International Journal of Engineering & Technology, 7 (3) (2018) 1918-1925

International Journal of Engineering & Technology

Website: www.sciencepubco.com/index.php/IJET
doi: 10.14419/ijet.v7i3.15296
Research paper

DVFS Algorithms of GPU and Memory for Mobile GPGPU Applications: A case study

SeongKi Kim[1] and Seok-Kyoo Kim[2*]

[1]*Division of Computer Science Engineering, Keimyung University, Daegu, Republic of Korea*
[2]*Department of Game Design and Development, Sangmyung University, Seoul, Republic of Korea*
[*]*E-mail:skkim@smu.ac.kr*

Abstract

Although both OpenCL and RenderScript have allowed the General-Purpose Graphics Processing Unit (GPGPU) to be used even for mobile GPUs, it is still difficult for mobile applications to use the GPGPU for several reasons. One of the reasons is that mobile devices place restrictions on GPU performance through power-saving technologies such as Dynamic Voltage and Frequency Scaling (DVFS). DVFS tries to control the balance between performance and energy consumption based on the application's requirements. This technology has been successful in many cases and is widely used; however, it significantly decreases the performance of GPGPU applications. In this paper, we propose novel DVFS algorithms for GPU and memory when the GPGPU applications run. The suggested algorithms decreased the energy consumption by more than 0.7 times without any algorithm changes, and improved the energy efficiency (performance per watt) by more than 3.42 times in comparison with the conventional interval-based algorithm.

Keywords: *DVFS; Embedded system; GPU; GPGPU; Mobile system; OpenCL*

1. Introduction

Since GPGPU technology has been enabled through Vulkan [1], CUDA [2], OpenCL [3], and RenderScript [4], many algorithms and applications such as games and virtual/augmented reality have tried to use them to meet the performance requirements of the algorithms. The technologies are particularly useful for processing repetitive tasks with different data. The fields of image processing [5], computer vision [6], collision detection [7], and cryptography [8] include such tasks. Thus, they have been successfully accelerated in desktop environments.

However, in mobile devices, the power required for the stable operation of the CPU and GPU is supplied by batteries with limited capacity, and power saving has traditionally been a critical issue. Thus, many researchers have studied technologies for saving energy and have developed many algorithms or techniques. Dynamic Voltage and Frequency Scaling (DVFS) is one of them, and most current mobile devices use it. The fundamental idea behind DVFS is the scaling of the frequency and voltage based on estimated workloads. If large workloads are expected, DVFS increases the frequency and the voltage for achieving better performance. Otherwise, the application cannot meet its performance requirements under large workloads. If small workloads are estimated, DVFS decreases the frequency and the voltage to save energy. Otherwise, the CPU or the GPU unnecessarily consumes energy by processing the small workloads at a high frequency, and the application cannot meet the energy requirements.

Through these fundamental mechanisms, DVFS tries to save energy with a minimum performance loss. In this scenario, the misprediction of future workloads can lead to low energy efficiency (performance over watts). When the frequency increases owing to the estimation of large workloads but only low workloads are given, the running application consumes unnecessary energy. When the frequency decreases owing to the prediction of low workloads but high workloads are given, the running application cannot meet the performance requirements that the application needs.

As a result, various estimation methods have been suggested and developed, and interval-based DVFS algorithms have been widely used. These algorithms periodically measure CPU, GPU, or memory utilization, and predict future workloads through previous utilizations. If the previous utilizations are high, DVFS estimates that the future workload will be also high, and increases the frequency and the voltage. If the previous utilizations are low, DVFS predicts that the future workload will be also low, and decreases the frequency and the voltage. These types of algorithms use thresholds that are criteria for increasing/decreasing the frequency.

This interval-based DVFS algorithm works well, especially in stable workload cases. If the workloads by an algorithm or an application are consistent, estimating future workloads through previous utilizations may be the best approach. However, the workloads vary in many cases, especially for the GPGPU, because the researchers or developers can program the GPU for any general purpose. If the workloads fluctuate significantly, the workload estimation through time may become incorrect, and the GPU cannot satisfy the performance requirements or the energy requirements.

This paper contributes to research on the GPGPU and DVFS. First, we suggest novel approaches to estimate future workloads in the GPGPU case. Second, we propose a novel DVFS algorithm that controls the GPUs and the memory frequencies for the GPGPU case.

Third, we decrease the energy consumption by 0.7 times and the energy efficiency by 3.42 times on average without any changes to running algorithms.
This paper is organized as follows: Section 2 is an overview of OpenCL, DVFS, and the characteristics of the GPU; Section 3 analyzes GPGPU applications for the DVFS algorithm; Section 4 describes our algorithms; Section 5 details our implementations and describes the performance results with regard to energy consumption/efficiency; Section 6 draws a conclusion.

2. Related works

This section introduces OpenCL, DVFS, and characteristics of the GPU to understand this paper, and describes related works.

2.1. OpenCL

OpenCL is a platform-independent programming framework maintained by the Khronos Group [9]. It can program CPU cores, GPU cores, and other Digital Signal Processors (DSP) for heterogeneous computing. The Khronos Group defines a *host* and *devices* in the specification [10]; the host is the primary device that executes the main program, and the device is a target that runs the parallel parts of a program. An OpenCL device can have many compute units, and each compute unit can include many processing elements.
An OpenCL application includes a *host code* that the host executes and a *kernel code* that the device runs. A host code enqueues the kernel code to devices as commands. The compute device internally chooses the compute units and the processing elements that will execute the commands. The selected compute unit and processing element execute the commands according to the device's scheduling policy. Programmers can write kernel code with the OpenCL programming language, which is a subset of the C language [11] and has some extensions for parallelization. Developers can statically compile the code before the kernel's executions with an offline compiler, or dynamically compile it during the runtime with the provided functions.
With regard to programming, OpenCL uses the *work-item* and the *work-group* when running the kernel. The work-item can be thought of as a thread, and the work-group can be thought of as a group of threads running the same code. When a host code enqueues a command to a compute device, it specifies the global and local sizes. The global size determines the total number of work-items, and the local size determines the number of work-items in a group. Physically, a work-item runs on a processing element, and a work-group runs on a compute unit. If the number of work-items is higher than that of processing elements, then the work-items are scheduled according to the vendor's policy.

2.2. Dynamic Voltage and Frequency Scaling (DVFS)

DVFS is a technology that saves energy and is widely used, particularly in mobile devices, because energy saving is one of the most critical issues. The following Eq. 1 [12], [13] shows the fundamental idea:

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

In Eq. 1, E, P, t, c, V, and F are the consumed energy of the CPU or GPU, the power consumption, working time, a constant specific to CPU or GPU, voltage, and frequency, respectively. As the frequency increases, the voltage required for the frequency increases, and the working time decreases linearly [14]. Thus, the consumed power and energy are proportional to the square of the required voltage for the frequency. Because of the quadratic increase in energy consumption owing to the frequency increase, the optimal frequency and voltage are important to consume minimum energy.

To find an optimal frequency, many researchers have suggested DVFS algorithms, which are largely classified into three categories: interval-based, inter-task, and intra-task algorithms [15]. The interval-based algorithms periodically measure the GPU's working time and calculate utilization U_i, the working time over a period. Then, the algorithms use an averaged utilization over the past n periods to estimate a future one, U_{i+1}.
U_{i+1} is formally defined by Eq. 2, where T_i, w_i and n are the period, the GPU's actual running time, and the window size, respectively.

$$U_{i+1} = \frac{1}{n}\sum_{j=0}^{n-1} U_j = \frac{1}{n}\sum_{j=0}^{n-1} \frac{w_{i-j}}{T_{i-j}} \text{ when } i \geq n-1 \tag{2}$$

After the average of previous utilizations, U_{i+1}, is calculated through Eq. 2, it is compared with up/down-thresholds. If it is less than a down-threshold, the frequency decreases; if it is higher than an up-threshold, the frequency increases. Otherwise, the current frequency is retained.
Inter-task algorithms investigate source codes, dynamically profile the real-time hardware information, and determine the frequency [16]. The intra-task algorithms monitor a single task or a process and determine the frequency [17].
The most similar work in this research is the GPGPU-Perf [18], which is an interval-based DVFS algorithm for mixed cases of graphics and GPGPU tasks. However, it is different in that the GPGPU-Perf does not control the frequency of the memory, but only that of the GPU. It temporarily sets the maximum frequency before running an OpenCL kernel, uses a weighted thresholding based on the types of works, increases or decreases a frequency more than 1 step based on the prediction, and adjusts the DVFS interval based on the changing degree of previous workloads. In this paper, we will compare our results with the conventional interval-based DVFS and the GPGPU-Perf [18].

2.3. Characteristics of GPU

One of the features of a GPU is that all of the work-items within the same work-group run the same instruction in lockstep way. Threads of the CPU can execute entirely different parts of code, but the work-items of the GPU cannot do this if they are in the same work-group. This lockstep execution can be one of the reasons that the memory workload increases. A GPGPU application can use a large number of work-items within the same work-group, and access the memory at the same time. These simultaneous accesses will decrease the performance and the energy efficiency because the GPU should wait for the completion of memory accesses by all work-items.

3. Analysis of GPGPU applications

This section describes our analysis for improving the energy-efficiency of GPGPU applications through static and dynamic methods.

3.1. Computation/Memory intensiveness of a kernel

While a GPGPU application is running, the frequencies of the GPU and memory should be determined based on the workloads that the application will deliver in the future. If the application makes computation-intensive requests, the frequency increases in memory cannot increase performance because the GPU is a bottleneck. If the application makes memory-intensive requests, the frequency increases in the GPU cannot increase performance because the memory is a bottleneck. Both cases consume energy without any performance benefits. Although the cache in the GPU can reduce the number of access requests to the memory, the cache size is not large in a mobile GPU, and the number of concurrently working work-items can be

huge in GPGPU cases. As a result, the memory intensiveness is higher than the cache capability in many cases. Thus, it is necessary to find the memory intensiveness of a kernel for the correct frequencies of GPU and memory. To find the intensiveness, we investigate the source codes of the kernel that the GPU will execute, and then use the information to determine the frequencies.

In addition to the number of memory instructions in a kernel, the number of working processing elements is also proportional to the memory intensiveness because the processing elements execute the same instruction in a lockstep way in the GPU, and simultaneously access the memory. From this number of memory instructions and number of processing elements, we can calculate the memory intensiveness of a kernel. The following Eq. 3 formalizes this approach:

$$T_M = N_P \frac{I_M}{I_M + I_G} \tag{3}$$

In Eq. 3, T_M is the memory intensiveness of a kernel, and N_P, I_M, and I_G are the numbers of working processing elements, memory instructions, and GPU instructions, respectively. Eq. 3 indicates that the memory intensiveness of a kernel is proportional to the ratio of memory instructions within a kernel and the number of processing elements that execute the kernel. This is because the processing elements access the memory at the same time.

When running a kernel, the number of physical GPU cores (processing elements), N_C, or the number of work-items within a work-group, N_W, determines the number of working processing elements that will simultaneously run the given kernel. From these, the number of working processing elements, N_P, can be calculated by the following Eq. 4:

$$N_P = \begin{cases} min(N_C, N_W), & \text{if } N_W \neq 0 \\ N_C, & \text{otherwise} \end{cases} \tag{4}$$

In Eq. 4, the minimum value of the count of physical cores, N_C, and the number of work-items, N_W, determines the number of working processing elements, N_P. For zero work-items, the number of physical cores, N_C, determines the number of working processing elements, N_P.

Through Eq. 3 and Eq. 4, we can calculate the memory intensiveness, T_M, before running a kernel. However, the memory intensiveness can fluctuate while running the kernel. The kernel includes computation instructions as well as memory instructions. The memory intensiveness will increase when the processing elements are simultaneously running the memory instructions, but it will decrease when the computation instructions are running. Because the memory intensiveness is dynamic during execution, we do not fix the frequency but set the frequency ranges of memory based on the memory intensiveness by Eq. 3.

To determine the computation intensiveness of a kernel, and also set the ranges of the GPU frequency, we do not multiply N_P in Eq. 4 because each processing element has the ability to run separately, and its computation intensiveness does not increase proportionally to the number of processing elements. Thus, we use Eq. 5 below to get the computation intensiveness, T_G, of a kernel:

$$T_G = \frac{I_G}{I_M + I_G} \tag{5}$$

3.2. Utilization Similarity

When running a GPGPU application, a kernel is repetitively executed in many cases with different parameters because the GPGPU is mostly used in repetitive algorithms. For example, the GPU should generate an image at the global illumination through procedures including ray generation, intersection tests, and shading. It should also perform the same procedures in each region within an image in the image processing field, and detect objects within an image in the computer vision field. Running the same kernel means that similar patterns of workloads are given, and similar utilizations are repeated. To correctly analyze these repetitions of the same kernel, we measured the utilization deviation between the repetitive executions of the same kernel and compared it with the utilization deviation between the old and next utilizations that the conventional interval-based DVFS algorithm uses. Fig. 1 and Fig. 2 show the results of the GPU and memory at the Rodinia benchmark [19].

In Fig. 1 and Fig. 2, the blue box indicates the utilization deviation between successive utilizations that the traditional interval-based DVFS uses. The orange box is the utilization deviation at the same step between the successive executions of the same kernel. In all cases, the utilization deviation of successive executions is less than or equal to that of successive utilizations, which means that the similarity between successive executions is higher than that between successive utilizations. The deviation between successive executions is 35.65% lower than that of successive utilizations in the GPU case, and 23.97% in the memory case. In particular, if we remove single-kernel cases such as nw, pathfinder, and backprop, then the similarity increases to 40.59% and 25.91% in the GPU and memory cases, respectively. Based on these results, we used the utilizations from the last kernel execution to predict future utilization and to determine the next frequency.

3.3. Execution Similarity

In traditional interval-based DVFS algorithms, interval determination is necessary because a too-long DVFS interval can result in the unnecessary consumption of energy, and a too-short DVFS interval can cause the frequent measurement of utilization and unnecessary changes in the frequencies. The GPU should maintain the frequency even after finishing running the kernel because it can change the frequency only after the expiration of an interval. The following Fig. 3 illustrates these wasting times when the fixed time and the previous running time are used as intervals.

In Fig. 3, the blue and orange boxes are the average wasting times when the fixed time (100 ms) and the variable time based on the previous running time of the same kernel are used as an interval, respectively. In the fixed case, although the GPU finishes the running of a kernel, the frequency can be modified after 84.75 ms in bfs, Gaussian, lud, nw, and backprop cases on average, which consumes additional energy at the GPU. In the adaptive case, the wasting time is reduced to 10.83 ms. In the pathfinder and kmeans cases, the fixed interval has shorter wasting times when compared with the adaptive interval. The pathfinder and kmeans cases have a common characteristic in that their kernels take a long time. Owing to the long interval by the previous execution, the GPU should maintain the frequency even after the kernel finishes, and consumes more energy as a result.

These reduced times of bfs, Gaussian, lud, nw, and backprop cases occur because the execution time of a kernel is similar to the previous running time. Fig. 4 shows the standard deviation of running times of the same kernel.

In Fig. 4, the blue box and the orange box are the standard deviation of a running time from the fixed time (100 ms) and the previous running time, respectively. In all cases, the standard deviation of the previous running time is less than that of the fixed time. Fig. 4 shows that we can expect the running time of a kernel to be 87.01% more correct than that of the fixed time.

Fig. 4 shows that the operating times of the same kernel are similar to each other, and Fig. 3 indicates that the interval based on the previous running time can decrease the wasting time in the short-taken kernel cases. From these results, we used the last running time as a next interval. In the cases of long-taken working kernels, we

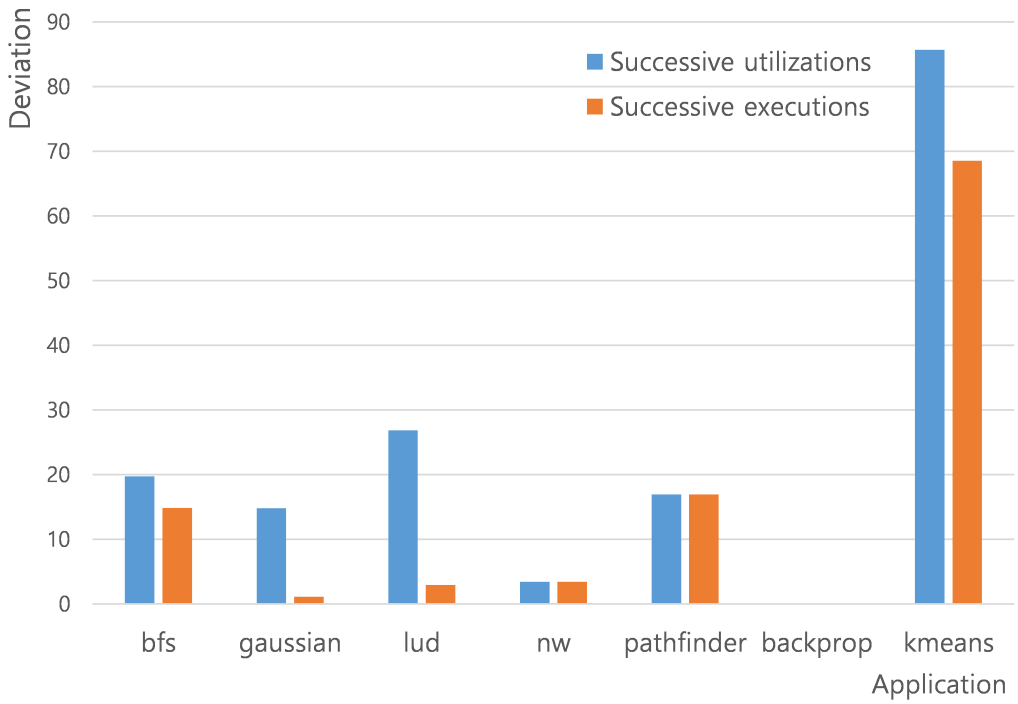

Figure 1: Utilization deviation of GPU

ALGORITHM 1: The algorithm for setting range and interval for GPU

Input: kernel name *kname*, called number of a kernel *no*, Parameter for ranging a frequency T_f, Threshold for interval-setting T_r
Output: The frequency range and DVFS interval of GPU are set

```
1  if IS_FIRST_CALL(kname, no) then
2  |   Find the number of computation instructions, I_G;
3  |   Find the number of memory instructions, I_M;
4  |   Calculate T_G through Eq. 5;
5  |   Record T_G for the kernel, kname;
6  end
7  Calculate the f_app = f_max × T_G;
8  Find i that is a minimum frequency satisfying f[i] ≥ f_app;
9  Temporarily set the frequency to f[i];
10 Set the frequency range from f[i − T_f] to f[i + T_f];
11 time = PREV_RUN_TIME(kname);
12 if time < T_r then
13 |   Set the DVFS interval to time;
14 end
```

ALGORITHM 2: Recording algorithm of running time

Input: kernel name *kname*, running time *run_time*

```
1  Record kname, run_time;
2  Remove the frequency range that is set by Algorithm 1;
```

used the default DVFS interval.

4. DVFS Algorithms for the GPGPU applications

Our DVFS algorithms are largely composed at two levels: application and GPU internal. The algorithms at the application level set the frequency ranges of the GPU and memory while running the kernel, and the algorithms at the GPU-internal level minutely control the frequencies during the execution. Algorithm 1 presents the application-level algorithm for the GPU and is called before a kernel runs at the GPU.

Algorithm 1 receives a kernel name *kname*, the called number *no* of the kernel, the range parameter T_f, and the interval threshold T_r as inputs. Algorithm 1 counts the numbers of computation and memory instructions in the OpenCL kernel to calculate Eq. 5 when a kernel is first called. Then, it tries to find a frequency proportional to the ratio of computation instructions to process the kernel in Lines 7 and 8. After finding the proportional frequency, it temporarily sets the frequency in Line 9 and sets the frequency range in Line 10. After setting the range, it modifies the DVFS interval into the previous running time of the kernel in Lines 11 to 14 if the time is less than a threshold T_r. Algorithm 1 uses the threshold when setting the DVFS interval because the DVFS interval can be too long, and then the GPU cannot handle the workload changes quickly.

Algorithm 1 temporarily sets an appropriate frequency for the given computation work, and sets a frequency range for the kernel execution. For that purpose, it uses Eq. 5 to find the ratio of computation instructions over the total instructions. When Algorithm 1 is first called, it investigates the executed codes of a kernel, calculates the rate T_G, and records it for future usage. After the first call, the rate is used to find a frequency and to set both the frequency and the range. We temporarily set the frequency because it is appropriate for the kernel, and use the frequency range from the threshold-less value to threshold-more value than the found frequency because we can expect the GPU's workload as defined by Eq. 5, but not all workloads are required during the entire execution of the kernel. Algorithm 1 also uses the previous running time as a DVFS interval because the GPU can waste the remaining time while maintaining the frequency even after finishing the kernel execution.

Algorithm 1 uses the previous running time of a kernel as a next DVFS interval and sets a frequency range. Thus, we should record the running time. Algorithm 2 is called to record a kernel name and its running time into a buffer after the execution of a kernel. It also removes the frequency range.

The algorithms for the memory are similar to Algorithms 1 and 2 but are different in that Eq. 3 and Eq. 4 are used in Line 4 of Algorithm 1.

Algorithm 3 presents the algorithms at the GPU-internal level. It is called during every expiration of a DVFS interval and controls the frequency for the GPU while running a kernel.

Algorithm 3 receives a kernel name *kname*, the called number of the

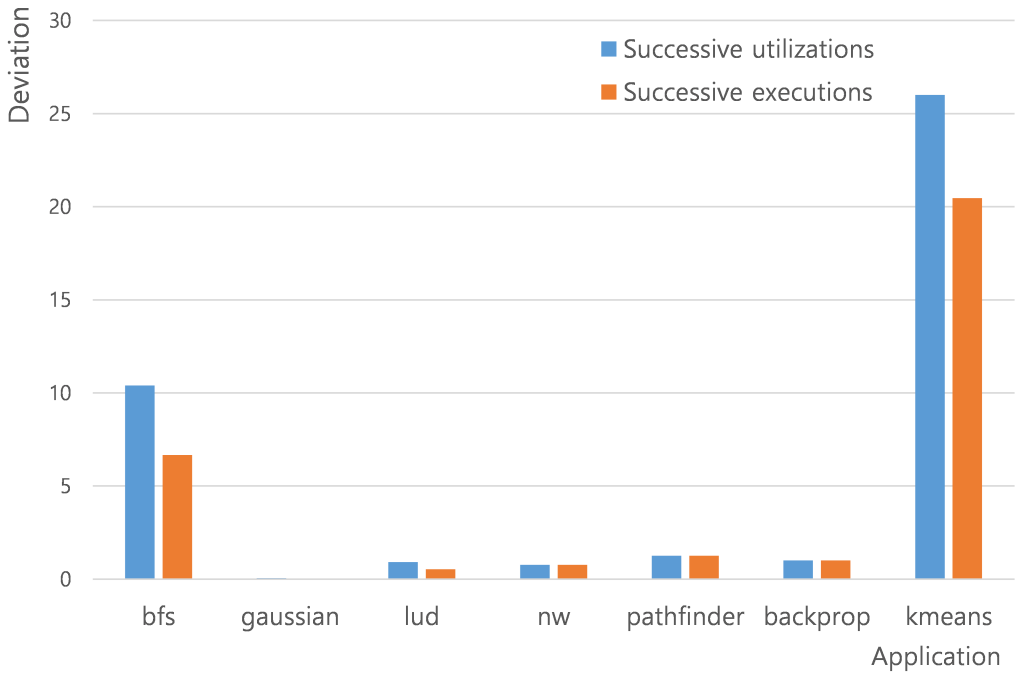

Figure 2: Utilization deviation of Memory

ALGORITHM 3: Frequency-setting algorithm for GPU

Input: kernel name *kname*, called number of a kernel *no*, called number within a kernel *co*, the current utilization *cutil*
Output: The frequency of GPU is set

```
1  if IS_FIRST_CALL(kname, no) then
2  |  Use an interval-based DVFS algorithm;
3  end
4  util = PREV_UTIL(kname, co);
5  if util > up-threshold then
6  |  Increase the frequency;
7  end
8  if util < down-threshold then
9  |  Decrease the frequency;
10 else
11 |  Maintain the frequency;
12 end
13 Record kname, co, and cutil;
```

kernel *no*, the DVFS-interval number *co* while running the kernel, and the current utilization *cutil* while running the kernel as inputs. It uses the conventional DVFS algorithm at the first call because our algorithm uses the utilizations during the previous execution, but the utilizations are not collected at the first call, and records the kernel name, the called number, and the utilization. From the second calls, it uses the collected utilization from the previous execution in Line 4 for the future determination of the frequency, and uses the up-threshold and the down-threshold for a next frequency.

Algorithm 3 uses the utilization similarity between kernel executions. Subsection 3.2 shows that the similarity between the same kernel executions is maintained more effectively than the similarity between the successive intervals, so we use the utilization of a previous kernel execution.

The frequency setting algorithm for the memory is also the same as Algorithm 3. It also uses an interval-based algorithm at the first call to a kernel, and uses the utilizations during the last kernel execution.

5. Implementation and Results

This section describes our implementations of the suggested algorithms, and the experimental results.

5.1. Implementation Details

Algorithm 1 temporarily sets a frequency and a frequency range that will be used during the kernel's execution. To set them, we should know the supported lists of frequencies. All of the devices have different frequency lists for the GPU and the memory. To obtain the lists, we define a new data type expressed in the C language as follows:

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

Then, we added the functions below to get a list of supported frequencies for the GPU and the memory.

```
cl_int GetGPUFrequencyList( int *num_frequencies,
type_frequency *pt_frequencies);
cl_int GetMEMFrequencyList( int *num_frequencies,
type_frequency *pt_frequencies);
```

We also added the functions below to temporarily set the frequency and the range of the GPU frequency.

```
cl_int SetGPUFrequency(int level);
cl_int SetGPUMinLockFrequency(int level);
cl_int SetGPUMaxLockFrequency(int level);
```

As the names indicate, the *SetGPUFrequency* function temporarily sets the frequency, the *SetGPUMinLockFrequency* function sets the minimum frequency, and the *SetGPUMaxLockFrequency* sets the maximum frequency for the GPU. We also added the functions below to set the range of memory frequency:

```
cl_int SetMEMFrequency(int level);
cl_int SetMEMMinLockFrequency(int level);
cl_int SetMEMMaxLockFrequency(int level);
```

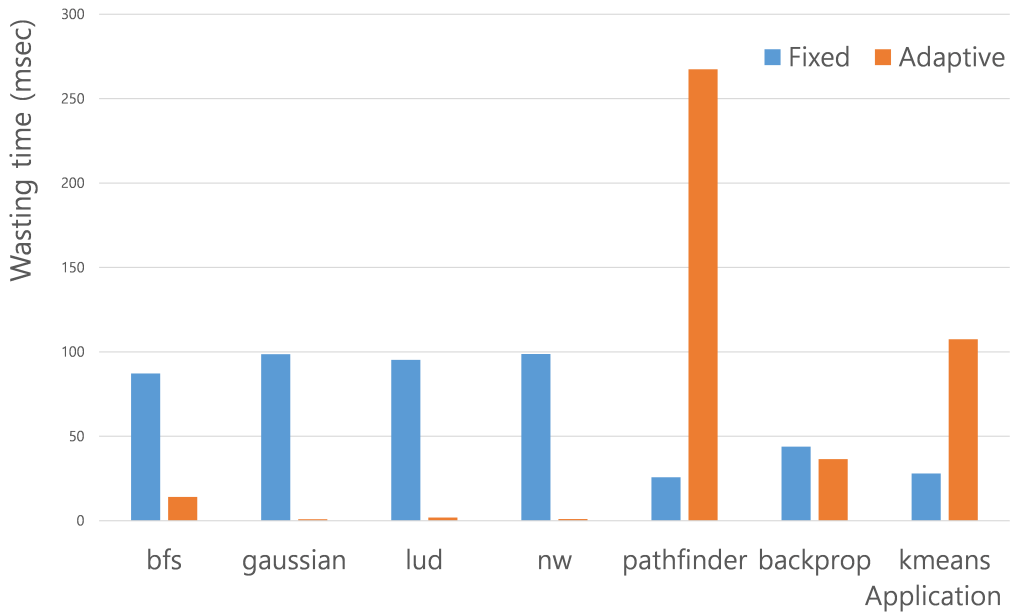

Figure 3: Wasting times by a wrong frequency setting

The functions above also do the same things as the GPU functions for the memory.
Algorithm 1 requires the numbers of the GPU and memory instructions. To find the numbers, we generate SPIR (Standard Portable Intermediate Representation) code [20] through the LLVM (Low Level Virtual Machine) [21], and count the numbers of total instructions and load/store instructions.
We implement Algorithms 1, 2 and 3 at the Odroid development board [22]. It includes Mali-T628 [23] as a GPU and INA231 [24] for the power measurement, and uses the interval-based DVFS algorithm for the GPU with the frequencies in Table 1.

Table 1: Frequencies and up/down-thresholds of the Mali GPU in Odroid

Frequencies (MHz)	Down-threshold (%)	Up-threshold (%)
177	0	90
266	60	90
350	70	90
420	78	90
480	90	99
543	99	100

The Odroid board has 2 GB, also uses the interval-based DVFS algorithm for the memory, and has the frequencies listed in Table 2.

We use the interval-based DVFS algorithms with Tables 1 and 2 only at the first call within Algorithm 3. All of these tables indicate that the DVFS algorithm increases the frequency if the one-step previous utilization is more than the up-threshold, and decreases it if the one-step previous utilization is less than the down-threshold. Otherwise, the previous frequency is maintained.

5.2. Results

As benchmarks for evaluations, we used seven GPGPU applications to measure the performance and energy consumption with

Table 2: Frequencies and up/down-thresholds of the memory in Odroid

Frequencies (MHz)	Down-threshold (%)	Up-threshold (%)
138	0	60
165	45	60
206	45	60
275	45	60
413	45	60
543	45	60
633	45	60
728	45	60
825	45	100

our algorithms: bfs, Gaussian, lud, nw, pathfinder, backprop and kmeans included in the Rodinia benchmark [19]. Bfs (breadth-first search) parallelly traverses a graph with 1,000,000 nodes, Gaussian (Gaussian elimination) solves a linear system using the Gaussian elimination method, and Lud (LU decomposition) calculates the solutions for a set of linear equations. Nw (Needleman-Wunsch) is an optimization method for DNA sequence alignments, pathfinder finds a path with the smallest accumulated weights in a 2D grid, backprop is the algorithm that trains the weights of connecting nodes, and kmeans is a clustering algorithm for data mining [25]. We ported these applications into Android and used them for performance comparisons, which are discussed in the remainder of this paper. In addition, when comparing the results, we used one as the window size in Eq. 2 because most of the recent mobile GPUs use it.

We implemented the Algorithms 1, 2 and 3 as described in Subsection 5.1, ran them after setting the parameters T_f and T_r of Algorithm 1 to 1 and 100, respectively and compared the energy consumption with the default interval-based algorithms and the GPGPU-Perf [18] in Fig. 5. When measuring the consumed energy, we measured the consumed power through the INA231 [24] inside the Odroid [22], and multiplied the time. We added the consumed energies at the GPU and memory.

In Fig. 5, the blue box indicates the increasing ratio of consumption over the interval-based DVFS algorithm, and the orange box is that over the GPGPU-Perf algorithm. If the value is less than 1 in Fig.

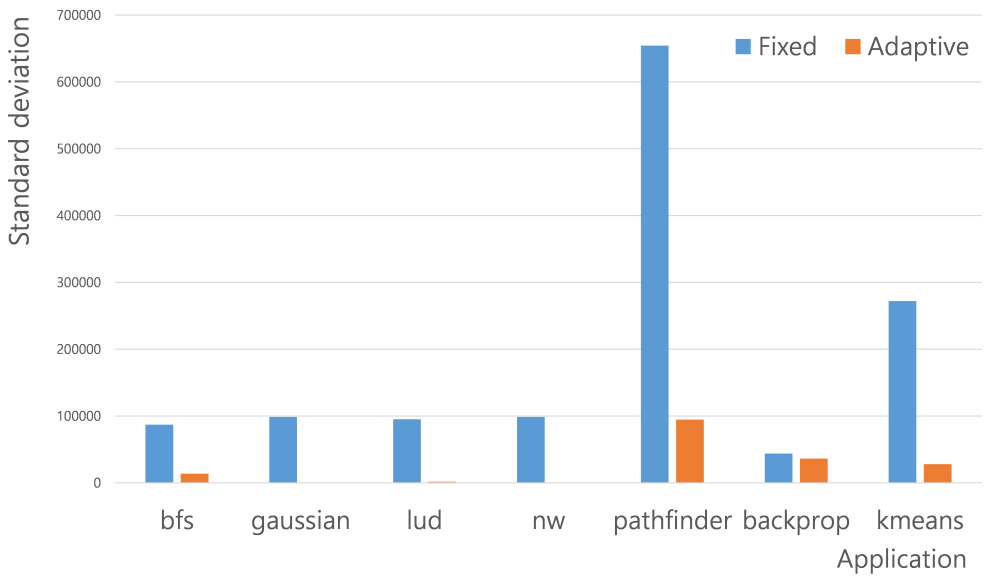

Figure 4: Standard deviation of running times

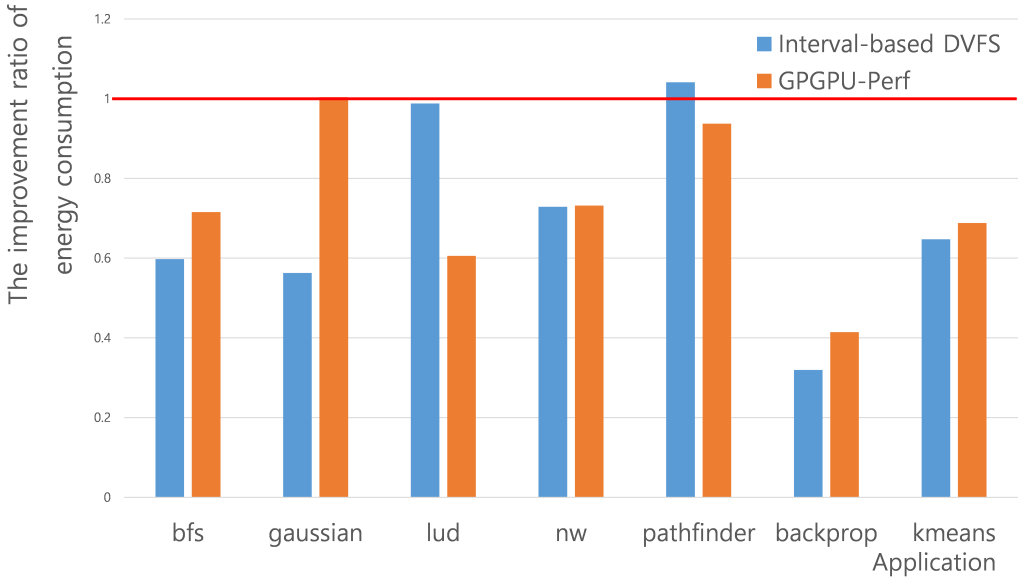

Figure 5: Improvements of energy consumption. If the ratio is less than 1 (red line), it means that the consumption decreases over the compared algorithm.

5 (red line), then the consumption decreases over the comparison targets. When compared with the conventional interval-based DVFS algorithm and GPGPU-Perf, the consumption decreases by 0.7 and 0.73 times on average, respectively.

If the suggested algorithms also decrease the performance, the decreases in energy consumption can be less meaningful. Thus, we measured the energy efficiency that is the performance increase over energy increase in Fig. 6.

In Fig. 6, the blue box indicates the increasing ratio of consumed energy over the interval-based DVFS algorithm, and the orange box is that over the GPGPU-Perf algorithm. Our suggested algorithms in this paper also increase the energy efficiency by 3.42 times, and by 1.74 times over the conventional interval-based DVFS and the GPGPU-Perf on average. Especially, our algorithms increase the energy efficiency in all of the tested cases.

6. Conclusions

In this paper, we suggest DVFS algorithms that improve the conventional interval-based DVFS algorithm. The interval-based DVFS algorithm uses a fixed interval for the utilization, an old utilization to predict the next utilization, and does not consider the memory intensiveness of a kernel. To overcome these problems, our algorithms use the similarity of execution time and the utilization similarity between each execution of a kernel. Our algorithms also investigate the kernel, approximate the computation/memory workloads, and set the frequency and the frequency range. Unlike GPGPU-Perf, our algorithms also control the frequency of the memory. Through these efforts, we increased the energy efficiency by 3.42 times and decreased the energy consumption by 0.7 times over the old algorithms.

However, our algorithms still have limitations. At this time, we cannot fully automate the processes of counting the numbers of the GPU and memory instructions. In addition, when measuring the results, we use the SPIR codes generated through LLVM because

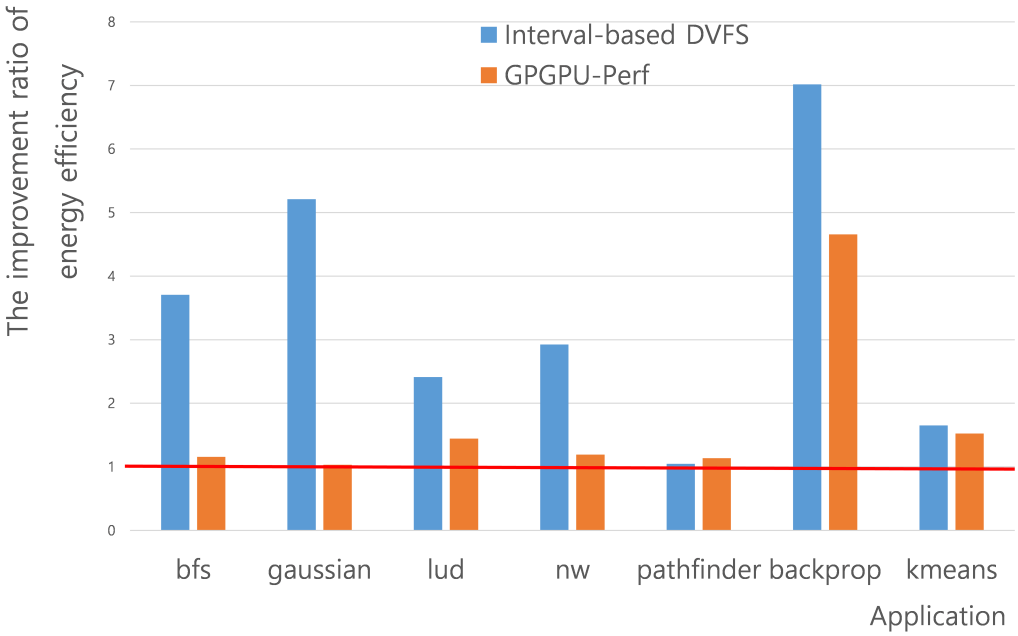

Figure 6: Improvements of energy efficiency. If the ratio is more than 1 (red line), it means that the efficiency increases over the compared algorithm

the vendor-specific codes are not available to the public, but the SPIR codes can be different from the real GPU codes. If we can obtain information about the real GPU instructions, we can measure the results more accurately. In addition, we assume no conditional branches or loops in the intensiveness-finding algorithm, which can make our model incorrect. However, the results are good in the many cases as shown in this paper. In the future, we plan to automate the counting processes of instructions from the real vendor-specific instructions, and create a new model that can cover the branches or loops.

Acknowledgement

This research was supported by NRF in Korea (NRF-2017R1A1A1A05069806). Seok-Kyoo Kim is the corresponding author.

References

[1] Sellers G, Kessenich JM. Vulkan Programming Guide: The Official Guide to Learning Vulkan. Always learning. Addison Wesley; 2016.
[2] Nickolls J, Buck I, Garland M, Skadron K. Scalable Parallel Programming with CUDA. Queue. 2008;6(2):40–53.
[3] Stone JE, Gohara D, Shi G. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. IEEE Des Test. 2010;12(3):66–73.
[4] Google Corporation. RenderScript; 2017. Available from: https://developer.android.com/guide/topics/renderscript/compute.html.
[5] Fernando R. GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics. Pearson Higher Education; 2004.
[6] Pharr M, Fernando R. GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems). Addison-Wesley Professional; 2005.
[7] Tang M, Manocha D, Lin J, Tong R. Collision-streams: Fast GPU-based Collision Detection for Deformable Models. In: Symposium on Interactive 3D Graphics and Games. I3D '11. New York, NY, USA: ACM; 2011. p. 63–70.
[8] Yang J, Goodman J. Symmetric Key Cryptography on Modern Graphics Hardware. In: Proceedings of the Advances in Crypotology 13th International Conference on Theory and Application of Cryptology and Information Security. ASIACRYPT'07. Berlin, Heidelberg: Springer-Verlag; 2007. p. 249–264.
[9] Khronos Group; 2017. Available from: https://www.khronos.org.
[10] Howes L, editor. The OpenCL Specification Version: 2.1 Document Revision: 23; 2015.
[11] ISO. ISO C Standard 1999; 1999.
[12] Tang L, Zhang Y. Low-power Task Scheduling for GPU Energy Reduction; 2011.
[13] Orgerie AC, Assuncao MDd, Lefevre L. A survey on techniques for improving the energy efficiency of large-scale distributed systems. ACM Computing Surveys (CSUR). 2014;46(4):47.
[14] Ge R, Vogt R, Majumder J, Alam A, Burtscher M, Zong Z. Effects of Dynamic Voltage and Frequency Scaling on a K20 GPU. In: Proceedings of the 2013 42Nd International Conference on Parallel Processing. ICPP '13. Washington, DC, USA: IEEE Computer Society; 2013. p. 826–833.
[15] Boyer M. Improving Resource Utilization in Heterogeneous CPU-GPU Systems. University of Virginia; 2013.
[16] Mochocki BC, Lahiri K, Cadambi S, Hu XS. Signature-based Workload Estimation for Mobile 3D Graphics. In: Proceedings of the 43rd Annual Design Automation Conference. DAC '06. New York, NY, USA: ACM; 2006. p. 592–597.
[17] Choi K, Soma R, Pedram M. Dynamic Voltage and Frequency Scaling Based on Workload Decomposition. In: Proceedings of the 2004 International Symposium on Low Power Electronics and Design. ISLPED '04. New York, NY, USA: ACM; 2004. p. 174–179.
[18] Kim S, Kim YJ. GPGPU-Perf: Efficient, Interval-based DVFS Algorithm for Mobile GPGPU Applications. Vis Comput. 2015;31(6-8):1045–1054.
[19] Che S, Boyer M, Meng J, Tarjan D, Sheaffer JW, Lee SH, et al. Rodinia: A Benchmark Suite for Heterogeneous Computing. In: Proceedings of IEEE International Symposium on Workload Characterization (IISWC); 2009. p. 44–54.
[20] Khronos. SPIR 1.2 Specification for OpenCL; 2013.
[21] Lattner C, Adve V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization. CGO '04. Washington, DC, USA: IEEE Computer Society; 2004. p. 75–88.
[22] Hard Kernel. ODROID-XU3; 2014. Available from: http://www.hardkernel.com/main/products/prdt_info.php?g_code=G140448267127.
[23] ARM. Mali-T628; 2014. Available from: http://www.arm.com/products/multimedia/mali-performance-efficient-graphics/mali-t628.php.
[24] Texas Instruments. INA231; 2016. Available from: http://www.ti.com/product/INA231.
[25] Shen J, Varbanescu AL. A Detailed Performance Analysis of the OpenMP Rodinia Benchmark. Delft University of Technology;. PDS-2011-011.