CGI 2015 manuscript

GPGPU-Perf: Efficient, Interval-based DVFS Algorithm for Mobile GPGPU Applications

SeongKi Kim · Young J. Kim

Abstract Although general purpose computation on graphics processing unit (GPGPU) technologies are available even on GPUs, its performance has been seriously affected by the underlying dynamic voltage and frequency scaling (DVFS) mechanism of GPU. In order to save the energy, eventually prolonging the battery life, the DVFS adjusts the GPU's frequency according to the past utilization. When the GPU processes graphic tasks only, it is enough to process them within a fixed time (typically 30∼60 frames per second), so the DVFS parameters can be conservatively set. However, in GPGPU case, the GPU should process them at much higher rates depending on applications. Although a modification to DVFS parameters may improve the GPGPU performance, the energy efficiency is sacrificed, and the performance of graphic tasks is affected, as these parameters are shared by both graphic and GPGPU tasks. In order to improve the GPGPU performance without influencing the graphic performance, we devise the new GPGPU-Perf algorithm that adjusts the DVFS parameters such as thresholds and an interval. The new algorithm controls the frequency more intelligently for mobile GPGPU applications, and thus the performance over energy increases by 1.44 times with no influences to graphic tasks and any modifications of GPGPU algorithms. To the best of our knowledge, this paper is the first work that proposes a GPU DVFS algorithm for GPGPU applications.

SeongKi Kim
E-mail: skkim9226@gmail.com

Young J. Kim
E-mail: kimy@ewha.ac.kr

Department of Computer Science and Engineering, Ewha Womans University, Seoul, Korea

Keywords DVFS · GPGPU · Mobile device · OpenCL · OpenGL ES

1 Introduction

For the last five years, mobile devices such as smartphones and tablets have been ubiquitous, and the related hardware including GPUs has been significantly improved. On the display side, the resolution of Nexus One was WVGA (800×480) in 2010, but those of Nexus 4 and Nexus 10 became WXGA (1280×768) and WQXGA (2560×1600) in 2012. This implies that the amount of graphically processed data has increased by approximately 2.56 times and 10.67 times in each case. In order to quickly process the huge amount of graphic data, a mobile GPU has been considered very more important and its hardware has improved accordingly as a result. With these hardware improvements, the programmable kernel of OpenCL [8] enables the use of GPU for general computation besides conventional graphic processing. In order to utilize the GPU for more general purposes, many researches have been conducted in numerous areas but most of these efforts are made on desktop, discrete GPUs, not on mobile GPUs, which has quite different characteristics from desktop counterparts; an obvious and notable example includes an issue of energy consumption on mobile GPUs.

There are a large number of successful GPGPU works on the desktop side. For instance, GPGPU is used for creating a visual model from multiple images [4], and quickly solving the image matting problem [7]. The construction of distance fields is accelerated by 35-64 times over a serial CPU implementation [11]. A collision-detection algorithm is accelerated by a factor of 71 times over the CPU Bullet Library [10]. In computer vision, the per-

formance of SIFT algorithm is accelerated by 4-7 times over an optimized CPU version, and by 6.4 times over a published GPU implementation with 87 percent reduced energy consumption [16]. GPGPU can be also used for accelerating feature extraction [1].

Although many researches have been conducted to successfully improve the GPGPU performance in the above fields, the energy saving in these applications an important factor in mobile applications. This energy saving factor is very significant for mobile devices because the energy of a mobile device is supplied by a battery with a limited capacity, but, unfortunately, the battery technologies grow relatively slowly. As a result, various researches have been proposed for energy savings on mobile devices, and the DVFS technologies have been adopted to the GPU. The dynamic voltage and frequency scaling (DVFS) adjusts the GPU's future voltage and frequency based on the past utilization or some signatures of processing tasks, and maintains them during the next time period to process the tasks more efficiently. As an example, if the GPU processed a lot of works during the last time interval, the DVFS increases the frequency and the voltage under the assumption that there will be also a lot of works generated in the future. This DVFS technology reduced GPU's energy consumption by 14.4% [9] in the GPGPU-Sim simulator [2], and reduced 19.28% energies with a little cost of no more than 4% performance loss [18].

There are many parameters to adjust with DVFS. The ratio between energy saving and performance increase, and the thresholds for increasing/ decreasing a frequency are among these configurable parameters. Compared to the number of graphic applications, GPGPU applications, not to mention of the killer applications, are relatively scare in the current market. Thus, the above DVFS parameters are usually determined in a graphic-centric way. Since $30 \sim 60$ FPS is enough for typical graphic applications, these existing graphics-centric DVFS parameters do not work well for GPGPU applications, as GPGPU applications require higher performance rates, hindered by a low frequency.

Our paper offers the following major contributions to GPGPU DVFS research. First, we provide the reasons and make a case of why the GPGPU DVFS should be different from the conventional DVFS for graphic applications. Second, we propose four novel strategies for efficient GPGPU- based DVFS targeted toward mobile applications, and measure their performance improvement and energy consumption. Third, we show that applications' performance/energy improvements by each strategy are affected by their characteristics. Finally, we propose a new GPGPU-Perf algorithm that integrates the four strategies and can improve the low performance problem of mobile GPGPU applications due to the graphic-oriented setting, and verify its effectiveness for different GPGPU applications by measuring the performance increase over the energy consumption increase, which can lead up to 1.44.

This paper is organized as follows: Section 2 describes DVFS; Section 3 mathematically models an interval-based DVFS algorithm and describes its problem; Section 4 describes our GPGPU-based DVFS strategies without sacrificing the performance of graphic applications; Section 5 implements and discusses them; Section 6 draws a conclusion.

2 Dynamic Voltage and Frequency Scaling

In order to utilize a battery for a long period of time, a mobile device should reduce energy consumption while maximizing performance. There are various technologies for such a purpose, and the dynamic voltage and frequency scaling (DVFS) is one of them. Its basic idea can be captured well by the following Eq. 1 [14]:

$$E = Pt = cV^2Ft, \tag{1}$$

where E is the consumed energy, P is the consumed power, c is a constant specific for the target device, F is the operating frequency, V is the required voltage for the frequency and t is the time period during which both the frequency F and the voltage V are maintained. The voltage, V, is not directly proportional to the frequency, F. However, as the frequency becomes higher, the required voltage also becomes higher for the device's stable operation. Thus, as the frequency becomes higher, the consumed power and energy also increase significantly higher. Thus, it is very important to find only the required frequency and voltage that an application requires, so that the consumed energy can be minimized, according to Eq. 1.

Using this simple idea, the DVFS dynamically scales up and down the voltage and frequency according to application's predicted workloads. A predicted-high workload requires high frequency to satisfy the application's performance requirements which in turn increases energy consumption.

Many DVFS algorithms to properly predict applications' workloads have been proposed originally on the CPU side. These can be largely classified into three categories: interval-based, inter-tasks, and intra-tasks algorithms [3]. Interval-based algorithms periodically measure how much busy a device is during the given time (i.e. utilization), and set the next voltage and frequency based on the current measurement of utilization. The

utilization, U_i, can be formulated by Eq. 2 when T_i is a time interval at time frame i, and the GPU spends time, w_i, for working.

$$U_i = \frac{w_i}{T_i} \tag{2}$$

When n is the window size, the next utilization U_{i+1} at time $i+1$ is predicted using Eq. 3.

$$U_{i+1} = \frac{1}{n}\sum_{j=0}^{n-1} U_{i-j} \text{ when } i \geq n-1 \tag{3}$$

The current frequency will be adjusted after the averaged utilization U_{i+1} in Eq. 3 is compared to a pre-determined threshold. As choices for a DVFS algorithm of the CPU, the current Linux implementation includes many *governors* such as Performance, Powersave, Userspace, On-demand, Conservative and Interactive. On-demand, conservative and interactive governors are interval-based algorithms, typically with $n = 1$ [15].

The DVFS algorithm is executed at every periodic DVFS event according to the sequences shown in Fig. 1. The DVFS event is periodically triggered by a kernel timer. The algorithm in Fig. 1 investigates the past utilization, and predicts the next utilization. If the predicted utilization is higher than the pre-determined threshold, then the kernel increases both the frequency and the voltage; otherwise the kernel decreases them. The pre-determined thresholds include up/down-thresholds that serve as a basis for increasing/decreasing the frequency. For utilization locating between the up-threshold and down-threshold, the kernel maintains the past frequency.

Inter-task algorithms investigate source codes, dynamically profiled the data or the hardware performance counter, and determine the frequency based on these values. The signature-based algorithm is an inter-task algorithm that maintains a global table to include signatures and an anticipated workload. A signature is mapped to a workload. As signature information for the GPU, an average triangle area, a triangle count, an average triangle height and a vertex count are used in [13]. By relying on these information, the corresponding workloads are looked up, and the voltage and the frequency are set accordingly.

The intra-task algorithms monitor a single task or a process, and determines the frequency based on it. In [6], a program's execution time is decomposed into two parts: on-chip computation and off-chip access latencies. The frequencies of a GPU and a memory are determined based on the ratio between them.

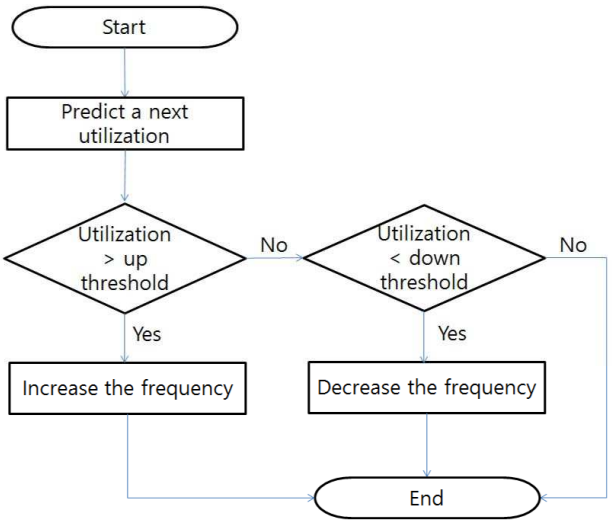

Fig. 1 Processing sequences within a single DVFS event

These DVFS technologies have been actively researched for more than 10 years mostly on the CPU side, and have been already applied to many existing CPU systems. But, there are very few DVFS researches on the GPU. Furthermore, most of proposed DVFS algorithms for GPUs use a graphics-specific information in order to adjust the GPU's frequency and voltage. We only find that the works in [3] and [12] might be possibly applied to GPGPU applications. However, both [3] and [12] focus only on saving energy, whereas our work focus on the performance increases of GPGPU applications while minimizing the energy increases. In this paper, we will use interval-based algorithms because it is most popular and included in most GPU kernel drivers.

3 Problem Formulation

This section describes a mathematical model of an interval-based DVFS algorithm for mobile GPGPU applications with $n = 1$ in Eq. 2. We propose the case of $n = 1$ because it is the simplest form of interval-based algorithms and has been most widely used in the fields. We also discuss the tracking results of GPU utilizations and frequencies with an existing interval-based DVFS algorithm.

3.1 Ideal DVFS Algorithm

In this section, we assume that the voltages and the frequencies are reversely sorted in their own sets; F_i^k represents the kth member in the sorted set of available frequencies is used at time i, and V_i^k, which is the kth member in the sorted set of available voltages.

And, the next frequency F_{i+1} is determined based on the current utilization U_i in Eq. 4 with a conservative governor that increases/ decreases the frequency by a single step. The next voltage V_{i+1} is determined by the frequency F_{i+1}. In Eq. 4, H_D and H_U are the down and up thresholds for decreasing and increasing the frequency. In other words,

$$F_{i+1} = \begin{cases} F_i^{k+1}, \text{ if } U_i < H_D \\ F_i^{k-1}, \text{ if } U_i > H_U \end{cases} \quad (4)$$

In Eq. 4, the one-step higher frequency, F_i^{k-1} than the one at time i, is used at time $i+1$, and the one-step lower frequency, F_i^{k+1} is used. The final goal of DVFS is to find the best F_i that is proportional to U_i.

$$F_i \propto U_i \quad (5)$$

In this ideal case as in Eq. 5, a higher GPU utilization at time i guarantees a higher frequency at the time. If so, the energy is not unnecessarily consumed by the application. However, an interval-based algorithm guarantees only Eq. 6, instead.

$$F_i \propto U_{i-1} \quad (6)$$

In the above Eq. 6, a higher utilization at time i does not guarantee a higher GPU frequency F_i at time i, but U_{i-1} at time $i-1$ does. Thus, the interval-based algorithm greedily approximates the ideal case of Eq. 6, which can cause some problems. We will demonstrate this problem in the next section.

3.2 Interval-based DVFS Algorithm

In order to describe a problem of the current interval-based DVFS algorithm with $n = 1$, we use an application called SmallptGPU on Odroid XU3 that embeds an interval-based DVFS mechanism in it. This application is an open source ray tracing application that uses a Monte Carlo technique, and calculates every pixel through OpenCL and draws the calculated pixels into a screen as shown in Fig. 2. The Odroid XU3 will be described more in Section 5.

We plot the history of utilizations and frequencies for 30 DVFS events in Fig. 3 while running the SmallptGPU. By default, H_D is 60, and H_U is 85 in this experiment. For a fair comparison with normalized utilization, the current frequency is normalized to 100%, divided it by the maximum frequency.

In Fig. 3, the blue line is the utilization, and the red line is the frequency. In this experiment, the tracking result of the red line is not matched well with that of the blue line. The frequency changes follow the utilization changes one step behind. The Fig. 3 means that a high frequency is not used even when the application needs a high performance. The high performance with low energy consumption can be expected if we can match both lines. The next section will describe our new strategies and a new algorithm to address these utilization changes.

4 DVFS Strategies for Mobile GPGPU Applications

Graphics applications should display their contents at an interactive rate to users (i.e. 30 FPS $\sim$ 60 FPS). This means that the GPU should finish drawing at least within 16.67 milli-seconds $\sim$ 33.33 milli-seconds. This again means that there are no big differences to users if the GPU finishes its drawing within 1 milli-seconds or 33.33 milli-seconds. But, GPGPU applications do not have such loose time limitations. It is always good that GPGPU applications finish their calculations as early as possible, which is the main purpose of using GPU for GPGPU applications in the first place.

To maximize the performance of GPU, we may simply set the maximum frequency while running GPGPU applications. However, some applications are memory-intensive inside their OpenCL kernels. Therefore, the GPU's maximum frequency may act less effectively to increase the performance. In this case, the memory frequency should be also adjusted for maximum performance. In addition, most graphic applications have both graphic and computing tasks interwined and each core can process a different type of tasks. Setting the maximum frequency for GPGPU-processing cores causes unnecessary energy consumption for graphic-processing cores because per-core DVFS is not widely available for a mobile GPU, yet.

This section describes our four novel DVFS strategies and a new integrated algorithm to increase mobile GPGPU performance. Each strategy is proposed based on the following problems of the earlier graphics-centric DVFS techniques. First, the frequency is set very low in the beginning of DVFS time interval when applications start executing OpenCL kernels, so the GPU processes tasks poorly. Second, the thresholds for GPGPU tasks are determined by graphic applications, which may have totally different characteristics from GPGPU. Third, slow frequency changes can not catch up with abrupt utilization changes.

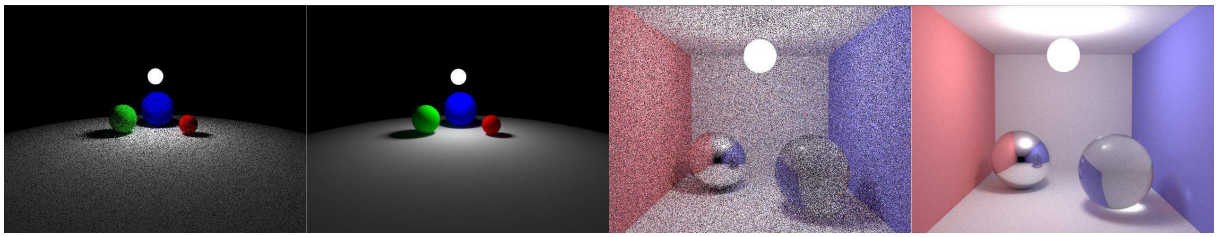

Fig. 2 SmallptGPU application

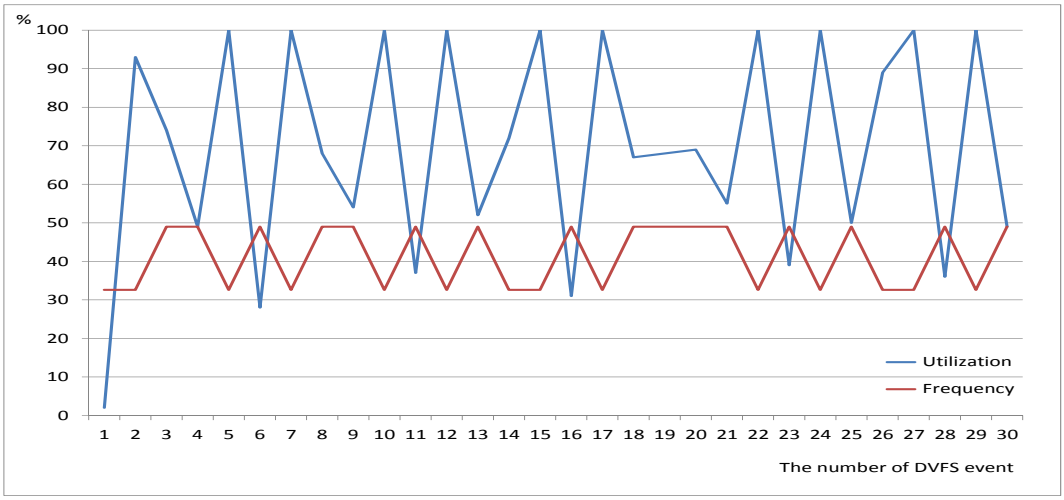

Fig. 3 The changes of utilizations and frequencies of SmallptGPU over 30 DVFS events

4.1 New DVFS Programming Interfaces

When an application tries to execute an OpenCL kernel, the GPU frequency is very low in many cases because it might be in an idle state or has only light-weight graphic tasks. So, we divide the applications' phase into a normal phase and a GPGPU phase. The normal phase represents a status that the GPU has no computing tasks assigned but graphic tasks still may be present, and the GPGPU phase represents the opposite. During the normal phase, the conventional interval-based DVFS proceeds. In case of GPGPU phase, the maximum frequency is set, called by our new DVFS application programming interfaces (APIs). After the GPGPU phase ends, the conventional DVFS resumes back.

In this strategy, we assume that applications know when the GPGPU phase is initiated because they should explicitly call *clEnqueue-* functions in OpenCL. In order to set the maximum frequency by calling these functions, we define a new C-like data type as follows.

```
typedef struct {
int voltage;
int frequency;
} type_frequency;
```

The above data structure is used for delivering available frequencies and voltages to user-side applications.

We now define two new APIs for user-side host applications: *GetFrequencyList* and *SetFrequency*. The former function gets a list of available frequencies, and the latter one temporarily sets the given value as a current frequency. We need these functions because every device has a different set of frequencies, so supported frequencies need to be searched first. The prototype of GetFrequencyList function is given below.

```
cl_int GetFrequencyList(int *num_frequencies,
                        type_frequency *pt_frequencies)
```

The above GetFrequencyList function retrieves the currently available frequencies. This function returns zero if there are no errors reported. Applications can set the frequency with the next function.

```
cl_int SetFrequency(int frequency)
```

The above SetFrequency function sets the given value by the user as a new frequency only for a single time interval. This function sets the frequency, and returns zero if there are no errors reported.

4.2 Weighted Thresholding

If only a single set of thresholds is available regardless of the presence of graphic or GPGPU tasks, it makes sense that graphic applications have a higher priority than GPGPU applications to determine the thresholds, H_U and H_D, mainly because the number of existing graphic applications in the fields is much higher than that of GPGPU applications. However, if there are more sets available for H_U and H_D, one can decide them more intelligently by executing and monitoring a sequence of GPGPU applications. In this case, the GPU's kernel driver uses a default set of thresholds when graphic tasks are processed, and switches to a new set of thresholds when computing tasks are processed. By adding new H_U and H_D for computing tasks, influences to graphic applications can be minimized because no modifications are made to the thresholds of graphic applications.

However, an application often uses OpenGL ES and OpenCL at the same time, and graphic and GPGPU tasks are generated at the same time. In this case, appropriate thresholds should be selected. For this, we use a weighted-threshold strategy like Eq. 7 as a new threshold.

$$H = \frac{g_i}{T_i} H_g + \frac{c_i}{T_i} H_c \tag{7}$$

Here, g_i and c_i are the working time of graphic and computing tasks at time i respectively. Furthermore, H_g and H_c are the set of thresholds for graphic and computing tasks respectively, which are determined after monitoring tasks. Using Eq. 7, the final thresholds are determined in proportional to the amount of working time of each task.

4.3 Adaptive Interval Adjustment

Generally, the workloads of GPGPU applications are difficult to accurately predict simply because the target GPGPU applications can be very diverse. In order to effectively adapt to these various workloads, we dynamically adjust the DVFS interval based on the standard deviation of past utilization factors like the next Eq. 8.

$$T_{i+1} = \frac{\sigma_{max}}{k\sigma} T \text{ when } \sigma = \sqrt{\frac{\sum_{k=1}^{n} (U_i - \mu)^2}{n}} \tag{8}$$

In Eq. 8, we record the past utilizations, and calculate the standard deviation of them, σ, at every event. The DVFS interval, T_{i+1}, is adjusted based on σ and the default DVFS interval, T. As the standard deviation becomes larger, the DVFS interval gets smaller. However, the downside of this approach is that many GPU events may be generated, which can deteriorate the overall system's performance. On the other hand, a very long interval results in slow responses of DVFS to workload changes. So, we clamp the DVFS interval into a fixed range. The variable k controls the rates of the length change of the interval. For instance, if a large value is used for k, the interval size is reduced quickly and reaches the minimum interval soon.

In case of quick utilization changes, the DVFS interval is shortened so that the DVFS algorithm can increase or decrease the frequency more responsively according to the utilization. In case of slow utilization changes, the DVFS interval increases under the assumption that the system is in a stable state, and both the utilization and the frequency are checked less often.

4.4 Multi-level Frequency Adjustment

Besides workload changes, another reason why an interval-based algorithm can not catch up with utilization changes is that it increases/ decreases the frequency only by a single level. In order to resolve this issue, we propose the use of multi-level frequency update based on the past utilization history as shown in Eq. 9.

$$F_{i+1} = \begin{cases} \min\{f_j | \forall j : \frac{F_i}{f_j} U_i < H_U\} \text{ if } U_i > H_U \\ \max\{f_j | \forall j : \frac{F_i}{f_j} U_i > H_D\} \text{ if } U_i < H_D, \end{cases} \tag{9}$$

where F_i is the past frequency at time i, and f_j is one of the available frequencies.

Eq. 9 finds an appropriate frequency within the range of H_U and H_D thresholds. If the utilization is higher than H_U, the minimum frequency that expects less utilization than H_U is selected. If the utilization is lower than H_D, the maximum frequency that expects higher utilization than H_D is selected. To summarize, Eq. 9 estimates the proper utilization U_{i+1} from U_i. If the estimated utilization with the corresponding frequency falls within a range of the up and down thresholds, the frequency is assumed to be appropriate for the past workload, and becomes one of the candidates for the future frequency selection. Among several candidates, one value is conservatively selected by the min or max operator in Eq. 9. Eq. 9 can increase the frequency by multiple levels if the calculated utilization is higher than the up-threshold. This strategy assumes that a frequency increase leads to a performance increase and a working time decrease, and the utilization in Eq. 9 is inversely proportional to the frequency.

The utilization is clamped into a range from 0 to 100. So, the true utilization can be higher than 100. For this special case, a small value ϵ is added to the utilization, U_i, at the beginning of this strategy. The GPGPU-Perf algorithm in the next section also shows this heuristic in more detail.

4.5 GPGPU-Perf Algorithm

All of the proposed strategies in the previous sections adjust different parameters of DVFS, and affect the performance of applications with different characteristics. An individual contribution to performance influence by each strategy will be demonstrated in Fig. 4, and will be also discussed in Section 5.2. The performance of general applications can be maximized by integrating all these strategies into a single algorithm, which we call GPGPU-Perf algorithm as shown in Algorithm 1.

Algorithm 1 GPGPU-perf algorithm

```
1: function HANDLE_DVFS_EVENT
2:     if there exist computing tasks then
3:         /* Weighted Thresholding */
4:         H = (g_i/T_i) H_g + (c_i/T_i) H_c
5:
6:         /* Adaptive Interval Adjustment */
7:         T_{i+1} = (σ_max/(kσ)) T
8:         Clamp T_{i+1}
9:
10:        /* Multi-level Frequency Adjustment */
11:        f_j = F_i
12:        if U_i ≥ 100 then
13:            U_i = U_i + ε
14:        end if
15:        /* Find the min f_j satisfying (F_i/f_j) H_U < H_U */
16:        if U_i > H_U and j > 0 then
17:            j = j − 1
18:            while j ≥ 0 do
19:                if (F_i/f_j) H_U < H_U then
20:                    break
21:                end if
22:            end while
23:        end if
24:        /* Find the max f_j satisfying (F_i/f_j) H_D > H_D */
25:        if U_i < H_D and j < n − 1 then
26:            j = j + 1
27:            while j < n do
28:                if (F_i/f_j) H_D > H_D then
29:                    break
30:                end if
31:            end while
32:        end if
33:        F_{i+1} = f_j
34:    end if
35: end function
```

When integrating our DVFS strategies into a single algorithm, the multi-level frequency adjustment, line number 10 in Algorithm 1, in Section 4.4 uses the thresholds, line number 4 in Algorithm 1, adjusted by Section 4.2. The adaptive interval adjustment, line number 6 in Algorithm 1, presented in Section 4.3 can be located before or after the weighted thresholding, line number 4 in Algorithm 1, or the multi-level frequency adjustment, line number 10 in Algorithm 1, because it has no conflicts with other thresholds or frequency. If we set the next frequency first and then set the next interval, unnecessary energy can be wasted before setting the next interval; since the GPGPU-Perf algorithm is repetitively triggered while the GPU works, the sum of energy wastes can be large, and thus we set the interval first, and then set the frequency. The interval adjustment can be located before the threshold adjustment, but it is located after the threshold adjustment because we want to minimize the inconsistency between the next frequency and the next interval.

5 Experiments and Discussions

In order to verify the effectiveness of the proposed four DVFS strategies as well as the new GPGPU-Perf algorithm, we used Android OS on the Odroid XU3 because OpenCL can be used on the board, and all the kernel and platform sources are open-sourced, so we can freely make any modifications to the system.

The Odroid XU3 device includes both A15 2.0Ghz quad core and A7 quad core as a CPU, and Mali-T628 MP6 as a GPU. By default, this GPU uses an interval-based DVFS algorithm with $n = 1$ in Eq. 2, with 100 milli-seconds as a fixed DVFS interval, and increases/decreases the frequency by a single step, namely the conservative governor in the CPU DVFS term.

We benchmarked eight GPGPU applications to measure the performance and energy consumption of our algorithms: SmallptGPU, myocyte, bfs, cfd, gaussian_elim, lud, nw and pathfinder available through the Rodinia benchmark [5]. Myocyte models and simulates the behaviors of cardiac myocyte (heart muscle cell). This application tries to solve a group of 91 ordinary differential equations. Bfs (breadth-first search) searches a graph in a parallel fashion. Cfd (computation fluid dynamics solver) is a solver of 3D Euler equations for compressible flow. Gaussian_elim (gaussian elimination) is a solver for systems of equations. Lud (LU decompsition) decomposes a matrix into a product of a lower triangular matrix and an upper triangular matrix. Nw (Needleman-Wunsch) is an optimization method for DNA sequence alignments. Pathfinder is a dynamic programming algorithm, and finds the shortest path of a 2D grid row by row [17]. The Rodinia benchmark does not include any graphic tasks, but include only computing

tasks. We port these eight applications into Android, and uses them for performance comparisons for the remaining of this paper.

5.1 Implementation Details

We implement our DVFS strategies as well as the GPGPU-Perf algorithm as follows. For the new DVFS programming interfaces presented in Section 4.1, the GetFrequencyList function reads a file from sysfs that includes the information of supported frequencies, parses the file, and gets the frequencies from it. The SetFrequency function sets a frequency also via the sysfs. We call the GetFrequencyList function before clEnqueue- functions in OpenCL, get the available frequencies, and then temporarily set the maximum frequency as a current one for a single DVFS interval. For the weighted thresholding, the adaptive interval adjustment, and the multi-level frequency adjustment presented in Sections 4.2, 4.3 and 4.4 respectively, we modify the Linux kernel so that Eq. 7, Eq. 8, Eq. 9 can determine the thresholds H_U and H_D, the interval T_{i+1} and the frequency F_{i+1}.

We set our DVFS parameters as follows. H_D is set to 60, and H_U is 85. As new thresholds for Eq. 7, 30 is experimentally chosen and set for H_D, and 70 is for new H_U. In Eq. 8, n, k, the min-interval and the max-interval are set to 3, 2, 50 milli-seconds and 200 milli-seconds, respectively. As an added value presented in Section 4.4, ϵ, 200 is experimentally chosen.

In order to measure the improvements by our strategies, we use the following three metrics: a performance increase (I_P), an energy increase (I_E) and the ratio (R) of the performance increase and energy consumption increase.

The performance increase, I_P, is calculated via Eq. 10 when T_I is the execution time of the original interval-based algorithm, and T_G is the reduced execution time by our algorithms.

$$I_P = \frac{T_I}{T_G} \tag{10}$$

The energy increase, I_E, is calculated via Eq. 11 where E_I and E_G are the consumed energy of the original interval-based and our algorithms respectively.

$$I_E = \frac{E_G}{E_I} \tag{11}$$

We also use the next metric, R, for checking the performance increases over the energy consumption increase via our algorithms.

$$R = \frac{I_P}{I_E} \tag{12}$$

Eq. 12 is the ratio of the performance increase and the energy increase of our algorithm over the original interval-based DVFS. The ratio of greater than one indicates that the new algorithm improves performance while saving more energy than the original interval-based DVFS algorithm.

5.2 Results and Analysis

Using the implementations and metrics proposed in the previous section, we repeat 30 iterations of the Smallpt-GPU application, and its execution times for an OpenCL kernel are averaged. We run the Rodinia applications, and measure their execution times through the *clGetEvent-ProfilingInfo* function. Through these execution times, the performance increase rate can be calculated using Eq. 10. For the energy increase rate, we get additional messages from the Linux kernel. These messages include the used frequency, voltage and utilization as well as the operating time with them. From these messages, we can mathematically calculate the consumed energy using Eq. 1. Through these consumed energies, the energy increase rate can be calculated using Eq. 11. Table 1 summarizes the results.

Table 1 Performance Improvements and Energy Consumption Improvements using different DVFS strategies. DPI is the DVFS Programming Interfaces, WT is the Weighted Thresholding, AIA is the Adaptive Interval Adjustment and MFA is the Multi-level Frequency Adjustment. The GPGPU-Perf is our integrated algorithm.

	Performance	Energy
DPI	2.02	1.61
WT	1.13	1.04
AIA	1.00	0.96
MFA	1.05	0.94
GPGPU-Perf	2.04	1.51

In the DPI case, the kernel-processing time decreases due to the maximum frequency. In the WT case, H_U is reduced from the default value of 85 with more OpenCL tasks because of the effects of new H_U (70 in this case) for computing tasks, and so is H_D from the default 60 because of the effects of new H_D (30 in this case). With these low thresholds, the frequency tends to become high and stay high, which makes the performance improvements as well as energy consumption high. In the AIA case, the kernel adjusts the interval according to the utilization changes, the applications react to

the changes more responsively. In the MFA case, the frequency increases and decreases in multiple steps depending on the past utilization. So, the performance and energy are also affected by AIA and MFA.

In the case of our integrated algorithm, GPGPU-Perf, the overall performance improve rates are 2.04 on average while the energy consumption rates are 1.51 on average. It has the best performance improvement compared to the individual DVFS strategy alone, and its energy increase is relatively low compared to the performance increases. The energy is not consumed as much as the performance increases because of two reasons: first, although the voltage and the frequency increase, the GPU working time is reduced as much. In the SmallptGPU case, the GPU works for 4.52 seconds with the original interval-based algorithm, whereas only for 2.03 seconds with the GPGPU-Perf algorithm. Similarly, in the cfd case, 25.02 seconds were spent with the original algorithm, but only 10.17 seconds with the GPGPU-Perf algorithm. Second, the original interval-based DVFS algorithm already uses a high frequency in many cases, and, consequently, the energy consumption does not increase significantly.

Fig. 4 shows the improvement ratio R, defined by Eq. 12, of each application in more details. The horizontal axis in the graph shows tested applications, and the vertical axis shows the ratio R. When R is 1, I_P is equal to I_E, which means that the performance increase rate is equal to the energy increase rate, and the same performance over energy as the original interval-based algorithm; the performance over energy is a processing capability with the same energy, and it represents the effectiveness of an algorithm.

In Fig. 4, the GPGPU-Perf algorithm has a good ratio of R in most cases such as SmallptGPU, bfs, cfd, gaussian, lud, nw and pathfinder. Especially, the performance increases almost double in most cases such as SmallptGPU (2.00 times), myocyte (1.37 times), bfs (2.17 times), cfd (2.51 times), gaussian (3.12 times), lud (1.50 times), nw (2.58 times) and pathfinder (1.07 times). In addition, the GPGPU-Perf algorithm complements different strategies. When a single DVFS strategy is used, the ratio becomes worse in some cases. For examples, the DPI is the best in the cfd and the lud cases, nevertheless, it is the worst in myocyte case. The AIA has a higher ratio than one in the nw and bfs cases, but it has a lower ratio than one in the lud, gaussian and pathfinder cases. Moreover, the WT has a higher ratio than one in the SmallptGPU and bfs cases, but it has a lower ratio than one in the cfd and pathfinder cases. However the GPGPU-Perf algorithm complements each individual DVFS strategy, and shows the best average ratio as summarized in Table 1. Overall, the ratios of the computation-intensive applications (cfd, gaussian, lud [5]) are remarkably increased by the DPI and GPGPU-Perf.

As a reason for the low ratio in the myocyte case, this particular application repetitively executes a short OpenCL kernel with a few memory operations. Although the DPI and the GPGPU-Perf algorithms set the maximum frequency in the GPU, memory operations inside the OpenCL kernel may prevent the performance enhancement. Accordingly, the GPU's maximum frequency does not improve the performance. In the pathfinder case, all of our DVFS strategies do not affect the performance and the energy. This application has extreme GPU workloads, and the maximum frequency is already set in the very beginning with the original interval-based DVFS algorithm. So, the performance and the energy are not affected by any strategies.

6 Conclusion

When GPGPU applications are executed with a DVFS algorithm on mobile devices, it is difficult to obtain the maximum performance of GPU since graphic applications requiring relatively low performance determine the DVFS parameters such as thresholds, an interval and a frequency. In order to resolve this problem without any influences to existing graphic applications, we suggested four DVFS strategies: new DVFS programming interfaces for executing an OpenCL kernel, weighted thresholding based on working times, adaptive interval adjustment based on utilization changes and multi-level frequency adjustment based on previous utilization. We proposed the GPGPU-Perf algorithm by integrating all the four strategies. We experimentally verify their effectiveness using various sets of benchmarks. We showed that each strategy has its own advantage, and the integrated GPGPU-Perf algorithm complements each strategy, and improves the performance 2.04 times with energy consumption 1.51 times via intelligent frequency controls.

Acknowledgements This work was supported in part by NRF in Korea (2012R1A2A2A01046246, 2012R1A2A2A06047007, 2014K1A3A1A17073365) and MCST/KOCCA in the CT R&D program 2014 (R2014060011). Young J. Kim is the corresponding author.

References

1. Altantsetseg, E., Muraki, Y., Matsuyama, K., Konno, K.: Feature line extraction from unorganized noisy point clouds using truncated Fourier series. The Visual Computer **29**(6-8), 617–626 (2013)

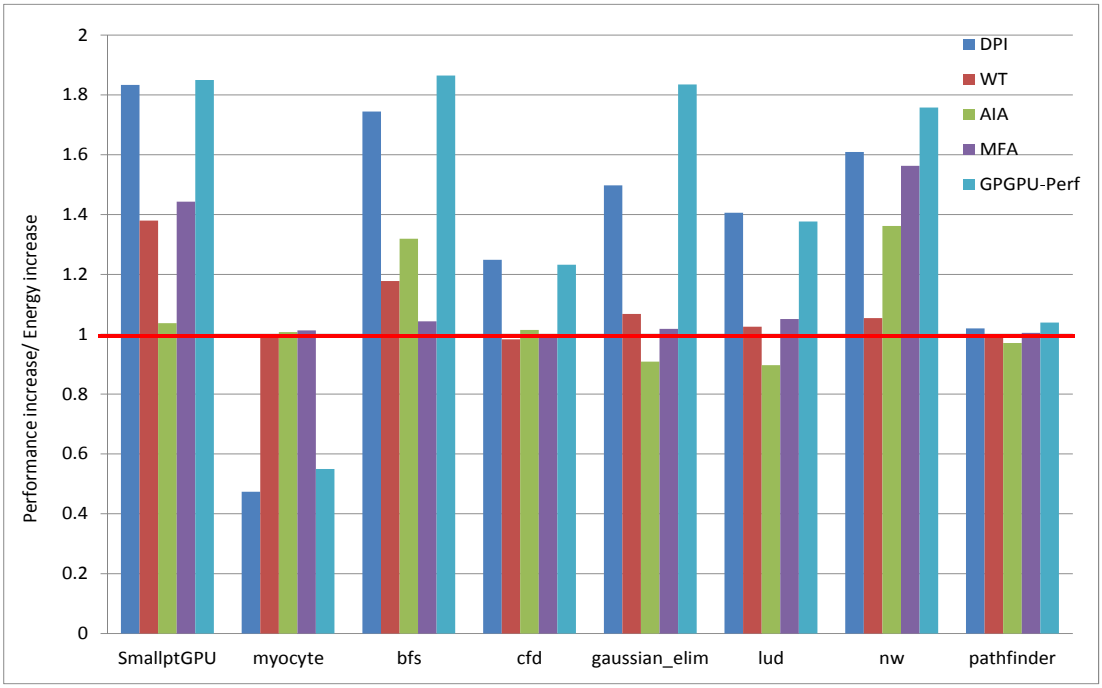

Fig. 4 The ratio of performance increase and energy consumption increase

2. Bakhoda, A., Yuan, G.L., Fung, W.W.L., Wong, H., Aamodt, T.M.: Analyzing CUDA workloads using a detailed GPU simulator. In: Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software, pp. 163–174 (2009)
3. Boyer, M.: Improving Resource Utilization in Heterogeneous CPU-GPU Systems. Ph.D. thesis, University of Virginia (2013)
4. Chang, B., Woo, S., Ihm, I.: GPU-based parallel construction of compact visual hull meshes. The Visual Computer **30**(2), 201–211 (2014)
5. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.H., Skadron, K.: Rodinia: A Benchmark Suite for Heterogeneous Computing. In: Proceedings of IEEE International Symposium on Workload Characterization (IISWC), pp. 44–54 (2009)
6. Choi, K., Soma, R., Pedram, M.: Dynamic voltage and frequency scaling based on workload decomposition. In: Proceedings of the international symposium on Low power electronics and design, pp. 174–179 (2004)
7. Huang, M., Liu, F., Wu, E.: A GPU-based matting Laplacian solver for high resolution image matting. The Visual Computer **26**(6-8), 943–950 (2010)
8. Khronos: The OpenCL C Specification Version: 2.0. Khronos Group (2014)
9. Leng, J., Hetherington, T., ElTantawy, A., Gilani, S., Kim, N.S., Aamodt, T.M., Reddi, V.J.: GPUWattch: enabling energy optimizations in GPGPUs. In: Proceedings of the 40th Annual International Symposium on Computer Architecture, pp. 487–498 (2013)
10. Liu, F., Harada, T., Lee, Y., Kim, Y.J.: Real-time Collision Culling of a Million Bodies on Graphics Processing Units. ACM Transactions on Graphics **29**(6) (2010)
11. Liu, F., Kim, Y.J.: Exact and Adaptive Signed Distance Fields Computation for Rigid and Deformable Models on GPUs. IEEE Transactions on Visualization and Computer Graphics (TVCG) **20**(5), 714–725 (2014)
12. Ma, K., Li, X., Chen, W., Zhang, C., Wang, X.: GreenGPU: A Holistic Approach to Energy Efficiency in GPU-CPU Heterogeneous Architectures. In: Proceedings of International Conference on Parallel Processing, pp. 48–57 (2012)
13. Mochockitt, B.C., Lahirit, K., Cadambit, S., Hut, X.S.: Signature-based workload estimation for mobile 3D graphics. In: Proceedings of Design Automation Conference, pp. 592–597 (2006)
14. Orgerie, A.C., Assuncao, M.D.d., Lefevre, L.: A survey on techniques for improving the energy efficiency of large-scale distributed systems. ACM Computing Surveys **46**(4), 47:1–47:31 (2014)
15. Pallipadi, V., Starikovskiy, A.: The ondemand governor: past, present and future. In: Proceedings of Linux Symposium, vol. 2, pp. 223–238 (2006)
16. Rister, B., Wang, G., Wu, M., Cavallaro, J.R.: A fast and efficient sift detector using the mobile GPU. In: Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), (2013)
17. Shen, J., Varbanescu, A.L.: A Detailed Performance Analysis of the OpenMP Rodinia Benchmark. Tech. Rep. PDS-2011-011, Delft University of Technology
18. Xinxin, M., Ling, S.Y., Kaiyong, Z., Xiaowen, C.: A measurement study of GPU DVFS on energy conservation. In: Proceedings of the Workshop on Power-Aware Computing and Systems (2013)