Low-power Task Scheduling for GPU Energy Reduction

Li Tang, Yiji Zhang
Department of Computer Science and Engineering, University of Notre Dame

Abstract—Graphics processing units (GPU) have been intensively used by high-performance computing applications. However, GPU’s large power consumption is a big issue coexisting with the high parallelism. Although Dynamic Voltage and Frequency Scaling (DVFS) [1] has been heavily studied and successfully applied to real products for saving CPU power consumption, DVFS is still relatively new for GPU energy studies. The lack of DVFS and other power management schemes on the GPU also makes its large power consumption significant in recent computer systems. In this Operating System course project, we propose a low-power scheme for GPU energy reduction. This project can be decomposed to GPU DVFS implementation and GPU linear regression power model. For enabling the DVFS on GPU, the settings of voltage and frequency in open-source Nouveau GPU driver [7] has been studied. For supporting the DVFS choosing voltage and frequency levels, a GPU linear regression power model has been built and evaluated.

I. INTRODUCTION

With the continuous technology scaling, traditional single-core architecture is not able to help people hold Moore’s Law [4]. Therefore, multi-core even many-core architectures are deployed to further improve the processor performance. GPU, a typical single instruction multi data (SIMD) architecture, possesses the capability of containing more than 500 processing cores [6]. The large number of processing cores permits extremely high parallelism for some applications. However, unfortunately, these cores are inevitable to bring us high power consumption. So it is very imperative to develop a efficient power management on GPU for reducing the GPU power consumption. This is also the motivation of this project.

Power management techniques can be sorted into static technique and dynamic technique [15]. DVFS is a dynamic power management technique in computer architecture and a method to provide variable amount of energy for a task by scaling the operating voltage/frequency. The basic motivation for dynamic voltage and frequency scaling is captured by the equations below:

\[ P = \alpha \times CV^2F \]  
\[ E = P \times t = \alpha \times CV^2F \times t \]

From these equations, the conclusion we can draw is that simply lowering the frequency can not necessarily lower the power, because \( F \times t \) is a constant value for a given task. Lowering the voltage and the frequency simultaneously is the way how DVFS efficiently saves the power consumption. Therefore, the settings of both voltage and frequency are both very necessary.

There are many chips in market can support DVFS. For instance, chips produced by Intel support SpeedStep [8], and chips by ARM [3] support IEM (Intelligent Energy Manager) and AVS (Adaptive Voltage Scaling) etc. But it is not enough to implement DVFS only by chip support, the design of software is also necessary. A typical workflow of DVFS system includes collection of current system load, prediction of the system performance within next period and settings of new voltage and frequency levels. By following this general process, we propose a simple low-power GPU DVFS scheme (shown in Figure. 1) in the operating system level.

The first challenge is to implement the dynamic scaling of voltage and frequency on GPU. Because the official GPU driver provided by NVIDIA is closesource and not able to allow us change the voltage and frequency levels. The first contribution in this project is the study of settings of voltage and frequency levels by using opensource Nouveau driver for enabling DVFS on NVIDIA GPUs. The second contribution is the establishing of GPU linear regression power model. This power model is used to predict the GPU average power used by the DVFS. The input of the power model is a set of performance counters and resource usage information of one targeting GPU application. The output of the power model is the predicted average GPU power of that CUDA application. Since the performance counters can be obtained by analyzing the GPU applications, the predicted power from static analysis can help the dynamic DVFS determine the suitable voltage and frequency levels for improving the GPU power efficiency before running that GPU application.

The rest of the paper is organized as follows. Section 2 gives some existing work related to GPU DVFS and GPU
power modeling. Section 3 presents the DVFS background. In section 4, we show some basic designs on the GPU DVFS. For supporting the GPU DVFS implementation, a GPU linear regression power model is presented in section 5. After that, the model has been evaluated in the section 6. We also draw the conclusion in section 7.

II. RELATED WORK

Comparing to new GPU DVFS, CPU DVFS is well-studied. B. Lin et al in [1] develop and evaluate two new, independently-applicable power reduction techniques for power management on processors DVFS: user-driven frequency scaling (UDFS) and process-driven voltage scaling (PDVS). The main contribution of these techniques is that UDFS leads to lower average frequencies and PDVS allows great decreases in voltage at low frequencies. Y. Liu et al in [2] propose design-time thermal optimization techniques for embedded systems by carefully planning DVFS at design time. The main contribution is that they provide a framework for system designers to determine a proper thermal solution and provide a lower bound on the minimum temperature achievable by DVFS. In this project, the main goal is to crack the settings of voltage and frequency on our NVIDIA GPU.

For the GPU power model, X. Ma in [10] presents a statistical power analysis and modeling scheme for GPU-based computing. The model can estimate the near future runtime power by dynamically obtaining power consumption, runtime performance, and dynamic work-loads of GPUs. The main contribution of the scheme is that the scheme can tell users the power consumption estimation during GPU runtime, and it can be used for dynamically optimization. H. Nagasaka in [13] presents a statistical approach for estimating power consumption of GPU programs. The main difference between this work and [10] is that this estimation only works after the execution of GPU programs. In this project, we use a method similar with [13] to predict the GPU power for supporting the GPU DVFS. The difference is we only consider four main GPU components which are instruction units, global memory, register files and shared memory for building the GPU power model.

III. DVFS BACKGROUND

A. DVFS implementation on CPU

There are many DVFS implementation on CPU, which can be regarded as reference when considering implementing DVFS on GPU. The implementation can be based on software or hardware. Software-based implementation of DVFS is done as follows. First, hooks are placed in kernel system calls of the operating system to collect the information of system call and thereby calculate the system load. For example, Scheduler and Timer are the two most common places for hooks to be placed, which are used to record the execution time of each task, and how long the time is when a task actively takes rest, respectively. Second, prediction of the system load in the next period of time should be made using the current system load and specific algorithm. As for hardware-based implementation of DVFS, it improves the accuracy of system load calculating. In addition, it also reduces the burden of CPU for keeping track of system load and performance prediction. The drawback is that it is not flexible to choose prediction algorithm, and this can be made up to some extent by setting different prediction parameters. Taking i. MX31 by Freescale Semiconductor Company as an example, Figure 2 shows the DVFS hardware mechanism in i.MX3.

In Figure 2, CCM stands for Clock Control Module, which is responsible for adjusting CPU frequency. PMIC is Power Management IC, which provides CPU voltage. The chip has two interfaces for CPU: the common Serial Programmable Interface (SPI) and Dynamic Voltage Scaling (DVS). The later interface consists of two lines, and different states of line indicate different voltage levels: 00 means no change on voltage; 01 means that the voltage increases by one unit; 10 means decreasing by one unit; and 11 means the voltage increase to maximum.

B. DVFS implementation on GPU

According to publicly available documentation [9] on NVIDIA’s implementation of DVFS, there are two predefined performance levels for their GPUs: Idle and MaximumPerformance. A third performance level HDVideo is designed for mobile GPU devices. The MaximumPerformance setting fixes the clock speeds to the highest setting to achieve the best possible performance. In the Idle setting, on the other hand, when the driver detects that the GPU is idling, it will automatically lower down the frequencies of the GPU cores and memory to the pre-defined Idle level, which is just enough for the display to work. Once the driver sees a computational kernel that is ready to be executed, the driver will increase the clocks up to the highest level set by user. NVIDIA provide frameworks [9] that allow the upper limit on the frequency and voltage to be
scaled by users. In particular, the core clock of the NVIDIA GTX 280 can be set between 300 MHz and 700 MHz while the memory clock can be set to any value between 300 MHz and 1200 MHz.

IV. DVFS DESIGN

A. Theory

A typical DVFS procedure has five steps. Step 1 collects the signals which are related to system load, and calculate the system load. This process can be implements by software or hardware. Step 2 predicts the system performance of next period based on current system load. There are different algorithms that can be applied in this step, and the choice of algorithm should be decided by the application. Step 3 transfers the performance to frequency, and adjusts the setting of frequency. Step 4 calculates the voltage by the new frequency. Step 5 resets the voltage and frequency. The whole workflow is shown in Figure 3

B. Design

Power management has been partly implemented in Nouveau, which is shown in Figure 4. There are two important structures which are Performance Level Tables and Voltage Tables. Performance Level Tables should contain the clocks (frequencies) need to be set, the specific voltage levels, and the fanspeed levels. On more recent GPU cards, they also contain the indices of the memory timings entries that should be used. Nouveau prints performance levels in the kernel log during loading. For the Voltage Tables, they contain a voltage label and the GPIO that should be set in order to achieve that voltage level.

Voltage adjusting is accomplished in two parts. First, the voltage can be read/written using the GPIO engine. Second, a control via onboard i2c chip should be implemented. For getting and setting different voltage levels, GPIO engines are used to activate the hook, as shown in Figure 5.

Getting and setting voltage are done by looking up the vid which is corresponded to each level of voltage. This step is implemented by the code shown in Figure 6 and Figure 7. Implementing a control via onboard i2c chip is left to be future work. After this is done, the experiment of getting and setting voltage externally can be really conducted.
C. Experiment

The configuration of the experiment is shown in Table II. In this project, Nouveau is chosen to be the driver. It uses reverse engineering techniques to provide a high-quality, opensource driver for NVIDIA GPU cards.

One way to verify voltage and frequency setting is to make performance levels printed by the following command. Nouveau prints these values in the kernel log file during loading process by using `available performance level` flag with `dmesg` command.

Figure 8 shows an example of printing performance levels. The output shows the values of several frequencies, voltage and fanspeed levels. These results can be used to check whether the value of frequency and voltage are set right.

V. GPU Power Analysis

A. GPU Architecture

The power of GPU comes from the SIMD architecture that many processing elements execute one instruction simultaneously. Latest NVIDIA’s Fermi GPU architecture (shown in Figure 9) usually consists of hundreds of Compute Unified Device Architecture (CUDA) cores, of which thirty-two are organized into one Streaming Multiprocessor (SM). All the CUDA cores in one SM execute the same instruction. One thread in CUDA program will be mapped to a physical CUDA core. For the threads inside one SM, each one has limited private register files. The threads in one SM can access a block of shared memory which enables the communication between these threads. Since the register files and shared memory are on-chip memory resource, they are fast but limited by small sizes. Therefore the GPU cards usually are equipped with more than one gigabytes external off-chip DRAM as the global memory that can be accessed by the threads from all the SM cores. The GPU DRAM subsystem is optimized for block data accesses by coalescing some consecutive accessing requests if they have closing addresses. The data loads and stores to DRAM in the GPU card can be configured to be cached in L1/L2, and 400-800 cycles will be paid for the ones whose requesting data is not cached in L1/L2.

For our experiments, we have used another high-end NVIDIA GeForce GTX 570 card for building the power model. Because this card is more power hungry than NVIDIA GeForce 210 [6]. GeForce GTX 570 belongs to the latest Fermi family and supports the Compute Capability up to 2.1. The detailed specifications are in Table II.

B. Programming Model: CUDA

CUDA [5] is an acronym for Compute Unified Device Architecture. CUDA could be considered as the extension of c language that allows the developers using GPU resource for computation. It brings the concepts of kernels, threads, thread blocks and grids. These concepts can describe be mapping relationships between CUDA programs and physical GPU resource. A GPU kernel is an invoking function happening in GPU. A CUDA thread is mapped to a CUDA core that can access its own registers, shared memory or global memory for executing instructions. A thread block containing several threads can be mapped to a SM core. One SM core is able to have multiple blocks serially run on itself. For the data in the host memory space, CUDA provides kernels to deliver the data to the global memory (On-board DRAM). The functions for data motion from global memory to host memory are also provided. A group of threads running on a single SM construct a thread block. A CUDA grid represents all the thread blocks that are launched for executing one function. In order to use CUDA to program on GPU, the system has been configured as following Table III.
### Table III
**SYSTEM CONFIGURATIONS.**

<table>
<thead>
<tr>
<th>Operating System</th>
<th>CentOS 5.6 64b</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>4.1</td>
</tr>
<tr>
<td>CUDA</td>
<td>3.2</td>
</tr>
</tbody>
</table>

#### C. GPU Linear Regression Power Model

Linear regression [12] is a technique to model the relationship between a scalar variable and one or more explanatory variables. In [14], Isci and Martonosi presented a power model for Pentium4 processors using performance counters. They decompose the power consumption of the target processor into 22 components, and manually derive a model for each component using processor performance counters. Similar with CPU, GPU can also be decomposed into multiple different components for power modeling. Since most components of GPU are power independent with each other, the linear regression model can be described as following:

\[
P = \sum_{i=1}^{n} (\alpha_i \times u_i) + \beta \quad (3)
\]

The letter \( n \) represents the number of selected components in GPU. \( \alpha_i \) is the maximum power contribution of the \( i^{th} \) selected component. \( u_i \) is the usage rate which could be obtained from the performance counters. \( \beta \) represents the intercept which means the basic power does not include the selected units. The model allows us to estimate the average power of a given CUDA program.

#### D. Power Measurement

The Figure 10 shows the schematic for the power supply on the target GPU system. For the GPU card, since GeForce GTX 570 may reach the power of more than 200w, an auxiliary 12V trail has been added to the GPU card. Another power supply goes through the PCI-E interface into the GPU card. Therefore, for measuring the GPU runtime power, it is necessary to read the DC current readings from both auxiliary trail and PCI-E interface. For the auxiliary trail, the wires transferring DC current can be easily picked and bundled together. However the difficulty comes from the PCI-E interface, because there are no wires between the pins of GPU card and the the PCI-E slot. So a PCI-E riser card is used to insert some wires for connecting the pins of GPU card and the PCI-E slot. After making auxiliary trail and PCI-E DC wires go through the two FLUKE 80i-110s clamps respectively, the clamps can generate variable voltages describing the DC currents of the wires through the clamp. NI USB 6126 data acquisition is used to capture the readings from the two clamps and deliver the data to a computer for recording. The real system is shown in Figure 11.

However, since that the measured real power also includes the power of the card electrical units and the fan. For obtaining the pure power for GPU computation, the idle power must be subtracted from the measured power. We also assume the runtime power of GPU for displaying the Linux desktop is the idle power. The measured idle is 28.5w. In the following sections, the power refers the average power subtracted the idle power from it during the program execution.

#### E. Performance Counters

For the CUDA 3.2 we have selected, it provides a performance profiling tool named Compute Visual Profiler [11]. By using this profiler to run the executable CUDA program, the statistical results for selected performance counters could be obtained. Since we mainly consider the power contributions of instruction, register file, shared memory and global memory, within all the provided performance counters by the visual profiler, only 5 counters have been chosen. They are shown in Table IV.

The value of the active warps divided by the active cycles can be used as the active rates of the GPU on the profiling CUDA program. The \( \text{inst}_{\text{executed}} \) is the number of instructions executed by one thread. The total number of DRAM operations of one thread is the sum of dram reads and dram writes. So we multiply the active rates with the number of instructions and use the product as the usage rate...
### Table IV

**SELECTED PERFORMANCE COUNTERS**

<table>
<thead>
<tr>
<th>Counter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inst_executed</td>
<td>Number of instructions executed</td>
</tr>
<tr>
<td>dram reads</td>
<td>Number of read requests to DRAM</td>
</tr>
<tr>
<td>dram writes</td>
<td>Number of write requests to DRAM</td>
</tr>
<tr>
<td>active_warps</td>
<td>Accumulated number of active warps per cycle</td>
</tr>
<tr>
<td>active_cycles</td>
<td>Number of cycles a multiprocessor has at least one active warp</td>
</tr>
</tbody>
</table>

instruction component of GPU. For DRAM usage rate, we apply the same formula.

### VI. Evaluation

#### A. Micro-benchmarks

In order to train the GPU linear regression power model, multiple sets of known total power and usage rates for selected components should be used. Micro-benchmarks that stress one type of operation on the associated component in GPU are suitable for getting the results. Because they could minimize the power noise comparing general CUDA programs. The micro-benchmarks will repeat the same operations and try to highly press the associated unit. Since we mainly consider four components in GPU, accordingly, we plan to develop four micro-benchmarks for each. Due to the time limitation, register and shared memory micro-benchmarks are still in progress.

Actually, the computation and global memory are the two most important power contributors to the total power. Because the maximum power contribution of each component is roughly proportional with its area size. Therefore, with the considerations only of instruction components and global memory, the linear regression power model can almost predict the GPU power misleadingly. In the final project report, we will finish the consideration of register and shared memory into the GPU linear regression power model. The hypothesis is the model should be more precise than the one in this draft paper.

#### B. Training

The micro-benchmark stressing on instruction component consumes most of its execution time on a large number of computation instructions. They are mainly floating point addition and multiplication. Since they are both executed by the CUDA cores, we consider them the same type of instruction. For the micro-benchmark stressing global memory (DRAM), we consider the total number of the global memory transactions of reads and writes, because the global reads and writes have similar power dissipation. For register and shared memory, the sizes of used registers and shared memory information can be obtained by settling `ptxas --options = -v` to the nvcc compiler. For each micro-benchmark, we set different number of threads to achieve different usage rates and GPU power.

Each configuration of the threads will be executed five times to get the average power. Since the performance counters for each CUDA program are fixed, one time running of each configuration in the visual profiler is sufficient. The average power and usage rates for instruction, global memory (GMEM), register and shared memory (SMEM) with different configurations are shown in Figure 12. The usage rates are increasing with the number of threads becomes large. When the number of threads gets close to 256, the usage rates for all micro-benchmarks are almost saturated. The power for the 512 threads configuration consumes some extra power on scheduling. However, the usage rates can predict the GPU power trend misleadingly.

By using the four sets of power and usage rates to train the GPU linear regression power model, the $a_i$ which is the maximum power contribution to the total power is shown in Figure 13. For this current power model, we only consider the power impacts from the instruction components and global memory. The other components in GPU have been included in the intercept which may increase the error rate of this power model.

#### C. Power Prediction

For evaluating the accuracy of the GPU power model, four open-source CUDA sorting programs have been selected for the comparison of real average power and estimated average power. The five GPU sorting programs are Thrust, SDK, BB-Sort, Bitonic and Hybrid in Table V. The accuracy comparison is shown in Figure 14. These results show the
average error rate is 12.25%. There are two error sources. One is the usage rates can only roughly describe the busy degree of some associated components. The other one is that this model only considers four contributors. The power impact from all the other components not considered is included in the intercept and assumed to be the average power of these components of the four micro-benchmarks. So, the intensively using of these components may make the prediction have high error ratio.

VII. CONCLUSION

DVFS is a dynamic power management technique, which has been implemented on CPU in many applications but rarely on GPU. DVFS implementation can be done by software or hardware. In software-based DVFS implementation, it is usually done by placing hooks in kernel system calls of operating system to collect the information of system call and thereby calculate the system load. The work of collecting voltage and frequency values and setting those values in Nouveau driver have been studied. For supporting the DVFS implementation choosing voltage and frequency levels of GPU, a GPU linear regression power model for power prediction is built. With the considerations of power contributions from instruction components, global memory, register and shred memory of GPU, the power model could achieve 12.25% average error ratio for the four chosen sorting CUDA programs on GPU.

REFERENCES