Activity-driven Task Allocation in Energy Constrained Heterogeneous GPUs Systems

Zhuowei Wang, Xiaoyu Song, *Senior Member*, *IEEE*, Lianglun Cheng, and Hao Wang, *Member*, *IEEE*

***Abstract*—As computing systems continue to increase in complexity, energy optimization plays a key role in the design and implementation of heterogeneous systems. Although the energy consumed by off-chip memory accounts for a large proportion of the total power consumed by the system as a whole, current research on energy optimization mainly focuses on optimizing the energy consumed by the processors. This paper explores the coordinated optimization of the holistic performance of the processors and memory system for heterogeneous systems with energy constraints. A communication–computing pipeline model for parallel executions is characterized to optimize program performance by simultaneously scaling the voltage and frequency of the processors and memory using task allocation strategies. A synergistic load-balancing optimization approach is presented to resolve the load imbalance among graphics processing units. Our experimental results substantiate the effectiveness of the approach in terms of execution times and throughputs with the energy constraints.**

***Index Terms*—Energy-constrained optimization, task assignments, load-balancing, multiple co-processors, heterogeneous system.**

I. Introduction

ACHEIVING high performance in heterogeneous hardware platforms is imposing new challenges to the design of computing systems. Conventionally, power optimization has been mainly focused on the average energy consumed by the system. Moreover, as the scale of high-performance computer systems has continued to increase, energy consumption has become one of the most important constraints affecting their design, operation, and management [1, 2]. In more recent times, research interest has focused on the problem of maximizing the execution performance of a system while not exceeding some predetermined energy constraints [3–5]. However, stringent power constraints are gradually starting to hinder system performance which affects their design and implementation.

Mainstream methods of optimization are aimed at optimizing the energy consumed by the processors of a heterogeneous system [6, 7]. Such methods involve the scaling of the voltage and frequency of the processors (mainly by inserting a voltage-scaling instruction into the code during its compilation). This helps designers realize their goal of reducing the energy consumption of the processors. In general, therefore, these methods involve establishing analysis models to delineate the voltage-setting and -scaling problems. Thereafter, the optimal solution is obtained using the solver for programming problems by virtue of the description methods appropriate to linear (or nonlinear) programming. For example, Hsu et al. [8] explored the possibility of stepping down the voltage in a partially-overlapping code segment and described the scaling optimization problem using a nonlinear programming approach. In this way, they managed to calculate the optimal solution. Using an integer linear programming method, Saputra et al. [9] determined the optimal voltage class required in each loop nesting. Xie et al. [10] studied the optimization of a dynamic voltage-scaling strategy during compilation and proposed a realistic model to comprehensively consider the characteristics of the programs and dynamic voltage-scaling characteristics.

However, the research mentioned above merely focuses on optimizing the energy consumed by the processors and ignores that associated with the off-chip memory. Research in more recent years has clearly demonstrated that the off-chip memory needs to be considered as well when optimizing energy consumption. For example, Fan et al. [11] showed that a better optimization result for the system as a whole can be achieved by incorporating memory with low power consumption into the energy optimization process. Indeed, the energy saved via voltage scaling alone (i.e. merely considering the processors) amounted to just 39%. On the other hand, that saved by simultaneously considering voltage scaling of the processors and use of power-aware memory reached 89%.

The main contributions of this work are as follows. First, we characterize the processor and memory as the objects of energy-constrained optimization. We propose a novel performance optimization method. Both the processors and memory are used in the energy optimization process are included. The method employs the parallel program OpenMP as the optimization object and simultaneously scales the voltage and frequency of the processors and memory using a software control method. The results correspond to the optimal system performance under the energy-constrained conditions. Second, we propose an energy-constrained modelling through analyzing the communication-computing pipeline. Depending on whether processor or memory idling occurs during the parallel execution of program tasks partitioned on different processors, program performance is optimized by

Zhuowei Wang is with the School of Computers, Guangdong University of Technology, Guangzhou 510006, China, and the Wuhan Donghu University, Wuhan 430074, China. E-mail: wangzhuowei0710@163.com.

Xiaoyu Song is with the Department of Electrical and Computer Engineering, Portland State University, Portland, OR 97207, USA. E-mail: songx@pdx.edu.

Lianglun Cheng is with the School of Computers, Guangdong University of Technology, Guangzhou 510006, China. E-mail: llcheng@gdut.edu.cn.

Hao Wang is with department of Computer Science, Norwegian University of Science and Technology, Gjøvik, Norway. E-mail: hawa@ntnu.no.

simultaneously scaling the voltage and frequency of the processors and memory. Third, we design an algorithm for synergistic energy-constrained of task assignment. Various conditions leading to a load imbalance among processors are analyzed theoretically in the heterogeneous multi-processor environment. Corresponding optimization methods to tackle load imbalance are provided.

The rest of the paper is organized as follows. Section 2 overviews work related to the current work. Section 3 presents the architecture of the target system and Section 4 introduces the task partitioning method used for energy-constrained optimization. Then, we provide a detailed description of energy-constrained modeling process in Section 5. Section 6 gives the algorithm used for the synergistic energy-constrained optimization process used in task assignment process and Section 7 analyzes the experimental results. Finally, Section 8 presents our conclusions.

II. Related work

In recent years, as the peak power consumed by processors has increased, progressively more research has focused on how to optimize program performance under given maximum power constraints [12-13]. Annavaram et al. [14] were the first to consider the problem of optimizing the performance of multi-core processors with maximum power constraints and hence formulated the EPI throttling maximum power management method.

Isci et al. [15] described a prediction-based management method based on MaxBIPS. In their work, by determining the power consumed by concurrent tasks at different operating frequencies of the processors, the frequency combination with the highest system throughput could be selected. Furthermore, the power constraints could be adapted by reducing the operating frequency of the processors. Sartori et al. [16] proposed a hierarchical method of power control based on optimal ascent. This involved dividing the multi-core processors into groups and adopting a two-level independent power-control mechanism within and between groups. As a result, the control complexity could be effectively reduced. Meng et al. [17] also presented a power management method which combined regulating the frequency of the processor core and adjusting the shared cache space. This allowed program execution performance to be optimized by combining two management strategies subject to the power constraints. Ma et al. [18] considered the criticality of different threads in a multi-threaded program oriented towards a multi-program execution environment. Based on existing methods of evaluating thread criticality, power could be preferentially allocated to processor cores with higher criticality for optimization purposes. Overall, there is an improvement in program execution performance. Cebri et al. [19] proposed a power-allocation strategy based on power tokens. In this case, power is preferentially allocated to processor cores with higher power requirements to improve parallel execution performance.

In the abovementioned research, performance optimization and power management were mainly proposed based on the constraints placed on energy consumption. The performance optimization problems were not modeled and analyzed subject to the constraints placed on energy consumption. Compared with general-purpose microprocessors, acceleration components such as graphics processing units (GPUs), field-programmable gate arrays (FPGAs), and many-integrated core (MIC) architectures have higher integration and operating power consumptions High-performance computer systems based on accelerated processors therefore generate larger amounts of heat per unit volume. This, of course, raises the energy requirements of the power supply and cooling systems. The maximum power consumed by the control system will become one of the most important issues for the power management of heterogeneous parallel systems in the future. Therefore, modeling the performance optimization process subject to limited energy consumption is a necessary and important step for low power design. The biggest difference between this paper and the abovementioned research is that it establishes an optimization model for the performance of a heterogeneous system under the condition of limited energy consumption.

The continuous improvement in transistor and system structure technology has led to processors entering a multi-core era. However, although the emergence of multi-core processors has improved computing speed, it has also widened the speed gap between processor and memory, leading to serious ‘memory wall’ problems [20, 21]. In recent years, optimizing the power consumption of storage systems has received more and more attention from researchers. Some researchers have pointed out that the effect of combining the optimization of the memory and processor at the same time is much better than that of just optimizing the processor [22]. With the progress made in architecture technology, energy-aware storage systems have emerged (e.g. configurable register files, partially closed caches, and memory with independent control memory) and there have been a large number of studies on the optimization of the frontal power consumption of storage systems [23, 24, 25]. A load-balancing method has also been proposed to coordinate the energy-constrained optimization of the task assignment process [26].

However, in the research work carried out on low-power optimization, the main focus has been the energy optimization of the processor. Little research has focused on the simultaneous energy optimization of the processor and off-chip memory. Moreover, the energy consumed by the off-chip memory accounts for a large proportion of the total energy consumed by the system and so it is of great significance in the energy optimization process. The work in [11] discusses the influence of different memory control strategies on processor frequency setting decision-making, to achieve the purpose of reducing system energy consumption. In this paper, the frequency regulation of the processor and memory is asynchronous. Based on the characteristics of application program execution, we determine the optimal processor and memory frequencies by analyzing the distribution of the communication computing pipeline, thus achieving the optimal performance under the condition of satisfying energy consumption constraints. In our previous research [25-26], the target system was to include the same computing capability of homogeneous multi GPUs. In such a system, due to the same type of GPUs, the task partition granularity was the same.

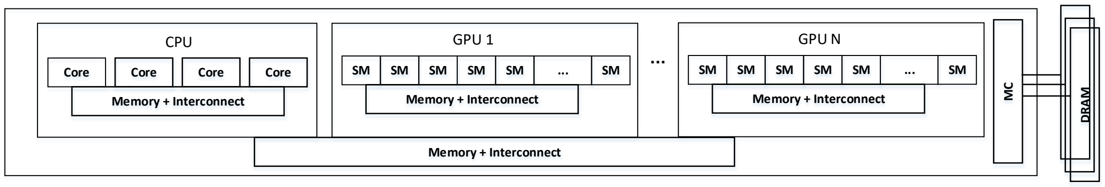

Fig. 1. Schematic diagram of a heterogeneous processor with integrated GPUs on a single chip.

Each GPU completes all tasks in the same or similar time, and there was no load imbalance. In this paper, our target system is to include the heterogeneous multi GPUs with different computing capabilities. Therefore, from the perspective of task partition, the task allocation of GPUs with different computing capabilities should be different, which will lead to the waste of power consumption due to the unbalanced load. This paper analyses the conditions of load imbalance between GPUs and proposes the load balance optimization method.

III. Target System Architecture

Fig. 1 shows a heterogeneous system architecture that includes a CPU and multiple GPUs on one chip. Both the CPU and GPU have their own memories. When the program is running, the CPU can send Direct Memory Access (DMA) commands and a DMA engine completes the data transfer between the main and GPU memories. Each GPU shares the main memory, so the CPU is only allowed to transmit data to one GPU at a time. The multiple GPUs can run independently. In this architecture, the CPU and GPUs share the same physical memory and memory controller. Therefore, the process of transferring data from the main CPU processor to a certain GPU is actually realized through memory.

To develop processor parallelism among multiple GPUs, the thread space of the kernel function can usually be divided and different subspaces are formed into different sub-kernels. These can then be assigned to different GPUs for simultaneous execution. Each sub-kernel needs the main processor to transmit the corresponding input data it requires, so the data communication process is serial. Furthermore, the calculations occurring in different GPUs can overlap. At the same time, as the DMA component is executed by a processor that is independent of the GPUs, the transmission of the next block of data and the calculation involving the previous block of data can also overlap within the same GPU.

GPUs generally contain multiple stream multi-processors (SMs). In order to make full use of the parallel processing performance of a single GPU, the tasks should be partitioned so that they have a basic granularity to ensure that each SM on each of the participating GPUs is assigned at least one thread block.

The application SWIM can be used as an example to illustrate the basic granularity of tasks. The SWIM program consists of three core processes, namely, CALC1, CALC2, and CALC3. We call these three core processes ‘three tasks’, each of which consists of a double parallel loop, and its iteration space is 2048×2048. We divide the original iteration space with a subspace of size 32×2048 as the granularity, and each task has 65 partitioning strategies (2048/32+1). Assume that each GPU contains 4 SMs. We set the size of each thread block on the GPU to 8×2048. When the number of thread blocks allocated to the GPU is not an integer multiple of the number of SMs, a load imbalance will occur, and some SMs will become idle. Therefore, during the execution of this application, the basic division granularity of tasks is $k = 4 \times (8 \times 2048)$. If there are more computing tasks $r \cdot k$ $(r = 1,2, \dots ,65)$ divided for a GPU than the basic division unit k, that is, when each SM can be divided into more than one thread block, the computational density inside the SM can be improved to certain extent.

In this paper, the actual platform and simulation platform are used for experimental verification. The actual platform is a heterogeneous system consisting of an Intel i7 920 quad-core CPU and 2*AMD 4870 GPU. The simulation platform is a GPU power simulator based on GPGPU-Sim. The programming model of the actual platform is OpenMP + brook+. The programming model of the simulation platform is GPGPUSim + nvcc.

IV. Task Partitioning for Energy-Constrained Optimization

Our objective is to optimize the performance of a heterogeneous system using processor–memory coordination subject to energy constraints. Our method involves task partitioning and frequency scaling and is capable of producing satisfactory execution times and processor profits while satisfying the given energy constraints. The task division process proposed in this paper uses thread blocks as the basic unit for task division. The energy-constrained model target communication is between inter-GPU tasks.

We need to define the following parameters in the energy-constrained optimization problem. k refers to the basic task partition unit. $r \cdot k$ denotes that the computation tasks partitioned to a co-processor are larger than the basic task partition unit k. μ denotes the number of cyclic iterations of the parallel program OpenMP. f_c refers to the processor frequency. f_m refers to the memory frequency. $comp(k)$ represents the amount of calculation required to complete the calculation task k. $Data(k)$ corresponds to the amount of data that the CPU needs to transmit to the GPU in order to complete the calculation task k. $C_c(comp(k))$ denotes the number of processor clock cycles taken to complete the calculation task k. $C_c(comp(r \cdot k))$ denotes the number of processor clock cycles taken to complete the calculation task $r \cdot k$. $C_m(Data(k))$ denotes the number of data access delay clock cycles generated by the transmission of the data amount $Data(k)$ in order to complete the calculation task k.

We note that there may be various relationships between the amount of calculation and amount of data required for the calculation. Take, for example, matrix multiplication: $\boldsymbol{C} = \boldsymbol{A} \times \boldsymbol{B}$, each assumed to be $M \times M$ in dimension. Suppose we choose to divide the B and C matrices and keep the A matrix intact in each GPU. The basic division unit k is also chosen as a column for calculating the C matrix. Therefore, the amount of calculation $comp(k)$ is equal to $M \times (M + (M-1))$ at this time, and the amount of data to be transmitted for the calculation task $Data(k)$ is equal to $M \times M + M$. If the calculation task assigned to a GPU is doubled, e.g. when calculating two columns of the C matrix, the calculation amount becomes $comp(2k) = 2M \times (M + (M-1))$, and the required data amount becomes $Data(2k) = M \times M + 2M$. Thus, in this case, the calculation space is three-dimensional, and the storage space is two-dimensional. Therefore, the calculation volume increases faster than the data transmission volume.

The amount of data transmitted does not depend linearly on the number of tasks. Without loss of generality, we relate the $Data(r \cdot k)$ required in order to complete the task $r \cdot k$ to $Data(k)$ via the function R:

$$Data(r \cdot k) = R\big(r, Data(k)\big) \qquad (1)$$

Accordingly, $C_m(R\big(r, Data(k)\big)$ denotes the number of clock cycles of memory access latency caused by transmitting data of size $Data(r \cdot k)$.

In OpenMP parallel programs, parallel loops are divided and mapped on heterogeneous processors for execution. As there are no data dependencies between iterations of parallel loops, each loop iteration can be mapped to multiple processors for parallel execution. When scheduling parallel loops (irrespective of whether the loop is nested or not), we only parallelize the outermost loop iterations and call these outermost loop iterations scheduling units.

Assuming that there are N different types of GPU in the system, the iteration space of *doall* is divided according to the task $r \cdot k$ (when a GPU divides more tasks than the basic unit k) and map it to a heterogeneous processor, recorded as:

$$F_{doall} = < r_1 \cdot k_1, r_2 \cdot k_2, \cdots, r_N \cdot k_N >$$

where $r_i \cdot k_i$ $(1 \le i \le N)$ represents a subset of the doall loop iteration set and is mapped to the i-th processor. We also use μ_i to represent the size of the subset $r_i \cdot k_i$, which is equal to the number of iterations.

A practical heterogeneous system consists of multiple heterogeneous co-processors. The heterogeneity is reflected in different ways: the task partition granularity, communication performance, and computation performance. We discuss each of these in turn:

(i) Task partition granularity – Task assignments to GPUs of different computing capability should be different. As already indicated, to avoid load imbalance in the SMs of a GPU, we define the basic granularity of the task division (k) to be the number of tasks when each SM in the GPU is assigned to a thread block. Therefore, different types of GPU will have different basic task division granularities. Assuming that there are N different types of GPU in the system, the granularity of the basic task division of these GPUs can be expressed in the form $k_1, k_2, \cdots, k_N$. When more tasks are assigned to each SM in the GPU than the basic granularity k, the granularity of tasks divided by the N different types of GPU can be similarly expressed in the form $r_1 \cdot k_1, r_2 \cdot k_2, \cdots, r_N \cdot k_\mathrm{N}$. $T(r_1, r_2, \cdots r_N)$ is used to denote the program execution time when the N GPUs use $r_1, r_2, \cdots, r_N$ as the granularity of the task division process.

(ii) Communication performance – As the off-chip DRAM chips used in different types of co-processor probably have different configurations, transmission times can differ for a given data size. The number of clock cycles required for communication using the i-th type of co-processor is given by $C_m(R(r_i, Data(k_i)))$.

(iii) Computation performance – Apart from their computation capacities, different types of co-processor may also have different on-chip memory hierarchies. Thus, the manner in which the computation time increases with task load will also be different. The number of clock cycles required for computation using the i-th type of co-processor is given by $C_c(comp(r_i \cdot k_i))$ when it is assigned r_i basic tasks.

The communication–computation time–space diagram needs to be analyzed taking into account the energy constraints. Two types of operation are involved: data communication and computation. We define a new function, RIO, corresponding to the ratio of the computation time to the communication time, i.e.

$$RIO_i(r_i, k_i) = \frac{C_c(comp(r_i \cdot k_i))/f_c^i}{C_m(R(r_i, Data(k_i))/f_m^i}$$

This ratio is the basic factor determining the distribution of the spatio-temporal diagram.

Different co-processors will generally refer to GPUs with different computing capabilities, that is, different types of GPU. Fig. 2 presents three spatial-temporal diagrams. In each case, the vertical axis in the pipeline represents the different heterogeneous GPUs and the horizontal axis represents execution time. A dotted box indicates a process in which the CPU processor is transferring data to the memory of a GPU. A solid box represents the time it takes for a GPU to execute a task. Clearly, the internal calculation parts in any given line in a spatial-temporal diagram cannot overlap. However, calculations in different lines, that is, in different GPUs, can overlap. As each GPU shares the same main memory, only the main processor is allowed to transmit data to a certain GPU at any given time. Therefore, memory access delays caused by data transmission cannot overlap in the entire pipeline. At the same time, the transmission of the next data block and calculation using the previous data block can also overlap within the same GPU as the DMA component is executed independently in the GPU.

Based on these basic construction rules, the total task amount of the program is assumed to be F and that the granularities of the task division process in the N GPUs are $r_1, r_2, \cdots, r_N$. Also, considering the need for load balancing, the same type of GPU is assumed to use the same task granularity. However, different types of GPU will use different task granularities and so three different pipeline distribution situations may occur.

As an example, consider the situation in which there are two different types of GPU (Fig. 2). In this case, the two GPUs have task granularities of r_1 and r_2. The task computing times of the two GPUs under the two task division granularities are therefore $C_c(comp(r_1 \cdot k_1))/{f_c}^1$ and $C_c(comp(r_2 \cdot k_2))/{f_c}^2$ where ${f_c}^1$ and ${f_c}^2$ are the core frequencies of the two GPUs. Similarly, $C_m(R(r_1, Data(k_1)))/f_m^1$ and $C_m(R(r_2, Data(k_2)))/f_m^2$ are the two communication times required by the GPUs to transfer the amount of data they require to complete the computation tasks when the task division granularities are r_1 and r_2. (This is the fetch latency and $f_m{}^1$ and $f_m{}^2$ are the fetch frequencies of the two GPUs). The corresponding spatio-temporal diagrams for the communication-computation pipelines are shown in Fig. 2.

In the case shown in Fig. 2(a), the task computing times of each co-processor are less than the sum of the task transferring times of the two co-processors. That is,

$$\begin{cases} \frac{C_c(comp(r_1\cdot k_1))}{f_c{}^1} < \frac{C_m(R(r_1,Data(k_1)))}{f_m{}^1} + \frac{C_m(R(r_2,Data(k_2)))}{f_m{}^2} \\ \frac{C_c(comp(r_2\cdot k_2))}{f_c{}^2} < \frac{C_m(R(r_1,Data(k_1)))}{f_m{}^1} + \frac{C_m(R(r_2,Data(k_2)))}{f_m{}^2} \end{cases} \tag{2}$$

Under these conditions, the total execution time of the program is determined by the memory access delay time (i.e. it is Communication-intensive), that is,

$$T_a(r_1,r_2) = \frac{F}{r_1\cdot k_1+r_2\cdot k_2}\sum_{i=1}^{2}\frac{C_m\big(R(r_i,Data(k_i))\big)}{f_m{}^i} + \varepsilon \tag{3}$$

where ε represents the time taken up by the final part of the computation (as the pipeline is evacuated). However, as long as the task load is large enough, this part of the computation time can be ignored in comparison to the overall time. The processor core is not a critical path restricting the performance of the program. Therefore, a reduction in processor frequency can save energy. The processor frequency can be adjusted until the boundary conditions shown in Eq. (2) are reached. However, the situation becomes computing-intensive if the processor frequency is further reduced. At this time, there will be a need to adjust the frequency of the memory in order to obtain the optimal energy result and best performance subject to the condition of limited energy consumption. However, as long as Eq. (2) is still satisfied, the computing time of each GPU is hidden by the memory access delay. This guarantees partial energy saving without performance loss. Under these conditions, the different types of co-processor do not show load imbalance. This is because the execution time of each co-processor is limited by the data communication latency. The difference in the computation capacities of these co-processors is hidden in the waiting time.

Now consider the case shown in Fig. 2(b). In this case, the data computing time of each co-processor is greater than the sum of the data transferring times of the two co-processors:

$$\begin{cases} \frac{C_c(comp(r_1,k_1))}{f_c{}^1} \geq \frac{C_m(R(r_1,Data(k_1)))}{f_m{}^1} + \frac{C_m(R(r_2,Data(k_2)))}{f_m{}^2} \\ \frac{C_c(comp(r_2,k_2))}{f_c{}^2} \geq \frac{C_m(R(r_1,Data(k_1)))}{f_m{}^1} + \frac{C_m(R(r_2,Data(k_2)))}{f_m{}^2} \end{cases} \tag{4}$$

Under these conditions, all the task communications are hidden by the computation tasks, except for the construction period of the pipeline. Due to the differences in performance of different types of co-processor this will probably lead to load imbalance. The overall execution time of the program is determined by the co-processor with the longest execution time, that is:

$$T_b(r_1,r_2) = \max_{i=1}^{2}\left\{\frac{F}{r_1\cdot k_1+r_2\cdot k_2}\frac{C_c(comp(r_i,k_i))}{f_c{}^i}\right\} + \varepsilon \tag{5}$$

where ε now denotes the data communication time in the initial period during the construction of the pipeline. As before, this time can be ignored in comparison with the overall time, as long as the task load is large enough. In this case, the total execution time of the program is determined by the calculation time (i.e. it is Computing-intensive). In other words, the fetch time is not a critical path that restricts the performance of the program. We can therefore reduce the memory frequency to obtain a reduction in energy consumption. However, the frequency reduction must be appropriate: one cannot continue to reduce the memory frequency because eventually a memory-intensive situation will arise. That is, one cannot achieve further energy reduction without performance loss. At this time, however, decreasing the frequency of the processor by an appropriate amount can further reduce energy consumption. It is clear that the frequencies of both components need to be adjusted to obtain the optimal performance result subject to the condition of limited energy consumption. However, as long as the conditions shown in Eq. (4) are still satisfied, the memory access latency of each GPU will be hidden by the calculation time. Therefore, it is guaranteed that partial energy savings can be obtained without performance loss.

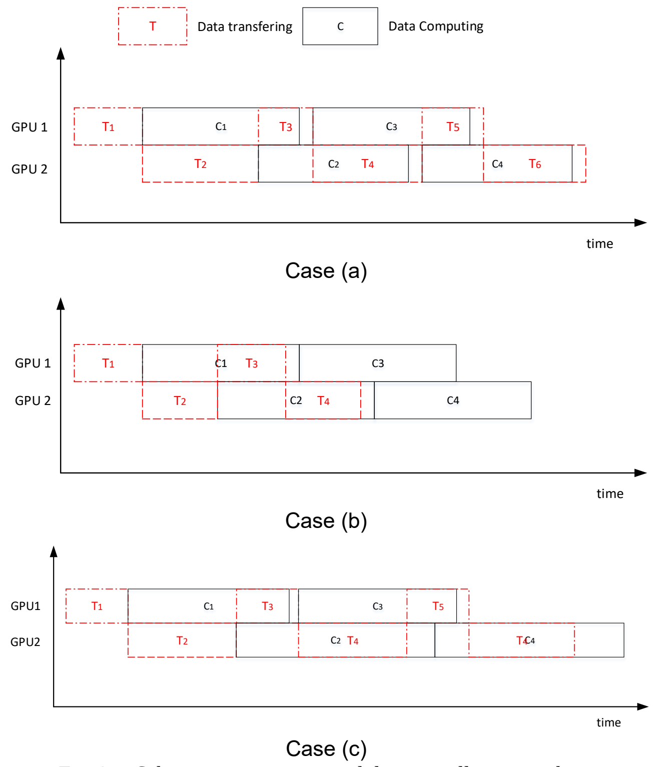

Fig. 2. Schematic spatio-temporal diagrams illustrating three communication–computing pipeline cases.

Finally, the situation is considered, as shown in Fig. 2(c). In this case, the task computing time of one co-processor is less than or greater than the sum of the task transmission times of the two co-processors:

$$\begin{cases} \frac{C_c(comp(r_1,k_1))}{f_c{}^1} < \frac{C_m(R(r_1,Data(k_1)))}{f_m{}^1} + \frac{C_m(R(r_2,Data(k_2)))}{f_m{}^2} \\ \frac{C_c(comp(r_2,k_2))}{f_c{}^2} \geq \frac{C_m(R(r_1,Data(k_1)))}{f_m{}^1} + \frac{C_m(R(r_2,Data(k_2)))}{f_m{}^2} \end{cases} \tag{6}$$

This time, the time consumed by one computation task in the first co-processor cannot hide the total time required for data communication between itself and the second co-processor. Therefore, the execution time of the first co-processor is determined by the overall data communication time. For the second co-processor, (except for the construction period of the pipeline) the data communication latency is completely hidden by the computation time, so the execution time of the second co-processor depends on the overall computation time. Thus, it is clear that the execution time of the second co-processor is larger than that of the first. Under such conditions, the overall execution time of the program is

$$T_c(r_1,r_2)=\frac{F}{(r_1\cdot k_1+r_2\cdot k_2)}\cdot\frac{C_c(comp(r_2,k_2))}{{f_c}^2}+\varepsilon \qquad (7)$$

where ε now represents the data communication time in the initial period during the construction of the pipeline. As in case (b) above, as long as the task load is large enough, this part of the communication time can be ignored in comparison with the overall time. There is also an obvious load imbalance among the co-processors.

At this point, we will discuss the two GPUs separately. The execution time of the first GPU is determined by the total data fetch time. In other words, the data fetch time of the first GPU is a critical path that restricts the execution time of the program. In this case, we can appropriately reduce the processor core frequency and increase the processor calculation time. As long as the condition shown in Eq. (6) is still satisfied, the calculation time of the first GPU will be hidden by the fetch delay. Partial energy savings can thus be guaranteed for the first GPU without performance loss.

In contrast, the execution time of the second GPU is determined by the total computation time. In other words, the computation time of the second GPU is a critical path that restricts the execution time of the program. In this case, we can appropriately reduce the frequency of the second GPU memory and increase the fetch time. As long as the conditions shown in Eq. (6) are still satisfied, the fetch time of the second GPU can be hidden by the calculation time. Therefore, the second GPU can be guaranteed to have partial energy savings without performance loss.

Eq. (6) shows that the execution time of the second GPU is always greater than that of the first GPU. Therefore, the total execution time of the program will be determined by the execution time of the second GPU. So, we can also coordinate the processor core frequency and memory access frequency in case (c) in order to improve the performance of the program (provided the energy consumption does not exceed the given energy constraint).

Eqs. (3), (5), and (7) can be used to calculate the overall execution times of a program corresponding to the three communication–computing pipeline conditions shown in Fig. 2. They can also be extended to systems consisting of a larger number of heterogeneous co-processors. For case (c), for example, this can be realized by replacing the quotient $C_c(comp(r_2,k_2))/f_c^2$ appearing in Eq. (7) with the largest quotient satisfying the expression:

$$\frac{C_c(comp(r_i,k_i))}{{f_c}^i}\geq\sum_{i=1}^{N}\frac{C_m\left(R(r_i,Data(k_i))\right)}{{f_m}^i} \qquad (8)$$

It can also be seen that case (b) is actually a special example of case (c). Therefore, the program execution time of a system consisting of N co-processors can be represented by the following piecewise function:

$$\mathrm{T}(r_1,r_2,\cdots r_N)=\begin{cases}T_a(r_1,r_2,\cdots r_N)\ \forall i\leq N{:}\ cond\ 9.1\\ T_c(r_1,r_2,\cdots r_N)\ \forall i\leq N{:}\ cond\ 9.2\end{cases}$$
$$cond\ 9.1{:}\ \frac{C_c(comp(r_i,k_i))}{{f_c}^i}<\sum_{i=1}^{N}\frac{C_m\left(R(r_i,Data(k_i))\right)}{{f_m}^i} \qquad (9)$$
$$cond\ 9.2{:}\ \frac{C_c(comp(r_i,k_i))}{{f_c}^i}\geq\sum_{i=1}^{N}\frac{C_m\left(R(r_i,Data(k_i))\right)}{{f_m}^i}$$

V. Energy-Constrained modeling

The energy-constrained optimization problem can be succinctly described as follows: given an OpenMP parallel loop segment, L, and a heterogeneous system with dynamically scalable voltage and frequency, we seek the optimal frequencies of the processors (f_c) and memory components (f_m). In addition, an appropriate task partition, P, is sought with which the thread space of L can be divided into multiple subspaces that can be distributed to, and then executed by, the multiple heterogeneous processors. The ultimate goal is to realize given optimal performance objectives (minimization of the execution time or maximization of processor profit) during program execution subject to keeping the energy consumption less than a given energy budget, E_{budget}.

In this context, we clarify the meaning of certain variables:

Minimization of program execution time, T: the minimum time that elapses between the beginning of the program and the end of the program.

The actual time taken by processors, T^*: The working time of processors.

Processor utilization, U: the ratio of the actual time taken by processors for program execution to the overall time consumed by program execution. In general, $0\leq U\leq 1$.

Processor profit, G: the ratio of the optimized processor utilization, U_p, to the utilization before optimization, U_{np}.

To accurately describe performance optimization subject to energy constraints, we have the following legitimate assumptions relating to the program, architecture, and circuit implementation. (i) The logic behavior of programs does not change with frequency. (ii) The frequencies of the processors and memory are scaled asynchronously. That is, we suppose that the frequency scaling of the processors and memory are independent of each other and have their own scalable ranges. (iii) The frequencies of the processors and memory are continuously scalable. (iv) The time and energy consumed during frequency conversion are not taken into account.

The performance optimization objectives and energy constraints of the problem are elaborated in detail on the basis of these assumptions. The consumption of energy can be broadly divided into the consumption of ‘dynamic’ and ‘static’ energy. Static power consumption is related to the operating voltage and temperature of the chip and is ever-present during the operation of the processor. This decomposition means that the assignable power consumption range is the difference between the energy constraints placed on the system and the static power consumption. The phrase ‘maximum power consumption’ in this research therefore refers to the maximum dynamic power consumption, unless otherwise stated. The dynamic energy consumption, E_d, consists of the energies associated with computations performed by the processors, E_c, and that used for memory access, E_m, and so we write:

$$E_d=E_c+E_m \qquad (10)$$

For a task partition of granularity k, the power consumed by the processors operating at frequency f_c is given by $p_c(f_c)$ and the time consumed by the computation of the processors in one iteration is $C_c(k)$ clock cycles. Then, the energy consumed by the processors for computational purposes in μ iterations is

$$E_c = p_c(f_c) \cdot \frac{C_c(comp(k))}{f_c} \cdot \mu \tag{11}$$

Similarly, the energy consumed due to memory access in μ iterations is

$$E_m = p_m(f_m) \cdot \frac{C_m(Data(k))}{f_m} \cdot \mu \tag{12}$$

so that

$$E_d = p_c(f_c) \cdot \frac{C_c(comp(k))}{f_c} \cdot \mu + p_m(f_m) \cdot \frac{C_m(Data(k))}{f_m} \cdot \mu \tag{13}$$

In a CMOS circuit, the dynamic power consumption, p, is directly proportional to the clock frequency, f, and square of the voltage, V, that is:

$$p \propto \alpha_0 C V^2 f \tag{14}$$

where α_0 represents the switching activity factor and C is the switching capacitance. Furthermore, the voltage and clock frequency are related through the following expression [27]:

$$f \propto \frac{(V-V_t)^{\alpha}}{V} \tag{15}$$

where V_t is the threshold voltage and α is a technology-related factor. The latter is different for different systems but is generally found to be such that $\alpha \in [1,2]$. When α is 2, the frequency can be considered to be approximately directly proportional to the power supply voltage. In this case, the dynamic power consumption takes the (approximate) form

$$p \propto \alpha_0 C f^3 \tag{16}$$

The dynamic power consumed by the processors and memory can then be expressed in the form

$$p_c(f_c) = \alpha_1 \cdot C_1 \cdot f_c^3 \tag{17}$$

$$p_m(f_m) = \alpha_2 \cdot C_2 \cdot f_m^3 \tag{18}$$

Substituting these into Eq. (13) yields:

$$E_d = M_1 f_c^2 C_c(comp(k))\mu + M_2 f_m^2 C_m(Data(k))\mu \tag{19}$$

where $M_1 = \alpha_1 \cdot C_1$ and $M_2 = \alpha_2 \cdot C_2$.

Using Eq. (9), the first performance objective (minimizing the execution time) is formulated in the form:

$$\min T(r_1, r_2, \cdots r_N) = \begin{cases} \min T_a(r_1, r_2, \cdots r_N) & i = 1, \dots, N, cond\ 9.1 \\ \min T_c(r_1, r_2, \cdots r_N) & i = 1, \dots, N, cond\ 9.2 \end{cases} \tag{20}$$

Next, the second performance objective (maximizing the processor profit G) is formulated. Processor utilization can be expressed as

$$U_p(i) = \begin{cases} \frac{C_c(comp(r_i,k_i))/f_c{}^i}{T_a(r_1,r_2,\cdots r_N)} & i = 1, \dots, N, cond\ 9.1 \\ \frac{C_c(comp(r_i,k_i))/f_c{}^i}{T_c(r_1,r_2,\cdots r_N)} & i = 1, \dots, N,\ cond\ 9.2 \end{cases} \tag{21}$$

and the processor utilization U_{np} before optimization is

$$\forall i \in [1, N]: U_{np}(i) = \frac{C_c(comp(r_i,k_i))/f_c{}^i}{C_m\left(R\left(r_i, Data(k_i)\right)\right)/f_m{}^i + C_c\left(comp(r_i,k_i)\right)/f_c{}^i} \tag{22}$$

The processor profit is defined as $G(i) = U_p(i)/U_{np}(i)$, and so the maximization of the processor profit (under energy constraints) can be formalized as

$$\max G(i) = \begin{cases} \frac{C_m\left(R\left(r_i,Data(k_i)\right)\right)/f_m{}^i + C_c(comp(r_i,k_i))/f_c{}^i}{T_a(r_1,r_2,\cdots r_N)} & i = 1, \dots, N, cond\ 9.1 \\ \frac{C_m\left(R\left(r_i,Data(k_i)\right)\right)/f_m{}^i + C_c(comp(r_i,k_i))/f_c{}^i}{T_c(r_1,r_2,\cdots r_N)} & i = 1, \dots, N, cond\ 9.2 \end{cases} \tag{23}$$

Now the two performance objectives have been obtained, we can formulate an expression for the energy constraints. The energy budget is taken to be E_{budget}. According to the total energy consumption given in Eq. (19), the energy constraint can be expressed in the form:

$$\begin{aligned} &\textstyle\sum_{i=1}^{N} M_1 (f_c^i)^2 C_c\left(comp(r_i, k_i)\right)\mu \\ &+ \textstyle\sum_{i=1}^{N} M_2 (f_c^i)^2 C_m\left(R\left(r_i, Data(k_i)\right)\right)\mu \le E_{budget} \end{aligned} \tag{24}$$

So, the performance model under energy constraints can be expressed in the form:

$$\begin{gathered} \min \mathrm{T}(r_1, r_2, \cdots r_N) = \begin{cases} min\ T_a(r_1, r_2, \cdots r_N)\ , & i = 1, \dots, N: cond\ 25.1 \\ min\ T_c(r_1, r_2, \cdots r_N)\ , & i = 1, \dots, N: cond\ 25.2 \end{cases} \\ \max G(i) = \begin{cases} \dfrac{\frac{C_m\left(R\left(r_i,Date(k_i)\right)\right)}{f_m^i} + \frac{C_c(comp(r_i,k_i))}{f_c^i}}{T_a(r_1,r_2,\cdots r_N)}, i = 1, \dots, N: cond\ 25.1 \\ \dfrac{\frac{C_m\left(R\left(r_i,Date(k_i)\right)\right)}{f_m^i} + \frac{C_c(\mathrm{comp}(r_i,k_i))}{f_c^i}}{T_c(r_1,r_2,\cdots r_N)}, i = 1, \dots, N: cond\ 25.2 \end{cases} \\ T_a(r_1, r_2, \cdots r_N) = \frac{F}{\sum_{i=1}^{N} r_i \cdot k_i} \sum_{i=1}^{N} \frac{C_m\left(R\left(r_i,Date(k_i)\right)\right)}{f_m^i} \\ T_c(r_1, r_2, \cdots r_N) = max_{i=1}^{N}\{\frac{F}{\sum_{i=1}^{N} r_i \cdot k_i} \frac{\mathrm{C}_c\left(\mathrm{comp}(r_i,k_i)\right)}{f_c^i}\} \\ \textstyle\sum_{i=1}^{N} M_1 (f_c^i)^2 \mathrm{C}_c(comp(r_i, k_i))\mu + \sum_{i=1}^{N} M_2 (f_m^i)^2 C_m\left(R\left(r_i, Date(k_i)\right)\right)\mu \le E_{budget} \\ cond\ 25.1\text{:}\ \frac{C_c(comp(r_i,k_i))}{f_c^i} < \sum_{i=1}^{N} \frac{C_m\left(R\left(r_i,Date(k_i)\right)\right)}{f_m^i} \\ cond\ 25.2\text{:}\ \frac{C_c(comp(r_i,k_i))}{f_c^i} \ge \sum_{i=1}^{N} \frac{C_m\left(R\left(r_i,Date(k_i)\right)\right)}{f_m^i} \\ \begin{cases} \frac{C_c\left(comp(r_i,k_i)\right)}{f_c^i} < \frac{C_m\left(R\left(r_i,Date(k_i)\right)\right)}{f_m^i} (i = 1, \dots, N)\ \ cond\ 25.1 \\ \frac{C_c(comp(r_i,k_i))}{f_c^i} \ge \frac{C_m\left(R\left(r_i,Date(k_i)\right)\right)}{f_m^i} (i = 1, \dots, N)\ \ cond\ 25.2 \end{cases} \\ f_c' \le f_c^i \le f_c''(i = 1, \dots, N) \\ f_m' \le f_m^i \le f_m''(i = 1, \dots, N) \end{gathered} \tag{25}$$

We give constraints $\begin{cases} \frac{C_c(comp(r_i,k_i))}{f_c^i} < \frac{C_m\left(R\left(r_i,Date(k_i)\right)\right)}{f_m^i} (i = 1,2, \dots, N)\ \ cond\ 25.1 \\ \frac{C_c(comp(r_i,k_i))}{f_c^i} \ge \frac{C_m\left(R\left(r_i,Date(k_i)\right)\right)}{f_m^i} (i = 1,2, \dots, N)\ \ cond\ 25.2 \end{cases}$ in formula (25), which guarantees that the type of the program will not be changed during the frequency adjustment process. In other words, if the program execution time on each coprocessor is initially determined by the time of data communication, after the frequency adjustment, the program execution time on each coprocessor is still determined by the time of data communication. If the program execution time on each coprocessor is initially determined by the data calculation time, after the frequency adjustment, the program execution time on each coprocessor is still determined by the data calculation time. Therefore, in the optimization of load balancing, we adjust the workload and always have to follow this constraint. Even if the load is changed, it does not affect the nature of the type of program. The constraints $f_c' \le f_c^i \le f_c''$ $(i = 1,2, \dots, N)$ and $f_m' \le f_m^j \le f_m''$ $(i = 1,2, \dots, N)$ limit the ranges over which the processor and memory frequencies vary (f_c' and f_c'' represent the lower and upper bounds of the processor frequency, while f_m' and f_m'' are those of the memory frequency).

VI. Algorithm for Synergistic Energy-Constrained Optimization of Task Assignment

The previous section reveals that load imbalance only occurs when a system contains a processor that finishes one computation and takes a longer time than the overall time for one communication by all processors. Two scenarios are considered below to illustrate how one can adjust the task partition to overcome this imbalance. The aim of the adjustment process is to basically assign balanced tasks to each processor when there is a large task load from the whole program. As a result, each processor can finish all its tasks at the same (or nearly same) time.

In the first scenario, the time taken by each processor for one computation is longer than the total time taken by all processors for one communication. To improve load balance, more tasks need to be allocated to processors with faster execution speeds. In general, we suppose that there are N processors in the system with a given task partition granularity. The times taken by these processors for one communication and one computation can be expressed as:

$$\left[\left(\frac{C_m}{f_m}\right)_1,\left(\frac{C_c}{f_c}\right)_1\right],\left[\left(\frac{C_m}{f_m}\right)_2,\left(\frac{C_c}{f_c}\right)_2\right],\cdots,\left[\left(\frac{C_m}{f_m}\right)_N,\left(\frac{C_c}{f_c}\right)_N\right]$$

According to the assumption made for this scenario, we therefore have

$$\forall x \in [1,N]: \left(\frac{C_c}{f_c}\right)_x \geq \sum_{i=1}^{N}\left(\frac{C_m}{f_m}\right)_i.$$

In order to achieve balance among the processors, we first seek the minimum common multiple, COM, of $(C_c/f_c)_1, (C_c/f_c)_2, \cdots, (C_c/f_c)_N$. We then calculate the quantities,

$$COM_x^B = \frac{COM}{(C_c/f_c)_x}$$

for each processor. B is a factor that describes the relationships between the two parameters $(C_c/f_c)_x$ and COM. The relationship between COM_x^B and $r_x \cdot k_x$ is that $r_x \cdot k_x$ is the basic task assigned to GPU of type x. COM_x^B means that, in order to achieve load balance, the task proportion coefficient should be added to the basic task $r_x \cdot k_x$. So, the overall program execution time is

$$T_b(r_1, r_2, \cdots r_N)' = \frac{F}{\sum_{i=1}^{N} COM_i^B \cdot r_i \cdot k_i} \cdot \frac{C_c(COM_i^B \cdot r_i \cdot k_i)}{(f_c)_i} + \varepsilon \quad (26)$$

In the second scenario, some of the processors finish one computation using a longer time than the overall time consumed by all the processors for one communication, while others do not. In this case, optimizing load balance is complicated. A system consisting of two heterogeneous processors is taken as an example.

The times used by the two processors for a single communication and a single computation are given by $[(C_m/f_m)_1, (C_c/f_c)_1]$ and $[(C_m/f_m)_2, (C_c/f_c)_2]$. Suppose that these times are such that $(C_c/f_c)_1 < (C_m/f_m)_1 + (C_m/f_m)_2$ and $(C_c/f_c)_2 \geq (C_m/f_m)_1 + (C_m/f_m)_2$. The given task partition granularities are r_1 and r_2. In this case, it is clear that the execution time of the first processor is the shorter of the two and so the overall execution time of the program depends on that of the second processor. Therefore, more tasks need to be assigned to the first processor.

Under these circumstances, the possible situations occurring each time a task is added to the load of the first processor need to be discussed according to the ratio of the computation and communication times of the processors, RIO. To begin with, certain basic tasks are separately added to the two processors. As a result, the communication and computation times of the two processors will change with the increase in task load. The adjusted communication and computation times are recorded as $[(C_m/f_m)_1{}', (C_c/f_c)_1{}']$ and $[(C_m/f_m)_2{}', (C_c/f_c)_2{}']$.

```
1.  Algorithm: OTA
2.  Input: $\left[\left(\frac{C_m}{f_m}\right)_1,\left(\frac{C_c}{f_c}\right)_1\right],\left[\left(\frac{C_m}{f_m}\right)_2,\left(\frac{C_c}{f_c}\right)_2\right],\cdots,\left[\left(\frac{C_m}{f_m}\right)_N,\left(\frac{C_c}{f_c}\right)_N\right]$
3.  Output: $T$
4.  $COM$ = The minimum common multiple of
    $(C_c/f_c)_1, (C_c/f_c)_2, \cdots, (C_c/f_c)_N$;
5.  for ($i = 1; i \leq N; i++$)
6.  $COM_i^B = COM/(C_c/f_c)_i$;
7.  if ($\forall i \in [1,N]: \left(\frac{C_c}{f_c}\right)_i \geq \sum_{i=1}^{N}\left(\frac{C_m}{f_m}\right)_i$) then
8.      $T = \frac{F}{\sum_{i=1}^{N} COM_i^B \cdot r_i \cdot k_i} \cdot \frac{C_c(COM_i^B \cdot r_i \cdot k_i)}{(f_c)_i} + \varepsilon$;
9.  else
10.     $T = \frac{F}{\sum_{i=1}^{N} COM_i^B \cdot r_i \cdot k_i} \max\left\{\sum_{i=1}^{N} COM_i^B \cdot \left(\frac{C_m}{f_m}\right)_i, COM_i^B \cdot \left(\frac{C_c}{f_c}\right)_i\right\} + \varepsilon$;
11. end
12. return $T$;
```

Fig. 3. The OTA algorithm.

The justification is that $RIO_1{}'(r_1, k_1) = \frac{(C_c/f_c)_1{}'}{(C_m/f_m)_1{}'}$ represents the ratio of the computation and communication times of GPU 1 after adding tasks. At this time, it is necessary to consider the possible situation after increasing the amount of tasks assigned to GPU 1 each time according to $RIO_1{}'(r_1, k_1)$. Increase the task mount of GPU 1, if $RIO_1{}'(r_1, k_1) \leq 1$, i.e. $(C_c/f_c)_1{}' \leq (C_m/f_m)_1{}'$, when the task amount of GPU 2 is increased or not, there are $(C_c/f_c)_1{}' \leq (C_m/f_m)_1{}' + (C_m/f_m)_2{}'$. The communication consumption is a bottleneck to program execution for the first processor, and the time taken to execute basic tasks by the first processor is $(C_m/f_m)_1{}' + (C_m/f_m)_2{}'$.

If $RIO_1{}'(r_1, k_1) > 1$, then $(C_c/f_c)_1{}' \geq (C_m/f_m)_1{}' + (C_m/f_m)_2{}'$ will probably hold (but may not) after adjusting the task load. Under these conditions, the program execution time of the first processor is the larger one out of the communication and computation times. The execution time of the program of the first processor is $\max\{(C_m/f_m)_1{}' + (C_m/f_m)_2{}', (C_c/f_c)_1{}'\}$. Two situations may also present themselves to the execution time of the second processor (dominated by the communication or computation time) and so the execution time of the program of the second processor is $\max\{(C_m/f_m)_1{}' + (C_m/f_m)_2{}', (C_c/f_c)_2{}'\}$. Therefore, the load-balancing optimization strategy is to find two additional coefficients COM_1^B and COM_2^B for basic tasks such that

$$\max\left\{COM_1^B\left(\frac{C_m}{f_m}\right)_1 + COM_2^B\left(\frac{C_m}{f_m}\right)_2, COM_2^B\left(\frac{C_c}{f_c}\right)_2\right\}$$
$$= \begin{cases} COM_1^B\left(\frac{C_m}{f_m}\right)_1 + COM_2^B\left(\frac{C_m}{f_m}\right)_2, & cond\ 27.1 \\ \max\left\{COM_1^B\left(\frac{C_m}{f_m}\right)_1 + COM_2^B\left(\frac{C_m}{f_m}\right)_2, COM_1^B\left(\frac{C_c}{f_c}\right)_1\right\}, & cond\ 27.2 \end{cases}$$
$$cond\ 27.1: RIO_1{}'(r_1, k_1) \leq 1$$
$$cond\ 27.2: RIO_1{}'(r_1, k_1) > 1 \quad (27)$$

As a matter of fact, as $(C_c/f_c)_1{}' \leq (C_m/f_m)_1{}' + (C_m/f_m)_2{}'$ exists when $RIO_1{}'(r_1, k_1) \leq 1$, the first scenario is a special case of the second one. Under these circumstances, the overall execution time of the program is

$$T_c(r_1, r_2, \cdots r_N)' = \frac{F}{\sum_{i=1}^{N} COM_i^B \cdot r_i \cdot k_i} \max\left\{\sum_{i=1}^{N} COM_i^B \cdot \left(\frac{C_m}{f_m}\right)_i, COM_i^B \cdot \left(\frac{C_c}{f_c}\right)_i\right\} + \varepsilon \quad (28)$$

The holistic algorithm for synergistic energy constrained optimization of task assignment (OTA) is shown in Fig. 3. We consider the following two cases in the load balancing optimization. Case 1 is the time taken by every processor for completing a computing task is longer than the total time taken by all the processors for completing a communication task. Case 2 is the time taken by some processor for completing a computing task is longer than the overall time consumed by all the processors for completing a communication task. The first ***else*** condition refers to Case 2. By incorporating Eq. (27) into Eq. (25), we arrive at the synergistic energy-constrained performance optimization model for task load balancing that we require. Therefore, the problem of optimizing the performance of heterogeneous multi-GPUs can be reduced to an M-extremum problem. That is, we need to find the task partitioning granularity k_i $(i \leq N)$ for each type of GPU (using the same task partitioning granularity for the same type of GPU) that yields the shortest program execution time and highest processor profit. Generally speaking, the number of GPUs in the system is limited due to the limitations of the bus ports. Therefore, the number of heterogeneous GPUs is small and so the searching process is generally fast to carry out and the problem quickly solved.

VII. Experiments and Evaluation

A. *Experimental Platform and Test Cases*

In this paper, the actual platform and simulation platform are both used for experimental verification.

The actual platform is a heterogeneous system consisting of an Intel i7 920 quad-core CPU and AMD 4870 GPU is employed as the experimental platform (Table I). In the system, the CPU and GPU have their own independent memory spaces.

TABLE I
Characteristics of the Processors Used in This Work

Parameter	Intel i7 920 CPU	AMD 4870 GPU
Processor frequencies (GHz)	2.67, 2.4, 2.0, 1.6	0.75, 0.65, 0.55
Memory frequencies (GHz)	1.33 (DDR3)	0.9/0.7/0.5 (GDDR5)
Cache	L1: I32 kB, D32 kB; L2: 256 kB; L3: 8 MB	–
Memory (GB)	8	1
Compiler	Intel Ifor/Icc v11.1-OpenMP -fast	Brook+ Brcc v1.4

At present, the number of GPUs included in actual systems is small, and most systems only contain homogeneous GPUs. Therefore, in this section, a simulation method is adopted in order to better test the impact of energy consumption constraints and load balancing strategies on the performance of a heterogeneous system.

GPGPU-Sim is a relatively mature GPU simulator [28]. GPGPU-Sim divides the GPU into five main modules: shader cores, interconnection network, L2 cache, DRAM, and memory controller. In this article, we use the Wattch power consumption model available in the GPGPU-Sim simulator to implement the power consumption modeling of the shader cores, L2 cache, memory controller, and other components in the GPU. For the interconnection network, we use the power modeling method used in PowerRed [29]. A modeling method available in the literature is used for the DRAM [30]. For each component, the simulator counts the activity and accumulates the power consumption within the clock cycle. Finally, it sums the results up to get the total GPU power consumption. We modified the simulator, on the basis of GPGPU-Sim, adding performance statistics to the data communication part and implementing a simple simulator at the application layer to simulate the simultaneous execution of multiple GPUs in an event-driven manner.

TABLE II
Parameter Settings Used in the Quadro FX 5600 GPU Power Simulator

SM	4	Network	Crossbar
Warp size	32	SIMT width	32
Max blocks per SM	8	Max threads per SM	1024
Shader Clock	1.35 GHz	Core Clock	600 MHz
Network Clock	650 MHz	DRAM Clock	800 MHz
Memory Clock	1.6 GHz	Memory Size	1.5 GB
Memory Bandwidth		76.8 GB/s	
Software		GPGPU-Sim 2.1.1b, NVCC 2.2, GCC 4.3	

TABLE III
Test Cases

Application	Description	Scale[a]	Kernel program
HotSpot	Thermal simulation tool	2048*2048	Hotspot
k-means	Clustering algorithm	819,200 (34 features)	Cluster
MGRID	Poisson equation solver	256*256*256	RESID, PSINV, RPRJ3, INTERP
SWIM	Shallow water modeling solver	2048*2048	CALC1, CALC2, CALC3
BS	BlackScholes	256*256	BS
LP	Laplace	16*16*2048	LP

[a]Number of data points.

GPGPU-Sim configures the simulated GPU by reading configuration files at runtime. Here, the configuration file parameters were set to represent a Quadro FX 5600 GPU. We maintained a global event list in the program. Each event contained a timestamp, GPU number, event type, and parameter list. ('Time stamp' indicates the clock cycle when the event occurred; 'GPU number' refers to the GPU to which the event belongs; 'Event type' includes four types: task communication start event, task communication end event, task computing start event, task computing end event. 'Parameter list' is a list of parameters required by the program to call communication and calculation operations.) We harnessed the event-driven simulation, taking the event with the smallest time stamp from the global event list each time. Table II shows some of the parameter settings used in the Quadro FX 5600 GPU power simulator.

Six applications were selected to use as test cases (Table III). MGRID and SWIM were chosen from the SPECOMP2001 benchmark test set and utilized as a Poisson equation solver and shallow water modeling solver, respectively. The other two, HotSpot and k-means, were selected from the Rodinia program set which is mainly used for assessing heterogeneous parallel systems. The HotSpot application was employed to simulate a chip temperature model while k-means was adopted in a form commonly used in data mining (as a clustering algorithm). BlackScholes (BS) was selected from the AMD APP SDK and the Laplace transform (LP) application is one commonly used in the scientific computing field. These applications are characterized by a large number of loop iterations in the kernel function. Abstract the loop iteration space as a task. Different task granularity is set to divide tasks, and different task load is mapped to multiple GPUs. It can meet the basic conditions of load balancing optimization.

B. Compiler Implementation and Example Code

We design and implement a source to source compiler model MPtoStream based on GCC compiler. The compiler model implemented in this work is one that we introduced in Ref. [23]. The compiler implementation is valid for both the actual platform and simulation platform.

```
//Insert frequency-scaling code
//Load input data into GPU space
Astream.Read(A);
//Set kernel execution domains
KerName. mv_kernel1(uint2(0, $N_{opt} - 1$));
KerName. mv_kernel2(uint2($N_{opt}, 2N_{opt} - 1$));
//Call GPU kernel
KerName (Astream, Bstream, Cstream);
//CPU execution
!OMP PARALLEL DO
Do I = 2$N_{opt}$, 2048
  Do J = 1, 2048
    C(J,I) = A(J,I)*B(J,I)
  End do
End do
//Write output data computed by GPU to CPU Space
Cstream.domain(int2(0,0), int2(0,total)).write(C)
```

Fig. 4. Matrix Multiplication implemented using parallel loop scheduling code.

The applications derived from SPECOMP2001 and Rodinia involve a large number of matrix operations. Fig. 4 presents a fragment of parallel loop scheduling code related to the implementation of matrix multiplication. The outer loop consists of 2048 iterations. Eq. (25) is used to derive the optimal task partition granularity $r_{opt} \cdot k$. The granularity of this task corresponds to the number of loop iterations that the application maps to the processor. Note that if the number of loop iterations mapped to the GPU is N_{opt}, then $N_{opt} = r_{opt} \cdot k$. We assume that the system contains two GPUs at this time. Hence, the subset of iterations $(0, N_{opt} - 1)$ is assigned to the first GPU, $(N_{opt}, 2N_{opt} - 1)$ is assigned to the second GPU, and the remaining cycles, $(2N_{opt}, 2048)$, are allocated to the CPU for execution. The execution code is shown in Fig. 4. First, the GPU data stream space is declared and the data loaded. Then, the iteration spaces mapped onto the GPUs are specified. Finally, the GPUs are called to perform the calculations. The CPU then completes the calculations using the remaining iteration space (so its iteration index starts from $2N_{opt}$). The kernel calculation process is completed on the GPU and the output results saved back to the CPU storage space. This completes the parallel loop partitioning process of the CPU–GPU heterogeneous parallel system. Experimental Results and Analysis

In this section, we report the implementation experiments and analyze the results from different perspectives. The two subsections C.1 and C.2 do not consider load imbalances situation, and do not need to simulate multiple heterogeneous GPUs environments. Therefore, the experimental contents of these two sections are completed on a real AMD cores platform. Subsection C.3 considers load imbalances situation and perform load balancing tests on multiple heterogeneous GPUs. However, the actual system contains fewer GPUs, and most systems only contain homogeneous GPUs. Therefore, in order to better test the impact of heterogeneous system load balancing strategies on performance under energy constraints, the experimental results of subsection (3) are completed on the GPGPUSim simulation platform.

The frequency assignment and task allocation policy can be implemented dynamically. By estimating the increase of power consumption, the appropriate frequency and task partition granularity are calculated to guide the real-time voltage regulation and task partition of the processor, to achieve the optimal power consumption. The data can be sampled periodically. The repetition period is set up in the process, each repetition period contains multiple sampling periods and a sampling guidance period. Through the statistics of the sampling period, the proper frequency values of processor and memory are calculated to guide the tasks for the rest of the sampling period. Assume that heterogeneous systems contain N GPUs with different configurations. Different types of GPUs have a different granularity of basic task partition: $r_1 \cdot k_1,\ r_2 \cdot k_2, \cdots, r_N \cdot k_N\ (i = 1, \dots, N)$, where $r_i \cdot k_i$ represents the amount of tasks assigned to GPU of type i, and the corresponding number of thread blocks is m_i. At this time, the number of the thread blocks allocated on N different types of GPUs is $m_1, m_2, \dots m_N$. The data are gathered every m_i threads on the type *i* GPU. The length of time that every m_i threads use is called a repetition cycle. During the repetition cycle, if the condition 25.1 (25.2) is met, the first n_i $(n_i < m_i)$ threads are executed in the state of allowing to adjust the task partition. Thus, the task execution time is assigned to $T_a(r_i \cdot k_i)$ $(T_c(r_i \cdot k_i))$, the processor computing power to $P_{ac}(r_i \cdot k_i)$ $(P_{cc}(r_i \cdot k_i))$, and the memory access power to $P_{am}(r_i \cdot k_i)$ $\left(P_{cm}(r_i \cdot k_i)\right)$, respectively. The next n_i threads are executed under the condition that the task partition are not allowed to be adjusted. In this case, the task execution time is assigned to $T_a{}'(r_i \cdot k_i)$ $(T_c{}'(r_i \cdot k_i))$, the processor computing power to $P_a{}'(r_i \cdot k_i)$ $(P_{cc}{}'(r_i \cdot k_i))$, and the memory access power to $P_{am}{}'(r_i \cdot k_i)$ $(P_{cm}{}'(r_i \cdot k_i))$, respectively. Performance gains are calculated by $gain = [T_a{}'(r_i \cdot k_i) - T_a(r_i \cdot k_i)]$ $(gain = [T_c{}'(r_i \cdot k_i) - T_c(r_i \cdot k_i)])$. In the remaining sampling guidance period of（$\mathrm{m_i} - 2 \times \mathrm{n_i}$）, if gain > 0, the optimal processor and

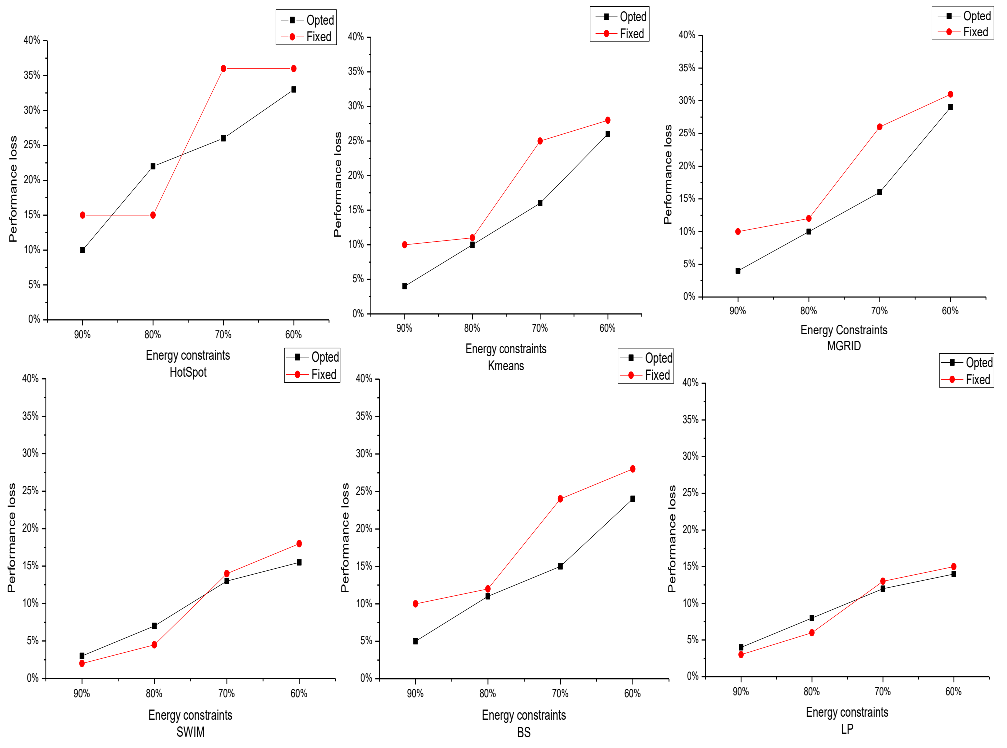

Fig. 5. Increase in performance loss (in terms of program execution time) as a function of energy constraint imposed.

memory frequencies $(f_c'(i), f_m'(i))$ can be adjusted under the current task partition r_i, or the task partition algorithm can determine the optimal task partition strategy r_i' under the current frequency $(f_c(i), f_m(i))$. If $gain < 0$, the remaining threads are executed in the current task division mode r_i with the frequency $(f_c(i), f_m(i))$.

1) Performance loss (in terms of program execution time)

In the experiments performed, the performance loss changes when energy constraints are applied. As a result, the power is consumed when the highest frequency used for parallel execution of each program is chosen to use as a benchmark. The change in performance loss (with respect to objective parameter 1: program execution time) when constraints are added is shown in Fig. 5 for each of the applications used.

The computation segment executed merely by the CPU consumes much less energy than the energy constraints imposed and so this execution time is not affected by the energy constraint applied. To more clearly reveal the effect of optimization, therefore, it is necessary to consider the contributions made to the computation segment by the parallel execution times of the CPU and GPU (Fig. 5). To allow informative comparisons to be made, execution performance using a fixed task partition also needs to be considered (such results are given the label 'Fixed'). In these experiments, just the operating frequency of the processor is scaled to meet the energy constraints while the task partition is fixed. The optimized results obtained using the method proposed in this research (coordinating task partition and frequency scaling) to satisfy the energy constraints are distinguished using the label 'Opted').

Fig. 5 shows that the performance of the programs suffers as the energy constraint placed on the system is decreased. (This is especially true for the more computationally-intensive applications like HotSpot, k-means, MGRID, and BS. However, the access-intensive application SWIM and LP are less significantly affected by energy constraint). As heterogeneous processing units have different trends in their power consumption and performance at different frequencies, a fixed-partition strategy fails to efficiently play to the advantages of the heterogeneous parallel processing methodology. In comparison, the Opted strategy coordinates task partitioning and frequency scaling which can effectively bring the advantages of heterogeneous parallel processing into full play. As a result, better execution performance is achieved when energy constraints are imposed. For example, when the energy constraint corresponds to 90%, the performance improvement gained using the Opted strategy is 7.3% above the performance of the Fixed strategy.

The results imply that the execution performance of the Opted strategy is lower than that of the Fixed strategy in a few cases. The main reason is that the proposed method is the theoretical optimal results under the assumption that the processor frequency is continuously adjustable. However, in the real GPU systems, the processor only supports a finite number of discrete physical frequency values. Therefore, we have to choose the maximum physical frequency value which does not exceed the optimal value of theoretical frequency. The state of discrete adjustable processor deviates from the theoretical optimal results when the energy constraint is 80%. The fixed method is to find the best frequency value in discrete space by traversing all frequency combinations of each processor. Therefore, in some cases, the Fixed method will get the better performance results.

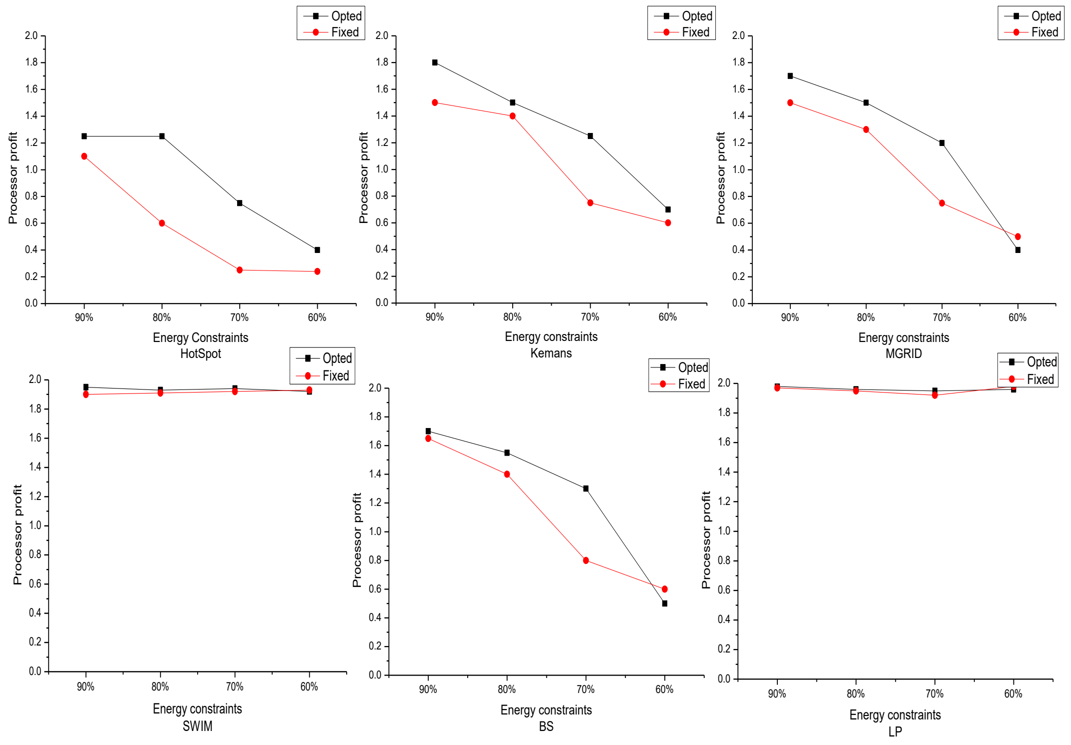

Fig. 6. Change in performance loss (in terms of processor profit) as a function of energy constraint imposed.

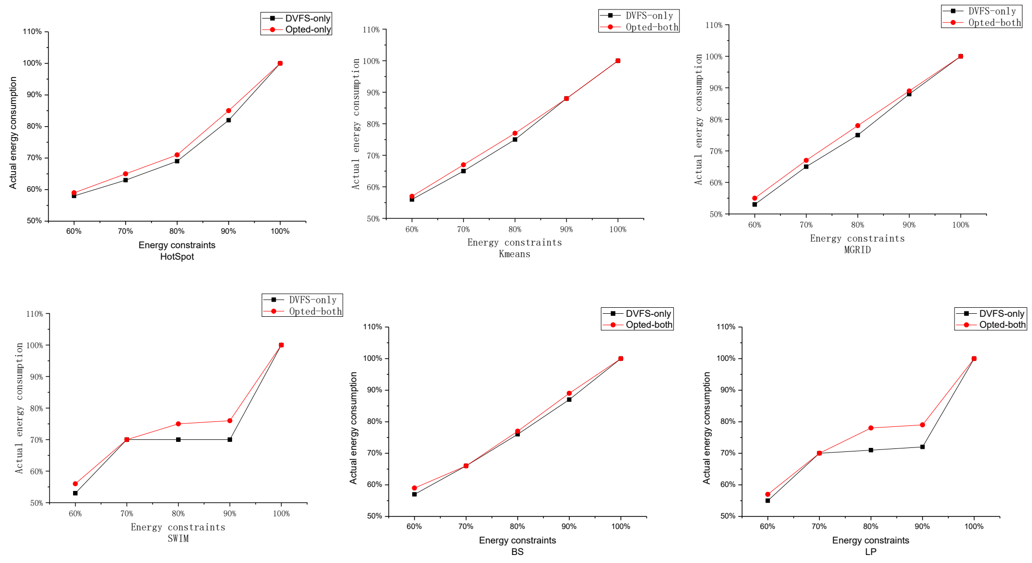

Fig. 7. Energy consumption control precisions

2) Performance loss (in terms of processor profit)

Fig. 6 shows the way in which the processor profit of the programs varies with the energy constraint decreased (in terms of objective parameter 2: processor profit). Once again, the change in processor profit with energy constraint is shown using the execution of each program at the highest frequency setting as the benchmark.

It can be seen from Fig. 6 that the Opted strategy generally provides a better execution performance than the Fixed strategy for a given energy constraint. For the access-intensive applications SWIM and LP, the optimal processor profit is always close to 2 irrespective of the energy constraint. In contrast, the processor profit drops in a simple manner in the computationally-intensive applications (HotSpot, k-means, MGRID, and BS) as the value of the energy constraint is decreased. As the experiments proceeded, it was found that the memory frequency remained at its lowest level (0.5 GHz) when performing the access-intensive applications. On the other hand, the processor frequency changed significantly, decreasing as the energy constraint decreased.

In fetch-intensive programs (SWIM and LP), the fetch time is a crucial factor affecting processor revenue. When the energy

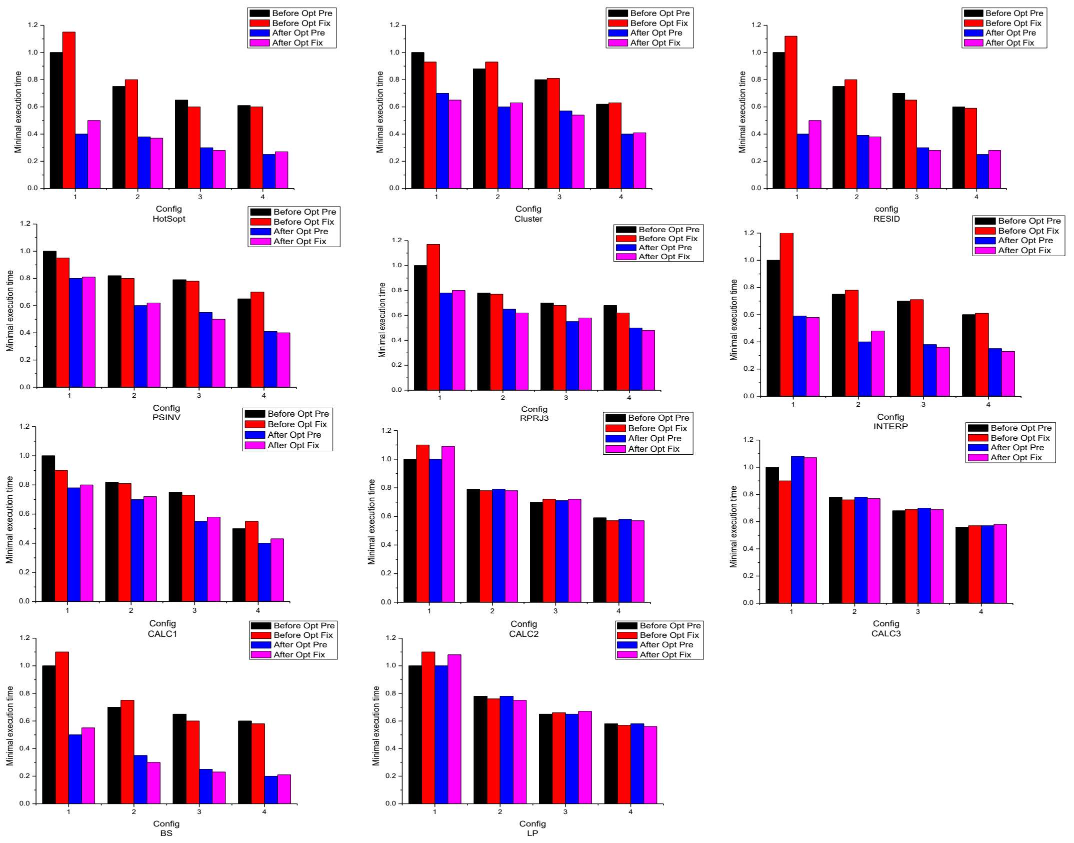

Fig. 8. Predicted and simulated minimum execution times for the heterogeneous multi-GPU environment (each normalized relative to the before-optimization time achieved using Configuration 1).

constraint becomes to decrease; the memory frequency becomes very sensitive and will quickly fall to its lowest allowable value. The processor gain value is close to two. In calculation-intensive programs, the calculation time is the key factor affecting the processor's revenue. Therefore, when the energy constraint becomes to decrease, the processor frequency is not as sensitive as the memory frequency. As a result, it gradually decreases as the energy constraint becomes smaller. At the same time, processor benefits decrease as processor frequency decreases.

We can conclude from our experimental results on heterogeneous parallel systems that the memory can be better optimized as a part of the overall system energy optimization (rather than just optimizing the processor). This is especially so when the program involved is memory-intensive as adjusting the memory frequency can achieve significantly better performance optimization.

The lower the energy constraint, the smaller the value of E_{budget}. As a result, we need to reduce both the processor and memory frequencies to reduce energy consumption to keep it less than E_{budget}. Our approach is to determine the optimal processor and memory frequencies using a power-constrained performance optimization model. This entails inserting instructions to adjust the processor and memory frequencies at the appropriate point in the source code to obtain the final optimized code. This guides the processor and memory to adjust their frequencies.

3) Energy consumption control precisions

For the similar performance optimization model with limited energy consumption, we compare the energy optimization method proposed in this paper with the hierarchical maximum power management method proposed in reference [24]. In reference [24], we only use DVFS technology (DVFS-only) to optimize hierarchical maximum power management. In this paper, Opted-both is used to represent the optimization of combining DVFS and task partitioning. The experimental results are shown in Fig. 7. The figure shows the actual energy consumption of the system under different energy consumption constraints. The abscissa represents the given energy consumption constraint, and the ordinate is the actual energy consumption value. It can be seen from the figure that the proposed energy control method can approach the given energy constraint more accurately without exceeding the energy constraint compared with the hierarchical maximum energy control method, and the accuracy can be increased by about 7%.

4) Minimum Execution Times for the heterogeneous multi-GPU environment

Actual systems contain only a few GPUs and most of these systems tend to be homogeneous in nature. Therefore, in this section, the GPGPU-Sim simulation method is used to test the impact of the heterogeneous load balancing strategy on the performance of a system subject to energy constraints. Except for the number of SMs, the other GPU parameters were set to those appropriate to a Quadro FX 5600 GPU. We modified the configuration files accordingly and varied the number of SMs from 4 ('GPU A') to 8 ('GPU B') to 16 ('GPU C'). We thus simulated 4 different heterogeneous multi-GPU configurations. Configuration 1(1,1,1), Configuration 2(2,1,1), Configuration 3(1,2,1), and Configuration 4(1,1,2). The three parameters in the brackets represent the number of GPU A, GPU B and GPU C respectively.

Fig. 8 shows the predicted and actual minimum execution times obtained using the four configurations. In each chart, ‘Opt Pre’ indicates a result predicted by the performance optimization model proposed in this paper. Similarly, ‘Opt Fix’ is used to indicate the result obtained via GPGPU-Sim simulation. Two other labels are also added: ‘Before’ indicates the result obtained before the system is optimized to balance load and ‘After’ the result obtained after load balancing. As different applications have execution times that vary greatly, each set of data is normalized relative to the minimum execution time obtained using Configuration 1 before load-balancing optimization. The results shown in Fig. 8 demonstrate that the pipeline model is able to effectively predict minimum program execution times in systems containing multiple heterogeneous GPUs. The programs were then guided to select the optimal processor and memory frequencies and task partition strategy to realize optimum performance under the given energy constraints. The minimum execution times subsequently predicted by the model are close to those obtained via simulation (the relative error is less than 10%). Except for CALC2 and CALC3 (in the SWIM tests) and LP applications, the performances of all programs were effectively improved after load balancing optimization. As LP, CALC2, and CALC3 are all memory-intensive programs, the calculation process is very simple and the communication overhead becomes the bottleneck to program operation. The load of each GPU has already reached balance and so load balancing optimization has no effect on it. The results also show that the optimal task partition strategy is obtained using a basic task partition granularity in the range 5–10. This is mainly because a task partition granularity that is too small fails to fully exploit the instruction level parallelism and hide memory access latency. As a result, the GPU cannot fully exploit its computation capacity.

VIII. Conclusion

When optimizing processors, it is very important to also consider the power consumed by off-chip memory. That is, both of these parts need to be optimized together to reduce the energy consumed by the system as a whole. In this work, energy consumption is optimized by coordinating the processors and memory of an energy-constrained system. A communication–computation pipeline for parallel programs was constructed to optimize a program’s performance by simultaneously performing voltage and frequency scaling of the processors and memory and using an appropriate task partitioning strategy. The conditions that lead to load imbalance between GPUs were also discussed in the context of a heterogeneous multi-processor environment. Corresponding load-balancing optimization methods were subsequently proposed. Our experimental results indicate that our proposed method is capable of producing satisfying execution times and processor profits while meeting the given energy constraints.

Acknowledgment

This work was sponsored in part by the National Natural Science Foundation of China (grant numbers 61672168, 61672172, 61772143, and 61803093), High Resolution Earth Observation Major Project (83-Y40G33-9001-18/20), and Provincial Agricultural Science and technology innovation and Extension project of Guangdong Province (2019KJ147).

References

[1] R. Raghavendra, P. Ranganathan, V. Talwar, et al. “NO “power” struggles: coordinated multi-level power management for the data center,” Proc. 13th Int Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS’13), pp. 48-59, 2008.

[2] J. Li, J F. Martinez, “Power-Performance Implications of Thread-level Parallelism on Chip Multiprocessors,” Proc. IEEE Int Conf. Symposium on Performance Analysis of Systems and Software (SPAS’05), Washington, DC, USA, pp. 124-134, 2005.

[3] J M. Cebrian, L. Juan, Aragón, M. José, García, et al. “Efficient microarchitecture policies for accurately adapting to power constraints,” Proc. 23th IEEE Int Conf Symposium on Parallel and Distributed Processing (IPDPS ‘2009), pp. 1-12, 2009.

[4] X. Wang, A. K. Singh, B. Li, Y. Yang, H. Li and T. Mak, “Bubble Budgeting: Throughput Optimization for Dynamic Workloads by Exploiting Dark Cores in Many Core Systems,” IEEE Transactions on Computers, vol. 67, no. 2, pp. 178-192, 2018.

[5] R. Martin, P. Anuj, and H. Jörg, “Pareto-Optimal Power- and Cache-Aware Task Mapping for Many-Cores with Distributed Shared Last-Level Cache,” In Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED ’18). New York, NY, USA, pp. 1–6, 2018.

[6] H.B. Yang, “Power-aware Compilation Techniques for High Performance Processors,” Doctor dissertation. University of Delaware. Winter 2004.

[7] R.C. Zhao, Z.M. Tang, Z.Q. Zhang, G.R. Gao, “Study on the Low Power Technology of Software Pipeline,” Journal of Software, vol. 14, no. 8, pp. 876-880, 2003.

[8] C.H. Hsu and U. Kremer, “Single region vs. multiple regions: A comparison of different compiler-directed dynamic voltage scheduling approaches,” Lecture Notes in Computer Science, vol. 2325, pp. 43-48, 2002.

[9] H. Saputra, M. Kandemir, N. Vijaykrishnan, M.J. Irwin, J. Hu, C. -H Hsu and U. Kremer, “Energy-conscious compilation based on voltage scaling,” in Proceedings of the joint conference on Languages, compilers and tools for embedded systems: software and compilers for embedded systems (LCTES/SCOPES’02), New York, NY, USA, pp. 2-11. 2002.

[10] F. Xie, M. Martonosi and S. Malik, “Compile-Time Dynamic Voltage Scaling Settings: Opportunities and Limits,” in Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, San Diego, California, USA, pp. 49-62, 2003.

[11] X.B. Fan, C.S. Ellis and A.R. Lebeck, “The Synergy between Power-aware Memory Systems and Processors Voltage Scaling,” in Proceedings of the 3th Int conference on Power - Aware Computer Systems (PACS’03), San Diego, CA, pp. 164-179, 2003.

[12] K.R. Basireddy, A.K. Singh, B.M. Al-Hashimi, et al. “AdaMD: Adaptive Mapping and DVFS for Energy-efficient Heterogeneous Multi-cores,” IEEE Transactions on Computer Aided Design of Integrated Circuits & Systems, pp.1–6, 2019.

[13] B. Donyanavard, T. Mück, S. Sarma and N. Dutt, “SPARTA: Runtime task allocation for energy efficient heterogeneous manycores,” 2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), Pittsburgh, PA, pp. 1-10, 2016.

[14] M. Annavaram, E. Grochowski, J. Shen, “Mitigating Amdahl’s Law through EPI Throttling,” Proc. 32th Int Conf. Symposium on Computer Architecture (SCA’05), Washington, DC, USA, pp.298-309, 2005.

[15] C. Isci, A. Buyuktosunoglu, C-Y. Cher et al., “An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget,” Proc. 39th Annual IEEE/ACM Int Conf. Symposium on Microarchitecture (SCA’06), Washington, DC, USA, pp. 298-309. 2006.

[16] J. Sartori, R. Kumar, “Distributed Peak Power Management for Many-core Architectures,” in Proceedings of Int Conference Design, Automation and Test in Europe. 3001 Leuven, Belgium, pp. 1556-1559, 2009.

[17] K. Meng, R. Joseph, R.P. Dick, et al., "Multi-Optimization Power Management for Chip Multiprocessors," in Proceedings of 15th Int Conference Parallel Architectures and Compilation Techniques. New York, NY, USA, pp. 177-186, 2008.

[18] K. Ma, X. Li, M. Chen, X.R. Wang, "Scalable power control for many-core architectures running multi-threaded applications." ACM SIGARCH Computer Architecture News. Vol. 39, No. 3, pp. 449-460, 2011.

[19] J.M. Cebri, J.L. Aragon, S. Kaxiras, "Power token balancing: Adapting cmps to power constraints for parallel multithreaded workloads," in Proceedings of 2011 IEEE International Parallel & Distributed Processing Symposium. pp. 431-442, 2011.

[20] Y. Yi, X. Ping, K. Jingfei, et al., "A GPGPU compiler for memory optimization and aprallelism management," in Proceedings of the 2010 ADM SIGPLAN conference on Programming language design and implementation. New York, USA, pp. 86-97, 2010.

[21] J. Byunghyun, S. Dana, M. Perhaad, et al., "Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures," IEEE Transactions on Parallel and Distributed Systems, pp. 105-118, 2011.

[22] J. Chen, Y. Dong, X-j. Yang, et al., "A Compiler-Directed Energy Saving Strategy for Parallelizing Applications in On-Chip Multiprocessors," in Proceedings of the 4th International Symposium on Parallel and Distributed Computing, Washington, DC, USA, pp. 147-154. 2005.

[23] Z.W. Wang, H. Wang, W. Zhao, et al., "Energy optimization of parallel programs in a heterogeneous system by combining processor core-shutdown and dynamic voltage scaling," Future Generation Computer Systems, vol. 92, pp. 198-209, 2019.

[24] Z.W. Wang, H. Wang, W. Zhao, et al., "Three-level performance optimization for heterogeneous systems based on software prefetching under power constraints," Future Generation Computer Systems, vol. 86, pp. 51-58, 2018.

[25] Z.W. Wang, L.L. Cheng, H. Wang, et al., "Energy Optimization by Software Prefetching for Task Granularity in GPU-Based Embedded Systems." IEEE Transactions on Industrial Electronics, vol. 67, no. 6, pp. 5120-5131, 2020.

[26] Z.W. Wang, T. Wang, G.P. Zhao, et al., "Coordinated optimization of the performance of processors and memory in a heterogeneous system under energy constraints," in Proceedings of the 15th International Symposium on Pervasive Systems, Algorithms and Networks (I-SPAN), pp. 100-106, 2018.

[27] W. Liao, L. He, and K.M. Lepak, "Temperature and supply voltage aware performance and power modeling at microarchitecture level," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 24, no. 7, pp. 1042-1053, 2005.

[28] A. Bakhoda, G.L. Yuan, W.L. Fung, H. Wong, and T.M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in Proceedings of IEEE Int Symp Performance Analysis of Systems & Software, pp. 163-174, 2009.

[29] K. Ramaniz, A. Ibrahim, S. Dan, "PowerRed: a flexible modeling framework for power efficiency exploration in GPUs," Worskshop on Gpgpu, pp. 1-8, 2007.

[30] Š. Tajana, B. Luca, D.M. Govanni, "Cycle-accurate simulation of energy consumption in embedded systems," in Proceedings of the 36th annual ACM/IEEE Design Automation Conference. New York, NY, USA, pp. 867–872, 1999.

Zhuowei Wang received Ph.D. degree in computer system architecture from Wuhan University, Wuhan, China, in 2012.

She is now an associate professor in the College of Computers at Guangdong University of Technology. Her research interests include high performance computing, low power optimization and distributed computing systems.

Xiaoyu Song (M'99-SM'04) received the Ph.D. degree from the University of Pisa, Italy, in 1991.

From 1992 to 1998, he was on the faculty at the University of Montreal, Canada. He joined the Department of Electrical and Computer Engineering at Portland State University in 1998, where he is now a professor. He was an editor of IEEE Transactions on VLSI Systems and IEEE Transactions on Circuits and Systems. He was awarded an Intel Faculty Fellowship from 2000 to 2005. His research interests include formal methods, design automation, embedded systems and emerging technologies.

Lianglun Cheng has a master degree in automation from Huazhong University of Science and Technology. He received the Ph.D. degree in machinery manufacturing and automation, from Changchun Institute of Optical Precision Machinery and Physics, Chinese Academy of Sciences.

He is now a professor in the College of Computers at Guangdong University of Technology. His research interests focus on IOT, CPS and sensor networks.

Hao Wang has a Ph.D. degree and a B.Eng. degree, both in computer science and engineering, from South China University of Technology.

He is an associate professor in the Department of Computer Science in Norwegian University of Science & Technology, Norway. His research interests include big data analytics, industrial internet of things, high performance computing, safety-critical systems, and communication security.