Comput Sci Res Dev
DOI 10.1007/s00450-012-0209-1

SPECIAL ISSUE PAPER

Performance characterization of data-intensive kernels on AMD Fusion architectures

Kenneth Lee · Heshan Lin · Wu-chun Feng

Abstract The cost of data movement over the PCI Express bus is one of the biggest performance bottlenecks for accelerating data-intensive applications on traditional discrete GPU architectures. To address this bottleneck, AMD Fusion introduces a fused architecture that tightly integrates the CPU and GPU onto the same die and connects them with a high-speed, on-chip, memory controller. This novel architecture incorporates shared memory between the CPU and GPU, thus enabling several techniques for inter-device data transfer that are not available on discrete architectures. For instance, a kernel running on the GPU can now directly access a CPU-resident memory buffer and vice versa.

In this paper, we seek to understand the implications of the fused architecture on CPU-GPU heterogeneous computing by systematically characterizing various memory-access techniques instantiated with diverse memory-bound kernels on the latest AMD Fusion system (i.e., Llano A8-3850). Our study reveals that the fused architecture is very promising for accelerating data-intensive applications on heterogeneous platforms in support of supercomputing.

Keywords GPU · AMD Fusion · Memory transfer

This work was supported in part by an AMD Research Faculty Fellowship and NSF grant IIP-0804155 for the NSF I/UCRC Center for High-Performance Reconfigurable Computing (CHREC).

K. Lee · H. Lin · W.-c. Feng (✉)
Department of Computer Science, Virginia Tech, Blacksburg, VA, USA
e-mail: feng@cs.vt.edu

K. Lee
e-mail: klee1@cs.vt.edu

H. Lin
e-mail: hlin2@cs.vt.edu

Published online: 23 May 2012

1 Introduction

Graphics processing units (GPUs) continue to be increasingly used for general-purpose computing, particularly in the sciences and engineering. Their high peak performance capabilities led to their inclusion into three of the top five fastest supercomputers on the TOP500 List [13]. The ubiquity of GPUs in home computers, via the commercial market for gaming, has also accelerated their adoption for use in HPC environments, especially for desktop HPC applications.

While GPUs can provide a tremendous amount of processing power, the performance of data-intensive applications on GPUs can be bottlenecked by the cost of data communication over PCI Express (PCIe). To best utilize a heterogeneous platform, it is often desirable to use fine-grained kernels and assign each kernel to the most appropriate device, due to the radical architectural differences between CPUs and GPUs. For instance, kernels with many divergent branches will execute more efficiently on the CPU. Fine-grained co-processing will increase CPU-GPU communication, further exacerbating the data communicating bottleneck on traditional discrete GPUs.

New "fused" CPU-GPU architectures, i.e., accelerated processing units (APUs), such as AMD Fusion, alleviate the PCIe bottleneck by placing the CPU and GPU on the same die, interconnected with high-speed memory controllers. The new architecture also features a single shared memory between the CPU and GPU. As a consequence, the CPU can directly access memory buffers on the GPU, and vice versa, enabling new memory-access techniques for heterogeneous computing.

In this paper, we seek to understand the challenges and opportunities brought by fused architectures to data-intensive GPU computing by systematically characterizing

the behavior of AMD Fusion with different memory-access techniques. Memory-bound kernels are used to illustrate the impact of these techniques on application performance. We find that although AMD Fusion offers a fast interconnection between the CPU and the GPU, taking advantage of this new architecture is nontrivial because of the performance heterogeneity along various memory paths in the system. Nonetheless, with proper memory access design, fused architectures can significantly outperform discrete architectures for memory-bound kernels.

The rest of the paper is organized as follows. Section 2 presents an overview of the fused architecture, specifically how it is different from a traditional discrete GPU. Section 3 describes the experimental setup and gives a brief overview of the applications under test. Section 4 presents our results. Section 5 provides a discussion on modeling the performance of applications. Section 6 outlines our related work. Finally, Sect. 7 presents a discussion of the implications of the results as well as opportunities for future work.

2 Background

In this section, we discuss GPU and APU architecture basics and present a brief overview of the OpenCL programming model.

2.1 GPU and APU architectures

GPUs and APUs share most of the same architectural features in their *compute units*, the heart of the computational power for a GPU or APU. However, the two differ in how their units are attached to the rest of the system. A compute unit is somewhat analogous to a multi-core chip. It contains registers, local memory, constant memory, and a single-instruction, multiple-data (SIMD) engine. This SIMD engine contains multiple computation cores called *processing elements* (PEs). Instructions to these cores are dispatched in lockstep, such that every PE performs either the same instruction or a nop (i.e., no operation) on every cycle. The local and constant memories of the compute unit act as user-managed caches of the high-latency global memory. There is a single global memory on the device, which is where communication between the host and device take place; it is also the location where persistent data is stored and loaded for kernel use. Every compute unit has access to the global memory. Data in the local and constant memories are not consistent between kernel executions.

The GPU and APU differ by where they are located in the system. The GPU is a stand-alone device separated from the CPU by the PCIe bus. The APU, on the other hand, is a fused architecture, where the CPU and GPU cores are co-located on the same die. In addition, the APU has a single shared memory space between the processors. Accesses

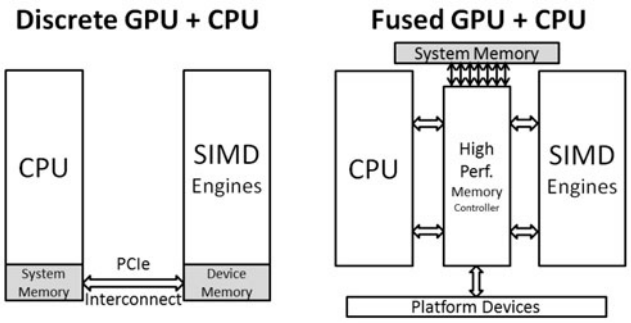

Fig. 1 Architectural differences of the discrete and fused GPUs

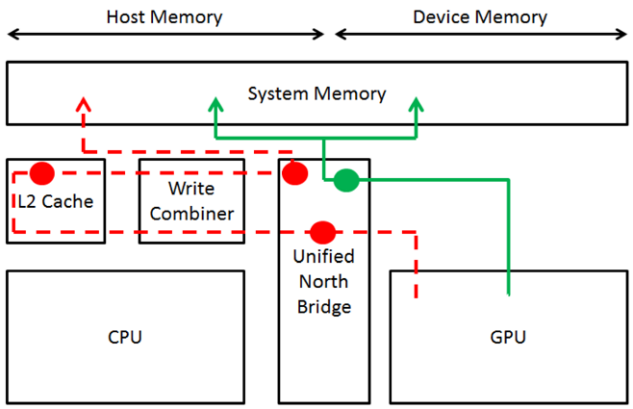

Fig. 2 Detailed APU Memory Architecture, showing the GARLIC (*solid*) and ONION (*dashed*) routes of data access from the GPU

to this shared memory space are facilitated through a high-speed memory controller. In reality, the memory space is still partitioned due to the relaxed constraints of GPU memory, but the memory controller allows for the user to see an apparent shared memory between the cores, while also allowing for faster transfers when compared to the PCIe bus. Figure 1 shows the differences between the discrete and fused GPU configurations.

2.2 APU memory paths

We present a brief overview of the APU memory system to better understand the performance of the different memory paths. Figure 2 shows a detailed model of the memory architecture of the AMD Fusion APU [3].

All memory accesses by the GPU to system memory go through the *Unified NorthBridge* (UNB), which is responsible for managing all GPU memory accesses. If the memory being accessed by the GPU is either device memory or uncached CPU memory, the GPU will gain access to it directly after the UNB, through the Radeon Memory Bus (aka the Garlic route). If the memory needed by the GPU is in cached CPU space, then the UNB must snoop the L2 cache of the CPU to ensure coherency of memory. Accesses of this type

are done through the AMD Fusion Complete Link (aka the Onion route), which has lower performance compared to the Garlic route.

CPU writes to cached host memory buffers are done through the L2 cache, as is typical for CPU systems. Uncached memory reads and writes to both host and device memory spaces are done through the *write combine* buffer, which is used to increase performance for writes to memory in contiguous memory spaces. Reading from the device memory by the CPU is very slow, because each read to device memory is uncached, and only one outstanding read is allowed at any given point.

2.3 OpenCL framework

The work presented here uses the OpenCL framework for GPU communication and computation. OpenCL is a vendor-agnostic parallel computing framework from the Khronos Group [10] that uses the following terminology. Each thread running in a kernel on a device is called a *workitem*. These workitems are combined into a *workgroup*. Workgroups are assigned to the compute units of the device, which can be a processor core on CPUs or a SIMD engine on GPU devices. Each workgroup can execute on only one compute unit. We will use the OpenCL terminology to discuss the work for the rest of this paper.

3 Characterization methodology

We aim to systematically investigate the challenges and opportunities introduced by fused CPU+GPU architectures to data-intensive computing. To this end, we first seek to quantify the performance difference in memory accesses on two types of architectures by running the BufferBandwidth micro-benchmark, shipped with the AMD SDK, on an AMD A8-3850 APU (Llano) and a AMD Radeon HD5870 discrete GPU. (Section 4 provides details about the two devices.) The BufferBandwidth benchmark exercises all possible memory-access paths in the system. Since the memory-access performance is impacted by the CPU performance, for a fair comparison, both the Llano and HD5870 were installed in the same machine.

Table 1 shows the memory performance results reported by the BufferBandwidth benchmark. There are several key observations from these results. *First*, copying data between a host buffer and a device buffer is much faster, i.e., by more than two-fold for both reads and writes, on Llano than the HD5870. This is mainly due to the fast interconnection between the CPU and the GPU on Llano.

Second, accessing host memory from the GPU is significantly faster on Llano, with an 11-fold improvement on read and a 3.9-fold improvement on write performance. Note that the HD5870 performance on reads and writes are as fast as memory transfers over the PCIe bus. This is due to the fact that these communications must take place over PCIe bus. Interestingly, on Llano, accessing the host memory from the GPU is also as fast as accessing device memory (16.26 GB/s vs. 17.54 GB/s). This opens up the possibility of having the GPU directly access data on the host side without copying them over for applications with little data reuse. It is worth noting that when accessing the host memory on Llano, the read bandwidth is also four-fold higher than the write bandwidth. This bandwidth imbalance needs to be carefully taken into account in designing efficient data-access flow for applications.

Table 1 BufferBandwidth Benchmark Results for AMD A8-3850 APU and AMD HD5870 GPU. The first two rows of data represent the transfer time between the host and device memory buffers, which is over the PCIe bus for discrete GPU systems. The remaining rows represent the read and write performance of the specified processor directly on the specified memory buffer

Transfer Type	AMD Llano	AMD HD5870
Host Buffer → Device Buffer	2.61 GB/s	1.25 GB/s
Host Buffer ← Device Buffer	3.17 GB/s	1.39 GB/s
CPU ← Host Buffer (Read)	5.67 GB/s	5.64 GB/s
CPU → Host Buffer (Write)	5.46 GB/s	5.44 GB/s
GPU ← Host Buffer (Read)	16.26 GB/s	1.46 GB/s
GPU → Host Buffer (Write)	4.96 GB/s	1.28 GB/s
CPU ← Device Buffer (Read)	0.01 GB/s	0.01 GB/s
CPU → Device Buffer (Write)	7.49 GB/s	1.52 GB/s
GPU ← Device Buffer (Read)	17.54 GB/s	128.74 GB/s
GPU → Device Buffer (Write)	14.31 GB/s	98.60 GB/s

Third, when accessing the device memory of the CPU, Llano again offers a much higher (i.e., *five-fold*) write bandwidth than the HD5870. Surprisingly, the read bandwidth from a device buffer to the CPU is awfully low at 0.01 GB/s for both Llano and HD5870. This is because each read to device memory is uncached, and only one outstanding read is allowed at any given point, as explained in Sect. 2.2. This slow read link can be a serious performance bottleneck if overlooked in the application design.

Finally, the memory bandwidth on the GPU device is much higher on HD5870 than on Llano. This is partly because the HD5870 is equipped with faster memory than Llano, i.e., DDR5 vs. DDR3. The HD5870 also has *four times* as many compute units than Llano, i.e., 20 vs. 5, which improves the aggregate memory access bandwidth.

From the above observations, we see that the fused architecture possesses a more diverse memory-access profile along various paths in the system. Although Llano offers more efficient interconnection between the CPU and the GPU, designing a data-access strategy that can best take advantage of this fast interconnection is nontrivial given the

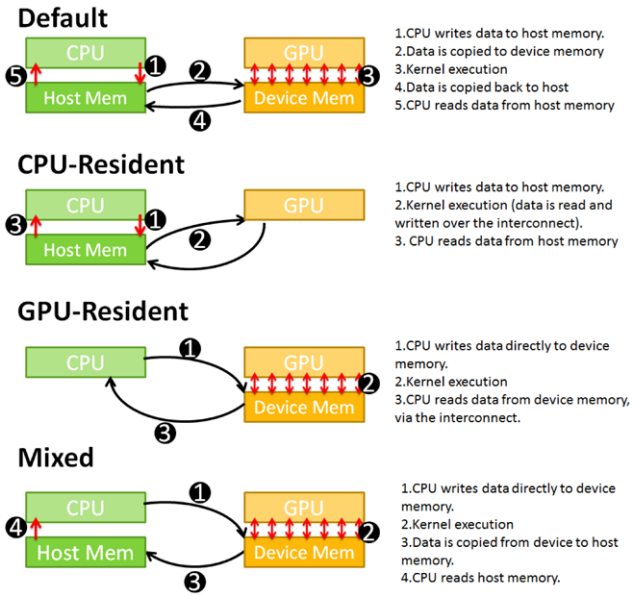

Fig. 3 Memory movement schemes

performance heterogeneity along different memory paths. To systematically evaluate the potential of fused architectures, we perform a comparison study between APUs and discrete GPUs by characterizing the behavior of a set of memory-access approaches instantiated with diverse memory-bound kernels, as described in the following sections.

3.1 Memory-Access Techniques

In our characterization study, we consider four different memory-access approaches for a GPU kernel: *Default*, *CPU-Resident*, *GPU-Resident*, and *Mixed*. Figure 3 provides a pictorial explanation of these methods.

The **Default** approach is the most commonly used data-access approach on discrete GPUs. With this approach, the input data is first copied from a host-side buffer to a device-side buffer. The kernel then reads the input data from and writes output data to device memory. Upon kernel completion, the output data is copied back to a host-side buffer.

The **CPU-Resident** approach stores all of the data in the host memory. The GPU kernel reads the data from and writes data to host-side buffers instead of device-side buffers. Compared to the Default approach, the CPU-Resident approach avoids data copies for both the input and output data to and from the device-side buffer. In discrete GPU systems, this memory-access approach causes an overhead because reads and writes to host-side buffers from the GPU must go across the PCIe bus.

The next approach is called **GPU-Resident** because all of the data of an application resides in the device memory. The CPU writes the input data directly to a device-side buffer, and then the GPU kernel operates on this buffer and writes the output data to another device-side buffer. Afterwards, the output data is read directly from the device-side buffer by the CPU. Compared to the Default approach, the GPU-Resident approach saves the copy of the input data from the host memory to the device memory, which is in favor of fused architectures with fast write bandwidth from the CPU to the device memory. Even on APUs such as Llano, reading a device-side buffer directly from the CPU is very slow as discussed above.

To address slow read performance of device memory by the CPU in the GPU-Resident approach, we designed a **Mixed** approach in which the data is written from the CPU directly to the device-side buffer, and then after the kernel completes the data is copied to a host-side buffer, as in the Default method. At that point, data can be read at normal speeds by the CPU. Note that copying a buffer from device memory to host memory is much faster than having the CPU read data elements within the device-side buffer.

3.2 Kernel benchmarks

We choose four memory-bound applications that have a variety of both memory accesses in the kernel as well as the amount of data needed to be transferred between the CPU and the GPU.

VectorAdd This application computes the value $\mathbf{C} = \mathbf{A} + \mathbf{B}$ for vectors of length n. The work for this application can be split by data partitions, and every three units of memory transfer requires only one addition operation, making this kernel extremely data-intensive. Each thread launched will compute one value in **C** by loading the necessary values from **A** and **B**.

Scan This application computes the exclusive prefix sum for a vector, **V**, of length n. That is $X_i = \sum_{k<i} V_k$. This computation requires a lot of synchronization between threads in order to be performed effectively in parallel. There are two steps for this algorithm, which both resemble a reduce operation. Each thread computes an element of the final result and may perform up as many local memory operations as there are number of threads in the workgroup.

Reduce The Reduce application computes the sum of a vector, **V**. To be performed in parallel, synchronization must occur between threads. There is roughly one addition operation preformed for each element of **V**. Each thread loads a value from the input into a local array, and then performs a reduction on that local array by having each thread in the group perform a binary tree reduction.

CRC This algorithm computes the *cyclic redundancy code* for a given stream of bits, **B**. CRC is a hash function and returns only a single value back to the host after completion. Each thread is responsible for a single byte of the data stream, and performs multiple data accesses to local memory. Afterwards, a reduction is performed for threads in a workgroup, and the workgroup writes the value of the CRC back to global memory.

All four of the applications send n elements of data to the accelerator. VectorAdd and Scan both return n elements, while the Reduce and CRC applications only return a single data element. Also, in each of these groupings there is one computationally inexpensive kernel, and one more intensive kernel. The VectorAdd and Reduce applications are computationally simple. On the other hand, the Scan and CRC applications require much more computation to complete. VectorAdd and Reduce require one read from global memory, while the Scan and CRC require more synchronization and memory accesses.

3.3 Comparison between fused and discrete architectures

One key question that we seek to answer is what are the advantages of fused architectures over discrete architectures for memory-bound kernels. Ideally, in comparing two architectures, we would like to use two GPU devices with the same processing power but different interconnection to the CPU. However, the GPU integrated in fused architectures today is less powerful than typical discrete GPUs. That is, Llano only has 5 compute units, but HD5870 has 20. One useful technique for this purpose is *device fission*, an OpenCL extension that allows dividing an OpenCL device into multiple sub-devices. For instance, we can create a sub-device out of HD5870 that consists of 5 compute units. This extension is being mainlined into the OpenCL v1.2 specification, but as of writing this feature has not be implemented for the AMD HD 5870 GPU we used for our testing purposes.

We address the above issue by controlling the number of workgroups of an OpenCL kernel. Since a workgroup can only be scheduled on one compute unit, we can have a kernel use a portion of the GPU device by using a number of workgroups less than the number of the compute units of the GPU. Each thread then supports computation of multiple data points to support various input sizes. Llano has 5 compute units while the HD5870 has 20. We use 5 workgroups in a GPU kernel when comparing the two to make the performance capabilities of the two devices comparable. We ensure that each workgroup is mapped to a different compute unit by allocate all of the local memory available for a compute unit. Because we allocate all of the local memory, the scheduler is unable to place any other workgroups on that compute unit.

Table 2 Test systems

Platform	Llano	HD5870
Stream Processors	400	1600
Memory Bus Type	DDR3	GDDR5
Device Memory	512 MB	1024 MB
Local Memory	32 KB	32 KB
Local Workgroup Size	256 threads	256 threads
Core Clock Frequency	600 MHz	850 MHz
Peak FLOPS	480 GFlops/s	2720 GFlops/s
CPU Clock Frequency	2.9 GHz	2.9 GHz

4 Characterization results

We first compare the performance of the two different memory paths, i.e., Garlic vs. Onion bus on the APU. Then we evaluate the efficacy of different memory access approaches on both discrete and fused architectures with memory-bound kernels described in Sect. 3. Finally, we evaluate the potential advantages of fused architectures over discrete architectures in accelerating two representative memory-bound kernels. We next present a detailed analysis on some of the observations in experimental results in Sect. 5.

The configurations of the systems used in our study are given in Table 2. We performed our experiments using the AMD APP SDK v2.5, which supports OpenCL version 1.1. The Windows 7 operating system was used for all of our tests.

4.1 Garlic vs. onion routes

We performed two tests to test the difference in performance when comparing the Garlic and Onion routes on the fused architecture. First we performed another BufferBandwidth test on cached and uncached memory on the GPU. GPU reads into uncached host buffer memory, via the Garlic route, achieved a maximum bandwidth of 16.26 GB/s. Reads to cached memory only achieved 6.36 GB/s.

In addition to seeing the results of the BufferBandwidth tests, we ran an experiment that augments the VectorAdd kernel[1] to use cached memory access instead of uncached ones. Using the Onion route had a negative impact on overall performance of the kernel, as shown in Fig. 4. As the problem size increases, we see an increased degradation of performance of the Onion route (note that the execution time shown in Fig. 4 is in log scale).

[1] When using CPU-Resident memory, the Garlic route can be accessed using the CL_MEM_(READ/WRITE)_ONLY flags when using the clCreateBuffer function.

4.2 Comparison of memory-access techniques

In order to understand the most efficient way of mapping memory-bound kernels on GPUs, in this experiment, we evaluate the four memory-access approaches with the four benchmark applications as discussed in Sect. 3, on both Llano and HD5870.

Figure 5 shows the performance results on Llano. Data transfer time, kernel execution time and total execution time are reported for each application and memory transfer scheme. The data transfer times reported here were computed as follows. Default memory transfers include the time taken to write to a buffer and copy that buffer to the GPU and also the time to copy the output buffer to the CPU and read from it. The other transfers include the time taken to map, write to, and unmap the buffers, as well as map, read from, and unmap the buffers after the kernel has executed. The Mixed case measures the map/unmap time to the buffer, as in the GPU- and CPU-resident cases. The read from the device is handled as it is for the Default memory movement case.

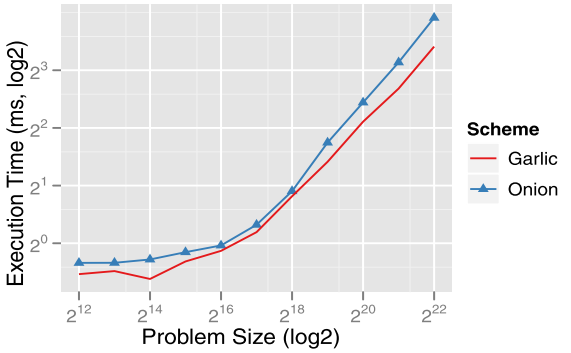

Fig. 4 Effect of Garlic and Onion routes on performance for the VectorAdd application on the Llano system

As can be seen in Fig. 5, the application performance profile varies considerably for different benchmarks. In the VectorAdd and Scan benchmarks, the amount of data being returned by the kernel is proportional to the problem size. As the problem size increases, the time spent on data transfer increases dramatically for the GPU-Resident memory due to the slow speeds of CPU reads from device memory. However, we also notice the higher cost of kernel execution for the CPU-Resident memory case. The other kernel times are almost identical because the memory is located in the same memory space for the other three schemes. In summary, we find the Mixed memory movement scheme shows the best performance for these applications as it is able to achieve good performance in the kernel execution, while also having much better data transfer speeds than the GPU-Resident approach. However, this assessment is dependent on the problem size.

Compared to the VectorAdd and Scan benchmarks, both the Reduce and CRC algorithms have little output data to

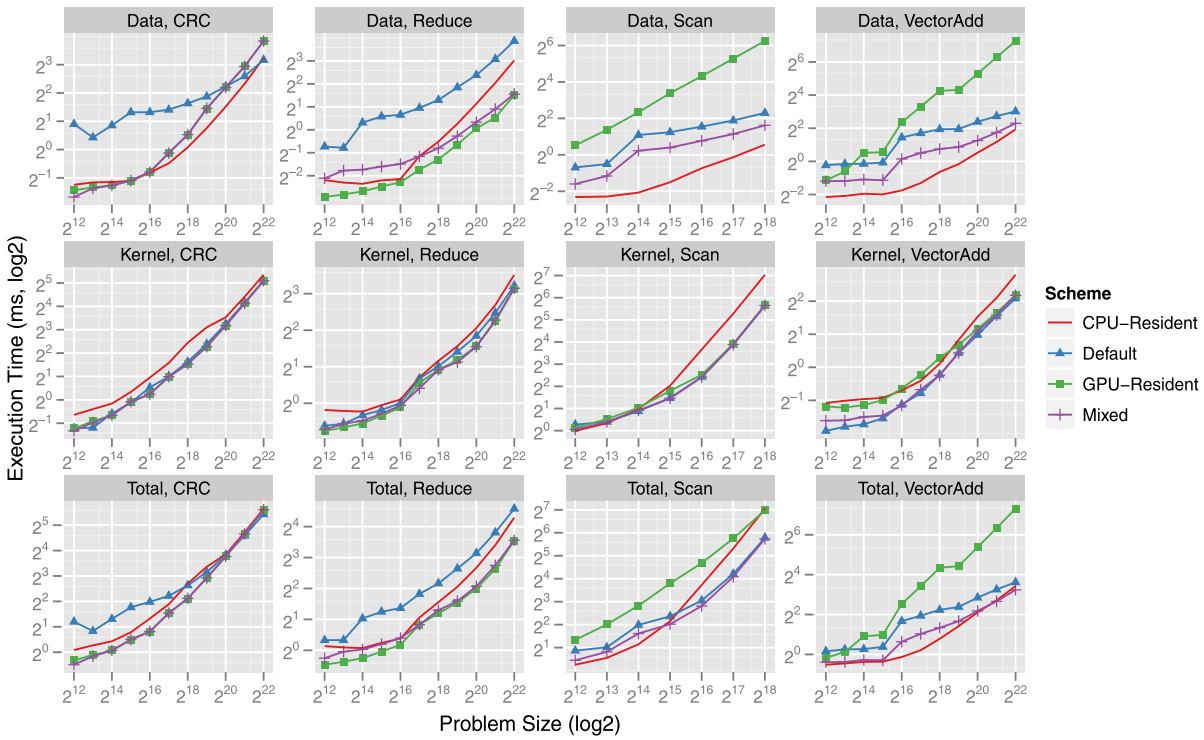

Fig. 5 Application performance broken down into data transfer and kernel execution time for the Llano system

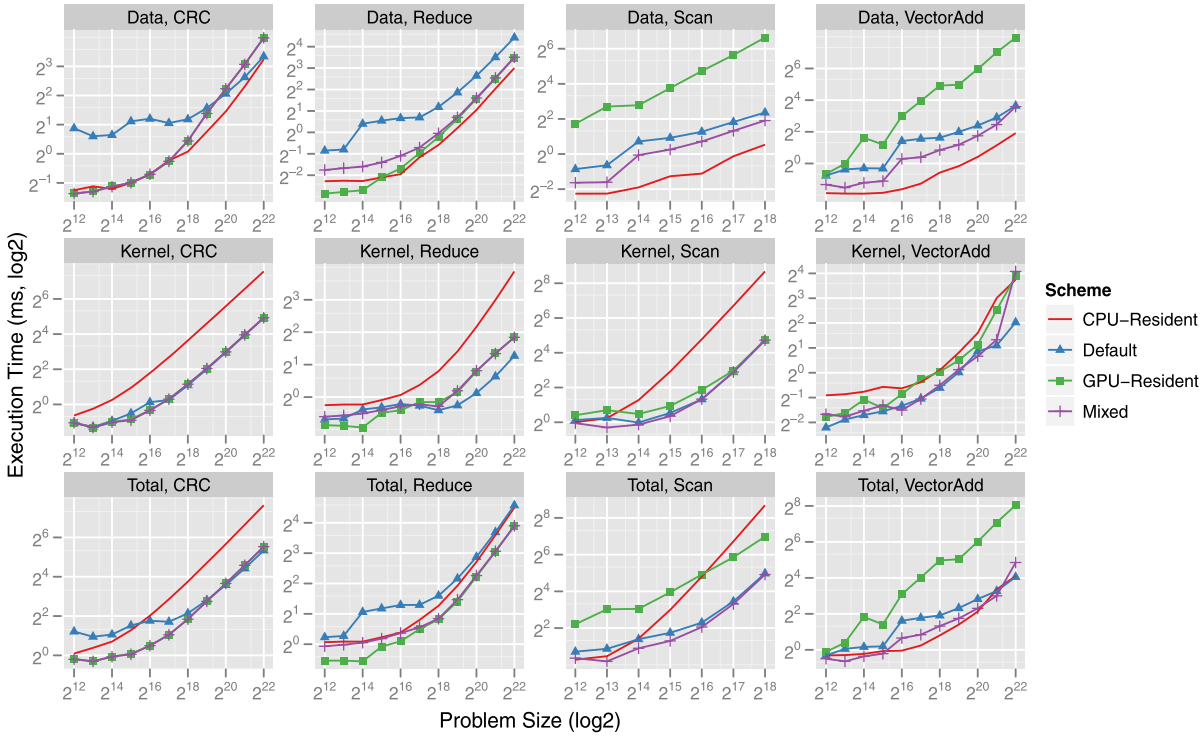

Fig. 6 Application performance broken down into data transfer and kernel execution time for the HD5870 system

return to the host after computation. In fact, GPU-Resident incurs the least data-transfer time for Reduce as this benchmark has only one data element for the output. CRC has a larger amount of output data, thus incur slightly higher data transfer overhead for GPU-Resident on large problem sizes. In terms of overall performance, the Mixed technique typically gives performance for data transfers somewhere between that of the CPU-Resident and the Default techniques. This makes sense logically because the Mixed technique performs on one side of the transfer in a similar way to CPU-Resident and on the other side similar to the Default. Overall, the kernel performance for Mixed is generally identical to the GPU-Resident as well as is near the performance of the Default.

In summary, the main trade-off between CPU- and GPU-Resident memory for the Llano APU is the difference in bandwidth writing data to and from the device and the speed of accesses to memory during kernel execution. Choosing the best memory technique is also dependent on the problem size as well as the execution profile of the kernel.

The results of HD5870 are given in Fig. 6. One observation here is that the Default approach generally delivers the best overall performance. This is because accessing data directly over PCIe is typically slow, thus copying data to/from the device is more efficient. The only exception to this rule is the Reduce application, where Mixed and GPU-Resident outperform Default. This is due to the fact that writing data from the CPU to device memory is more efficient than copying a host-side buffer to a device buffer (1.52 GB/s vs. 1.25 GB/s as shown in Table 1), and there is very little data to read back to the host side.

4.3 Comparing discrete vs. fused architectures

When comparing the Llano and HD5870 systems, we note a few key differences. For the discrete GPU, the kernel performance of CPU-Resident memory was typically very poor. The difference in performance between CPU-Resident memory and the next fastest memory technique is as much as 10 times. This is due to the fact that memory accesses to these units of memory must take place over the PCIe bus, which is high latency.

In order to fairly compare the APU and discrete GPU, we limited the number of compute units of each of the devices to *five*, which is the number of compute units found on the fused APU. Performing our comparison this way, we see a comparison of two roughly equally performing GPUs and can therefore better compare the effects of the fused versus discrete architecture. We assume that one main advantage of the discrete architecture is that it can be equipped with faster memory module and more powerful cores compared to the fused architecture. We would like to understand that under such an assumption, whether the fused architecture will be advantageous in accelerating memory-bound kernels.

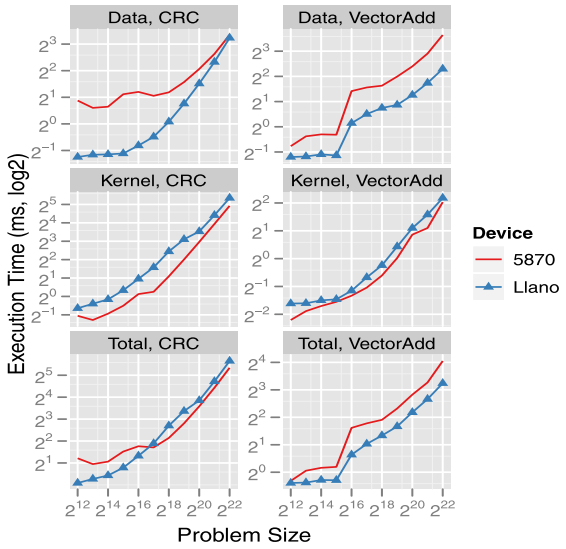

Fig. 7 The results of Fused (Llano) and Discrete (HD 5870) performance for the VectorAdd and Reduce applications

Figure 7 shows the performance of the Llano architecture versus the HD5870 discrete GPU for the CRC and VectorAdd applications. We used the Mixed approach on Llano and the Default approach on HD5870. For applications requiring more computation (Scan and CRC), the discrete GPU was able to make up for its slower data transfer speeds in the kernel execution, but because of the relatively small amount of computation that needed to be done per unit of transfer, it was often not enough to gain a clear advantage over Llano. In the cases where the application kernel was simpler (VectorAdd and Reduce), there was even less of a chance for the discrete GPU to catch up, which leads to improved performance on the fused architecture. In fact, Llano outperforms HD5870 by up to 57 % for the VectorAdd benchmark.

5 Discussion

To gain insight into the results of this experiment, we present a model describing the performance of applications for different memory speeds. The model is simple, but it can be used to understand the general trends in the results. This model is not intended to predict real performance of systems, but the model facilitates discussion of the results by illustrating the major components of application performance for data-intensive kernels. We compute the overall time taken to complete the application, T, based on the bandwidth to and from the device, B_w and B_r and the speed of reads and writes in the kernel itself, K_r and K_w. We also consider the size of the input data, kernel reads and writes, and output data (S_i, S_r, S_w, and S_o, respectively) to make an appropriate estimate.

$$T = \frac{S_i}{B_w} + \frac{S_r}{K_r} + \frac{S_w}{K_w} + \frac{S_o}{B_r} \tag{1}$$

Considering the difference between GPU-Resident and CPU-Resident memory on the Llano machine for the Reduce application. The GPU-Resident memory will always beat the CPU-Resident memory because the read transfer bandwidth (B_r) is slower for the GPU-Resdient memory, but the amount of data sent back is also very low (S_o) and is a constant. As we increase the problem size, therefore, we will always see better performance for the GPU-Resident memory. Conversely, in the VectorAdd application, the final term is not a constant, and has the largest growth as we increase problem size for GPU-Resident memory. In this case, we see that CPU-Resident memory performs much better by avoiding as large a growth in time.

The same kind of analysis can be performed on the CRC application to understand the cross-over point between the two devices. Substituting size values into Eq. (1), we have:

$$T = \frac{N}{B_w} + \frac{N}{K_r} + \frac{N}{K_w} + \frac{1}{B_r}$$

which can further be reduced to:

$$T = N\left(\frac{1}{B_w} + \frac{1}{K_r} + \frac{1}{K_w}\right) + \frac{1}{B_r}$$

This equation shows a linear growth of time based on the amount of data in the problem size, which has an intercept of $\frac{1}{B_r}$. The value of the intercept is lower for Llano, because of the increase bandwidth speed of reads from that memory. However, the faster speeds of kernel reads and writes enable the discrete GPU to outperform the fused architecture when the problem size is large enough. This model helps us to understand the architectural implications on application performance as well as enable us to develop guidelines for the usage of these memory techniques.

6 Related work

Because the AMD Fusion architecture is so new, there is a dearth of literature discussing its performance and power. Gutta et al. present an overview of the Fusion architecture [8], but do not discuss its performance in GPU applications. Daga et al. present an investigation similar to ours, but only compares the importance of increased PCIe bandwidth using the basic memory movement technique [6].

There are traditionally three methods for characterizing GPU system performance: reverse engineering, benchmark suites, and performance modeling. Relative to reverse engineering, Wong et al. present a method of micro-benchmarking to understand the performance characteris-

tics as well as underlying architecture of a NVIDIA graphics card using CUDA [14]. Che et al. investigate the applications for the GPU and how being aware of the GPU's parallel architecture can generate speedups over traditional serial implementations [4]. Similarly, in [11, 12], Ryoo et al. present a method of optimization for discrete NVIDIA GPUs.

Benchmark suites can be used to better understand the characteristics of a system by comparing the results of a wide array of different application profiles. The SHOC [7] benchmark suite makes use of the micro-benchmarks to determine performance characteristics for discrete GPUs. Likewise, the Rodinia [5] benchmark suites present information about performance differences between GPUs based on the benchmark results.

With respect to performance modeling, Hong and Kim present a model that fully takes memory bandwidth and thread-level parallelism into account to improve the models performance estimate to 5 % for micro-benchmarks [9]. Baghsorkhi et al. present a performance model for data parallel kernels using the CUDA platform [2]. Finally, Aji et al. present a model that takes global memory partition camping into account to improve performance error rates, capturing more system complexity [1].

7 Conclusion

This paper makes three major contributions in support of heterogeneous supercomputing. First, we present a characterization of memory-access techniques on the latest fused CPU+GPU processor, namely the AMD Llano A8-3850 APU from their Fusion project. Second, with an understanding of the characteristics of AMD Llano APU in place, we can configure the less powerful Llano APU to outperform more powerful discrete GPUs on memory-bound kernels by as much as 55 %. By paying attention to the residency of data in an application, we can achieve performance gains against a naive implementation on the same system by as much as 20 % on a typical data transfer. Third, we present a analytical model that can be used to guide the use of these memory accesses and to explain the performance for different memory-bound applications being used. This work can be extended into a framework to automatically optimize memory transfers for applications.

References

1. Aji A, Daga M, Feng W (2011) Bounding the effect of partition camping in GPU kernels. In: 8th ACM int'l conference on computing frontiers. doi:http://doi.acm.org/10.1145/2016604.2016637
2. Baghsorkhi S, Delahaye M, Patel S, Gropp W, Hwu W (2010) An adaptive performance modeling tool for GPU architectures. ACM SIGPLAN Not 45:105–114. doi:http://doi.acm.org/10.1145/1837853.1693470
3. Boudier P, Sellers G (2011) Memory system on fusion APUs: The benefits of zero copy. In: AMD Fusion developer summit, AMD. http://developer.amd.com/afds/assets/presentations/1004_final.pdf
4. Che S, Boyer M, Meng J, Tarjan D, Sheaffer J, Skadron K (2008) A performance study of general-purpose applications on graphics processors using cuda. J Parallel Distrib Comput. doi:10.1016/j.jpdc.2008.05.014
5. Che S, Boyer M, Meng J, Tarjan D, Sheaffer JW, Lee S-H, Skadron K (2009) Rodinia: A benchmark suite for heterogeneous computing. In: IEEE int'l symp. on workload characterization. doi:10.1109/IISWC.2009.5306797
6. Daga M, Scogland T, Feng W (2011) Architecture-aware mapping and optimization on a 1600-core GPU. In: IEEE int'l conf. on parallel and distributed systems
7. Danalis A, Marin G, McCurdy C, Meredith J, Roth P, Spafford K, Tipparaju V, Vetter J (2010) The scalable heterogeneous computing (shoc) benchmark suite. In: 3rd workshop on general-purpose computation on graphics processing units. doi:10.1145/1735688.1735702
8. Gutta S, Foley D, Naini A, Wasmuth R, Cherepacha D (2011) In: Int'l solid-state circuits conference digest of technical papers. doi:10.1109/ISSCC.2011.5746314
9. Hong S, Kim H (2009) An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. Comput Archit News 37:152–163. doi:10.1145/1555815.1555775
10. Khronos Group (2008) The khronos group releases opencl 1.0 specification
11. Ryoo S, Rodrigues C, Stone S, Baghsorkhi S, Ueng S, Hwu W (2007) Program optimization study on a 128-core GPU. In: 1st workshop on general purpose processing on graphics processing units
12. Ryoo S, Rodrigues C, Baghsorkhi S, Stone S, Kirk D, Hwu W (2008) Optimization principles and application performance evaluation of a multithreaded GPU using cuda. In: 13th ACM SIGPLAN symp. on principles and practice of parallel programming. doi:http://doi.acm.org/10.1145/1345206.1345220
13. Top500 (2011) http://www.top500.org/
14. Wong H, Papadopoulou MM, Sadooghi-Alvandi M, Moshovos A (2010) Demystifying GPU microarchitecture through microbenchmarking. In: IEEE Int'l symp. on performance analysis of systems software. doi:10.1109/ISPASS.2010.5452013

Kenneth Lee received the BS degree in computer science from Virginia Tech in 2008. He is currently working toward the MS degree in computer science at Virginia Tech under the direction of Dr. Wu-chun Feng. His research interests include high performance computing and GPU architectures, with an emphasis on memory management and movement.

Heshan Lin received the BS degree in applied math from South China University of Technology in 1998, the M.S. degree in computer science from Temple University in 2004, and the Ph.D. degree in computer science from North Carolina State University in 2009. He is a Research Scientist in the Department of Computer Science at Virginia Tech. His research interests include data-intensive parallel and distributed computing, bioinformatics, cloud computing, and GPU (graphics processing unit) computing.

Wu-chun Feng became an Associate Professor in the Department of Computer Science and Department of Electrical & Computing Engineering at Virginia Tech (VT) in January 2006. He leads the Synergy Lab and serves as site co-director of the NSF Center for High-Performance Reconfigurable Computing at VT. He received B.S. degrees in Computer Engineering and Music (Honors) and M.S. degree in Computer Engineering at Penn State University in 1988 and 1990,

respectively. He then earned his Ph.D. in Computer Science at the University of Illinois at Urbana-Champaign in 1996. His research interests encompass a broad range of topics in efficient parallel computing, including high-performance computing and networking, energy-efficient (or green) supercomputing, accelerator-based computing, cloud computing, grid computing, bioinformatics, and computer science pedagogy for K-12.