clear that saturation is a cliff and if requests exceed the link capacity, an extreme performance drop will occur. Also, reinforcing the bandwidth plot, each benchmark has its unique maximum concurrency point. A group of four benchmarks show very similar profiles: ImageDiff at x16 decimation, the mixed workload, SparseMatVec, and PageRank. These benchmarks peak at about 40 threads of concurrency, with



(a) Bandwidth.



(b) Latency.



(c) Power.

Figure 6: Performance of benchmarks, 32-byte packets.

maximum bandwidth of 50–55 GB/s. Latency for this group is about 120 ns. ImageDiff at full resolution peaks at 56 threads and 54.3 GB/s. KMeans has the most threads, 84, and uses about 53 GB/s. In contrast, RandomAccess has the least concurrency of 36 threads, but uses the highest bandwidth 64.1 GB/s.

Figure 6c, the power profile, shows that power tracks bandwidth. At nominal bandwidth as requested by single threads, power is about 11 W, near the idle power draw. As bandwidth demand increases, power increases linearly to a maximum of 19.7 W at the maximum demand of 64.1 GB/s.

4.4 Discussion

The results highlight notable characteristics of the HMC and its behavior on data-centric workloads.

4.4.1 Scalability

The HMC shows exceptional linear scalability in bandwidth as demand increases. To achieve maximum bandwidth, the application workload must exhibit high concurrency. Serial, latency-sensitive code will not be able to get

performance, but a throughput-driven application (or workload) with many independent concurrent threads can effectively utilize the memory. Full thread-level concurrency or even over-subscribed thread concurrency is well suited to the HMC.

4.4.2 Link saturation

The HMC shows linear scaling in bandwidth and a modest increase in latency up to link saturation. However, once the link is saturated, bandwidth declines, and an extremely long latency suddenly occurs. However, we have found that each benchmark reaches saturation at a different level of concurrency, making it difficult to use a fixed level of concurrency in CPU task scheduling. It may be desirable in future architectures to have logic in the CPU that tracks requested load and throttles task parallelism as the load approaches peak bandwidth for the channel.

4.4.3 Read/write mix

As noted in Section 4.2 the data-centric benchmarks are highly skewed toward reads. The HMC has independent read/write channels, which is an advantage when there is a mix of reads and write. There is no delay to switch the channel direction because each channel only goes in one direction. However, if the read/write mix is highly unbalanced, the workload can only use a portion of the HMC's enormous bandwidth. The measured bandwidth of these benchmarks is a fraction of peak achievable bandwidth on the HMC (see Figure 3). In the best case, with 32-byte packets and a 50/50 read/write mix, bandwidth is 80 GB/s. Our application-specific workloads at peak concurrency get from 63% to 80% (RandomAccess) of the 80 GB/s and at most 50% of the measured peak of 128 GB/s (50/50 read/write, 128-byte packets).

4.4.4 Irregular access patterns

Of all the benchmarks tested, only RandomAccess is a truly irregular benchmark. However, due to the folded traces, memory requests are issued in seemingly random order even when a single thread issues sequential requests. The differentiating factor is the amount of each cache line an application uses before it issues the next request.

The degree of cache line use is demonstrated by two versions of ImageDiff. The ImageDiff benchmark, depending on the reduction factor, uses all the data in a cache line or very little. Pixels are represented in memory by 32-bit words that contain 8-bit red, green, blue, and alpha subcomponents. Pixels are read sequentially from the source images and subtracted to compute the difference. A 32-byte cache line can hold 8 pixels. When subtracting full images, all of the pixels in a cache line are compared making effective use of a cache line load. In contrast, with a reduction factor of 16, only one out of every 16 pixels is needed from the source images. A cache line load by the processor contains several pixels even if only one is used. With a cache line size of 32 bytes, only 1 in 8 pixels is used. Since the next pixel in sequence skips a cache line, a prefetch of the adjacent cache line will be entirely wasted. The results are consistent with this scenario. The 16x decimation version requires greater bandwidth than full resolution because more memory must be read to accomplish less work than the corresponding full resolution version.

This study was done using emulator traces with 32-byte cache lines. The HMC shows higher bandwidth at larger packet sizes, indicating that CPU requests using longer cache lines will get higher bandwidth. However, for irregular applications, the bandwidth may be wasted if only a small section of the cache line is used, as in RandomAccess and ImageDiff x16. Our HMC tools can handle all the packet sizes, and in future work, we will evaluate traces obtained from CPU simulators such as gem5 using 64 and 128-byte cache lines and 64-bit processor models.

5. CONCLUSIONS

Using data-centric application traces collected from a custom built FPGA-based emulator, we have evaluated a novel 3D memory, the Hybrid Memory Cube. The HMC shows linear bandwidth scaling up to channel saturation, but performance degrades sharply if the demand exceeds that limit, We find that read-dominated, throughput-driven highly concurrent applications with 32-byte memory requests can achieve up to 80% (64 GB/s) of peak bandwidth given the separate read/write channels, with latency up to 130 ns.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract No. DE-AC52-07NA27344. LLNL-CONF-676776.

6. REFERENCES

[1] Arira Design. http://www.ariradesign.com, accessed 2015.

[2] High bandwidth memory (HBM) DRAM (JEDEC). http://www.jedec.org/standards-documents/docs/jesd235, accessed 2015.

[3] JEDEC Wide IO and Wide IO 2 specs. http://www.jedec.org/standards-documents/results/jesd229-2, accessed 2015.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[5] H. Gahvari, M. Hoemmen, J. Demmel, and K. Yelick. Benchmarking sparse matrix-vector multiply in five minutes. In *SPEC Benchmark Workshop*, Austin, TX, January 2007. http://bebop.cs.berkeley.edu.

[6] HMC-Consortium. HMC specification 1.1. http://www.hybridmemorycube.org/files/SiteDownloads/HMC%20Rev%201%5F1%20Specification.pdf, accessed 2015.

[7] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[8] Micron. Hybrid Memory Cube. http://www.micron.com/products/hybrid-memory-cube, accessed 2015.

[9] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[10] P. Rosenfeld, E. Cooper-Balis, T. Farrell, D. Resnick, and B. Jacob. Peering over the memory wall: Design space and performance analysis of the Hybrid Memory Cube. Technical Report UMD-SCA-2012-10-01, University of Maryland Systems and Computer Architecture Group, October 2012.