On the Efficacy of APUs for Heterogeneous Graph Computation

Karthik Nilakant
University of Cambridge
United Kingdom
Email: karthik.nilakant@cl.cam.ac.uk

Eiko Yoneki
University of Cambridge
United Kingdom
Email: eiko.yoneki@cl.cam.ac.uk

Abstract

Accelerated Processing Units (APUs) are central processors that feature integrated GPU cores. In this study, we show that this architecture is well-suited to the domain of graph analytics. Our evaluation shows that a current-generation integrated GPU can outperform an externally-connected discrete GPU by up to 50% for the breadth-first search and PageRank algorithms. Furthermore, by operating on data with different characteristics in unison, the CPU and integrated GPU can halve the running time of PageRank on a scale-free dataset.

1. INTRODUCTION

Large-scale graph-based computation is a key analytical tool in a range of disciplines, including social network analysis, scientific computing, and e-commerce data mining. Graph-based computation is characterised by large structural definitions, which must often be accessed in an irregular fashion. Such irregularity is often poorly managed by traditional execution frameworks. One way to alleviate this problem is to take advantage of processor heterogeneity to adapt to structural patterns in the data.

Graphics processors (GPUs) can offer several advantages over traditional CPUs when applied to specific graph-centric problems. When encountering regions of data or code that call for added parallelism, GPUs offer a much higher hardware thread count than CPUs (up to three orders of magnitude more) and also have access to higher memory bandwidth. However, the local memory capacity on a GPU card will typically be lower than RAM capacity on the host system. Furthermore, GPUs are typically connected to a machine via an external peripheral bus (PCI Express), which means that data transfer throughput between the host and the GPU is restricted.

An alternative architecture, which is currently a key area of development for hardware manufacturers, is to incorporate GPU cores into CPU chip design. These CPUs with integrated GPUs (also known as accelerated processing units or APUs) are primarily being developed to address the need for miniaturisation and energy efficiency. However, the APU architecture also presents a possible opportunity for graph analytics, by removing the bottleneck imposed by the peripheral bus.

Until recently, the performance gap between integrated GPUs and discrete cards has been too large to warrant serious consideration. However, current-generation hardware offers a level of performance that was previously only present in discrete GPUs. Our interest in the technology is primarily application-driven; we theorise that graph applications have a set of requirements that would benefit from tighter integration and increased bandwidth between processing units. To this end, in this paper, we seek to analyse the costs and benefits of adopting a current-generation APU platform for graph computation, compared with discrete GPU computation. We have found that:

1. The integrated GPU consistently outperformed the discrete GPU by 15% to 50%, when reading graph data from a host-allocated memory buffer.

2. Despite having a higher bandwidth channel to local device memory, the need to copy data over the peripheral bus hampered the performance of the discrete GPU.

3. CPU performance on PageRank (where communication costs are dominant) is usually better than both GPU units tested, when each unit is used in isolation. However, the integrated GPU performs better on graphs with a low average degree.

Based on these findings, we also show how the APU architecture could support two schemes for hybrid, heterogeneous graph computation that have been successfully deployed in other systems: switching and partitioning (described further in the next section). Our results show that while the test platform was not conducive to the switching method, heterogeneous partitioning provided a performance improvement of up to $2\times$ over the CPU-only benchmark.

We begin by discussing the characteristics of graph-centric computation, and how this relates to heterogeneous processing. We then outline the parameters of our evaluation test bed, including the algorithms that were tested and the memory layout that was adopted. We then report our findings, analysing the feasibility of two possible hybrid processing schemes. We then examine the results in the context of related work before concluding.

2. GRAPH PROCESSING OVERVIEW

In this section, we will review the features of graph computation, and show how a tightly integrated heterogeneous processing environment might be utilised to improve performance.

2.1 Graph processing

In data-parallel graph computation, the developer typically defines one or more granular operations to be applied to a locality in a graph (for instance, a vertex and its neighbours, or an edge and its associated endpoints). These operations are then executed in parallel. The selection of graph elements to operate on, and the order in which these operations execute may affect the semantics (requiring strict ordering) or the performance of the program.

For example, in PageRank [16], every vertex is updated in each pass, with no constraints on the order that the vertices are updated. However, ordering operations to follow the sequential layout of data in memory will (usually) improve performance. Alternatively, for traversal style algorithms such as breadth-first search, operations proceed in a frontier radiating from the source - in this algorithm, it is necessary for operations to execute one frontier level at a time, in order to preserve the semantics of the equivalent sequential algorithm.

A common approach taken by some current graph analysis platforms is to adopt a static data-parallel execution strategy, which is usually necessitated in the case of networked machines. In this approach, the graph structure is (randomly) partitioned across the cluster, and each node executes the operations on its local elements. Updates to the program metadata (algorithmic state) are propagated along edges, with message-passing used for remote updates. A synchronisation barrier at the end of each iteration ensures all updates have completed before progressing to subsequent iterations. In theory, such a scheme could also be adopted for distributed computation across a CPU and GPU, however the limitations of communication over an external peripheral bus usually mean that this is impractical. Furthermore, this approach does not to take advantage of the differing performance characteristics of each type of device, which will be discussed in the next section.

2.2 Heterogeneous Graph Computation

GPUs feature a number of advantages and constraints when compared with CPUs, which must be considered carefully when building a heterogeneous execution platform. The following list characterises some of these differences:

- Parallelism model. Modern multicore CPUs allow full thread flexibility, with a multiple-instruction multiple-data (MIMD) processing model. In contrast, the "virtual cores" of a GPU require the use of SIMT (single instruction multiple thread) computation.

- Parallel processing power. Current CPU sockets accommodate less than a hundred concurrent hardware threads. In contrast, the vector registers of a GPU offers several thousand hardware threads.

- Local memory capacity. A commodity server may accommodate several hundred gigabytes of system RAM, but less than ten gigabytes of graphics RAM. Note however that the graphics hardware can access system memory via DMA across the PCI express bus, albeit at a higher latency cost.

- Memory bandwidth. With high memory clock speeds and multiple channels, graphics hardware typically has much higher local memory bandwidth than the CPU (up to ten times more). GPU programs also have the option to specifically utilise small areas of low-latency cache memory if desired.

To align these differences to the characteristics of graph programs, it is necessary to select graph elements with relatively uniform properties to be analysed by the GPU. The aim of this is to ensure that branch divergence is kept to a minimum. Divergent branches in an SIMT kernel will result in partial serialisation, which affects the GPU's overall throughput. Assuming that appropriate graph elements are dispatched to the GPU in batches of appropriate size, the associated overhead of kernel invocation can be masked by the added parallelism and throughput of the GPU.

Processor / Graphics Card	AMD A10-7850K		AMD R7 250
	CPU	iGPU	dGPU
Cores	4	8	6
Clock MHz	4000	720	1050
RAM	8GB DDR3		1GB GDDR5
Local Read GB/s	–	21.5	55.6
DMA Read GB/s	–	8.0	5.8

Table 1: Testing platform layout. "Local read" speeds indicate throughput from device-visible RAM, whereas "DMA Read" speeds are between the GPU device and host memory.

In situations where the elements to be processed are not well-suited to SIMT processing, an alternative is to process these directly on the CPU. This forms the basis of a heterogeneous runtime platform for graph computation, using a data-centric scheduler that selects the correct device to process a given workload. Current systems in this domain can be categorised into two basic approaches:

Heterogeneous Switching: in this approach, computation proceeds on a single device until the scheduler detects that the data to be processed could be better managed by another device. The data and computation are then switched to the other device. To operate successfully in this manner, the overhead of switching and measuring the characteristics of the current workload must be offset by gains in processing efficiency.

Heterogeneous Partitioning: in this approach, computation proceeds in parallel on two or more heterogeneous devices, with the data to be processed by each device tailored to fit its characteristics. For this approach to be successful, the overhead from partitioning the input data and synchronising the output data must be offset by the gain in overall computation throughput.

A system could support both approaches simultaneously, however in this study we will analyse each scheme in isolation. In particular, we show how each scheme could be respectively implemented with two separate algorithms, breadth-first search and PageRank. However, a key ingredient in ensuring such schemes operate effectively is high-performance interoperability between the CPU and GPU – our evaluation shows that integrated GPUs may be able to provide this capability in particular scenarios.

3. TESTING METHODOLOGY

Our main aim is to highlight some key differences in application performance, when comparing graph processing on integrated GPUs (iGPUs) discrete GPUs (dGPUs). We also compare GPU performance to CPU performance on two graph algorithms with different characteristics – breadth-first search and PageRank. Finally, we explore ways in which these differing performance characteristics can be exploited in the context of the heterogeneous processing models discussed in Section 2.2.

3.1 Graph Algorithms

For these trials, we implemented multi-threaded kernels for two representative graph algorithms: breadth-first search (BFS) and PageRank (PR). The BFS algorithm is a fundamental building block for a variety of other algorithms in the graph domain, such as weak and strongly-connected component detection, shortest path algorithms, graph colouring, and centrality measurement. PR is representative of a class of graph algorithms based on fixed-point iterative computation.

Both algorithms proceed in a distributed fashion by dividing the job into vertex-centric subtasks, which execute repeatedly until a solution condition is reached. However, the two algorithms exhibit contrasting patterns of computation and I/O:

- In BFS, subtasks are executed in a level-synchronous fashion, which means that only vertices at a fixed distance from the source (the "frontier") are active in any iteration. In PR, computation is required at every vertex in each iteration.

- In each iteration of PR following initialisation, all graph data is read and all algorithmic state is updated. In contrast, the amount of data to be read or written in a BFS iteration is dependent on the degree distribution of vertices in the frontier set.

Results from our evaluation show that although the iGPU outperforms the dGPU when operating from host memory for both algorithms, the CPU typically performs better than the iGPU on the I/O-intensive PageRank workload. Further analysis is provided in Section 4.

3.2 Graph Structure and Layout

Since our focus in this study is on comparative performance between heterogeneous processors, we have adopted a straightforward graph data format, ensuring that it is preserved on each device. We use an adjacency list representation to store a graph's edges, with an associated index to allow efficient iteration. Prior to running each algorithm, we read the adjacency and index data from disk into arrays pinned in host memory. Since this step is necessary for all trials, it is excluded from the reported runtime figures.

In addition to each graph's structural data, each algorithm stores state in auxiliary data structures. We allocate buffers in local device memory to store these structures. In the case of the iGPU, local device memory refers to an area of RAM reserved as cache by the OpenCL runtime platform. The main graph data structures were also loaded into local device memory in some trials, if sufficient capacity was available. In other cases, the adjacency list and index arrays were read directly from host memory by the GPU device. This is also known as a "zero-copy" transfer, as blocks of data are only loaded from host memory when required.

For the trials, we used two forms of synthetic random graph. The R-MAT [3] graph generator synthesises graphs with a scale-free degree distribution, mirroring the structure of many real-world networks. For comparison, we also used an Erdos-Renyi [5] (ER) random graph generator, which produces graphs with a binomial degree distribution. We generated a variety of graphs with differing sizes and degrees (the degree of a graph is the average number of neighbours per vertex). Unless otherwise specified, generated graphs have a degree of 16, and are referred to by their "scale". For instance, a "Scale-22" graph has 2^{22} vertices (these are the default parameters adopted by the Graph500 benchmark [14]).

3.3 Platform and Implementation

To run the experiments in the evaluation, we assembled a PC based on the components described in Table 1. Henceforth, we refer to the integrated GPU as an "iGPU", and the discrete GPU as a "dGPU". The device-local and host DMA read speeds in the table were measured using AMD's APP SDK [1]. The APU we used is based on AMD's "Kaveri" architecture, which was released in January 2014. The installed dGPU provides a comparison point with

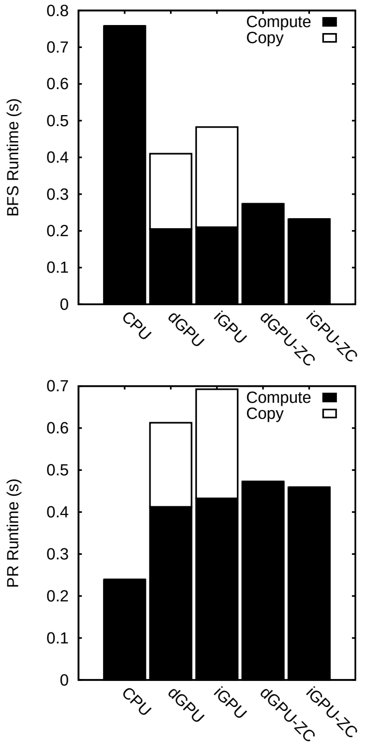


Figure 1: Running times on RMAT Scale-20 graph, for BFS (top) and PR (bottom). ZC denotes Zero Copy mode, where graph data is read from a host-allocated buffer.

the iGPU, and is connected via a PCI Express 3.0 bus. The dGPU was also equipped with one gigabyte of dedicated GDDR5 graphics memory. GDDR5 is optimised for high bandwidth streaming throughput, whereas DDR3 is optimised for low latency at the expense of bandwidth. As a result, there is a difference in throughput when operating from local memory on the dGPU. In contrast, the iGPU has higher rates of throughput when operating from host-allocated memory.

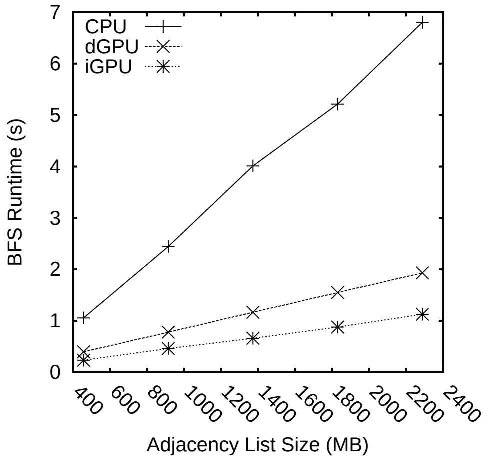
The testing machine was installed with the Windows 8.1 64-bit operating system, with the AMD Catalyst 14.1 Beta device driver, which provides OpenCL support for both the iGPU and dGPU. We developed kernels for the BFS and PR algorithms with OpenCL, which were designed to work with the graph format described in the previous section.

4. EVALUATION

We begin by examining performance on a graph that can be held within dedicated RAM on the discrete GPU card, before analysing performance on larger and more varied graph data. Based on these results, we can then predict whether or not the heterogeneous computation models described earlier can be applied in this environment. Our main finding is that heterogeneous switching during BFS is unlikely to yield major performance benefits, however heterogeneous partitioning may be feasible in some circumstances.

4.1 Zero-Copy Efficiency

For our first set of trials, we used an RMAT graph with 2^{20} (approximately one million) vertices and 16 million undirected edges. The resulting graph data occupies approximately 500 megabytes of memory, which is small enough to fit within a single OpenCL



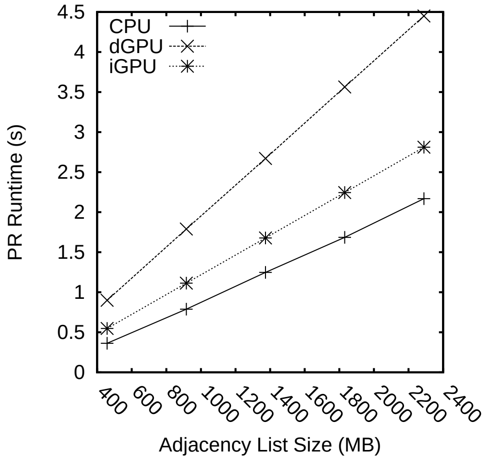


Figure 2: Running times for BFS (top) and PR (bottom) on ER graphs of varying size. GPUs operate in zero-copy mode.

buffer object. Figure 1 shows the runtime performance of BFS and PR with this graph. In the CPU's case, graph data is read from the arrays pinned in RAM. For the iGPU and dGPU, these arrays are first copied into local device memory. As one might expect, the discrete GPU exhibits the fastest compute performance for this workload. However, there is a significant cost in transferring the graph data to the dGPU so that it can be processed.

As discussed earlier, an alternative approach is to map a buffer into host memory, and load data directly from the host when required. This approach is known as "zero copy" mode, as it avoids the need to allocate and transfer data in advance. The results for BFS show that the runtime performance of the iGPU in zero-copy mode is better than the dGPU operating from local device memory, after adding the time taken to finish the initial copy. However, the results for PageRank show that the CPU outperforms both forms of GPU, regardless of the memory transfer mode. These results are indicative of what we found in most trials on this testing platform.

Another reason to make use of zero-copy host memory is that it allows access to data that exceeds the local memory capacity of a device, without requiring the developer to perform the cumbersome task of splitting up the source data into manageable chunks for processing in local memory. For the remainder of the results reported in this section, we have used the zero-copy method to map the graph data structures.

In Figure 2, we show how the running time scales as we increase the size of the input graph for BFS and PR. For these trials, we generated graphs with a constant average degree (set to 20 neighbours), but with a varying number of edges, between 20 million to 100 million. The Erdos-Renyi generator was used to create these graphs, since it allows free control of the desired number of vertices in the graph (whereas the RMAT generator requires the number of vertices to be a power of 2). The figures show the running times plotted against the memory capacity requirements for each graph. Note that the dGPU has a maximum local memory capacity of one gigabyte, but no change is evident in the running time figures for graphs that exceed this threshold. The likely explanation for this is that neither algorithm exhibits data access patterns that are well suited to caching (which is normally the case for irregular graph applications).

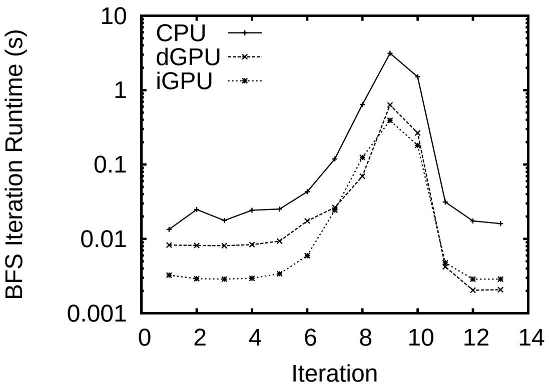


Figure 3: Breakdown of running time per iteration for BFS on ER graph with an average of four neighbours per vertex.

The results reinforce the trends evident in the first figure: in the compute-dominant BFS workload, the GPUs outperform the CPU when using an identical OpenCL kernel, by a factor of 3× (dGPU) to 6× (iGPU), whereas for the I/O-intensive PageRank workload, the CPU outperforms the GPUs by a factor of 1.2× to 2×. In the following sections, we turn our attention to the heterogeneous computation schemes discussed in Section 2.2, exploring scenarios where these may be applicable on this platform.

4.2 Heterogeneous Switching

Recall that the switching method involves shifting all computation to a single device at a particular point in the algorithm, based on the nature of the work that is about to be processed. Prior work by Hong et al. [7] has focused on building a hybrid CPU/GPU workflow for BFS, based on this model. In that work, the authors found that the CPU was more suited to handling iterations in the distributed computation where only a few vertices existed in the frontier set, but would switch to processing on the GPU when exponential growth in the frontier set was observed.

To test the feasibility of a similar scheme in this environment, we looked at the running time in each iteration of BFS on each device. Figure 3 shows one such trace, for BFS on an ER graph with 16 million vertices and 64 million edges. In each trace, the only point at which the CPU outperforms the GPUs is in the first iteration. In BFS, the first iteration is a trivial step, which does not require any parallelism. However, for all remaining iterations (in every BFS trial), the performance on the GPUs ran faster than the OpenCL CPU implementation.

In the trace shown in the figure, the running time for the 8th iteration on the dGPU was faster than the iGPU. In all of the traces

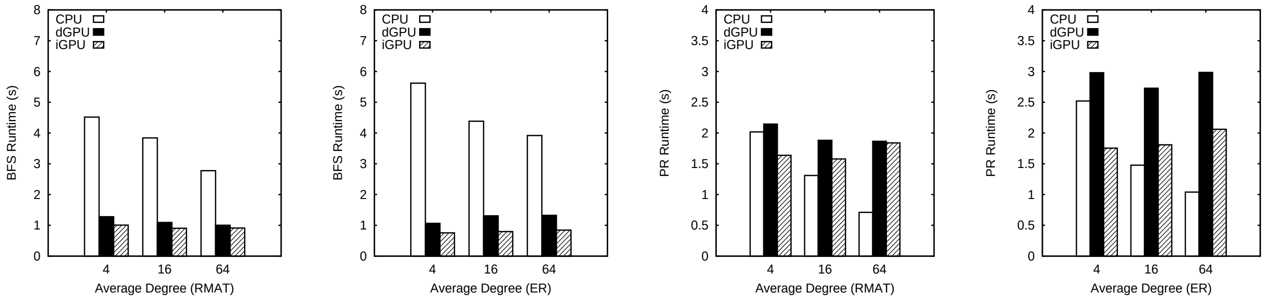


Figure 4: Running times for BFS (left) and PR (right) on RMAT and ER graphs with varying mean degree and a fixed number of edges.

we examined, this was one of the few cases where this condition existed. In theory, it would be possible to take advantage of the runtime savings (which amounts to approximately 60 milliseconds, offset by the cost of shifting algorithmic metadata which we estimate to be 5-10 milliseconds in this instance). However a predictive mechanism is also required, to allow a switching condition to be recognised. When comparing with results from other runs, we found several other instances where the same level of growth in the frontier set did not lead to improved performance on the dGPU.

At this stage, we cannot find any evidence to suggest that hybrid switching is a feasible tactic on this testing platform. Further work is required to determine whether an optimised CPU BFS implementation could change this.

4.3 Heterogeneous Partitioning

As described in Section 2.2, a common cause of performance degradation in SIMT GPU applications is thread divergence. In both of these algorithms, the degree distribution of processed nodes directly impacts the amount of work that needs to be completed per vertex. In Figure 4, we show the impact of varying the degree distribution on BFS and PR. For these trials, we synthesised graphs with a fixed number of edges (2^{26} or approximately 64 million), and with the number of vertices set to either 2^{24}, 2^{22} or 2^{20}, resulting in graphs with a mean degree of 4, 16 or 64 neighbours per vertex respectively. We generated graphs using both the ER and RMAT generators for this experiment – recall that the degree distribution of the ER graphs is more tightly clustered around the mean degree.

The results for BFS show that the performance gap between the CPU and GPUs reduces as the average degree increases. However, the effect on running times for the GPU devices is minimal. The PageRank results show more marked differences; similarly to the BFS results, the CPU's performance increases as the average degree of the input graph increases, whereas the GPU running times remain unaffected. However, in this case, for the graphs with the lowest average degree, CPU performance degrades to a level that is poorer than the iGPU (but not the dGPU).

These results indicate that a simple partitioning scheme, where the GPU processes low degree vertices only, might lead to a performance improvement. To test this hypothesis, the PageRank kernel was modified to accept an additional parameter, namely the "partition point". On the GPU, all vertices with degrees greater than the partition point are rejected, and vice versa for the CPU. Using this new kernel, the test launcher was modified to instantiate two OpenCL contexts concurrently, on the CPU and iGPU. After each iteration, it is necessary to synchronise the PageRank value vector across both partitions, however this is a relatively inexpensive operation compared to the runtime of the main kernel. By varying the partition point, it was then possible to find a point where the running time of each kernel was similar.

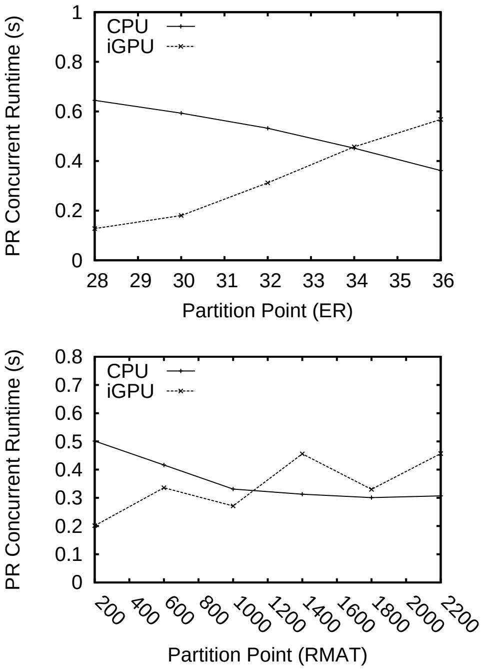


Figure 5: Running times for the hybrid partitioning prototype, on the ER (top) and RMAT (bottom) scale-22 graphs. The partition point was varied to find the optimal balance in runtimes.

The results of the hybrid partitioning prototype is shown in Figure 5, when tested on the ER and RMAT scale-22 graphs. In the case of the ER graph, there is a clear crossover point, just above the mean number of edges per vertex. For the RMAT graph, variance in the iGPU runtime means that the crossover point is not as clear, however there is a range of partition values that will yield near-optimal performance. Comparing these results with the CPU-only trials reveals that the hybrid partitioning prototype yields a speedup of approximately $1.6\times$ on the ER graph and $2\times$ on the RMAT graph.

5. RELATED WORK

The results from this pilot study seem to indicate that graph computation on APUs will be a promising area for further investigation. In this section, we will explore possible extensions and optimisations of this work, in the context of related research.

Heterogeneous switching for breadth-first search was previously investigated by Hong et al. [7]. Memory transfer overhead made switching back forth between the CPU and GPU infeasible in that study. Although the APU platform reduces the transfer overhead, in our trials the CPU was simply not as efficient as the GPUs. The CPU implementation of the OpenCL kernel may not take advantage of the differences in device architecture. In Hong's work, the CPU-based implementation of BFS uses the classical queue-based algorithm, adapted to multiple threads. Numerous other optimisations for multi-threaded BFS have been posited in the literature, for both CPUs [2, 4] and for GPUs [13, 11, 10]. A possible direction for future research could be to investigate how these algorithms could interoperate in the APU platform.

We showed that the CPU and GPU cores in an APU can work effectively in unison, when processing different types of data. Instead of determining such partitions dynamically at runtime, an alternative approach would be to statically pre-process the graph into an optimally partitioned format. Graph partitioning is a well-studied domain, with tools such as METIS [8] allowing graph data to be organised with balanced cuts. However, finding such cuts is computationally expensive, which has led other systems to employ random partitioning schemes; Pregel [12] is an example of a distributed graph processing system that takes this approach. Future research may seek to investigate how partitioning schemes can be adapted to heterogeneous processors.

We intend to explore other GPU-based graph algorithms, and possibly optimise these for the APU platform. Other research has explored how to take advantage of architectural features such as memory coalescing [6]. A number of other papers have analysed optimisations for multi-GPU platforms [20, 18, 9] – this need to be investigated in the context of the APU environment.

Our broader aim is to investigate how heterogeneous (and in particular, APU-based) processing can augment existing execution platforms for graph computation. Systems such as Galois [15] and Ligra [19] provide a simple programming interface, and a multi-threaded engine for executing these programs, but do not currently utilise GPU-based execution. In contrast, Dandelion [17] is an example of an existing system that takes a task parallel approach to executing programs across heterogeneous devices.

6. CONCLUSION

This investigation has shown that APUs can provide an effective platform for graph computation. The high bandwidth channel between the CPU and iGPU cores allows computation to proceed faster than it does on an externally-attached discrete GPU. This is true for both the compute-intensive graph traversal workload, and for the I/O-intensive PageRank workload. Furthermore, by adapting the PageRank algorithm to employ heterogeneous partitioning, the combined performance of the APU outperforms the CPU by a factor of up to $2\times$.

Further work is required to extend the scope of this study to a wider variety of algorithms, graph types, CPU/GPU optimisations and other types of accelerator hardware.

7. REFERENCES

[1] `http://developer.amd.com/`.

[2] AGARWAL, V., PETRINI, F., PASETTO, D., AND BADER, D. A. Scalable graph exploration on multicore processors. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (2010).

[3] CHAKRABARTI, D., ZHAN, Y., AND FALOUTSOS, C. R-mat: A recursive model for graph mining. *Computer Science Department* (2004), 541.

[4] CHHUGANI, J., SATISH, N., KIM, C., SEWALL, J., AND DUBEY, P. Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency. In *Parallel Distributed Processing Symposium (IPDPS)* (2012).

[5] ERD6S, P. On the evolution of random graphs. *Publ. Math. Inst. Hungar. Acad. Sci 5* (1960), 17–61.

[6] HONG, S., KIM, S. K., OGUNTEBI, T., AND OLUKOTUN, K. Accelerating cuda graph algorithms at maximum warp. In *Principles and Practice of Parallel Programming (PPoPP)* (2011).

[7] HONG, S., OGUNTEBI, T., AND OLUKOTUN, K. Efficient parallel graph exploration on multi-core cpu and gpu. In *Parallel Architectures and Compilation Techniques (PACT)* (2011), pp. 78–88.

[8] KARYPIS, G., AND KUMAR, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput. 20*, 1 (1998).

[9] KIM, J., KIM, H., LEE, J. H., AND LEE, J. Achieving a single compute device image in opencl for multiple gpus. In *Principles and Practice of Parallel Programming (PPoPP)* (2011).

[10] LI, D., AND BECCHI, M. Deploying graph algorithms on gpus: An adaptive solution. In *Parallel Distributed Processing (IPDPS)* (2013).

[11] LUO, L., WONG, M., AND HWU, W.-M. An effective gpu implementation of breadth-first search. In *Design Automation Conference (DAC)* (2010).

[12] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: a system for large-scale graph processing. In *PODC* (2009).

[13] MERRILL, D., GARLAND, M., AND GRIMSHAW, A. Scalable gpu graph traversal. In *Principles and Practice of Parallel Programming (PPoPP)* (2012).

[14] MURPHY, R. C., WHEELER, K. B., BARRETT, B. W., AND ANG, J. A. Introducing the graph 500. *Cray User?s Group (CUG)* (2010).

[15] NGUYEN, D., LENHARTH, A., AND PINGALI, K. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), SOSP '13.

[16] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The pagerank citation ranking: Bringing order to the web, 1999.

[17] ROSSBACH, C. J., YU, Y., CURREY, J., MARTIN, J.-P., AND FETTERLY, D. Dandelion: A compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 49–68.

[18] SCHAA, D., AND KAELI, D. Exploring the multiple-gpu design space. In *Parallel Distributed Processing (IPDPS)* (2009).

[19] SHUN, J., AND BLELLOCH, G. E. Ligra: A lightweight graph processing framework for shared memory. In *Principles and Practice of Parallel Programming (PPoPP)* (2013).

[20] SPAFFORD, K., MEREDITH, J. S., AND VETTER, J. S. Quantifying numa and contention effects in multi-gpu systems. In *Workshop on General Purpose Processing on Graphics Processing Units* (2011).