Sustainable Computing: Informatics and Systems 19 (2018) 72–85

Contents lists available at ScienceDirect

Sustainable Computing: Informatics and Systems

journal homepage: www.elsevier.com/locate/suscom

A survey on techniques for cooperative CPU-GPU computing

Raju K*, Niranjan N. Chiplunkar

Dept. of CS& E, NMAMIT, Nitte, Karkala, Karnataka, India

ARTICLE INFO

Article history:
Received 30 August 2017
Received in revised form 18 April 2018
Accepted 21 July 2018
Available online 3 August 2018

Keywords:
Heterogeneous system
CPU-GPU
Multicore CPU
Cooperative execution
Task allocation
Energy saving

ABSTRACT

Graphical Processing Unit provides massive parallelism due to the presence of hundreds of cores. Usage of GPUs for general purpose computation (GPGPU) has resulted in execution speedup and energy efficiency. In addition, the modern CPUs also possess multiple cores, with enormous computational power. In general, the current programming frameworks for CPU-GPU heterogeneous computing system do not facilitate the efficient utilization of computational resources so as to further improve the performance and reduce the energy consumption. Several researches have been carried out to improve the resource utilization and reduce the energy consumption of heterogeneous systems by cooperatively performing the computation on both multicore CPUs and GPUs. In this survey paper we review the various techniques available for CPU-GPU cooperative computing.

1. Introduction

As the clock frequency and powerconsumption of unicore processors reached its peak, processor architects switched to the concept of multicore processors. A multicore processor is one in which two or more execution cores are placed in a single chip. The operating system perceives each of these execution cores as a discrete physical processor with all the associated execution resources. The main objectives of multi-core architecture are to reduce power consumption, and enable efficient simultaneous processing of multiple tasks.

OpenMP [82] is an application programming interface that provides the developers with a simple and flexible interface making the development of parallel applications on multicore processors easier. OpenMP is based on the concept of multi-threading wherein a master thread forks several slave threads and the work is distributed amongst them. The threads run concurrently and are scheduled to run on different processor cores.

The Graphics Processing Unit is a co-processor, originally developed to accelerate graphical applications. Modern GPUs consists of hundreds of cores and capable of processing thousands of threads. The cores of a GPU execute the same instruction sequence in lockstep but possibly on different data elements. GPU computing has become one of the interesting areas of research due to the suitability of the architecture for executing applications that exhibit massive parallelism.

Though the primary reason of introducing GPU was for graphical purposes, it is now being used for general purpose parallel computing. During 2006–2007, a massively parallel architecture was introduced by NVIDA called as “CUDA” which revolutionized the concept of GPGPU programming. CUDA C/C++ [80] is an extension of the C/C++ programming languages for general purpose computation. CUDA provides the programmers with massive parallel computational power using the NVIDIA graphics card. OpenCL [81] is another programming model for GPU programming which is supported by almost all GPU vendors. The usage of CUDA or OpenCL for GPU computing helped in improving the performance of many data parallel and compute-intensive applications.

Owing to the wide acceptance of CUDA in GPGPU computations, we have used CUDA terminologies in this paper to describe the GPU thread organization and program execution flow. In CUDA terminology, the CPU and its memory are called as the *host* whereas the GPU and its memory are called as the *device*. Any code running on the device is called as the *kernel*. CUDA architecture provides *grids*, *blocks*, and *threads* which enable the utilization of the entire computational capability provided by the graphics cards. In OpenCL, the *grids*, *blocks* and *threads* are referred to as *NDRange*, *workgroup*, and *work item* respectively with the same semantics as that of CUDA.

The threads are grouped into thread blocks and thread blocks are grouped into a grid. Each thread within a grid executes an instance of the kernel. Each thread within a thread block is a separate entity and is associated with a thread id, program counter, registers, per thread private memory, input data and output results.

* Corresponding author.
E-mail addresses: rajuk@nitte.edu.in (R. K), nchiplunkar@nitte.edu.in (N.N. Chiplunkar).

https://doi.org/10.1016/j.suscom.2018.07.010
2210-5379/© 2018 Elsevier Inc. All rights reserved.

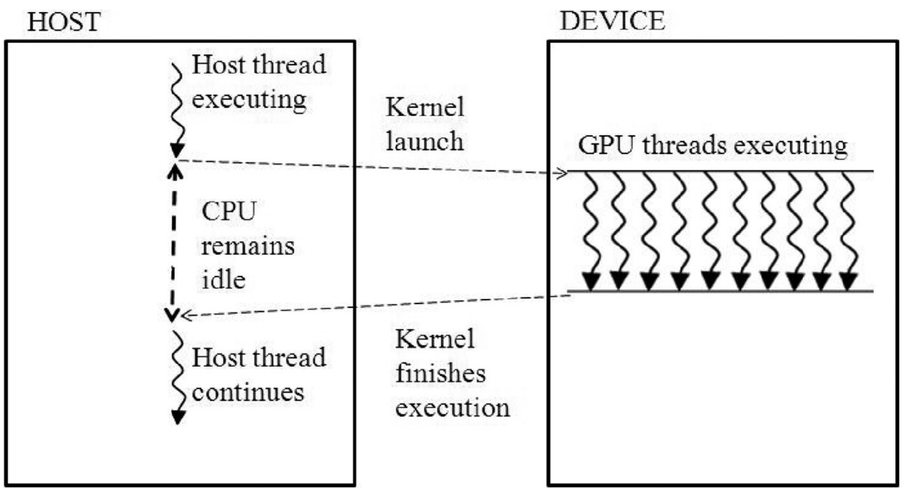

Fig. 1. Execution flow of CUDA program.

The threads within a block share the data among themselves using shared memory and barrier synchronization mechanism. A thread block within a grid has a unique block id. All threads within a grid share the global memory for reading the input and writing the output. Number of threads throughout the execution of the kernel will be same. Threads within the block share the per block shared memory.

The flow of execution of a typical CUDA program is illustrated in Fig. 1. A CUDA program consists of host code and the device code. The host and the device memory address spaces are different. Prior to kernel invocation the input data to be processed by the kernel resides in the host memory. As the device can not directly access the host memory, the input data has to be moved from the host memory to the device memory through PCI-Express. Initially, the CPU sends the input data to GPU and launches the kernel. Now device begins the execution of the kernel, during which all the threads in the grid execute the kernel code on different input data. Here kernel launch is an asynchronous (non-blocking) operation, meaning that immediately after the kernel launch the control is returned to the CPU without waiting for the GPU to finish the execution of that kernel. After reacquiring the execution control, the CPU thread can only perform data transfer between host and the device, and launch any other kernels using CUDA streams. Huge amount of computational power of the multi-core CPU is wasted as these processor cores are neither involved in the execution of the kernel nor in performing any other independent tasks.

The CPU-GPU heterogeneous hardware architecture and the execution model described above leads to several shortfalls such as underutilization of computational resources, communication overhead between the processing units, lack of portability between the hardware architectures, load balancing among different processing units, etc. Improved performance and energy efficiency can be observed by properly handling these issues. The above lacunae in the CPU-GPU heterogeneous computing can be overcome by utilizing the computing resources in a cooperative manner.

Several researches have been carried out to develop mechanisms for cooperative execution of tasks on CPU and GPU. In this survey paper we focus on those research works that use both CPU and GPU collaboratively to solve the above problems. A survey paper by Mittal et al. [76] discusses heterogeneous computing techniques under the following four categories:

1 Algorithm and Program-Execution level workload partitioning techniques
2 Programming languages, frameworks and related development tools
3 Techniques for improving energy efficiency
4 Fused CPU-GPU chips and comparison with discrete systems

The difference between the paper [76] and our paper lies in the perspective and granularity of categorizing the research works. The categories are chosen by considering the subtle specificities of techniques used in improving the performance. In addition to this, we have reviewed very recent research works in the area of cooperative CPU-GPU heterogeneous computing. We classify the research works as follows:

A *Frameworks for cooperative (collaborative) execution of a single kernel on CPU and GPU.*
B *Application specific CPU-GPU hybrid implementation techniques.*
C *Frameworks for concurrent execution of multiple kernels using CPU and GPU.*
D *Minimizing CPU-GPU data transmission overhead through collaborative CPU-GPU execution.*
E *Compilers and runtimes for portability among GPU and multicore processors.*
F *Task scheduling schemes in CPU-GPU cooperative execution.*
G *Cooperative computing using coupled CPU-GPU architectures.*

The performance of GPGPU computing can be further improved if we can use the idle CPU cores [1] by cooperatively executing the kernel on both CPU and GPU. That is, while the GPU is running the kernel some amount of GPU workload can be assigned to the CPU cores. Thus the cooperative CPU-GPU computing can exploit parallelism across both GPU and the CPU cores. In Section 2.1, we present the mechanisms available for the cooperative execution of a single kernel on both CPU and GPU.

Several applications have been successfully implemented using cooperative CPU-GPU computing. The parallelization strategy adopted in each of these implementations is specific to the individual application. However, all these research works intend to utilize both CPU and GPU computational power with the objective of improving the performance. The two major parallelization strategies used are pipelined execution and concurrent execution approaches. In pipelined execution approaches some phases of an algorithm are executed on the CPU and the remaining phases are executed on the GPU. In concurrent execution approach both CPU and GPU simultaneously perform the computations involved in the algorithm. Some of the research works towards application specific CPU-GPU hybrid implementation techniques are summarized in Section 2.2.

The GPGPUs are used for executing compute intensive applications that exhibit massive parallelism. When used for executing small scale computations, the computational resources of the GPU are not fully utilized. Moreover, when a GPU is shared by many concurrent kernels like in the cloud environments, throughput is a most desirable metric of system performance [40]. High throughput is a resultant of high degree of resource utilization. A memory intensive kernel underutilizes the compute resources of the device. Similarly a compute intensive kernel underutilizes the memory resources of the device. For a GPU application developer it is difficult to determine the ratio of compute instructions to memory instructions such that the GPU is fully utilized. Hence to improve the GPU performance it is necessary to run multiple kernels concurrently in such a way that these kernels utilize the memory and compute resources most efficiently. Even in the case of small scale computations the underutilization of the GPU can be overcome by concurrently executing multiple independent kernels from one or many applications. Some prominent approaches towards increasing the CPU and GPU utilization through concurrent execution of multiple kernels are discussed in Section 2.3.

As described earlier, in CUDA program execution flow the input data has to be transferred from host memory to the device memory through PCI-Express. This data transfer is a time consuming task as the bandwidth of PCI-Express acts as a bottleneck. The data transfer overhead can be minimized either by overlapping the data transfer and the kernel execution or by processing a portion of input data in the host itself. Such techniques are discussed in the Section 2.4.

The suitability of a processor for an application depends on several factors such as the size of input data, the type of application such as memory intensive or control intensive applications etc. Hence there exists the need for mechanisms that allow for a GPU kernel to be executed efficiently on a most suitable processor if one is available or fall back to a slower processor. This feature can be enabled by generating different binary versions of a kernel corresponding to different processor architectures available in the heterogeneous computing system. Compiler frameworks are necessary to generate optimized binary versions of the same kernel for different target processor architectures. These frameworks make the kernel execution portable among CPU and accelerators. Thus a set of independent kernels can be scheduled to concurrently execute on any of the available processors. In Section 2.5 we analyze the research works carried out in this direction.

In CPU-GPU heterogeneous computing systems several algorithms have been developed that analyze a parallel program and partition it into subprograms of CPU and GPU kind. The partitioning is based on some metrics such as the characteristics of the sub-programs, the hardware features of the processor, the current workload of the processor, energy consumption of processor, etc. The sub-programs are then optimally allocated to the most suitable processors (CPU or GPU) and executed concurrently. The task allocation is performed with the objective of improving the performance, throughput, and energy saving. Task allocation techniques combined with the exclusive techniques for energy saving can further scale up the energy efficiency. Task allocation schemes that improve performance and energy efficiency by increasing the possibility of cooperative CPU-GPU computing are discussed in Section 2.6.

The research works reviewed in the Sections 2.1 through 2.6 are based on discrete CPU-GPU heterogeneous systems. In such systems, the GPU is considered as an independent computational device from the CPU with its own physical memory. As discussed earlier, the data transfer between CPU and GPU memory happens through PCI-Express bus, which acts as performance bottleneck. In a coupled CPU-GPU architecture, the CPU and GPU are integrated into a single chip. The CPU and GPU share a single physical memory. Hence the data transfer between CPU and GPU can happen at the speed of main memory. Thus the data transfer overhead of the PCI-e bus is eliminated in the coupled CPU-GPU architecture. Some examples for coupled CPU-GPU architecture include AMD APU (Accelerated Processing Unit), Intel's Sandy Bridge, and Ivy Bridge processors. The Section 2.7 presents the research works that use coupled CPU-GPU architectures for cooperative computing.

In Section 3, we present the future directions and we conclude this paper in Section 4.

2. Literature survey

2.1. Frameworks for cooperative (collaborative) execution of a single kernel on CPU and GPU

A CUDA or OpenCL program is the combination of both host and device code. The CPUs and GPUs are based on different instruction set architectures. Hence in CUDA, the *nvcc* (Nvidia CUDA compiler) separates these codes during the compilation process. The host code is converted into x86 object code using the host compiler. The device code is translated into PTX (Parallel Thread eXecution), a virtual ISA form or *cubin* object form. The PTX form is then compiled into device specific binary object.

To enable the cooperative execution of a single GPU kernel on both CPU and GPU, a mechanism is needed to partition and translate a GPU kernel from PTX to x86 form so that it can be run on multicore CPU without any intervention from the programmer.

CHC [1] is a framework that enables the utilization of idle CPU cores along with the GPU cores for the execution of a CUDA kernel. The framework assigns the workload statically between the CPU and GPU cores. The number of thread blocks to be removed from the kernel for execution on the CPU is statically determined based on a heuristic approach. This partitioning information is used to generate new execution configurations for CPU and GPU. The framework takes PTX form of a CUDA kernel as the input and using Ocelot [29] translates the PTX code to LLVM IR (Low Level Virtual Machine Intermediate Representation) at runtime. Also Ocelot optimizes the PTX and LLVM code for execution on x86 architecture. Finally the LLVM-JIT is used to execute the LLVM IR on the CPU. The GPU and CPU sub kernels use separate memory spaces for storing the input data and computed results. To access a variable in two different memory spaces, two separate pointers are maintained. Whenever the pointer variable is referenced, a memory abstraction layer maps the pointer to the respective memory addresses for both CPU and GPU. Executing a kernel using this framework provides thrice the amount of speedup as compared to GPU only execution of that kernel. The advantage of this framework is that it enables the execution of a CUDA binary concurrently both on CPU and GPU without the need for source recompilation. However this framework lacks a method to dynamically distribute the workload among CPU and GPU. Moreover the framework is limited for the concurrent execution of only CUDA binaries on CPU and GPU.

FluidiCL [2] is a framework that takes a kernel written for a single OpenCL device and executes it concurrently using both CPU and GPU. It is a mechanism that does not require any prior training. For every OpenCL API, a corresponding FluidiCL API has been designed. In OpenCL, the programmer can choose a target device for the kernel to run on. In other words, the OpenCL compiler generates target code for the compute device chosen by the user. The kernel is given a flattened workgroup ID in which work groups in a two dimensional grid are given one dimensional numbers. The CPU executes the workgroups in the decreasing order of the flattened workgroup IDs and the CPU executes in the increasing order. The CPU and GPU continue the execution of the work groups until there is an unprocessed work group. In this manner the workload is dynamically partitioned among CPU and GPU, and the number of workgroups executed by either CPU or GPU is proportional to execution speed of the respective hardware. This technique provides a speedup of 64% over the GPU-only execution. The significant feature of this approach is the dynamic distribution of the workload among CPU and GPU which results in efficient utilization of computational resources. However the FluidiCL can be used only with OpenCL programs and, the CPU and GPU kernels have to be modified manually by adding the appropriate checks.

SKMD [3] (Single Kernel Multiple Device) is a technique wherein a single OpenCL kernel is split into sub kernels to be executed on CPU and GPU cores. The framework uses a profiler to collect performance metrics for each compute device by varying the workload. A dynamic compiler is used to transform the data parallel kernel so that it can be executed on different devices. The dynamic compiler consists of kernel transformer, buffer manager and partitioner. The output of the kernel transformer is partition-ready kernels which work only on smaller data sets. The buffer manager determines the memory access pattern of each work-group by statically analyzing the kernels. If the memory access range of each work-group can be analyzed then the buffer manager transfers only necessary data to and from each device. Otherwise the entire input data is transferred to each device and output must be merged on the CPU. The partitioner divides the input data sets into smaller sets considering the performance metrics provided by the profiler and host to device data transfer costs. The buffer manager transfers these data sets from the CPU to other compute devices and the results are merged together after the completion of the assigned work. This technique results in an average speed up of 29% on a system with one multicore CPU and two asymmetric GPUs compared the fastest GPU-only execution. However, the SKMD system requires a profiling to determine the number of work-groups to be executed on a device. The system supports both offline and online profiling. If offline profiling is used before the real execution the profile-run has to be executed. The online profiling, if used, adds overhead to the initial execution time. Like FluidiCL, even SKMD can be used only with OpenCL environment.

To enable a single kernel execution on multicore CPU and GPU, Piao et al. [4] introduce a JavaScript framework for adaptive work sharing (JAWS). The Web Workers [78] are the thread-like programming construct supported by JavaScript engines which does not support shared memory semantics. Hence the high communication cost between the parallel contexts, and the overhead of distributing input data to and merging the kernel output from different contexts degrades the performance of kernel execution. WebCL [79] is a parallel programming framework for data-parallel applications, which allows the execution of a kernel either on CPU or GPU but not both. JAWS addresses these challenges by providing shared arrays which is accessible by all parallel contexts (workers). The runtime system partitions the input data into chunks by flattening the input data as in [2] and distributes them to CPU and GPU. The CPU runs a JavaScript kernel on multiple Web Workers and the GPU runs a WebCL kernel. To balance the workload between CPU and GPU the task scheduler dynamically adapts the chunk size. The scheduler begins with a predetermined chunk size and multiplicatively increases the chunk size by a constant factor until the throughput of any one device is stabilized. The remaining workload is distributed to each device in proportional to each device's throughput. For both CPU-friendly and GPU-friendly benchmarks JAWS outperforms the best single-device execution.

2.2. Application specific CPU-GPU hybrid implementation techniques

An approach to develop libraries for dense linear algebra (DLA) on CPU-GPU hybrid systems is provided by Tomov et al. [5]. It is observed that some algorithms map well on CPU and others on GPU. The DLA algorithms are represented as a collection of BLAS based tasks along with the dependencies among them. A hybrid algorithm splits and schedules the computations to the most suitable processors in a hybrid system. The hybrid coding approach described in this paper plans to execute small, non-parallelizable kernels on the CPU and the large, data-parallel kernels on the GPU. The hybrid algorithms yield orders of magnitude performance improvement compared to those algorithms that use only multicores. Also the hybrid approach is scalable with the increase in the number of GPUs. The small kernels on the CPU and the large kernels on the GPU are executed concurrently. Implementation uses CUDA and BLAS library.

Agulleiro et al. [6] have used CPU-GPU hybrid approach to the problem of 3D reconstruction in Electron Tomography (ET). The system keeps a pool of slabs of four slices to be reconstructed. The 3D reconstruction problem is decomposed into a set of 2D reconstruction sub problems, where each sub problem corresponds to a slice. The system maintains a pool of slabs of four slices each. The four slices in a slab can be reconstructed simultaneously by taking advantage of vector instructions in the CPU cores. The system concurrently runs multiple CPU level and GPU level threads on the CPU. Each CPU level thread runs on one of the CPU core and a slab of four slices is computed simultaneously. A GPU level thread is responsible for sending input data to the GPU and receiving the reconstructed slices from the GPU. To minimize the competition of GPU level threads with the CPU level threads for the CPU time, the total number of CPU and GPU level threads is limited to the number of available CPU cores. The CPU and GPU level threads demand for the next slab when the reconstruction of the current slab is complete. Hence this method achieves dynamic load balancing and the threads process the different slabs concurrently. The implementation uses CUDA programming language. The collaborative approach achieves performance improvement of a factor of 1.5 to 2 compared to the CPU-only or GPU-only approaches.

An approach for dense matrix computations is proposed by Song et al. [7] with the objective of utilizing all CPU cores and all GPUs in multicore and multi-GPU heterogeneous systems. As the matrix tiles of uniform size for both CPU and GPUs is not a suitable choice from the performance and load balancing perspective, the algorithm uses smaller tiles for CPU cores, and larger tiles for GPUs. An auto-tuning method is used to adapt the partition size such that the difference between the host and the GPU execution time for a partition is minimal. A static two-level 1-D block cyclic distribution method is used to allocate the workload to host and GPUs evenly with minimal communication. The runtime system identifies the data dependencies between the tasks and sends the output of parent task to its children automatically. Thus the user is relieved of providing communication codes for parent-child communication. The experiment environment includes Intel MKL and Intel compilers in the host side, and CUDA, CUDA BLAS, and MAGMA in the device side. The experiments of this approach demonstrate high performance, great scalability, and good load balancing.

A CPU-GPU hybrid approach for the computation of discrete particle simulation is proposed by [8] in which the fluid flow is computed by CPU(s) while the particle motion is computed by GPU(s). The parallelization strategy adopted here is concurrent execution method and the scheduling of computations is performed statically. This approach is implemented using CUDA programming language.

Irregular wavefront propagation pattern (IWPP) algorithms involve irregular data accesses and computations. Teodoro et al.

[9] have developed a tile-based hybrid implementation of IWPP on multiple CPUs and GPUs, where input data domain is divided into non-overlapping tiles. Tiles are dynamically mapped to both CPU and GPU for processing. The implementation uses CUDA for GPU and OpenMP for CPU side parallelization.

Papadrakakis et al. [10] have presented the parallel implementation of the finite element tearing and interconnect (FETI) domain decomposition method (DDM) on CPU-GPU heterogeneous platform. Task queue method is used for the implementation of the dynamic load balancing. Both CPU and GPU extract tasks from the queue in an asynchronous manner. By performing the computation concurrently on CPU and GPU the available processors are efficiently utilized. CUDA is used for the implementation.

Chakroun et al. [11] have proposed branch and bound algorithms for solving tree-based combinatorial optimization problems by combining both GPU and multicore CPU computing. They have implemented two approaches viz. concurrent and a cooperative to perform the tree exploration process. In the concurrent approach the tree exploration problem is partitioned among CPU and GPU cores and the exploration is performed concurrently. If the new solution obtained by the CPU and GPU threads is the best one then a shared variable corresponding to the best solution is updated. The CPU and GPU use a shared list to hold the newly generated subproblems. Synchronization is necessary to update the shared list and the best solution variable. The synchronization appears to be overhead and due to which this approach does not improve the performance even if all the CPU cores are used for the exploration process. Hence the authors have observed that in general for any tree based optimization problem, all four operators (selection, branching, bounding and pruning) of the exploration process should be performed on the GPU. In the cooperative approach, each CPU thread asynchronously transfers its partition of the pools of tree nodes to the GPU using data streaming. Then it calls the kernels for branching, bounding and pruning on its partition. Finally each CPU thread copies the output data from the GPU to CPU. The GPU performs the exploration of the tree corresponding to the subproblem. This approach minimizes the CPU–GPU communication overhead by overlapping kernel calls with the data transfer operations. Also, it overcomes need for synchronization between cooperative GPU threads. This approach is implemented using CUDA.

The tile based algorithms the use a fixed block size for CPU and GPU. The fixed block size finalizes the total work before execution and hence it is difficult to achieve accurate load balance even when dynamic scheduling is used. The idling of CPU or GPU can be minimized by adjusting the block size at each iteration of the computation. Chen et al. [12] have developed an adaptive scheme to determine the optimal block size in the CPU-GPU hybrid implementation of QR factorization. The timing performance for a given size is predicted using a linear regression model. The regressors are chosen based on offline training and complexity analysis. Based on regression models two workload functions and the corresponding optimization problems are formulated using which optimal block size for CPU and GPU is determined in each iteration of the computation. Thus the scheduling strategy is dynamic with adaptive block size. Pipeline based parallelization strategy is employed in this work. CUDA is used for GPU side and parallel versions of OpenBLAS and Intel MKL are used in the CPU side parallelization.

Zhang et al. [13] presented a CPU-GPU collaborative parallel aerial image simulation algorithm. OpenMP and AVX instructions are used to parallelize the algorithm on the CPU. The GPU side parallelization is implemented using CUDA. Load balancing is dynamically performed through a rectangle-based dynamic task scheduling strategy. Multiple tasks corresponding to the simulation are processed concurrently on CPU and GPU.

A method to implement two-list algorithm to solve subset-sum problem by using both CPU and GPU cooperatively is proposed by Wan et al. [14]. This method focuses on utilizing the computing powers of CPU and GPU efficiently by effectively distributing the computation to both processors. The task distribution ratio is determined based on the parameters such as execution speed and memory size of host and device, the bandwidth, the CPU-only and GPU-only execution time for a program, and the CPU-GPU data transfer overhead. A feedback based dynamic task distribution scheme is used to overcome the load imbalance that may arise between CPU and GPU during runtime. This scheme uses the load statuses of CPU and GPU at runtime and accordingly generates a new task distribution ratio. To overcome the CPU-GPU communication overhead, an incremental data transfer method is used. This method improves the data reusability by overcoming the transfer of frequently used data to and fro between the CPU and GPU. The CPU-GPU cooperative method is implemented using OpenMP and CUDA. The parallelization strategy used here is concurrent where both host and device sides concurrently perform computations involved in different stages of the two-list algorithm. The performance of cooperative computing method is significantly better compared to CPU-only or GPU-only computing methods. However the implementation of two-list algorithm through CPU and GPU cooperative computing requires tedious efforts from the developer.

Image simulation process is used to quantitatively estimate atom species and numbers of the high quality atom resolution images acquired through scanning transmission electron microscopes (STEM). To speed up the simulation process Yao et al. [15] have developed a parallel program that distributes the computational load to CPU and GPU dynamically. Both CPU and GPU perform the computation of the simulation process concurrently. Along with GPU processing power, this approach takes advantage of the computational power of multiple CPU cores as well as the large physical memory available in the CPU. The CPU side parallelization is implemented using standard Fortran code and OpenMP. OpenACC is used to evoke the computation o GPU. During the compilation, Intel MKL and CUDA cuFFT libraries were included to use the optimized Fast Fourier transformation (FFT) routines.

GPUs are suitable for the implementation of algorithms that are regular with less control and memory divergence. Tree traversal algorithms exhibit highly irregular control flow and memory access patterns. A framework that automatically schedules and executes the tree traversals is developed by Liu et al. [16]. It is observed that the future behavior of a traversal is correlated with its past behavior. There are more chances of two traversals behaving similarly during the second half of their execution if they have behaved similarly during their first half. Based on this observation, the framework profiles the behavior of every traversal by performing partial traversals on the GPU and the behavior of each traversal is recorded in a matrix. This matrix is transferred to the CPU. The CPU uses this information to assign the threads into different buckets. Threads corresponding to traversals that access the same branches of the tree are assigned to same bucket which improves the locality. The rest of the traversal is executed on the GPU according to this new schedule. Though the tree traversal is not executed on the CPU, it is used to reorder the traversals that greatly improves locality and hence the performance of the algorithm.

A CPU-GPU parallel implementation of variable neighborhood search method and its application to inventory optimization problems is presented in [17]. CPU threads send the input data from host memory to device memory, and receive the computed values from the GPU. Then the values received by each thread are compared to select the minimal value. This process is continued until the given time limit is reached. Two GPUs are used to execute the variable neighborhood functions in parallel. The search function is implemented using OpenACC and the CPU side parallelization is implemented using OpenMP programming languages.

2.3. Frameworks for concurrent execution of multiple kernels using CPU and GPU

Wende et al. [18] have proposed an approach to concurrently execute multiple kernels of the multithreaded CUDA program on the GPU. This approach avoids the underutilization of the GPU for the small scale computation. A single NVIDIA Fermi GPU supports concurrent execution of up to 16 kernels. Concurrent kernel execution within a single-threaded CUDA program is straightforward. However in a multi-threaded CUDA application in which each host invoke multiple GPU kernels in succession, the concurrency breaks down if the kernel actions are not explicitly synchronized. To avoid this concurrency breakdown the authors propose Kernel Reordering scheme through a producer-consumer mechanism. In this scheme multiple host threads can execute their kernels on the same GPU without the need for explicit synchronization. Here each host thread is treated as producer and is assigned with a queue. Host threads enqueue their GPU kernels in the associated queues and continue their computations on the CPU. The consumer inspects the queues in round-robin fashion and dequeues one kernel per queue and invokes the respective kernel. Also the consumer notifies the corresponding producer that its GPU kernel was invoked. On receiving the notification the producer waits for its kernel to finish. With this scheme, the kernel invocations are made such that successive kernels are placed onto different CUDA streams, which is an ideal condition for concurrent kernel execution. The authors have reported a speedup of almost a factor 14 over the sequential kernel execution. The kernel reordering scheme is useful only with the Fermi GPUs that has one kernel queue which is shared by all host threads using the same device concurrently. However with the advent of Kepler GPUs, the scheme becomes obsolete as Kepler consists of 32 kernel queues.

Auerbach et al. [19] have developed compiler and runtime called Liquid Metal for the new programming language called Lime. This language enables programming heterogeneous systems involving CPU and GPU or FPGAs. Also the compiler and the runtime support enable the co-execution of the programs on these heterogeneous platforms. For a source file Liquid Metal produces a collection of artifacts, which are executable entities for different architectures. An artifact corresponds to either the whole source program or the subset of the input program. For executing the entire program on the CPU using the JVM the front end of the compiler generates Java bytecode. The backend which compiles only the subsets of the input program generates code for GPUs and FPGAs using GPU and FPGA compilers respectively. The programmer can use relocation brackets ([]) around a subset of the program to inform the compiler and runtime that the specific subset has to be co-executed. Liquid Metal runtime chooses which implementation of a task to use from a set of functionally equivalent artifacts and schedules it for execution. Thus the compiler and the runtime automatically perform the dynamic partitioning of the computations between host and the accelerators.

Robson et al. [20] have described a framework to coordinate the execution of tasks on a heterogeneous environment. This work is an extension of the CHARM++ parallel programming library [55]. The framework generates CUDA kernels from programmer-annotated functions in the host code and dynamically schedules kernels to host or device based on a user provided heuristic. The data transfer to and from the GPU is managed by system automatically. The system also allows the use of tags to specify whether a kernel is splittable so that the sub-tasks after splitting can be executed on different platforms. Using this framework the CPU-GPU coordinated execution results in speedups of up to 3.09 compared to CPU-only or GPU-only execution.

2.4. Minimizing CPU-GPU data transmission overhead through collaborative CPU-GPU execution

Data transmission bandwidth between the CPU and GPU is a bottleneck due to which the programs cannot fully exploit the computational power of GPUs. Haung et al. [21] have proposed an approach to reduce the overhead of data transmission between CPU and GPU. Generally CPU remains idle while the GPU is performing the computations. Therefore, in this approach instead of sending all the data to the GPU and processing there, a part of the data is held in the CPU and processed. By retaining a part of the data in the CPU side the data transmission overhead is minimized. Since the CPU and GPU perform the computations in parallel, the system leads to efficient utilization of all the available compute resources. The data set is divided into chunks of suitable size. Based on the processing speed of the CPU and GPU, and the data transfer rate of PCI-E, chunks of data of suitable size are distributed between CPU and GPU cores. With this method, the authors have reported a speedup of 20% compared to the GPU-only execution.

Before an application developer begins his/her efforts to port a workload to the GPU, it is necessary to guarantee the performance benefits of porting. Boyer and Kumaran [22] have proposed a framework to predict the kernel execution time and data transfer time. The framework assists the developer to estimate the overall speed up that can be obtained by porting a CPU code to GPU. The framework uses the code skeleton, which is the simplified description of the CPU code. It analyses the data usage of the code skeleton and calculates the total time needed for data transfer between CPU and GPU. Also, using the code skeleton the framework explores different code transformations and estimates the optimal kernel execution time. The sum of the data transfer time and execution time gives the overall time taken by the GPU to complete the workload. With high accuracy for the prediction of the data transfer time the performance model can be effectively used in cooperative CPU-GPU computing.

Mokhtari and Stumm [23] have proposed BigKernel**,** a scheme to reduce CPU-GPU communication overhead and provide coalesced access to the GPU memory. With this scheme the overlap of data transfer and computation is achieved in four stages. In the first stage the addresses of the data needed by the GPU computation is recorded in an address buffer in the CPU memory. In the second stage a dedicated CPU thread fetches the data into prefetch buffer in the CPU memory using the addresses available in the address buffer. The data is placed in the prefetch buffer so as to enable coalesced accesses once it is transferred to GPU memory. In the third stage the DMA engine of GPU transfers the data from prefetch buffer to the data buffer in the GPU. The fourth stage executes the kernel using the data from the data buffer. The kernel is suitably transformed to replace the original memory accesses with the accesses to data in the data buffer. This technique enables the execution of kernels that process arbitrarily large data sets which cannot be fitted in the GPU memory by partitioning them into chunks. With this method, an average speed up of 1.7X is obtained compared to the schemes that use double buffering. Though the BigKernel eliminates the bottleneck due to PCIe, the GPU memory access still remains to be the bottleneck.

Sunitha et al. [24] have demonstrated the effects of overlapping data transfer and the kernel execution on overall execution time of some CUDA applications. CUDA streams are used for this purpose. A stream is a set of device operations. Data transfer and kernel execution operations invoked by the host are queued in the stream and are executed in the order in which they are added into the stream. When the programmer does not specify the stream, the operations are enqueued into the default stream. It is possible to concurrently execute the operations from two or more non-default streams on the GPU. In this work the benefits of using different levels of concurrencies such as 2, 3, 4 and 4+ way concurrency are explored. In 2-way concurrency, initially the entire input data is transferred to the device. Now the kernel is launched which processes a chunk of the input data. When the first kernel finishes the execution, the same kernel is launched again to process the next chunk of the input data. When the second kernel is being executed the results produced by the first kernel-launch is transferred back to the host from the device. In this way the data transfer operation is overlapped with the kernel execution. In the same way, the 3-way concurrency overlaps data transfer from host to device, device to host and the kernel execution. The 4-way concurrency extends 3-way concurrency by assigning some workload to one of the idle CPU core. The 4+ -way concurrency further extends the 4-way concurrency to utilize multiple CPU cores. With this method enhanced performance is observed for a set of applications. Also, using this method both the CPU and GPU cores are utilized efficiently. However the techniques used in [21] and [24] are applicable to only those cases in which the workload of an application can be divided into many blocks.

The current NVIDIA hardware scheduler uses only the resource requirement information of the kernel to determine the order of launching the kernels. Thus the scheduler does not consider the possibilities of overlapping between the data transfer and kernel launching commands of different independent tasks. A model to determine a task execution order has been developed in [25] that minimizes the total execution time by increasing GPU usage, and exploiting opportunities for overlapping the data transfer and kernel computation commands. Compared to the Hyper-Q, the concurrent kernel execution feature supported by NVIDIA, this model has obtained performance improvement of up to 19% for real workloads.

2.5. Compilers and runtimes for portability among GPU and multicore processors

MCUDA [26] is a compiler that translates the CUDA kernels into a form suitable for efficient execution in multi-core CPUs. The kernel function is translated from a per-thread code form to a per-block code form using a technique called *thread loop*, which is an iterative structure around the entire body of the kernel code. Thus each CUDA thread becomes a loop iteration over the kernel code. To implement the barrier synchronization within thread loop, loop fission technique is applied at the synchronization point. To enforce synchronization within the control structures a technique called *deep fission* is used. This technique creates new thread loops within the scope containing the control statements. MCUDA supports both static and dynamic scheduling methods to assign thread blocks to the worker threads. With MCUDA the CUDA applications written for GPU architecture can be efficiently executed on multicore CPU architectures.

Harmony [27] is a programming and execution model for heterogeneous many core systems which simplifies the application development and also enables binary portability across different architectures. A kernel is dispatched to the runtime as and when the application invokes the kernel through non-blocking APIs. The programming model supports a shared address space which is accessed by all kernels. The shared address space enables the runtime dependency checks. Synchronization is achieved by having the application check for the availability of the output of a dispatched kernel through blocking calls. Harmony provides syntactic structures known as control decisions to explicitly specify the control dependent kernel execution. A control decision specifies the next kernel to be executed. The execution model follows the style of execution employed in out-of-order (OOO) superscalar processor. A kernel and a heterogeneous processor in Harmony correspond to an instruction and a functional unit respectively of an out-of-order execution processor. The runtime provides for each kernel the binaries for different architectures. This ability to execute the kernel on different architectures enables the performance to scale up the as more processors are added to the system. Dependence between kernels is updated as the kernels complete the execution. When a kernel becomes free of any dependence, it is dynamically mapped to any heterogeneous core. For a particular input data, the execution time of a kernel is a function of underlying architecture. To make the scheduler more effective, the authors plan to use online monitoring support through which the execution time can be modeled as a function of data value and the underlying architecture.

Like GPUs, FPGAs are also widely used as accelerators in CPU based heterogeneous computing systems. Through their reconfigurable fabric FPGAs enable exploitation of application specific parallelism. However the performance benefits of FPGAs are subsided by the difficulty involved in programming them. FCUDA [28] provides a mechanism that enables the execution of CUDA kernels on FGPA. It takes a CUDA program as the input and transforms it to C code annotated for coarse-grained parallelism. It then uses Autopilot, a tool that enables coarse grained FPGA programming. The Autopilot maps the coarse grained parallelism into application specific processing engines in FPGA. Finally, it generates RTL description for the specific FPGA, which is then synthesized before downloading into the FPGA. The authors have reported a speed up of up to 2X using FPGA compared to GPU implementation for a set of CUDA kernels. The objective of the FCUDA is to transform a CUDA kernel to parallel C code so that it can be executed on FPGA.

Ocelot [29] is complier framework that maps the PTX BSP (Bulk-Synchronous Parallel) execution model used in CUDA programs to a set of many core architecture execution models. It uses LLVM (Low Level Virtual Machine) code generator to translate PTX instructions to x86 and other instruction set architectures. The advantage is that no recompilation is required as it is not a source to source compilation technique like MCUDA. The translation process of the CUDA binary in the form of PTX to x86 begins with the PTX to PTX transformations so as to enable the creation of LLVM intermediate representable form. The next step is to translate PTX to LLVM IR. The LLVM IR is subjected to further optimizations and the target code is generated. To execute PTX program on many core architecture, along with the translation from PTX to x86, Ocelot also allocates memory space for statically allocated variables. Ocelot uses the Hydrazine thread library to bind worker threads to the available CPU cores. The main thread will assign a subset of GPU threads (Cooperative Thread Arrays) to the worker threads. To preserve the global barrier semantics of the bulk synchronous execution model, the main thread then blocks itself until all the worker threads have completed its execution. Like MCUDA, Ocelot enables the execution of CUDA kernels efficiently on multicore x86 architecture. The frameworks presented in [26], [28], and [29] does not facilitate the cooperative execution of a kernels on both CPU and GPU. But, just like the usage of Ocelot in developing the CHC frame work [1], these frameworks can be used to develop other frameworks that facilitate CPU-GPU cooperative execution.

Gummaraju et al. [30] present *Twin Peaks*, a framework that enables the execution of OpenCL kernels on multicore CPUs. With this feature the framework achieves portability and load balancing among GPU and CPU. The compilation system of the *Twin Peaks* generates device specific binaries using LLVM compiler framework [56]. A runtime system schedules the tasks to the underlying compute devices. All threads that belong to a workgroup are combined into a single OS-thread called as light-weight thread (LWT). A light-weight thread is mapped to a single CPU core. The use of LWT reduces the context switching overhead that will arise when each GPU thread is mapped to an OS thread. The barrier synchronization on the CPU is implemented by executing a work item until the synchronization point, save its context, switch to the next work item using *setjmp()*, and so on until all work items of the workgroup are executed till the synchronization point. On encountering the last work item of the workgroup restore the state of the first work item using *longjmp()* routine, execute till its completion or till the next synchronization point, restore the state of the next work item, and so on. The runtime uses optimized setjmp() and longjmp() routines which save and restore only a few registers during context switching between work items. The optimized routines minimize the overhead that incurs when the generic setjmp() and longjmp() library routines are used. By enabling the applications to utilize both CPU and GPU *Twin Peaks* improves the performance.

Hong et al. [31] have developed a framework called MapCG that enables source code level portability of MapReduce applications across CPU and GPU platforms. GPU does not support dynamic memory allocation which is very much essential for the MapReduce framework to dynamically allocate the memory space for intermediate data. To fill this gap MapCG supports lightweight memory allocator on both CPUs and GPUs. MapCG uses a hash table to efficiently group the intermediate data on GPU. The framework provides considerable performance advantage over the state-of-the-art MapReduce implementations on CPU and GPU.

StarPU, a runtime layer to provide portability of code execution across multicore processors and accelerators, is developed by Augonnet et al. [32]. The APIs provide a convenient way to generate parallel tasks and develop task schedulers to exploit heterogeneity among multicore processors, CUDA-enabled GPUs, and the CELL processor. For each architecture programmers provide the abstraction of a task, known as Codelet which is implemented using programming languages like CUDA or libraries like BLAS routines. Dependencies between the Codelets are enforced by the StarPU using the dependencies between tasks provided by the programmers. Support for execution of a Codelet on a particular architecture is provided by the respective driver. Scheduler programmers can use the low level scheduling mechanisms provided by StarPU in high level to implement portable scheduling policies. The need to provide different implementation of a Codelet for different architectures requires more effort from the programmer.

2.6. Task scheduling schemes in CPU-GPU cooperative execution

The task schedulers efficiently allocate the tasks to different processing units with the objective of improving the performance or reducing the power consumption. A task scheduler for a particular purpose is based on a scheduling model that determines the order in which several tasks are to be executed and the processor where the task is to be executed. The decision of the task scheduling model is based on several features of the processor and the task to be executed. The task schedulers can be classified as static and dynamic schedulers. A static scheduler allocates tasks to processors even before the execution begins. In dynamic scheduling tasks are allocated to processors during the execution of the program. We present here the various task scheduling approaches for performance improvement and energy saving. Most of the approaches presented below utilize the processing power of both CPU and GPU to attain these objectives.

2.6.1. Task scheduling schemes for performance improvement

A CPU-GPU collaborative computing model supported by a task scheduling algorithm to schedule the subtasks of an application to CPU or GPU is given by Wang et al. [33]. This algorithm schedules the tasks based on the computing capability of processor, communicating cost, and size of data sets. According to this algorithm, when the size of the data sets for two subtasks is same, a compute intensive subtask is scheduled to GPU, and a communication intensive subtask is scheduled to CPU. The algorithm improves the performance by using both CPU and GPU collaboratively to complete a task.

Qilin [34] describes a technique to adaptively map computations to CPU and GPU cores. It relies on training and profiling to maintain a database which facilitates adaptive mapping. The database contains information about the execution time of a computation on CPU as well as GPU. The technique easily adapts to varying size of input or varying hardware configuration of the system. An API for parallelizing operation is provided which makes the job of the compiler easier as it no longer have to search for parallel code from the serial version. To adapt to the changes in the runtime environment, Qilin uses Dynamic compilation technique to convert the API calls into a target machine code. A speedup of 25% over static mapping techniques is obtained using this scheme. Further, this technique provides a 20% reduction in energy consumption. The profiling phase to build the database of execution times is an overhead. Moreover the system relies on the programmer to explicitly express the parallelizable computations through API. Also the dynamic compiler would add compilation overhead during run time.

In CPU-GPU heterogeneous systems the execution time, power consumption, and response time can be improved, if different applications can fully utilize all the available computational resources. In this regard, a predictive user-level scheduler is developed by Jiménez et al. [35], with the objective of efficiently scheduling the tasks to the most suitable computing unit (CPU or GPU) considering the characteristics of the code to be scheduled and computing units present in the system. The scheduler is implemented as a dynamic library for the Linux operating system. The scheduling process involves PE (Processing Element) selection and task selection steps. The PE selection step chooses a PE for the execution of the next task. This step is accomplished by using a variant of the first-free algorithm. The basic first-free algorithm first tries to schedule tasks in a PE which is currently free. The modified first-free algorithm tries to achieve load balance between the PEs by introducing a bias in the scheduling system. However the first step does not choose the best computing unit for a given application. The task selection step chooses the next task to be executed on a given PE, when the PE finishes the execution of the current task. The second step is based on performance history which is obtained by executing a task on different PEs. A task is scheduled to any of the PEs among which the performance unbalance is least. Compared to the GPU-only execution, by scheduling on to both CPU and GPUs this scheduler achieves speed up ranging from 30% to 40%. However the need of profiling to obtain the performance history for every PE and task pair is an overhead in this approach.

KernelMerge [36] is a kernel scheduler that runs two OpenCL kernels concurrently on one device. Executing one memory-bound kernel at a time in the device may leave the compute resources underutilized. Similarly executing only a compute bound kernel may leave the memory resources underutilized. Hence the authors propose that running two kernels concurrently can take advantage of the underutilized resources, improving overall system throughput. KernelMerge combines two invocations of OpenCL kernels into a single kernel launch called as scheduler kernel. The scheduler creates specified number of “scheduler workgroups”. The scheduler then dispatches a kernel workgroup from the individual kernels to each scheduler workgroups. A technique called “spoofing” is used to convert the global scheduler workgroup IDs to the original kernel workgroup IDs. Two scheduling algorithms, viz. work stealing and partitioning by kernel, are used in KernelMerge. With the work stealing algorithm the allocation of resources to each kernel is not fixed, and it depends on the run time of each kernel. With the partitioning algorithm the scheduler can dedicate a fixed percentage of resources to each kernel. The authors report a maximum speed up of 18% for a pair of concurrent kernels over sequential execution of the same.

KernelMerge is advantageous only when the application has more than one kernel. Another limitation of KernelMerge is the overhead involved in the process of spoofing. Also the number of the available registers and the shared memory available in a GPU limits the number of workgroups that may simultaneously occupy the device. Since the amount of shared memory on a GPU is limited, a kernel using more shared memory limits all schedulable kernels.

A Functional Performance Model (FPM) represents the processor speed as a function of problem size, and characteristics of architecture and application. Zhong et al. [37] build a FPM for heterogeneous multicore and multi-GPU systems by executing several optimized GPU kernels. They use the FPM model in their data partitioning algorithm for the CPU-GPU hybrid systems. They have demonstrated that the algorithm optimally distributes the matrices among the multicore CPU and GPUs, and balances the workload efficiently.

OpenCL allows the portability of tasks among the multiple heterogeneous compute devices. However the scheduler is not capable of automatically choosing a compute device for a given task. Sun et al. [38] have proposed an API extension for OpenCL which helps the programmer to develop scheduling schemes that can automatically schedule tasks to the compute devices. A kernel is combined with additional information to enable the execution of kernel on multiple devices. Such kernels are called work units. The scheduler mechanism uses work pools where several work units can be enqueued. The scheduler dequeues a work unit from the work pool and schedules it on the compute device specified by the programmer. Thus the scheduling policy is static. The task queueing scheduler improves the performance gain on single GPU device and it is scalable on multi GPU systems.

Grasso et al. [39] have developed an automatic task partitioning framework for OpenCL programs on heterogeneous systems. This framework uses Insieme Parallel Intermediate Representation (INSPIRE) compiler infrastructure [77]. The task partitioning process is problem size sensitive which consists of training and deployment phases. The purpose of the training phase is to build a partitioning prediction model which is based on machine learning approach. In the training phase static program features are collected for a set of OpenCL programs using INSPIRE and stored in a database. Also the backend of the framework generates binary code for different OpenCL devices. These codes are executed with different problem sizes and task partitioning. The runtime features thus obtained are added to the database. A machine learning prediction model is built using the static program features and the runtime features. In the deployment phase, the static program features and the runtime features for the current OpenCL program is collected and provided to the prediction model. The prediction model gives an optimal partitioning for a given problem size and the program is executed on the chosen device with the predicted partition. For a set of benchmark programs the framework exhibits performance improvement compared to CPU-only or GPU-only execution. However the process of collecting runtime feature for a given program is an overhead in this approach.

Kernelet [40] is a runtime system to enhance the performance in environments like clouds and clusters where several kernels are submitted to the GPU by the different users. Executing a single kernel at a time on a GPU underutilizes the GPU resources. Kernelet is based on a feature called concurrent execution of multiple kernels, which is supported by the recent GPUs that enables the efficient utilization of all SMs of the GPU. It makes use of two techniques, kernel slicing and co-scheduling. A kernel is divided into sub kernels called as slices, each consisting of multiple thread blocks. By slicing the kernel, the Kernelet provides more opportunity for concurrent execution of slices from different kernels, which in turn improves the throughput of concurrent kernel executions. The size of the slices should be such that slicing overhead is less than 2% of the kernel execution time. To handle the scheduling of the slices two schedulers are used: memory command scheduler to handle data transfer instructions and kernel scheduler to co-schedule the slices from the different kernels that are ready for the execution. Co-scheduling enables concurrent execution of the slices from different kernels. It uses a greedy algorithm for the scheduling of the two ready kernels. The authors have reported a performance improvement of up to 31% on NVIDIA Tesla C2050 GPU. With Kernelet, overhead of data transfer is masked by overlapping data transfer with kernel execution. However, Kernelet is useful only in multiuser environment such as clusters and clouds where different users submit their kernels to execute on shared GPUs.

Yao et al. [41] describe strategies for partitioning workload between the CPU and GPU. The C source program is preprocessed to determine a computing resource (CPU or GPU) suitable for the given application. The characteristics of the given application such as memory access, arithmetic operations, control flow structure, and data parallelism are considered to determine whether CPU or GPU is suitable for a given application. If a subprogram consists of frequent memory accesses with high locality then the algorithm assigns it to the CPU due to the availability of the cache. Subprograms having high continuity of memory accesses are assigned to GPU. The algorithm computes the arithmetic density of a subprogram using abstract syntax tree. Based on a threshold which is related to the FLOPS (floating point operations per second) of CPU and GPU, the algorithm determines if a subprogram with a given arithmetic density is to be executed on CPU or GPU. If the control flow structure of a program is constructed mainly of loops, body of which contains many branches then such program is assigned for execution on CPU. A sub-program is suitable for GPU in cases in which the control flow structure is constructed by loops and loop iteration is very big and computations in the loop body is complex, or control flow structure is constructed by branches in which few statement blocks process most of the data. Sub-programs with high degree of data parallelism are more suitable for GPUs. To represent the data parallelism of a sub-program, an attribute of the sub-program called arithmetic intensity is used. It is the ratio of total number of arithmetic operations and the total amount of data in data dependencies. Higher is the amount of data dependencies lower will be arithmetic intensity and hence the amount data parallelism. The sub-programs with high arithmetic intensity are given to the GPU. With the proposed algorithms and strategies the authors have reported better performance for programs on CPU-GPU heterogeneous environment.

Aciu and Ciocarlie [42] propose an algorithm for executing code on a CPU-GPU heterogeneous platform. Some applications are not suitable for the GPU due to the drawbacks of the GPU such as its poor capabilities for pointer exchange with the host, stackless execution model, etc. To select the most suitable processing resource, CPU or GPU, for a given application, the authors propose an algorithm. According to the algorithm, a thread is created for each of the CPU core and each thread is assigned with the set of CPU and GPU functions. In order to utilize all the computing power of the GPU, once a host thread reaches the GPU function in the assigned set of functions, the proposed algorithm pauses the thread until sufficient kernel calls are collected from the different host threads. These kernels are then executed on the GPU and results are passed to the respective host threads. Then host threads resume their execution. To avoid the wastage of CPU cycles when the host thread is paused, cooperative threads are used which are supported in mainstream operating systems as APIs. Every thread has a pool with its associated fibers. For each thread function a fiber is created and added to thread's fibers pool. Whenever the thread encounters an invocation of a GPU kernel, a GPU scheduler object yields the execution to the next fiber from the pool. Thus the threads waiting time is efficiently utilized to execute other fibers. For a test application the cooperative CPU-GPU execution model results in significant amount of performance improvement. However to make the framework more suitable for real world applications, accurate metrics need to be used to evaluate the performance of a given function on different computing resources.

Wen et al. [43] have presented an OpenCL runtime task scheduler for CPU-GPU heterogeneous systems that schedules multiple kernels from different programs such that a given kernel utilizes the best computing resource for its execution. The scheduling priority is decided based on the factors such as the predicted speedup and input data size. According to the scheduling policy the high speedup kernels are scheduled to GPU and the low speedup kernels are scheduled to CPU. The kernels with same speedup will be scheduled based on the input size, i.e., the kernels with smaller input sizes will be scheduled to the CPU. Classification of kernels into high and low speedup categories is performed by a support vector machine based classifier. This classifier is trained offline. The training process involves measuring the GPU execution speedup over CPU execution for each kernel of a set of training programs. Also the training process collects static features such as the number of instructions and dynamic features such as the number of work items for each training program. Considering the throughput and average normalized turnaround time as the performance metrics, and comparing to FluidiCL [2] this technique gives significant performance improvement.

Li et al. [44] have proposed a scheduling technique for concurrent execution of kernels on GPU. The scheduler is based on symbiosis between the kernels. If two kernels improve the performance when executed concurrently by efficiently utilizing GPU resources they are said to be symbiotic. For every pair of kernels a symbiotic score is calculated offline by analyzing the resource requirements and execution characteristics. The symbiotic score is stored in a table. Among a set of kernels, the scheduling algorithm selects a pair of kernels with highest symbiotic score. The number of kernels executed concurrently depends on the resource requirements of the kernels to be executed concurrently and the available GPU resources such as registers, shared memory, number of threads etc. Compared to the basic scheduling algorithms their approach provides a near optimal scheduling order which improves performance and energy efficiency. However, the possibility of CPU-GPU cooperative execution of kernels is not considered in this algorithm.

Vilches et al. [45] have proposed an algorithm that partitions the parallel loops in irregular applications into appropriate chunks of iterations for parallel execution on CPU and GPU. The partitioning strategy monitors the throughput of computing device during the execution of the allotted iteration space. The chunk size is dynamically adapted to balance the workload among the CPU and GPU, thus optimizing the throughput. The authors extended the *parallel_for* template of the Intel Threading Building Blocks (TBB) to provide a programming interface to allow the exploitation of the parallelization scheme. For a set of irregular benchmarks the partitioning strategy improves the performance and saves the energy.

A task allocation model that classifies and allocates tasks of a given application for efficient utilization of CPU and GPU resources is implemented by Wang et al. [46]. Based on the size of input data, output data, the degree of computations and control flow involved the tasks of an application are defined as input type, output-type, computing-type and logic-type respectively. The tasks are classified into two sub sets, CPU-kind and GPU-kind, using SVM (Support Vector Machine) by considering the above characteristics of the tasks. Once the classification of tasks is made the sub-groups are then adjusted and mapped onto processors in such a way that both CPU and GPU require almost equal time to complete the allocated tasks. The mapping is based on the current running status and computing ability of processors. The mapping of tasks avoids one processor completing its tasks much ahead of the other due to which the faster processor remains idle too long while the other processor is busy executing the tasks. The authors have claimed an average performance improvement of up to 23.43% compared to some state-of-the-art allocation techniques. Since the task allocation model does not consider the effects of task synchronization and global memory accessing on CPU and GPU execution times respectively a deviation of measured value from value calculated by the model is observed.

A Service Level Agreement (SLA) aware user mode runtime framework vHybrid is proposed in [47] to efficiently utilize the CPU and GPU resources in the cloud platforms. The framework utilizes two control policies (open-loop and adaptive control) for CPU and a scheduling policy for GPU. To control the resource utilization rates of different applications vHybrid sets proper sleep time for each workload. This is achieved by exploiting the library interposition. For a cloud environment, vHybrid is able to provide optimal CPU-GPU utilization and response time while maintaining the desired quality of service (QoS).

2.6.2. *Task scheduling approaches for energy saving*

Wang and Ren [48] propose a method to distribute the workload of a single application between CPU and GPU, and also to scale the CPU and GPU frequency with an objective of minimizing energy consumption under a given scheduling length. The authors have developed an energy efficient work distribution model for a given scheduling length constraint and a hardware-supported running level of CPU and GPU. An algorithm based on this model takes dynamic power consumption of processor at different running levels as input and traverses all possible combinations of different running levels for CPU and GPU to find out the optimal work distribution ratio. It is reported that the method reduces the energy consumption significantly. However the computational complexity of the algorithm may seem to be an overhead for large number of running levels of modern processors.

Rong Ge et al. [49] have developed PEACH (Performance and Energy Aware Cooperative Hybrid computing), a model to predict the performance, and energy efficiency for cooperative CPU-GPU heterogeneous computing systems. Using PEACH, for a given performance and energy efficiency requirement the user can determine the optimal workload distribution between host and device, and DVFS scheduling that adjusts the host and device frequencies to meet the performance and energy saving requirements. The authors have given basic analytical model for the overall compute rate and energy consumption of a hybrid system during workload execution. To handle both workload distribution and DVFS scheduling, the basic model is integrated into the CPU and GPU DVFS. This is done by substituting the relation between the basic model parameters and the CPU/GPU frequencies. Some important among the different observations made by the authors are that for cooperative hybrid computing the best performance distribution is different from best energy distribution. And for a given CPU speed, both energy efficiency and performance improvement can be achieved by running the GPU at higher speed and offloading larger percentage of workload to GPU. For a set of codes, PEACH predicts the performance and energy with an overall error rate of less than 3%. However the current model of prediction is not automatic.

An energy efficient CPU-GPU parallel computation approach for Conjugate Gradient Method (CGM) is proposed by Lang and Runger [50]. A model to predict energy efficient distribution of workload to CPU and GPU is introduced. This model uses parameters such as frequency and power consumption of CPU and GPU, the speed of CPU-GPU data transfer and energy needed for the same, and the choice of voltage and frequency scaling. During each refinement step of the finite element method (FEM), the execution time for the subsequent step is predicted. Based on this feedback the workload distribution is dynamically adapted. Thus the model dynamically responds to the changes in the parameters such as the increase in data size or changes in the underlying hardware. This online autotuning feature makes this model suitable for cloud computing, where the hardware on which the application runs is not known beforehand. Using this model the authors find that on a Sandybridge machine with a Tesla C2075 GPU the CPU-GPU collaborative execution of CGM is most energy efficient.

GreenGPU [51] is a framework for reduction of energy consumption by distributing the workload on both CPU and GPU. The framework consists of two tiers. Workload distribution module makes the first tier. The workload is distributed in such a way that the completion of assigned workload by the CPU and GPU cores takes the same amount of time, thereby avoiding wastage of energy due to idling. The second tier involves evaluating the utilizations of GPU core and memory, and then throttling of the frequencies accordingly to conserve energy. This method provides an energy saving of around 21%. The main objective of GreenGPU is lower the energy consumption of GPU-CPU heterogeneous architectures. But this objective is achieved at the cost of a marginal degradation of the application performance.

A framework to optimize the performance and energy consumption is developed by Siehl and Zhao [52]. The framework uses an energy prediction model. It executes the computation of the input program for a short time interval and measures the performance data. Using the energy prediction model the framework dynamically fine tunes the workload distribution for the next interval. The authors report considerable energy saving compared to CPU-only and GPU-only execution of the computations.

Chau et al. [53] have proposed a heuristic algorithm to schedule tasks on CPU-GPU heterogeneous systems that minimizes the total energy consumption. The algorithm considers that the number of cycles needed to complete a task depends on the type of the processor and the processors are capable of scaling the speed. A job is assigned to a processor such that it modify the least the maximum workload among all processors. They have also extended their algorithm for online case where the jobs have different release times. For different simulated cases the heuristic algorithms achieve near optimal performance.

An application specific technique to save energy is proposed by Gong et al. [54]. They have used a CPU-GPU cooperative DVFS scheme to reduce energy consumption in the implementation of the High Efficiency Video Coding (HEVC) decoder on an embedded CPU-GPU platform. The initial stage of the decoding task of a frame is performed by CPU and the later stages are performed in the GPU. When GPU is performing the decoding task of frame i-1, CPU performs the decoding task of frame i. The synchronization between CPU and GPU is necessary to ensure that GPU has finished the task of frame i-1 before the CPU launches the task of frame i on the GPU. To estimate the GPU or CPU frequency it is necessary to predict the workload of the incoming task. It is observed that the workload of the current decoding frame has a monotonic correlation with the coding parameter, i.e. the total number of transform units (TUs). The predictor uses this coding parameter to improve the workload prediction of a frame. Based on the workload prediction the DVFS chooses minimal frequency for CPU and GPU which meets the requirement of a given frame rate. Through experiments the authors demonstrate that their DVFS scheme is significantly saves energy for HEVC decoding.

Stafford et al. [57] analyze the advantages of dividing the workload of a single kernel among the CPU and GPU of a heterogeneous system. They develop several models to efficiently partition the workload among the computing devices in order to increase the performance and decrease the power consumption.

2.7. Cooperative computing using coupled CPU-GPU architectures

Several researches have been carried out to assess the effectiveness of coupled CPU-GPU architectures for high performance computing. Daga et al. [58] have reported that on AMD Fusion architecture the data transfer time for a set of benchmarks is 1.7–6.0 times better compared to discrete GPUs. The improved data transfer time results in improved overall performance of the application. The authors show that for *reduction* benchmark, the AMD Fusion with only 80 GPU cores provides a performance improvement of 3.5 times compared to the AMD Radeon HD 5870 GPU with 1600 GPU cores. Lee et al. [59] show that by properly designing the memory accesses in a data-intensive kernel, a less powerful coupled CPU-GPU architecture can outperform the a more powerful discrete GPU. The research works in papers [60] and [61] analyze the efficiency benefits of coupled CPU-GPU architectures compared to the discrete GPUs. The authors of the paper [61] carried out the performance and the power efficiency evaluation and comparison of AMD APU (Kaveri) against discrete GPU (Tahiti). The authors find that for some medium-sized or communication intensive problems the performance of the APU (Kaveri) can be as good as or better than that of discrete GPUs. The authors also show that for all memory bound and compute bound workloads, the APUs are more power efficient than discrete GPUs. Dashti et al. [62] analyze the performance/functionality trade-offs of memory management methods supported by different programming frameworks for heterogeneous systems.

There are research works on implementation of some applications on coupled CPU-GPU architectures. Even though tree searches are data parallel, accelerating tree searches using discrete GPUs face the challenge due to the irregular representation of the tree in the memory and also due to the need to copy the tree from CPU to GPU memory through PCI-e bus. Acceleration of B + tree searches on an APU is presented by Daga et al. [63]. The authors find that discrete GPU performs better than APU when the tree resides in the GPU memory. However the memory size of the discrete GPU limits the size of the tree that can be operated on. When the tree does not reside in the GPU memory, the performance of discrete GPU is much lower than that of APU.

Performance of implementation of Deep Neural Network (DNN) models on discrete GPUs and APUs is presented by Gu et al. [64]. The authors find that though the discrete GPUs are faster than APUs, the performance per watt efficiency of APUs is up to 2 times higher than that of discrete GPUs. Hence the APU servers can be an energy efficient alternative for accelerating DNN applications. Similar results were obtained by Wyrzykowski et al. [65] for the execution of Reed-Solomon erasure codes on AMD APU architecture. Even though the performance of discrete GPU is higher than the APUs, taking into account the power consumption, the authors opine that the APU architecture is a better choice for the implementation of the Reed-Solomon codes.

The above research works on coupled CPU-GPU implementations show the significance of these architectures in the high performance computing. But in these research works only the GPU of the coupled architecture is used to perform the given computation. However, hybrid implementation of an application that utilizes both CPU and GPU of a coupled architecture can provide better performance than the CPU-alone or GPU-alone implementation of the corresponding application. The research works in this direction attempt to improve the resource utilization of the coupled CPU-GPU architecture which in turn results in improved performance.

Delorme et al. [66] have implemented a parallel radix sort that simultaneously uses both CPU and GPU of an APU. The authors identify three major factors that affect the performance of the parallel sort: the granularity of sharing the data between CPU and GPU devices, the load balancing schemes, and determining the region of the memory where the sort buffer is allocated. The input data is partitioned into tiles and assigned to CPU and GPU. The data sharing granularity determines whether CPU and GPU independently sort the allocated tiles (coarse-grain) or whether the sort buffers are used by both devices (fine-grain). The synchronization overhead is less in the coarse-grain sharing. At the same time the tiles can be allocated in the memory which is preferred by corresponding device. Allocating the data in the preferred memory of a device enables fastest access for that device. The drawback of the coarse-grain sharing is that after each device completes sorting its tile, the sorted tiles need to be merged. Fine-grain sharing increases the synchronization overhead. Also fine-grain sharing complicates the allocation of data in the preferred memory. However the need to combine the sorted results is eliminated. The load balancing schemes determine the size of the tiles assigned to CPU and GPU. The size of the CPU and GPU tiles are chosen such that both the devices sort the assigned tiles roughly at the same time. Accordingly there are two partitioning schemes: Fixed partitioning and variable partitioning. Implementation of fixed partitioning is simpler as it allows the allocation of data in the preferred memory of a device. Variable partitioning achieves better load balancing, as a result it improves the performance. The authors present three different implementations of parallel radix sort based on the design issues discussed above. Among these implementations the fine-grain sharing with variable data partitioning provides best performance. Further the authors report that the parallel radix sort using both CPU and GPU of an APU is 1.8 and 1.9 times faster than the CPU-only and GPU-only parallel radix sorts respectively.

He et al. [67] study the fine-grained co-processing schemes for hash joins on coupled CPU-GPU architecture. The usage of a same physical memory in a coupled CPU-GPU architecture enables the fine-grained co-processing and data sharing. The authors show that the fine-grained co-processing improves the performance significantly compared to the CPU-only, GPU-only and conventional CPU-GPU co-processing.

An in-cache query co-processing system on coupled CPU-GPU architectures is proposed in [68]. In this system the CPU-assisted prefetching scheme and the decompression schemes are adapted to reduce the cache misses and improve query execution performance respectively. This query co-processing system employs heterogeneous workload distribution approach where both CPU and GPU can perform decompression, and query processing. In addition the CPU can also perform memory prefetching. The experiments show that this query co-processing system with the workload distribution schemes can significantly improve the performance.

Eberhart et al. [69] present CPU-GPU hybrid implementation of stencil computations on a coupled CPU-GPU architecture. Computation of the borders of stencil domain may cause compute divergence. The authors propose that the CPUs are suitable than GPUs when the amount of the compute or memory divergence is high. They call this deployment as task-parallel. The regions within the domain, with low divergence are executed on the GPU in a data-parallel fashion. For the hybrid implementation of a memory-bound stencil, the authors report a 30% speed up over the GPU-only deployment.

An approach for energy efficient query co-processing in CPU-GPU embedded environments is given by Cheng et al. [70]. The authors find that the CPU is more energy efficient than GPU for selection and aggregation, where as GPU is energy efficient for sort and hash join. Their results prove the energy efficiency of the CPU-GPU query co-processing in embedded systems.

Zhang et al. [71] analyze the CPU-GPU co-running behavior of several benchmark programs on integrated architectures. They observe that not all programs benefit from co-running. Hence the authors develop a model to predict the performance benefit of co-running a program on integrated CPU-GPU architecture. They have also developed a profiler-based predictor that enables optimal workload partitioning between CPU and GPU. Co-running the programs using these methods achieved 34.5% and 20.9% performance improvement compared to the CPU-alone and GPU-alone execution of those programs respectively.

Lupescu et al. [72] use the GPU in a coupled CPU-GPU architecture to accelerate the CPU sorting. In their approach, the GPU partially sorts the chunks of input data set. Once all the chunks are partially sorted the CPU sort algorithm is called to sort the whole data set. With this method, the authors have shown significant improvement in the performance for different CPU sorting algorithms.

Co-processing of irregular applications such as graph processing on coupled CPU-GPU architecture pose challenges in the effectiveness of workload partitioning. Zhang et al. [73] et al. have developed a software framework called *FinePar* that can efficiently partition the workload for co-processing of irregular applications considering the architectural differences between the CPU and integrated GPU. Their results show that *FinePar* improves the resource utilization and hence improves the performance compared to the optimal coarse-grained partitioning method.

Zhu et al. [74] present a study of challenges of co-scheduling independent jobs on coupled CPU-GPU architectures considering the power cap. The authors find that factors such as memory contention, power contention and job lengths affect the performance. The proposed heuristic algorithms find efficient co-schedules that provide significant throughput compared to the default schedules.

Fang et al. [75] analyze the effects of data partitioning on the performance of co-processing different applications on coupled CPU-GPU architectures. They observe that not all applications are benefited from data partitioning. They report that data partitioning has little effect when the data set is small and one of the processor is computationally very powerful.

3. The future research directions in the area of cooperative CPU-GPU computing

The heterogeneous computing systems with multicore CPUs and GPUs are widely used for solving compute intensive and data parallel applications. Due to the availability of large number of cores the GPUs provide high computational power. Also, the modern CPUs possess multiple processor cores and each core has the support for vector (SIMD- Single Instruction Multiple Data) operations. With this ability even the CPUs provide huge computational power. The current programming models for GPUs do not support mechanisms for efficient utilization of available CPU and GPU computing resources. Along with this there exist other challenges like reducing the power consumption and minimizing the data transmission overhead between CPU and GPU. The literature survey in Section 2 brings out the different research approaches that try to overcome these challenges by involving CPU and GPU cooperatively for performing the computations. From the literature survey we observe that there are still some issues that need to be addressed in the future research. Some of these issues are as follows:

- The frameworks for cooperative execution of a single kernel on both CPU and GPU need to dynamically determine which of these two processors is most suitable for the execution of the given kernel and accordingly partition the workload in such a way that the computational resources are utilized efficiently. Such frameworks also need to focus on minimizing the overhead of dynamically translating the kernel from one instruction set architecture to the other.

- The number of registers and the amount of shared memory available in a GPU is limited. Hence the runtime systems that enable concurrent execution of different kernels either from single or multiple users need to employ efficient mechanisms for partitioning the kernels and scheduling the partitions so that optimal number of kernels can be executed simultaneously.
- The overhead of data transfer between CPU and GPU is major hurdle in the performance improvement of CPU-GPU heterogeneous computing. The existing research works overlap data transfer and kernel execution by partitioning the data into chunks and invoke the kernel to process the previously transferred chunk while the next chunk is being transferred. This approach requires the application developer to divide the input data into chunks of suitable size, call the kernel multiple times, and merge the results produced by each kernel invocation. Future researches are needed to develop runtime systems that can automatically handle the above issues.
- Currently available compiler frameworks that enable portability of GPU code to multicore CPU target does not make use of the SSE (Streaming SIMD Extension) support provided by the modern CPUs. Also the translation of multithreaded CPU code to GPU target and mapping of pointers from CPU address space to GPU is not achieved effectively.
- The GPUs have higher performance per Watt compared to CPUs. However the power consumption in large computing systems involving GPUs is a major concern. In such systems, even a small percentage of power saved per watt consumed can cause huge quantity of reduction in overall power consumption. Also power is the major constraint in mobile devices. Mobile devices have become ubiquitous and GPUs have become integral part of such devices providing high performance for graphics applications. Hence energy saving in both large computing systems and mobile platforms demands for significant research efforts.
- The task allocation algorithms aid in utilizing CPU and GPU computing resources efficiently thereby increasing the throughput and execution speedup. Reduced execution time means reduced energy consumption. There are techniques exclusively meant for energy saving such as dynamically scaling the frequency and voltage of CPU and GPU. More research is necessary towards the process of using task allocation algorithms and frequency/voltage scaling techniques together. Further these techniques can be benefited by the usage of artificial neural networks and machine learning algorithms to dynamically determine the kind of a task (CPU or GPU) and the suitable voltage or frequency for CPU and GPU.
- The elimination of PCI-e data transfer overhead encourages the use of coupled CPU-GPU architectures for the implementation of applications with high communication requirements. Moreover, the integrated GPUs also provide best power-performance ratio compared to the discrete GPUs. However the tighter integration of CPU and GPU in a coupled architecture imposes new challenges such as mapping the kernels to suitable processing cores, finding the optimal data partitions for load balancing, finding the optimal job co-schedules considering the memory contentions and power cap etc. Hence the future researches need to develop programming models that can address the above challenges.

4. Conclusion

There have been several research works aiming at achieving different objectives such as execution speedup, minimizing the communication overhead, reducing power consumption, enabling portability of applications etc. in CPU-GPU heterogeneous computing systems. In this survey paper we have presented a study of techniques that achieve the above objectives by utilizing the CPU and GPU cooperatively. We have presented our perspectives for future research in the area of cooperative CPU-GPU heterogeneous computing. We hope that this paper can benefit the future researchers in this area.

References

[1] C. Lee, W.W. Ro, J.L. Gaudiot, Boosting CUDA applications with CPU-GPU hybrid computing, Int. J. Parallel Program. 42 (2) (2014) 384–404.

[2] P. Pandit, R. Govindarajan, Fluidic kernels: cooperative execution of OpenCL programs on multiple heterogeneous devices, Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (2014) 273.

[3] J. Lee, M. Samadi, Y. Park, S. Mahlke, SKMD: single kernel on multiple devices for transparent CPU-GPU collaboration, ACM Trans. Comput. Syst. (TOCS) 33 (3) (2015) 9.

[4] X. Piao, C. Kim, Y. Oh, H. Li, J. Kim, H. Kim, J.W. Lee, JAWS: a JavaScript framework for adaptive CPU-GPU work sharing, ACM SIGPLAN Notices 50 (8) (2015) 251–252.

[5] S. Tomov, R. Nath, H. Ltaief, J. Dongarra, Dense linear algebra solvers for multicore with GPU accelerators, 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW) (2010) 1–8.

[6] J.I. Agulleiro, F. Vazquez, E.M. Garzon, J.J. Fernandez, Hybrid computing: CPU+GPU co-processing and its application to tomographic reconstruction, J. Ultramicroscopy 115 (2012) 109–114.

[7] F. Song, S. Tomov, J. Dongarra, Enabling and scaling matrix computations on heterogeneous multi-core and multi-GPU systems, Proceedings of the 26th ACM International Conference on Supercomputing (2012) 365–376.

[8] M. Xu, F. Chen, X. Liu, W. Ge, J. Li, Discrete particle simulation of gas-solid two-phase flows with multi-scale CPU-GPU hybrid computation, Chem. Eng. J. 207 (2012) 746–757.

[9] G. Teodoro, T. Pan, T.M. Kurc, J. Kong, L.A. Cooper, J.H. Saltz, Efficient irregular wavefront propagation algorithms on hybrid CPU–GPU machines, Parallel Comput. 39 (4) (2013) 189–211.

[10] M. Papadrakakis, G. Stavroulakis, A. Karatarakis, A new era in scientific computing: domain decomposition methods in hybrid CPU–GPU architectures, Comput. Methods Appl. Mech. Eng. 200 (13) (2011) 1490–1508.

[11] I. Chakroun, N. Melab, M. Mezmaz, D. Tuyttens, Combining multi-core and GPU computing for solving combinatorial optimization problems, J. Parallel Distrib. Comput. 73 (12) (2013) 1563–1577.

[12] R.B. Chen, Y.M. Tsai, W. Wang, Adaptive block size for dense QR factorization in hybrid CPU-GPU systems via statistical modeling, Parallel Comput. 40 (5) (2014) 70–85.

[13] F. Zhang, C. Hu, P.C. Wu, H. Zhang, M.D. Wong, Accelerating aerial image simulation using improved CPU/GPU collaborative computing, Comput. Electr. Eng. 46 (2015) 176–189.

[14] L. Wan, K. Li, J. Liu, K. Li, Efficient CPU-GPU cooperative computing for solving the subset-sum problem, Concurr. Comput. Pract. Exp. 28 (2) (2016) 492–516.

[15] Y. Yao, B.H. Ge, X. Shen, Y.G. Wang, R.C. Yu, STEM image simulation with hybrid CPU/GPU programming, Ultramicroscopy 166 (2016) 1–8.

[16] J. Liu, N. Hegde, M. Kulkarni, Hybrid CPU-GPU scheduling and execution of tree traversals, Proceedings of the 2016 International Conference on Supercomputing (2016) 2.

[17] N. Antoniadis, A. Sifaleras, A hybrid CPU-GPU parallelization scheme of variable neighborhood search for inventory optimization problems, Electron. Notes Discret. Math. 58 (2017) 47–54.

[18] F. Wende, F. Cordes, T. Steinke, On improving the performance of multi-threaded CUDA applications with concurrent kernel execution by kernel reordering, 2012 Symposium on Application Accelerators in High Performance Computing (SAAHPC) (2012) 74–83.

[19] J. Auerbach, D.F. Bacon, I. Burcea, P. Cheng, S.J. Fink, R. Rabbah, S. Shukla, A compiler and runtime for heterogeneous computing, Proceedings of the 49th Annual Design Automation Conference (2012) 271–276.

[20] M.P. Robson, R. Buch, L.V. Kale, Runtime coordinated heterogeneous tasks in charm++, Proceedings of the Second International Workshop on Extreme Scale Programming Models and Middleware (2016) 40–43.

[21] W. Huang, L. Yu, M. Ye, T. Chen, T. Hu, A CPU-GPGPU scheduler based on data transmission bandwidth of workload, 2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT) (2012) 610–613.

[22] M. Boyer, J. Meng, K. Kumaran, Improving GPU performance prediction with data transfer modeling, 2013 IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW) (2013) 1097–1106.

[23] R. Mokhtari, M. Stumm, BigKernel–high performance CPU-GPU communication pipelining for big data-style applications, 2014 IEEE 28th International Parallel and Distributed Processing Symposium (2014) 819–828.

[24] N.V. Sunitha, K. Raju, N.N. Chiplunkar, Performance improvement of CUDA applications by reducing CPU-GPU data transfer overhead, 2017 International Conference on Inventive Communication and Computational Technologies (ICICCT) (2017) 211–215.

[25] A.J. Lázaro-Muñoz, J.M. Gonzalez-Linares, J. Gomez-Luna, N. Guil, A tasks reordering model to reduce transfers overhead on GPUs, J. Parallel Distrib. Comput. 109 (2017) 258–271.
[26] J.A. Stratton, S.S. Stone, W.H. Wen-mei, MCUDA: an efficient implementation of CUDA kernels for multi-core CPUs, LCPC 2008 (2008) 16–30.
[27] G.F. Diamos, S. Yalamanchili, Harmony: an execution model and runtime for heterogeneous many core systems, Proceedings of the 17th International Symposium on High Performance Distributed Computing (2008) 197–200.
[28] A. Papakonstantinou, K. Gururaj, J.A. Stratton, D. Chen, J. Cong, W.M.W. Hwu, FCUDA: enabling efficient compilation of CUDA kernels onto FPGAs, 2009 IEEE 7th Symposium on Application Specific Processors, SASP'09 (2009) 35–42.
[29] G.F. Diamos, A.R. Kerr, S. Yalamanchili, N. Clark, Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems, Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (2010) 353–364.
[30] J. Gummaraju, L. Morichetti, M. Houston, B. Sander, B.R. Gaster, B. Zheng, Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors, Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (2010) 205–216.
[31] C. Hong, D. Chen, W. Chen, W. Zheng, H. Lin, MapCG: writing parallel program portable between CPU and GPU, Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (2010) 217–226.
[32] C. Augonnet, S. Thibault, R. Namyst, P.A. Wacrenier, StarPU: a unified platform for task scheduling on heterogeneous multicore architectures, Concurr. Comput. Pract. Exp. 23 (2) (2011) 187–198.
[33] L. Wang, Y.Z. Huang, X. Chen, C.Y. Zhang, Task scheduling of parallel processing in CPU-GPU collaborative environment, International Conference on Computer Science and Information Technology, 2008. ICCSIT'08 (2008) 228–232.
[34] C.K. Luk, S. Hong, H. Kim, Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping, Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (2009) 45–55.
[35] V.J. Jiménez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, N. Navarro, Predictive runtime code scheduling for heterogeneous architectures, HiPEAC 9 (2009) 19–33.
[36] C. Gregg, J. Dorn, K.M. Hazelwood, K. Skadron, Fine-grained Resource sharing for concurrent GPGPU kernels, Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism (HotPar'12) (2012).
[37] Z. Zhong, V. Rychkov, A. Lastovetsky, Data partitioning on heterogeneous multicore and multi-GPU systems using functional performance models of data-parallel applications, 2012 IEEE International Conference on Cluster Computing (CLUSTER) (2012) 191–199.
[38] E. Sun, D. Schaa, R. Bagley, N. Rubin, D. Kaeli, Enabling task-level scheduling on heterogeneous platforms, Proceedings of the 5th Annual Workshop on General Purpose Processing With Graphics Processing Units (2012) 84–93.
[39] I. Grasso, K. Kofler, B. Cosenza, T. Fahringer, Automatic problem size sensitive task partitioning on heterogeneous parallel systems, ACM SIGPLAN Notices 48 (8) (2013) 281–282.
[40] J. Zhong, B. He, Kernelet: high-throughput GPU kernel executions with dynamic slicing and scheduling, IEEE Trans. Parallel Distrib. Syst. 25 (6) (2014) 1522–1532.
[41] D. Yao, G. Zeng, C. Ding, Partition strategies for C source programs to support CPU+GPU coordination computing, International Conference on Information Science and Cloud Computing (2013) 39–48.
[42] R.M. Aciu, H. Ciocarlie, Algorithm for cooperative CPU-GPU computing, 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC) (2013) 352–358.
[43] Y. Wen, Z. Wang, M.F. O'Boyle, Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms, 21st International Conference on High Performance Computing (HiPC) (2014) 1–10.
[44] T. Li, V.K. Narayana, T. El-Ghazawi, Symbiotic scheduling of concurrent GPU kernels for performance and energy optimizations, Proceedings of the 11th ACM Conference on Computing Frontiers (2014) 36.
[45] A. Vilches, R. Asenjo, A. Navarro, F. Corbera, R. Gran, M. GarzarÃ¡n, Adaptive partitioning for irregular applications on heterogeneous CPU-GPU chips, Procedia Comput. Sci. 51 (2015) 140–149.
[46] Y. Wang, J. Qiao, S. Lin, T. Zhao, Performance Optimization for CPU-GPU Heterogeneous Parallel System, 2016 12th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD) (2016) 1259–1266.
[47] B. Wang, R. Ma, Z. Qi, J. Yao, H. Guan, A user mode CPU-GPU scheduling framework for hybrid workloads, Future Gener. Comput. Syst. 63 (2016) 25–36.
[48] G. Wang, X. Ren, Power-efficient work distribution method for CPU-GPU heterogeneous system, 2010 International Symposium on Parallel and Distributed Processing With Applications (ISPA) (2010) 122–129.
[49] R. Ge, X. Feng, M. Burtscher, Z. Zong, PEACH: a model for performance and energy aware cooperative hybrid computing, Proceedings of the 11th ACM Conference on Computing Frontiers (2014) 24.
[50] J. Lang, G. Runger, An execution time and energy model for an energy-aware execution of a conjugate gradient method with CPU/GPU collaboration, J. Parallel Distrib. Comput. 74 (9) (2014) 2884–2897.
[51] K. Ma, Y. Bai, X. Wang, W. Chen, X. Li, Energy conservation for GPU-CPU architectures with dynamic workload division and frequency scaling, Sustain. Comput. Inform. Syst. 12 (2016) 21–33.
[52] K. Siehl, X. Zhao, Power-aware heterogeneous computing through CPU-GPU hybridization, Energy 20 (40) (2016) 60.
[53] V. Chau, X. Chu, H. Liu, Y.W. Leung, Energy efficient job scheduling with DVFS for CPU-GPU heterogeneous systems, Proceedings of the Eighth International Conference on Future Energy Systems (2017) 1–11.
[54] F. Gong, L. Ju, D. Zhang, M. Zhao, Z. Jia, Cooperative DVFS for energy-efficient HEVC decoding on embedded CPU-GPU architecture, Proceedings of the 54th Annual Design Automation Conference (2017) 42.
[55] L.V. Kale, S. Krishnan, CHARM++: a portable concurrent object oriented system based on C++, ACM Sigplan Notices 28 (10) (1993) 91–108.
[56] C. Lattner, V. Adve, LLVM: a compilation framework for lifelong program analysis & transformation, Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (2004) 75.
[57] E. Stafford, B. Pérez, J.L. Bosque, R. Beivide, M. Valero, To distribute or not to distribute: the question of load balancing for performance or energy, European Conference on Parallel Processing (2017) 710–722.
[58] M. Daga, A.M. Aji, W.C. Feng, On the efficacy of a fused CPU+ GPU processor (or APU) for parallel computing, 2011 Symposium on Application Accelerators in High-Performance Computing (SAAHPC) (2011) 141–149.
[59] K. Lee, H. Lin, W.C. Feng, Performance characterization of data-intensive kernels on AMD fusion architectures, Computer Science-Research and Development 28 (2-3) (2013) 175–184.
[60] K.L. Spafford, J.S. Meredith, S. Lee, D. Li, P.C. Roth, J.S. Vetter, The tradeoffs of fused memory hierarchies in heterogeneous computing architectures, Proceedings of the 9th Conference on Computing Frontiers (2012) 103–112.
[61] I. Said, P. Fortin, J.L. Lamotte, R. Dolbeau, H. Calandra, On the efficiency of the accelerated processing unit for scientific computing, Proceedings of the 24th High Performance Computing Symposium (2016) 25.
[62] M. Dashti, A. Fedorova, Analyzing memory management methods on integrated CPU-GPU systems, Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management (2017) 59–69.
[63] M. Daga, M. Nutter, Exploiting coarse-grained parallelism in b+ tree searches on an apu, High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion (2012) 240–247.
[64] J. Gu, M. Zhu, Z. Zhou, F. Zhang, Z. Lin, Q. Zhang, M. Breternitz, Implementation and evaluation of deep neural networks (DNN) on mainstream heterogeneous systems, Proceedings of 5th Asia-Pacific Workshop on Systems (2014) 12.
[65] R. Wyrzykowski, M. Woźniak, L. Kuczyński, Efficient execution of erasure codes on AMD APU architecture, International Conference on Parallel Processing and Applied Mathematics (2013) 613–621.
[66] M.C. Delorme, T.S. Abdelrahman, C. Zhao, Parallel radix sort on the AMD fusion accelerated processing unit, 2013 42nd International Conference on Parallel Processing (ICPP) (2013) 339–348.
[67] J. He, M. Lu, B. He, Revisiting co-processing for hash joins on the coupled CPU-GPU architecture, Proceedings of the VLDB Endowment 6 (10) (2013) 889–900.
[68] J. He, S. Zhang, B. He, In-cache query co-processing on coupled CPU-GPU architectures, Proceedings of the VLDB Endowment 8 (4) (2014) 329–340.
[69] P. Eberhart, I. Said, P. Fortin, H. Calandra, Hybrid strategy for stencil computations on the APU, Proceedings of the 1st International Workshop on High-Performance Stencil Computations (2014) 43–49.
[70] X. Cheng, B. He, C.T. Lau, Energy-efficient query processing on embedded CPU-GPU architectures, Proceedings of the 11th International Workshop on Data Management on New Hardware (2015) 10.
[71] F. Zhang, J. Zhai, B. He, S. Zhang, W. Chen, Understanding co-running behaviors on integrated CPU/GPU architectures, Ieee Trans. Parallel Distrib. Syst. 28 (3) (2017) 905–918.
[72] G. Lupescu, E.I. Sluşanschi, N. Tăpuş, Using the integrated GPU to improve CPU sort performance, 2017 46th International Conference on Parallel Processing Workshops (ICPPW) (2017) 39–44.
[73] F. Zhang, B. Wu, J. Zhai, B. He, W. Chen, FinePar: irregularity-aware fine-grained workload partitioning on integrated architectures, 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (2017) 27–38.
[74] Q. Zhu, B. Wu, X. Shen, L. Shen, Z. Wang, Co-run scheduling with power cap on integrated CPU-GPU systems, 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS) (2017) 967–977.
[75] J. Fang, H. Chen, J. Mao, Understanding data partition for applications on CPU-GPU integrated processors, International Conference on Mobile Ad-Hoc and Sensor Networks (2017) 426–434.
[76] S. Mittal, J.S. Vetter, A survey of CPU-GPU heterogeneous computing techniques, ACM Computing Surveys (CSUR) 47 (4) (2015) 69.
[77] Insieme compiler and runtime infrastructure. Distributed and Parallel Systems Group, 2012. University of Innsbruck. URL http://insieme-compiler.org.
[78] Web Worker. URL http://www.w3.org/TR/workers.
[79] WebCL Standard. URL www.khronos.org/webcl/.
[80] CUDA C Programming Guide, Version 8.0, Nvidia Corporation (2017). URL www.nvidia.com.
[81] OpenCL Programming User Guide, rev 1.0, Advanced Micro Devices, Inc. (2013). URL www.amd.com.
[82] OpenMP Application Program Interface, Version 4.0 (2013). URL www.openmp.org.