CPU-GPU System Designs for High Performance Cloud Computing

Yiran Chen, Jie Guo, and Zhenyu Sun

Abstract Improvement of parallel computing capability will greatly increase the efficiency of high performance cloud computing. By combining the powerful scalar processing on CPU with the efficient parallel processing on GPU, CPU-GPU systems provide a hybrid computing environment that can be dynamically optimized for cloud computing applications. One of the critical issues in CPU-GPU system designs is the so called memory wall, which denotes the design complexity of memory coherence, bandwidth, capacity, and power budget. The optimization of the memory designs can not only improve the run-time performance but also enhance the reliability of the CPU-GPU system. In this chapter, we will introduce the mainstream and emerging memory hierarchy designs in CPU-GPU systems, discuss the techniques that can optimize the data allocation and migration between CPU and GPU for performance and power efficiency improvement, and present the challenges and opportunities of CPU-GPU systems.

1 Introduction

The core of cloud computing requires to have capability of *exascale computing* to process big data applications. General-purpose Programming on Graphics Processing Unit (GPU) architectures [18, 33] have been introduced to process vertices and fragments in parallel, which frequently occur in online data/transaction processing. By leveraging the intensive computing capability of GPGPU and the functional flexibility of Central Processing Unit (CPU), the hybrid CPU-GPGPU architectures [28, 37] have been proposed to satisfy the ever-increasing computing capacity of big data applications.

Y. Chen (✉) • J. Guo • Z. Sun
University of Pittsburgh, Pittsburgh, PA, USA
e-mail: yic52@pitt.edu; jig26@pitt.edu; zhs25@pitt.edu

K.J. Han et al. (eds.), *High Performance Cloud Auditing and Applications*,
DOI 10.1007/978-1-4614-3296-8_11,

Data management and communication are two critical problems in hybrid CPU-GPGPU computing system design and programming. The technology fusion between CPU and GPGPU dramatically raises the complexity of data management because of the different requirements on memory metrics in CPU and GPGPU as well as the throughput discrepancy. Many memory designs have been proposed to mitigate these issues and optimize the data allocation and migration between CPUs and GPUs. The reliability and power consumption of the memory hierarchy in CPU-GPGPU systems are also major considerations in these designs. All the above problems will be discussed in this chapter.

The rest of the chapter is organized as follows. Section 2 presents CPU and GPU hierarchies, and limitations of CPU and GPU. Section 3 provides an overview of discrete CPU-GPU architecture, CPU-GPU communication, optimization and scalability. Section 4 provides an overview of integrated CPU-GPU architecture, memory hierarchy for CPU-GPU systems and design issues. Finally, Section 5 presents our conclusions.

2 Motivation for CPU-GPU Systems

2.1 CPU Memory Hierarchy

In computer systems, CPU is in charge of processing various applications. The executions of instructions in pipelined CPU designs require low data access latency to memory hierarchy. Figure 1 shows a typical memory hierarchy design [9], which consists of cache, main memory and secondary storage. As the component closest to the microprocessor, cache is usually implemented by static random-access memory

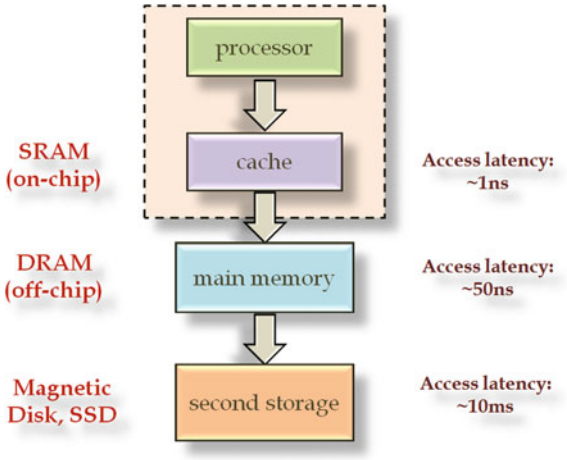

Fig. 1 The conventional memory hierarchy

(SRAM) to offer the fastest access time. However, the high power consumption and large memory cell size limit the cache capacity, which is usually from tens KB (L1 cache) to a few MB (L2 cache and beyond). The spatial locality of the cache access ensures a high cache hit rate (e.g., higher than 90%) [9] to supply the data for most of the accesses without referring to lower-level storage component. Main memory, which is implemented by dynamic random-access memory (DRAM), usually has a much larger capacity than cache but also a longer access latency. Finally, the second storage system (e.g., hard disk drive (HDD)) offers the largest capacity in the memory hierarchy, but also the longest access latency up to milliseconds. Very recently, NAND flash technology is applied as the second storage system (e.g., solid state drive (SSD) in embedded applications.

2.2 GPU Memory Hierarchy

A graphics processing unit (GPU) consists of an array of highly scalable multi-core processors that were specially designed for accelerating the display output generation. Recently, GPU has been also applied in general-purpose parallel computing applications due to its powerful data processing capability. A typical present-day GPU is composed of multiple streaming multiprocessors (SM) which are hardware multi-threaded and each SM contains eight or more streaming processors (SP). Such a highly parallel architecture requires also high data transfer bandwidth.

Compute unified device architecture(CUDA) is a widely adopted parallel programming model mainly designed for NVDIA GPUs. To provide high data throughout required by multi-processors under general computing and visual display applications, CUDA uses a memory subsystem distinct from CPU. The subsystem consists of *cache*, *share memory*, *constant memory*, *texture memory* and *global memory*. Similar to the scenario in CPU designs, cache is on the top of this memory hierarchy and closest to the processors. Instead of building a large memory capacity for hit rate improvement in CPU designs, GPU cache designs focus on promoting the memory bandwidth for highly threaded applications. In GPU memory hierarchy, below the cache, shared memory supports both read and write accesses to each processor in the same SM. Due to requirements on data throughput, both cache and shared memory are implemented by on-chip multi-bank SRAM. Constant memory, texture memory and global memory reside at the lower levels of memory hierarchy.

Both constant memory and texture memory store read-only data and support access from all GPU threads. In particular, constant memory has relatively short access latency for simultaneous requests to the same word while texture memory is designed to support the requests with strong spatial locality. Global memory is for the data transfer between the GPU and the host applications. It is also accessible to all threads of the GPU. In general, constant memory, texture memory, and global memory all reside in the external memory. Therefore, the external memory has to provide enough bandwidth for the multi-processors in the GPU and the data

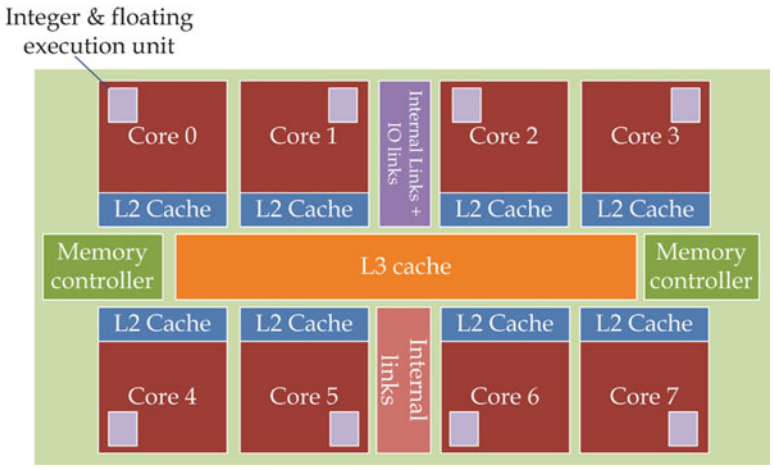

Fig. 2 IBM® POWER7™ processor chip view (Adapted from Ware et al. [29])

migration between the host memory and the global memory to prevent performance degradation. Usually, graphics double data rate (GDDR) memory is employed as the external memory.

2.3 Limitations of CPU and GPU

The high performance of modern CPUs is due to the combination of many technology advancements: high transistor density and performance, pipeline technology, superscalar execution, speculative execution, caching and so on. The evolution of the compiler also greatly reduces the programming difficulty of the modern CPU as the hardware details of micro-architecture is almost invisible to the software developers. However, CPUs are mainly designed for a wide variety of applications with a balanced response time to different tasks. Due to the complex control logic, the integer and floating-point execution units only occupy a very small fraction of the die area in a modern CPU, as shown in Fig. 2. It is no surprise that CPU is relatively inefficient for data-intensive high performance computing applications with high parallelism.

The multi-core structure and high memory bandwidth make GPUs ideal computing platforms for graphic applications with a large degree of data parallelism. Compared to CPUs, the smaller cache capacity and lower cache hit rate (i.e., lower than 90%) and longer memory access latency limit the performance of GPUs for single-threaded applications. Nonetheless, the high tolerance of graphic applications to memory access latency allow GPUs to trade single-thread performance for parallel processing capability. The GPU performance is ultimately limited by the percentage of the scalar section of the program [15].

In modern CPU-GPU systems, CPUs mainly handle the dynamic workloads with short sequences of computational operations and unpredictable control flow while GPUs are mainly in charge of gaming, multimedia encoding and other popular PC applications. Combining the computing flexibility of CPUs and the parallel computing strength of GPUs, the CPU-GPU system has become one of the popular architectures in both PC and enterprise markets. In the next section, we will introduce two kinds of mainstream heterogeneous systems.

3 Discrete CPU-GPU Systems

3.1 Overview of Discrete CPU-GPU Architecture

The architecture of a typical discrete CPU-GPU system is shown in Fig. 3. GPU is connected to a chipset, which is attached to CPU via a $\times 16$ peripheral component interconnect express (PCIe) link. Both CPU and GPU have their own memory subsystems which are disjoint from each other. The data computing in the GPU invokes data movement between the main memory of the host and the memory of the GPU on the PCIe bus. In the existing program models such as CUDA, the host-to-device data transfer is manually and explicitly managed. Since all the data transfer between host and GPU is through the PCIe bus, additional timing and control overhead costs are incurred. As a consequence, the PCI data transfer is often the bottleneck of the GPU performance [2].

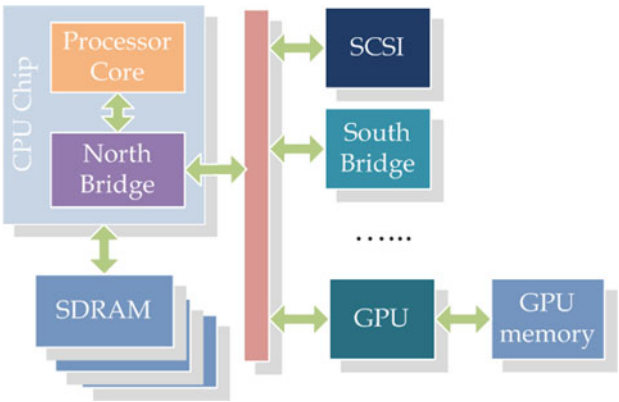

Fig. 3 CPU-GPU architecture

3.2 CPU-GPU Communication and Optimization

If the CPU or GPU needs data outside their memories, the data must be explicitly loaded from other memories into their own memories. The process of copying data between these memories is called *managing communication*. There are two generic types of data communication, *cyclic communication* and *acyclic communication*. Because cyclic communication is often several orders of magnitude slower than acyclic communication, transforming cyclic communication patterns to acyclic patterns is of importance to computing system designs, and is called *optimizing communication*. In CPU-GPU systems, copying data to GPU memory, generating a GPU function, and copying the results back to CPU memory yield typical cyclic communication patterns. Copying data to the GPU in the preheader, generating multiple GPU functions, and copying the result back to CPU memory at the loop exit lead to a typical acyclic communication pattern [10]. An effective communication optimization can minimize the number of access states and prevent the accessing of inconsistent data. However, the communication patterns between CPUs and GPUs could be very complicated because of the mismatch of data rate, access patterns, processing throughput, etc.

A fully automatic CPU-GPU communication management and optimization is proposed in [10]. The proposed scheme consists of a run-time library and a set of compiler transformations working together for the management and optimization of the CPU-GPU communications. This scheme does not rely on the strength of the static compile-time analysis or programmer-supplied annotations. The optimized automatic GPU parallelization yields on average a 5.36× speedup of the overall system over the best sequential CPU-only execution.

In [27], a CUDA-lite communication scheme translates low-performance, naive CUDA functions into high performance code by coalescing and exploiting GPU shared memory. For execution efficiency improvement, programmers need to provide annotations describing certain properties of the data structures and code regions designated for GPU execution. The CUDA-lite tool analyzes the code with these annotations and determines if the memory bandwidth can be conserved and the access latency can be reduced by utilizing any special memory types and/or by massaging the memory access patterns. In [32], JCUDA uses the Java type system to automatically transfer GPU function arguments between CPU and GPU memories. An annotation is required to indicate whether each parameter is live-in, live-out, or both. If Java implements multidimensional arrays as arrays of references, JCUDA uses the type information to flatten these arrays to Fortran-style multidimensional arrays. In [30], the Portland Group, Inc (PGI) Fortran and C compiler provides a mode for semi-automatic parallelization of GPUs. The PGI compiler cannot parallelize loops containing general pointer arithmetic, while CPU-GPU Communication Manager (CGCM) preserves the semantics of pointer arithmetic. Unlike CGCM, the PGI compiler does not automatically optimize the communication across GPU function invocations.

Acyclic communication can be achieved in the rare instances when dynamic dependence information is reusable. In [24], program annotations are used to prevent the undesired reuse of the dynamic dependence information. In [23], a dynamic check on the relevant program state is adopted to determine if the dependence information is reusable. However, the dynamic check is usually very costly and if the check fails, the system goes back to cyclic communication by default.

In [14], an automatic compiler is proposed to conduct the source-to-source translation from standard OpenMP applications to CUDA-based GPGPU applications. The system automatically transfers the named regions between CPU and GPU using two passes: The first pass copies all named annotated regions to the GPU for each GPU function, and the second cleanup pass removes all the copies that are not live-in. The two passes produce a communication pattern equivalent to an un-optimized CGCM communication.

Recently, message passing interface (MPI) [17] has been incorporated in the CUDA programming model to facilitate the data transfer via the PCIe bus, due to its outstanding inter-node communication capacity. However, the duplicate buffers introduced by the mixed MPI and CUDA programming model lead to the waste of memory space and code complexity. Furthermore, the MPI library is not able to support intra-node communication and the data movement between the host and memory is required under the current mixed MPI-CUDA model. However, such an approach results in longer latency and decreases bandwidth due to the significant amount of message transfers. In [11], the authors propose an address-aware MPI-GPU programming model to eliminate the redundant data copying between intermediate buffers by adding support to the GPU memory buffer in MPICH2 that is a open source of the MPI implementation. Asynchronous direct memory access (DMA) mode is also adopted to further improve the data copying throughput.

3.3 Power Efficiency Optimization

To satisfy the high data throughput requirements in video display applications and high performance computing, GDDR memories are specialized for current CPU-GPU systems. GDDR memories feature high working frequency and extended bus width. However, the increase in working frequency significantly raises power consumption and higher heat sinking cost. Although voltage and frequency scaling techniques can reduce the power consumption to a certain degree, they may also result in significant performance degradation. To improve the power efficiency of CPU-GPU systems, Zhao et al. [35] present a design to integrate GPU processor, GDDR-like graphic memories and their controllers in a single package. The data throughput is dramatically improved by reducing the number of memory package pins, without incurring any performance overheads. To further reduce power consumption, a reconfigurable memory controller is designed to dynamically adjust the bus width and frequency, based on the characteristics of different applications.

3.4 Scalability

To maximize parallel computing performance, multiple GPUs may be integrated in each core (or a node) in some CPU-GPU systems, such as Tianhe-1A, Tsubame, and KIDS. However, such designs impose new challenges for system architects in the design of efficient data transfer algorithms. In addition, increasing the number of GPUs introduces a performance bottleneck to the discrete heterogeneous system under the current topology where each node connects the GPU to the I/O hub via the PCI Express bus. Compared to the memory bandwidth of GPUs, PCIe bus is still quite narrow: it has a maximum of 16 PCI lanes, which can support only a peak bandwidth of 8 GB/s while the peak bandwidth of GPU memory (e.g., Nvidia Tesla M2070) can reach up to 148 GB/s [16]. This great throughput gap has emerged as the major performance bottleneck in modern CPU-GPU systems.

4 Integrated CPU-GPU Systems

4.1 Overview of CPU-GPU Systems

Following the advances of VLSI technologies, the integration of CPUs and GPUs on the same chip has become a hot research area. Such CPU-GPU systems provide the opportunity to leverage both the high computational power from GPUs for regular applications and flexible execution from CPUs for irregular workloads. In this section, we will introduce the latest integrated CPU-GPU architectures, including Intel's Sandy Bridge, AMD's Fusion, and NIVIDIA's Denver. We will also discuss the memory designs in such systems.

4.1.1 Intel Sandy Bridge

Figure 4 shows the architecture of the Sandy Bridge processor [34] that integrates four high performance Intel Architecture cores, one GPU, on-chip cache, memory controller and peripheral component interconnect (PCI) controller on the same die. A 8 MB L3 cache memory is shared by both CPUs and GPU. The data flow is optimized by a high performance on-die interconnect fabric (called "ring"). The ring connects the CPUs, the GPU, the L3 cache and the system agent (SA) unit, which houses a 1,600 MT/s dual-channel DDR3 memory controller, a 20-lane PCIe gen2 controller, 2 parallel pipe display engines, a power management control unit and testability logic. Sandy Bridge also has 6 power planes, 13 PLLs driving independent clock domains, temperature control with 2 types of thermal sensors, independent debug bus, as well as variable power supply for power and performance

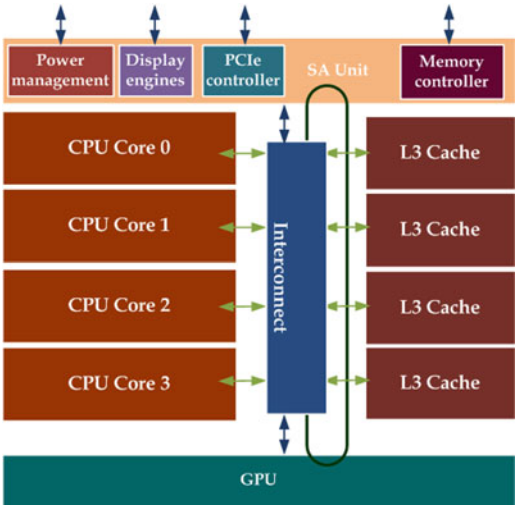

Fig. 4 Block diagram of Sandy Bridge (Adapted from Zhao et al. [35])

optimization. Because the L3 cache capacity of Sandy Bridge is large (3, 4 or 8 MB depending on the configuration), the power dissipated by the cache memory accounts for a big portion of the overall design.

4.1.2 AMD Fusion

An accelerated processing unit (APU) is a system with additional processing capability designated to accelerate one or more types of computations outside a CPU. Some examples are general-purpose computing on graphics processing unit (GPGPU), a field-programmable gate array (FPGA), or similar specialized processing system. In some marketing relevant usage of the term, APU also describes the processing device which integrates a CPU and a GPU on the same die, thus improving the data transfer rates between these components while reducing the total power consumption.

AMD Fusion is the series of such APU designs [1, 7, 8], including the version for *desktop processors*, *mobile processors*, *ultra-mobile processors*, and *embedded processors*. As the most representative version, the *Llano* variant is a complex integration of processor, graphics, and multimedia resources. Here, we use Llano architecture as the example to discuss the CPU-GPU system design in detail.

As shown in Fig. 5, the Llano variant combines four $\times 86$ processor cores, a unified video decoder, an integrated DirectX11 graphics core, and an integrated two-head display controller. The entire integrated complex fits in a die area of 227 mm^2 at 32-nm silicon on insulator (SOI) process. Twenty-four lanes of PCIe Gen2 are also supported for high performance connectivity. Among them, 4 lanes are dedicated to the Llano-to-Fusion controller hub (FCH) link, 4 lanes to general-purpose ports, and the other 16 lanes to the discrete graphics that can expand

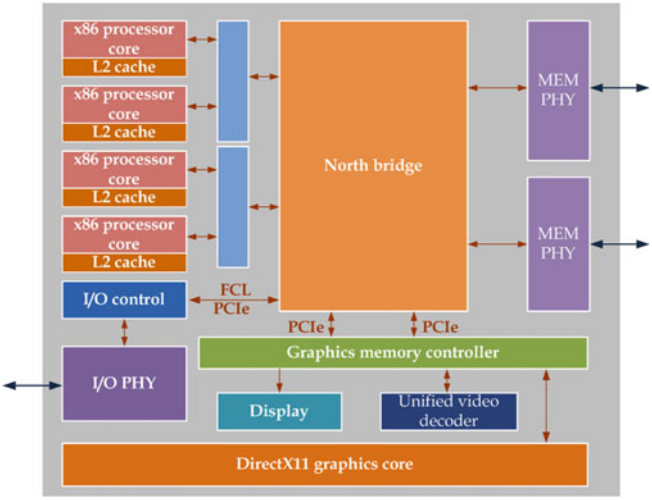

Fig. 5 Block diagram of a fusion APU of AMD (Adapted from Branover et al. [1])

computing and graphics stream capabilities. Two display streams can be directed to any combination of up to six display ports.

The integrated NorthBridge is the node where the processor, graphics, and multimedia devices are jointly managed. It also mainly determines the APU performance and power consumption. In NorthBridge, processor cores, I/O interfaces, graphics and video accelerators are connected to two 64-bit channels of DRAM through a multistage memory controller. Memory operates at data transfer rates up to DDR3-1866 megatransfers per second (MT/s).

Llano uses a unified memory architecture (UMA) in which the processor and graphics share a common memory. A portion of this memory is also dedicated to graphics frame buffer. Graphics, multimedia, and display memory traffic is routed through the graphics memory controller (GMC), which arbitrates between the requestors and issues a well-behaved (from a memory access perspective) stream of memory requests over the Radeon memory bus (RMB) to the NorthBridge. The accesses from GMC to frame buffer memory are non-coherent and do not snoop the processor caches. The RMB supports a 256-bit read data path and a 256-bit write data path for each of the two memory channels.

The coherent accesses from graphics or multimedia to the DRAM are directed over the fusion control link (FCL). The FCL consists of separated 128-bit read and write data paths to serve as the path for the processor accesses to the I/O devices and the dedicated graphics memory, and for the I/O accesses to the DRAM. Also, display traffic imposes a real-time requirement on the whole memory system. The display controller maintains a local frame buffer to store the next few lines to be sent to the display panel; buffer underrun must be avoided because it results in visible tearing effects on the panel. The DRAM controller prioritizes display requests indicating a

nearly empty buffer over other traffics. Rather than supplying a continuous request stream to refill the frame buffer, the display controller issues the requests to memory in a burst, followed by a period of no request activities. In an idle system, this lets the memory refresh itself between bursts, thus maximizing power efficiency.

The other graphics-related streams are managed in both the GMC and DRAM controllers to ensure that the average memory latency never exceeds a configurable threshold. The utilizations of processors and I/O traffic are arbitrated in the NorthBridge front-end stage by allocating proper bandwidth to each of them. The front-end output stream is arbitrated with the graphics-related stream in the DRAM controller. The DRAM controller contains a starvation-prevention mechanism that tracks the outstanding activity and ensures that the minimum required bandwidth is allocated to each stream. Starvation timers indicate when a specific traffic class needs to be stalled longer than the dynamically configurable threshold, and allows forward progress. Software drivers or on-die firmware can dynamically adapt the arbitration timers for specific workload scenarios.

4.1.3 NVIDIA Denver

Project Denver was developed by NVIDIA [3]. The innovative features of Denver include an NVIDIA CPU running the ARM instruction set, which will be fully integrated on the same chip as the NVIDIA GPU. By extending the performance range of the ARM instruction-set architecture, project Denver opens a new era for computing that enables the ARM architecture to cover a large portion of the computing space. Coupled with an NVIDIA GPU, it provides the hybrid CPU-GPU computing platform of the future with ultra-high performance and energy efficiency.

4.2 Memory Hierarchy for CPU-GPU Systems

The main bottleneck for most discrete GPU-based applications is the cost of data transfers to and from the GPU over PCIe. The recent introduction of the AMD Fusion provides a novel architecture to eliminate this bottleneck. By placing the CPU on the same die as a SIMD engine, the fused device is able to eliminate most of the costs of data transfer (e.g., PCIe transfers). The typical memory hierarchy for such a fused system is shown in Fig. 6.

Table 1 summarizes the information of three AMD Fusion systems including memory hierarchy configurations [2]. All systems have only one CPU core and the same memory capacity of 2 GB. CPU cache hierarchies of the three processors are all 2-level but with different capacities. The numbers of processing elements for GPU vary from 80 to 1,600. The GPU side has a similar memory hierarchy but a different number of processing units, yielding different throughput. Also, system throughput is determined by not only the memory but also other factors such as

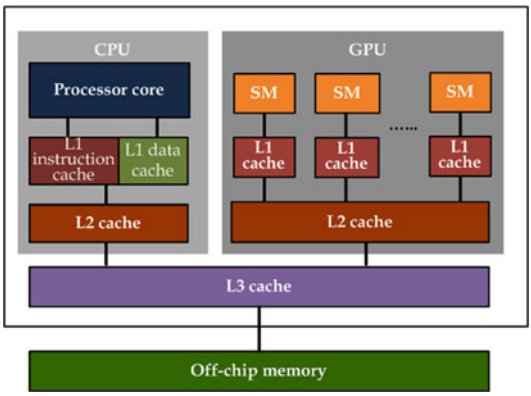

Fig. 6 Memory hierarchy for CPU-GPU systems

Table 1 Information of different fusion systems

Platform	AMD Zacate APU	AMD Radeon HD 5870	MD Radeon HD 5450
Host			
Processor	AMD Engg. sample	Intel Xeon E5405	Intel Celeron 430
Frequency	1.6 GHz	2.0 GHz	1.8 GHz
System memory	2 GB (NA)	2 GB DDR2	2 GB DDR2
Cache	L1: 32K, L2: 512K	L1: 32K, L2: 6M	L1: 32K, L2: 512K
Kernel	Ubuntu 2.6.35.22	Ubuntu 2.6.28.19	Ubuntu 2.6.32.24
Stream processors	80	1,600	80
Compute units	2	20	2
Memory bus type	NA	GDDR5	DDR3
Device memory	192 MB	1,024 MB	512 MB
Local memory	32 KB	32 KB	32 KB
Max. workgroup size	256 threads	256 threads	128 threads
Peak core clock freq.	492 MHz	850 MHz	675 MHz
Peak FLOPS	80 GFlop/s	2,720 GFlop/s	104 GFlop/s

operating frequency, on-chip interconnect, timing and memory controller. Different workloads also generate different system performances on different platforms [2].

4.3 Design Issues in CPU-GPU Systems

As aforementioned, CPUs are designed for a wide range of applications while GPUs are specialized for large scale parallel computing. Latency-sensitive CPUs tend to adopt the memory with large capacity, like the DDR3 in the Llano APU. In contrast, GPUs conventionally use GDDR memory with higher clock frequency and wider buses to provide high data throughout. Compared to DDR3 memory,

longer-latency GDDR memory reduces its capacity to minimize the power budget. In integrated CPU-GPU systems, these two diverse cores share the same type of memory, incurring performance degradation for either counterpart due to the mismatch of memory requirements [25]. Furthermore, sharing memory between CPU and GPU invokes access contention. It has been shown that the performance penalty incurred by contention is up to 20% [25].

4.4 Adopting Emerging Nonvolatile Memory in CPU-GPU Systems

Besides software-level data managements and optimizations, many hardware solutions can be also implemented for the performance and power improvements of CPU-GPU memory systems. As an example, emerging nonvolatile memory technologies such as spin-transfer torque random access memory (STT-RAM) and phase change memory (PCM) have been adopted for both cache and main memory designs in CPU memory hierarchy.

Due to its relatively fast access speed and zero-leakage power, STT-RAM has been adopted to replace SRAM as on-chip cache, which accounts for a large portion of energy consumption. However, there are two major obstacles in using STT-RAM for on-chip caches: long write latency and high write energy. When a STT-RAM cell operates in the sub-10 ns region, the resistance switching mechanism of *magnetic tunnel junction* (MTJ), which is the data storage device in STT-RAM, is dominated by *spin precession*. The required switching current rises exponentially as the MTJ switching time reduces. As a consequence, the size of MOS driving transistor increases accordingly, leading to a large memory cell area. The lifetime of the STT-RAM cell also degrades exponentially as the voltage across the oxide barrier of the MTJ increases. Therefore, a 10 ns programming time is widely accepted as the performance limit of STT-RAM designs in mainstream STT-RAM research and development [4, 5, 13, 26, 31]. Several proposals have been made to address the write speed and energy limitations of the STT-RAM. For example, the early write termination scheme [36] mitigates the performance degradation and energy overhead by terminating the unnecessarily long write pulse to STT-RAM cells. The dual write speed scheme [31] improves the average access time of a STT-RAM cache by having a fast and a slow cache partition. A SRAM/STT-RAM hybrid cache hierarchy with 3D stacking structure was proposed in [26] to compensate the performance degradation caused by STT-RAM writes by migrating write intensive data block into SRAM-based cache way.

Due to its small cell size and multi-level cell (MLC) capability, PCM is considered to be the replacement of DRAM in high-density main memory designs. The first issue in PCM is its long programming latency and high programming energy. A DRAM/PCM hybrid memory structure was proposed in [22] to merge frequent writes into the DRAM cache so that the number of write accesses to

the PCM devices can be greatly reduced. Write pausing [19] was proposed to allow performance critical read operations to preempt writes. A morphable memory system [20] was proposed to improve the read latency of MLC PCM by converting one MLC PCM page into two SLC pages when there is sufficient memory. Another issue in PCM is its limited programming endurance, since a PCM cell can only be reprogrammed about 10^6 times [6]. A start-gap scheme was used in [21] to efficiently enhance the PCM main memory lifetime by distributing the write accesses among all the PCM cells. In [12], line-level mapping and salvaging are combined to raise the limitation of wear-leveling technique for PCM-based main memory.

5 Conclusion

CPU-GPU systems are widely employed in high performance cloud computing to offer the computing capability in diverse scientific research fields. The major issues in the existing CPU-GPU system designs include data communication performance bottleneck, power efficiency and system scalability. In this chapter, we introduced the currently prevalent CPU-GPU system architectures. The memory hierarchies of these two kinds of computing units are extensively discussed. Based on the architectural characteristics, we conducted a deep analysis on the performance bottleneck of data transferring between CPU and GPU, the power efficiency of the GPU memory, and their architecture limitations on scalability. Finally, we discussed the adoption of emerging non-volatile memories in CPU-GPU systems for performance optimization and power reduction.

Acknowledgements This material is based upon work partially supported by the National Science Foundation (NSF) grant CNS-1116171 and the Air Force Research Laboratory (AFRL) Visiting Faculty Research Program (VFRP) extension grant LRIR 11RI01COR. We are grateful to Prof. Hai (Helen) Li from the University of Pittsburgh Department of Electrical and Computer Engineering for generous help.

References

1. Branover, A., Foley, D., Steinman, M.: AMD fusion APU: Llano. IEEE Micro **32**(2), 28–37 (2012). doi:10.1109/MM.2012.2
2. Daga, M., Aji, A.M., Feng, W.c.: On the efficacy of a fused CPU+GPU processor (or APU) for parallel computing. In: Proceedings of the 2011 Symposium on Application Accelerators in High-Performance Computing, SAAHPC'11, Knoxville, pp. 141–149 (2011). doi:10.1109/SAAHPC.2011.29
3. Dally, B.: nvidia.com, PROJECT DENVER: Processor to usher in new era of computing. http://goo.gl/HepP5 (2011)
4. Desikan, R., Lefurgy, C., Keckler, S., Burger, D.: ibm.com, On-chip MRAM as a high-bandwidth low-latency replacement for DRAM physical memories. http://goo.gl/lyvV2 (2008)

5. Dong, X., Wu, X., Sun, G., Xie, Y., Li, H., Chen, Y.: Circuit and microarchitecture evaluation of 3D stacking magnetic RAM (MRAM) as a universal memory replacement. In: Proceedings of the 45th Annual Design Automation Conference, DAC'08, Anaheim, pp. 554–559. ACM, New York (2008). doi:10.1145/1391469.1391610
6. Ferreira, A.P., Zhou, M., Bock, S., Childers, B., Melhem, R., Mossé, D.: Increasing PCM main memory lifetime. In: Proceedings of the Conference on Design, Automation and Test in Europe, DATE'10, Leuven, pp. 914–919. European Design and Automation Association, Leuven (2010)
7. Foley, D., Bansal, P., Cherepacha, D., Wasmuth, R., Gunasekar, A., Gutta, S., Naini, A.: A low-power integrated x86–64 and graphics processor for mobile computing devices. IEEE J. Solid-State Circuits **47**(1), 220–231 (2012)
8. Gutta, S.R., Foley, D., Naini, A., Wasmuth, R., Cherepacha, D.: A low-power integrated x86-64 and graphics processor for mobile computing devices. In: Proceedings of the 2011 IEEE International Solid-State Circuits Conference (ISSCC), San Francisco, pp. 270–272. IEEE (2011). doi:10.1109/ISSCC.2011.5746314
9. Hennessy, J., Patterson, D.: Computer Architecture: A Quantitative Approach, 4th edn. Morgan Kaufmann, Burlington (2007)
10. Jablin, T.B., Prabhu, P., Jablin, J.A., Johnson, N.P., Beard, S.R., August, D.I.: Automatic CPU-GPU communication management and optimization. SIGPLAN Notice **47**(6), 142–151 (2011). doi:10.1145/2345156.1993516
11. Ji, F., Aji, A.M., Dinan, J., Buntinas, D., Balaji, P., Feng, W.c., Ma, X.: Efficient intranode communication in GPU-accelerated systems. In: Proceedings of the 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IPDPSW'12, Shanghai, pp. 1838–1847 (2012). doi:10.1109/IPDPSW.2012.227
12. Jiang, L., Du, Y., Zhang, Y., Childers, B.R., 0002, J.Y.: LLS: Cooperative integration of wear-leveling and salvaging for PCM main memory. In: Proceedings of the 41st International Conference on Dependable Systems & Networks (DSN), Hong Kong, pp. 221–232. IEEE (2011). doi:10.1109/DSN.2011.5958221
13. Kawahara, T., Takemura, R., Miura, K., Hayakawa, J., Ikeda, S., Lee, Y., Sasaki, R., Goto, Y., Ito, K., Meguro, T.: 2Mb SPRAM (SPin-Transfer Torque RAM) with bit-by-bit bi-directional current write and parallelizing-direction current read. IEEE J. Solid-State Circuit **43**(1), 109–120 (2008)
14. Lee, S., Min, S.J., Eigenmann, R.: OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'09, Raleigh, pp. 101–110. ACM, New York (2009). doi:10.1145/1504176.1504194
15. Lee, V.W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A.D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., Dubey, P.: Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. SIGARCH Comput. Archit. News **38**(3), 451–460 (2010). doi:10.1145/1816038.1816021
16. Meredith, J., Roth, P., Spafford, K., Vetter, J.: Performance implications of nonuniform device topologies in scalable heterogeneous architectures. IEEE Micro **31**(5), 66–75 (2011). doi:10.1109/MM.2011.79
17. mpi-forum.org, MPI: A message-passing interface standard version 2.2. http://goo.gl/SEqm1 (2009)
18. Nere, A., Lipasti, M.: Cortical architectures on a GPGPU. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU'10, Pittsburgh, pp. 12–18. ACM, New York (2010). doi:10.1145/1735688.1735693
19. Qureshi, M.K., Franceschini, M., Lastras-Montaño, L.A.: Improving read performance of phase change memories via write cancellation and write pausing. In: Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA), Bangalore, pp. 1–11. IEEE Computer Society (2010). doi:10.1109/HPCA.2010.5416645

20. Qureshi, M.K., Franceschini, M.M., Lastras-Montaño, L.A., Karidis, J.P.: Morphable memory system: a robust architecture for exploiting multi-level phase change memories. SIGARCH Comput. Archit. News **38**(3), 153–162 (2010). doi:10.1145/1816038.1815981
21. Qureshi, M.K., Karidis, J., Franceschini, M., Srinivasan, V., Lastras, L., Abali, B.: Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 42, New York, pp. 14–23. ACM, New York (2009). doi:10.1145/1669112.1669117
22. Qureshi, M.K., Srinivasan, V., Rivers, J.A.: Scalable high performance main memory system using phase-change memory technology. In: Proceedings of the 36th Annual International Symposium on Computer Architecture, ISCA'09, Austin, pp. 24–33. ACM, New York (2009). doi:10.1145/1555754.1555760
23. Rauchwerger, L., Amato, N., Padua, D.: A scalable method for run-time loop parallelization. Int. J. Parallel Program. **23**(6), 537–576 (1995)
24. Saltz, J., Mirchandaney, R., Crowley, K.: Run-time parallelization and scheduling of loops. IEEE Trans. Comput. **40**(5), 603–612 (1991)
25. Spafford, K.L., Meredith, J.S., Lee, S., Li, D., Roth, P.C., Vetter, J.S.: The tradeoffs of fused memory hierarchies in heterogeneous computing architectures. In: Proceedings of the 9th Conference on Computing Frontiers, CF'12, Caligari, pp. 103–112. ACM, New York (2012). doi:10.1145/2212908.2212924
26. Sun, G., Dong, X., Xie, Y., Li, J., Chen, Y.: A novel architecture of the 3D stacked MRAM L2 cache for CMPs. In: IEEE Symposium on High-Performance Computer Architecture (HPCA), Los Alamitos, pp. 239–249. IEEE Computer Society, Los Alamitos (2009). doi:10.1109/HPCA.2009.4798259
27. Ueng, S.Z., Lathara, M., Baghsorkhi, S.S., Hwu, W.M.W.: CUDA-Lite: reducing GPU programming complexity. Languages and Compilers for Parallel Computing, pp. 1–15. Springer, Berlin/Heidelberg (2008)
28. Venkatasubramanian, S., Vuduc, R.W.: Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. In: Proceedings of the 23rd International Conference on Supercomputing, ICS'09, Yorktown Heights, pp. 244–255. ACM, New York (2009). doi:10.1145/1542275.1542312
29. Ware, M., Rajamani, K., Floyd, M., Brock, B., Rubio, J., Rawson, F., Carter, J.: Architecting for power management: The IBM® POWER7™ approach. In: Proceedings of the 16th IEEE International Symposium on High Performance Computer Architecture, HPCA'10, Bangalore, pp. 1–11 (2010). doi:10.1109/HPCA.2010.5416627
30. Wolfe, M.: Implementing the PGI accelerator model. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU'10, Pittsburgh, pp. 43–50. ACM, New York (2010). doi:10.1145/1735688.1735697
31. Xu, W., Sun, H., Wang, X., Chen, Y., Zhang, T.: Design of last-level on-chip cache using spin-torque transfer RAM (STT RAM). IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **19**(3), 483–493 (2011). doi:10.1109/TVLSI.2009.2035509
32. Yan, Y., Grossman, M., Sarkar, V.: JCUDA: A programmer-friendly interface for accelerating Java programs with CUDA. In: Proceedings of the 15th International Euro-Par Conference on Parallel Processing, Euro-Par'09, Delft, pp. 887–899. Springer-Verlag, Berlin/Heidelberg (2009). doi:10.1007/978-3-642-03869-3_82
33. Yang, Y., Xiang, P., Kong, J., Mantor, M., Zhou, H.: A unified optimizing compiler framework for different GPGPU architectures. ACM Trans. Archit. Code Optim. **9**(2), 9:1–9:33 (2012). doi:10.1145/2207222.2207225
34. Yuffe, M., Knoll, E., Mehalel, M., Shor, J., Kurts, T.: A fully integrated multi-CPU, GPU and memory controller 32nm processor. In: Proceedings of the 2011 IEEE International Solid-State Circuits Conference (ISSCC), San Francisco, pp. 264–266 (2011). doi:10.1109/ISSCC.2011.5746311

35. Zhao, J., Sun, G., Loh, G., Xie, Y.: Energy-efficient GPU design with configurable package graphic memory. In: ISLPED'12, Redondo Beach, pp. 403–408 (2012)
36. Zhou, P., Zhao, B., Yang, J., Zhang, Y.: Energy reduction for STT-RAM using early write termination. In: Proceedings of the 2009 International Conference on Computer-Aided Design, ICCAD'09, New York, pp. 264–268. ACM, New York (2009). doi:10.1145/1687399.1687448
37. Zidan, M.A., Bonny, T., Salama, K.N.: High performance technique for database applications using a hybrid GPU/CPU platform. In: Proceedings of the 21st Edition of the Great Lakes Symposium on VLSI, GLSVLSI'11, Lausanne, pp. 85–90. ACM, New York (2011). doi:10.1145/1973009.1973027