Exploiting DVFS for GPU Energy Management

Diogo Mineiro Cameirinha

INESC-ID
Instituto Superior Técnico - Universidade de Lisboa
Email: diogo.cameirinha@tecnico.ulisboa.pt

Abstract—The IT infrastructure represents, nowadays, a significant part of the worldwide electricity consumption, with a tendency to increase. General-Purpose Computing on Graphics Processing Units (GPGPU) has become a suitable target for energy reduction techniques due to its maturity and the penetration of GPU hardware in nowadays server and workstations.

A technique for energy usage reduction in GPGPU is proposed in this manuscript, by exploiting dynamic scaling of the GPU core and memory frequency levels. To accomplish this, the GPU’s memory channel is monitored to determine the nature of the task being executed and to estimate the optimal GPU clock frequencies. Two methods are used to monitor the memory channel, Access Rate Threshold (ART) and Frequency Variation (FV). In the former, the memory channel is considered to be saturated if the current data rate is above a predetermined threshold. In the latter, the state of the memory channel is determined by inducing a change in the GPU frequency and verifying the effect it has on the rate of memory transactions. To evaluate the proposed technique it was implemented and tested on a system with an Nvidia K20c GPU. CUPTI and NVML are used to perform the required GPU monitoring and the latter is also used to set GPU clock frequencies. Testing was made using a set of benchmarks from the Rodinia suite and two extra benchmarks developed around cuBLAS. The obtained experimental results show that energy reductions of up to 24% can be attained with 45% performance penalty.

Index Terms—Energy usage optimization, GPU, Frequency Scaling, Kernel classification

I. INTRODUCTION

Heterogeneous computing systems composed of Central Processing Units (CPUs) and Graphics Processing Units (GPUs) have become common in nowadays server and workstations. In particular, almost all devices capable of computation, from smartphones to Personal Computers (PCs), are heterogeneous platforms of this kind.

With the appearance of General-Purpose Computing in Graphics Processing Units (GPGPU), programming languages and frameworks, such as Compute Unified Device Architecture (CUDA), have been developed to allow programmers exploiting GPU architectures to maximize application performance. However, such a subject was particularly difficult to be studied due to the difficulty in measuring kernel power consumption, the introduction of energy counters in modern GPU architectures has opened the way for such investigations. Accordingly, this subject is being increasingly studied in the literature, by predicting the energy usage of a GPU kernel and on the effects that clock frequency and gating have on the energy usage of a kernel. In particular, recent research suggests that if a GPU task is memory intensive, it is usually beneficial (from an energy usage standpoint) to keep the GPU memory at a high clock frequency, whereas it is beneficial to reduce the memory clock frequency in other cases. Although a few techniques that optimize specific tasks or GPU kernels can be found, no techniques for online and continuous GPU energy reduction exists in the literature, according to the authors knowledge.

The objective of the presented work was to develop a technique for online optimization of energy usage in GPUs. To do this, however, it was required that any targeted GPU should be able to provide software access to performance counters, its power or energy consumption in real-time, and a method to set clock frequencies. For the moment, only Nvidia’s Tesla family of GPUs fulfills these requirements.

Two Nvidia GPUs were available for development, a K20c and a K40c. The Nvidia Management Library (NVML) was useful for having methods that provided the GPU’s current power usage at a satisfactory rate. Through NVML is was also possible to configure the clock frequency settings of the GPU board in real-time. To extract event counts from the GPU’s performance counters, CUDA Profiling Tools Interface (CUPTI) was used. With CUPTI it is possible to obtain in real-time the current data rate of the GPU Random Access Memory (RAM) memory. The developed technique uses the state of the memory channel to determine whether it is recommended to reduce the memory clock frequency in order to reduce the energy usage. With this technique it was possible to reduce the energy used in the execution of a set of benchmarks by 24% with a 46% increase in execution time.

II. METHODS FOR ENERGY REDUCTION

The majority of a processor’s power usage comes from dynamic power, which is given by the proportionality:

\[ P_{\text{dynamic}} \propto C \times V^2 \times A \times f \]  \hspace{1cm} (1)

In (1), \( C \) is the aggregate capacitance, \( V \) is the supply voltage, \( A \) is an activity factor that refers to how often wires transition from 0 to 1 or vice-versa, and \( f \) is the operating frequency. The capacitance can be influenced by the architecture of the processor because it depends largely on the wire lengths of on-chip structures. The activity factor \( A \) can be reduced using clock gating techniques. The clock frequency has a potentially far-reaching impact on power usage. It can impact power not only directly but also indirectly due to its effect on supply voltage. Typically, maintaining higher clock
frequencies requires maintaining higher supply voltages. Thus, the combined $V^2 \times f$ of (1) can have a cubic impact on power. Dynamic Voltage and Frequency Scaling (DVFS) strategies recognize periods when lower performance is acceptable (e.g., memory-bound or latency-tolerant regions of code) and reduce both the supply voltage and the clock frequency accordingly [1]. On modern multicore CPUs, operating frequency can be set individually on each core[2].

Several successful cases of software-driven power optimization on GPUs can be found in the literature.

In [3] a study on performing matrix multiplications with a K20c GPU under different GPU and memory clock settings is presented. In this work, two important conclusions are attained. First, the GPU cannot sustain the overclock frequency for sustained periods of time because the board’s power management system throttles the system back to the default clock settings after about 60 seconds. Second, for the problem of matrix multiplication, the efficiency of the GPU measured in GFLOPS per Watt behaves differently from the CPU. For matrices large enough, while lower clock frequencies attain higher energy efficiency levels on the CPU, on the GPU all the frequencies yield similar efficiency except for when the memory frequency was dropped from 2600 MHz to 324 MHz [3]. In the latter case, i.e., with the lower memory frequency, the total amount of energy required to finish the task is actually higher than at other operating points [3]. In [4] it is demonstrated that varying the clock frequency of the memory can effectively save energy. Energy usage in computational intensive kernels generally benefits from lowered memory frequency and, likewise, energy usage in memory intensive kernels generally benefits from increased memory frequency.

In [5] an integrated power and performance prediction model for a GPU architecture to predict the optimal number of active processors for a given application. The basic intuition is that when an application reaches the peak memory bandwidth, using more cores does not result in performance improvement. This is an approach that makes use of clock gating. The low-level code of a kernel is analyzed and the number of optimal active cores is predicted. Power savings obtained with this solution averaged at $\approx 11\%$.

The greenGPU framework proposed in [6] uses a combination of workload division between GPU and CPU, and frequency scaling on the GPU achieving energy savings of $\approx 21\%$. The principle behind this approach is that the workload of a task is distributed in a way that both GPU and CPU finish at the same time, minimizing the energy that would be wasted if one of the processors was idle and waiting for the other to finish. The solution is achieved iteratively. When a task is first queued for execution, the workload is divided arbitrarily between GPU and CPU (e.g., 50%-50%). Depending on the execution times of each workload, the workload division for the next iteration of the task is adjusted in steps of 5%. During each iteration, the frequency of the GPU is adjusted based on a GPU occupation metric provided by NVML [6].

### III. POWER AND PERFORMANCE MONITORING IN NVIDIA GPUs

#### A. Power Monitoring

There are two documented methods to estimate energy consumption in real time. The first one is by estimating the instantaneous power, by measuring the voltage drop across shunt resistors mounted in series with the chip’s power source. This voltage drop $U_R$ is linear with the current by a factor of of the resistance of the shunt $R_{shunt}$. Power can then be obtained by multiplying the power source’s delivered voltage $V_{dd}$ by the obtained current $P = V_{dd} \times I_{est}$ [7].

Another solution is to estimate power using an indirect method with hardware performance counters, as explained by an Nvidia’s patent [8]. Power usage is estimated based on the combined activity of various functional blocks of the chip, in particular the number of flip-flops active in each block at a given time. This number is estimated by monitoring a representative set of enable signals supplied to the flip-flops. After selecting which enable signals are representative enough, weighting factors are determined for each set of signals. For each block, the representative signals are summed up and the scale factor is applied to the sum. The scaled sums are added together to yield an interim power usage estimate. Several interim power usages are low-pass filtered and integrated over the refresh period of the estimator to achieve the effective power usage. This method as described by the patent is part of an energy control system that lowers the GPU frequency to avoid excessive power usage[8]. This method is similar to the approach taken in [9] where data from performance counters is used to estimate power usage. However, Nvidia’s method is performed by the hardware itself and produces results in real-time.

In [10], it is proposed that the NVML power readings are accurate within 5% of the actual power draw and the sampling frequency of the sensor was measured to be about 66.7 Hz. It is also demonstrated that the power usage value reported by NVML has a capacitor-like behavior in which the time constant $r$ of is $r = 0.8333 s$. With the time constant, it is possible to calculate the time it takes for the value reported by NVML to reach a certain percentage of the actual power usage.

\[ X_N / 100 = 1 - e^{-0.8333 t} \Leftrightarrow t = -\frac{\ln(1 - \frac{X_N}{100})}{0.8333} \]  

Using equation 2 it is possible to calculate the establishment times in table I.

<table>
<thead>
<tr>
<th>$X_N$</th>
<th>Time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>0.832</td>
</tr>
<tr>
<td>75</td>
<td>1.664</td>
</tr>
<tr>
<td>90</td>
<td>2.764</td>
</tr>
<tr>
<td>95</td>
<td>3.596</td>
</tr>
<tr>
<td>99</td>
<td>5.328</td>
</tr>
</tbody>
</table>

To put in context the impact of this value for the time constant an example where a GPU kernel is causing a power
usage of 190 W can be considered. The time it takes for the value reported by NVML to reach a value within the margin of error indicated by Nvidia, $190W - 5\% = 180.5W$ starting at an idle power usage of 50 W would be of around 3.23 s. This result is demonstrated in (3).

\[
\begin{align*}
190W - 50W & = 140.0W \\
180.5W - 50W & = 130.5W \\
130.5W & = 140W \\
\frac{140W}{0.8333s^{-1}} & = 0.9322 \\
\frac{\ln(1 - 0.932)}{0.8333s^{-1}} & = 3.23s
\end{align*}
\]

Because it is possible that the method used by the GPU to report power can be updated via firmware or driver update [7] a confirmation of the of the establishment time of the reported power usage and the rate at which new values are produced presented in [10] was needed as these parameters are critical for a real-time application. It was discovered that on the K20c system, a new value was given at each 110 milliseconds and on the K40c system, a new value was given at each 20 milliseconds. This means that the sample rate is 10Hz on the K20c and 50Hz on the K40c. Furthermore, the establishment times were also different from expected. A test was developed that produced a behavior on the K20c GPU similar to the example calculated in (3) where the idle power is 50W and the GPU kernel increases power usage to 190W, constant throughout the remaining of the execution. In this case, a power usage value of 183W within the error margin was reached in 2 samples, meaning that the establishment time was only 220 milliseconds. On the K40c system, within 11 samples the value rose from 66 W to 210 W within 5% of the final 220W, meaning an establishment time of 220 milliseconds similar to the K20c.

B. Performance Monitoring

Power alone cannot provide a meaningful indication on whether energy usage will benefit from lowered clock frequencies. Through CUPTI it is possible to read event counts that indicate the number of occurred data transactions to the RAM memory, providing a GPU-wide performance indicator.

Although the process to configure and read event counts using CUPTI is well documented by Nvidia, there are some limitations that had to be checked experimentally through trial and error. Some of the events are sectioned, meaning that events like the number of memory read transactions are divided in two or more sections. For this particular example, the number of read transactions made on the RAM memory is divided into the two events. Each of these events counts the number of read accesses to one of two sectors of the on-board RAM. One of the limitations is that, for example, although event counts for the memory read and memory write transactions of the same section can be accessed simultaneously, the memory read event counts of more than one of the memory’s section cannot be accessed simultaneously. This puts a constraint on real-time applications using the data because to obtain the total amount of RAM memory read transactions, approximations have to be made. To overcome this limitation, an approximation can be made by simply counting only one of the events and multiplying it by 2 since the memory is partitioned in 2 sections. The error associated with this approximation is difficult to measure because the way that the partitions are divided is unknown. The worst case scenario would be that each event corresponded to either the lower or the higher half of the memory addresses and the memory usage of the monitored GPU kernel were contained within the not monitored half. The memory data rate can be, then estimated using equation (4) which includes the estimation of all read and write transactions, the memory bus width, the time delta for which the memory events were counted, and a normalization factor indicated in CUPTI’s documentation.

\[\text{MemRate}_{bps} = \frac{(\text{Read} + \text{Write}) \times 2 \times \text{Norm} \times \text{BusWidth}_{bit}}{\text{ElapsedTime}_s} \]

C. Benchmark profiling

To test the performance of the solution, both the Rodinia benchmark suit and self-developed benchmarks were used. To complement Rodinia, matrix multiplication, and array elements swap benchmarks were created. In the matrix multiplication benchmark two matrices are loaded into the GPU memory, multiplied 5 times in a row using the cublasSgemm function, and the resulting matrix retrieved to CPU memory. The array element swap benchmark uses cublasSswap to swap all the elements from one array to another and vice-versa 10 times in a row. Matrix multiplication can be a good test for overall performance as it requires a large number of calculations by the GPU and it demands a large number of accesses to memory for both writing and reading. The swap benchmark is a pure memory-oriented benchmark as it produces a high number of memory accesses that do not require processing by the GPU.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Repetitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>gaussian</td>
<td>20</td>
</tr>
<tr>
<td>heartwall</td>
<td>5</td>
</tr>
<tr>
<td>hotspot</td>
<td>5</td>
</tr>
<tr>
<td>kmeans</td>
<td>5</td>
</tr>
<tr>
<td>lavaMD</td>
<td>5</td>
</tr>
<tr>
<td>lud</td>
<td>5</td>
</tr>
<tr>
<td>myocyte</td>
<td>5</td>
</tr>
<tr>
<td>particlefilter</td>
<td>5</td>
</tr>
<tr>
<td>rad</td>
<td>5</td>
</tr>
<tr>
<td>streamcluster</td>
<td>5</td>
</tr>
<tr>
<td>gemm</td>
<td>5</td>
</tr>
<tr>
<td>swap</td>
<td>5</td>
</tr>
</tbody>
</table>

Because the objective is to optimize energy consumption and the available modifiable parameters are the GPU and memory clock frequencies, the energy used by each benchmark on each frequency allowed by each card was measured using NVML. To mitigate errors, each benchmark was executed repeatedly for at least 5 times or for the number of iterations required for the total run time to be larger than 10 seconds. The used benchmarks are presented in table II in the order
of their execution, along with the number of iterations. This method would be fallible if the objective was to measure actual GPU kernel energy usage because portions of time when the benchmarks are performing I/O and GPU work are also accounted. The objective is to determine which benchmarks benefit energy-wise from being executed at different frequency settings.

On both GPUs, results show the most drastic differences are between frequency sets where the memory is operating at different frequencies. On the K20c, energy differences between the GPU frequencies for which the memory operates at 2600MHz stay within a maximum of 4% between them. Furthermore, in some cases the energy and frequency do not have a direct proportion, for example, in particlefilter 544 J are measured at 614 MHz, 551 at both 640 and 666 MHz, and energy drops to 541 J at 705 MHz. Likewise, on the K40c GPU differences between the GPU frequencies at which the memory operates at the default 3004 MHz frequency stay within a narrow margin. It can be concluded, therefore that, for the sake of performance, it is most efficient to use the default GPU frequency when the default memory frequency is used.

Comparing the default clock settings to the lowest clock frequencies (324/324), it is expected that tasks not dependent on memory performance decrease execution times only approximately to the GPU frequency proportion (i.e., K20c \( \approx 0.56 \), K40c \( \approx 0.43 \)) and that their energy usage decreases. On the other hand, tasks with higher demand for memory frequency performance should decrease their execution times even further and see their energy usage increase. The expected results are summarized in table III.

### TABLE III

**Expected results when decreasing memory frequency**

<table>
<thead>
<tr>
<th>Task type</th>
<th>Execution time</th>
<th>Energy Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not memory bound</td>
<td>Decreases with GPU frequency</td>
<td>Decreases</td>
</tr>
<tr>
<td>Memory bound</td>
<td>Decreases greatly</td>
<td>Increases</td>
</tr>
</tbody>
</table>

In section II, several successful approaches to energy reduction on GPUs are presented. However, the available methods are tied to the optimization of specific kernels of tasks, rather than optimizing the GPU in real-time. Despite successful, there are usability limitations when the optimization is tied to specific kernels or tasks. This is a more severe problem for offline methods that statically match tasks to the best processor either at compile time or by intercepting library calls at execution time. These methods require that the known

![Fig. 1. Memory rate while executing the benchmark set on the K20c GPU](image)

Unfortunately, it was not possible to do the same analysis on the K40c board. This was due to an issue with the used Nvidia software version that would not allow CUPTI to count events from GPU tasks launched from separate CPU processes. Since the methodology to obtain the memory profiles consisted of launching the benchmarks and reading the event counts from a separate process, it became impossible to replicate the test for the K40c. It would be interesting to observe the differences in the data rates especially for the benchmarks that changed classification.

In conclusion, the technique for energy usage reduction should identify when the GPU task is not memory bound and reduce the memory clock frequency. Also, the clock frequency should also be reduced when the GPU is not being used. The data rate obtained through CUPTI can, therefore, be used to provide a basis for the decision of changing the clock frequencies on the GPU.

### IV. Energy Reduction Through Dynamic Frequency Scaling

Table IV presents the benchmark classifications for both GPUs on whether they are memory bound or not for energy usage. When compared to the execution on K20c, myocyte and particlefilter become memory bound on the K40c.

Figure 1 and provide the data rate in MB/s over time throughout the execution of the benchmarks in table IV by the K20c GPU. The vertical lines mark the points in time where a benchmark ends and another begins. The benchmarks were executed in the top-down order presented in table IV. Throughout the benchmarks’ execution, sudden spikes in data rate both upwards and downwards can be observed. Further observation of benchmarks with fairly constant data rate values indicates that a spike in a sample is immediately compensated. In other words, a spike downwards is preceded by an equally large spike upwards and vice-versa. In the last benchmark, some of these spikes upwards cause the estimated data rate to go over the maximum theoretical value of around 200 GByte/s. These spikes can, therefore, be attributed to some error in the values reported by CUPTI.

![Fig. 1. Memory rate while executing the benchmark set on the K20c GPU](image)

In conclusion, the technique for energy usage reduction should identify when the GPU task is not memory bound and reduce the memory clock frequency. Also, the clock frequency should also be reduced when the GPU is not being used. The data rate obtained through CUPTI can, therefore, be used to provide a basis for the decision of changing the clock frequencies on the GPU.

### TABLE IV

**Benchmark classifications by GPU**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Classification on K20c</th>
<th>Classification on K40c</th>
</tr>
</thead>
<tbody>
<tr>
<td>gaussian</td>
<td>not memory bound</td>
<td>not memory bound</td>
</tr>
<tr>
<td>heartwall</td>
<td>memory bound</td>
<td>memory bound</td>
</tr>
<tr>
<td>hotspot</td>
<td>not memory bound</td>
<td>not memory bound</td>
</tr>
<tr>
<td>kmeans</td>
<td>not memory bound</td>
<td>not memory bound</td>
</tr>
<tr>
<td>lalad</td>
<td>memory bound</td>
<td>memory bound</td>
</tr>
<tr>
<td>myocyte</td>
<td>not memory bound</td>
<td>memory bound</td>
</tr>
<tr>
<td>particlefilter</td>
<td>not memory bound</td>
<td>memory bound</td>
</tr>
<tr>
<td>rad</td>
<td>memory bound</td>
<td>memory bound</td>
</tr>
<tr>
<td>streamcluster</td>
<td>not memory bound</td>
<td>not memory bound</td>
</tr>
<tr>
<td>gemm</td>
<td>not memory bound</td>
<td>not memory bound</td>
</tr>
<tr>
<td>swap</td>
<td>memory bound</td>
<td>memory bound</td>
</tr>
</tbody>
</table>

In section II, several successful approaches to energy reduction on GPUs are presented. However, the available methods are tied to the optimization of specific kernels of tasks, rather than optimizing the GPU in real-time. Despite successful, there are usability limitations when the optimization is tied to specific kernels or tasks. This is a more severe problem for offline methods that statically match tasks to the best processor either at compile time or by intercepting library calls at execution time. These methods require that the known
behavior for each GPU function and its CPU counterpart is checked for changes each time a library is updated. This can be specially exhausting because matching decision usually depends on the data set size of the task or function. Also for online methods, if the granularity of the optimization is at a kernel or task level, there is the possibility that part of the task behaves differently. For example, a small part of a task that would benefit from a slower memory clock could go undetected. Furthermore, with the ability for the GPU to generate work for itself, it is possible that a GPU kernel withing a larger task generates workloads that benefit from different GPU settings.

The solution for these problems is to develop a method that optimizes the clock frequencies of the GPU in real-time based on its current state. The usage of hardware performance counters provides real-time data on the state of the GPU and the task being executed. In chapter III the conclusion was reached that the GPU retains its energy efficiency across the allowed clock frequencies. For the used GPUs, the notable differences in energy usage were achieved when comparing the benchmark executions at different memory clock settings. It is possible to know in real-time the data rate of the whole GPU’s physical memory through a method described in III. Taking these assumptions to consideration, a method to adjust in real-time the memory frequency relative to the GPU is herein proposed.

The objective of this method is to minimize the GPU energy usage independently of the task being executed. This can be done by decreasing the clock frequency of the memory when the GPU’s current task is not memory intensive enough. The frequency reduction should lead to a reduction in power usage and a reduction in performance. The reduction in performance should be lower than the reduction in power usage. Reducing frequencies during the execution of a task would be beneficial if, for example, power is reduced to a third and execution time to double (half the performance) to achieve two thirds of the original required energy, \( \frac{1}{3} \text{Power} \times 2 \text{Time} = \frac{2}{3} \text{Energy} \).

The developed method can be divided in two components: Memory Usage Detection (MUD), and Dynamic Frequency Scaling (DFS). Execution consists of infinite iterations of a cycle where the MUD component analyses the GPU memory data rate and decides whether the GPU is idle, on a memory bound task or on a compute bound task, and the DFS component manages the state of the GPU frequencies accordingly. When a compute bound task is detected, energy reduction is achieved by setting the system into a power saving mode for a predetermined time period and during which the clock frequency of the GPU memory is reduced. Several parameters of both components can be adjusted to the user’s preference.

To avoid naming confusion with other types of cycles such as clock cycles, the MUD and DFS cycle will be referred to as an E-cycle. The general operation of an E-cycle is presented in figure 2. Each E-cycle starts with the thread sleeping for a predetermined amount of time. This allows for the necessary GPU events to be counted and provides a time interval to calculate the needed performance metrics. After sleeping, the MUD component is responsible for acquiring and analyzing the event counts and classifying the current status of the GPU task as a compute bound or memory bound task, or as being in an idle state. Afterwards, and based on the classification made by the MUD component, a set of decisions are made by the DFS component. If the GPU was considered to be idle, its clock frequencies are immediately set to the lowest and the E-cycle returns to the sleeping step. If the GPU is indeed being used, a “Power Saving” flag, initially set to "false", is inspected to verify if the system is already in the power saving mode. If not, a decision to set the power saving mode and lower the clock frequencies is made if the GPU task is compute bound. The system remains in power saving mode until a predetermined number of E-cycles have passed (corresponding to the power saving period).

A. Memory Usage Detection

The MUD component analyses the memory usage rate and classifies the current state of the GPU. Accordingly, at each E-cycle, the MUD component collects the CUPTI events necessary to obtain the current data rate using equation (4). The collected data is analyzed using two methods, Access Rate Threshold (ART) and Frequency Variation (FV). The first method, ART, classifies the GPU task by comparing the current memory data rate to a threshold value. The second method, FV, classifies the GPU task by analyzing the effect that a small variation on the GPU clock frequency produces on the memory data rate while maintaining the memory clock frequency fixed. The reason both are used simultaneously is that they have complementing advantages and disadvantages that will be presented and further explained in this section.

Both ART and FV can output three kinds of classifications: IDLE

![Basic E-cycle flowchart](image)
The GPU is currently idle, meaning that there memory activity is inexist or negligible.

**MEM_BOUND**
The GPU is currently executing a memory bound task and the memory performance should be high to minimize energy usage.

**NOT_MEM_BOUND**
The GPU is currently executing a task not limited by memory performance which can be lowered to minimize energy usage.

As already discussed, occasional errors occur in the event counts that produce sudden increases and decreases in the read values. To mitigate this problem, a sliding window for each method is used to determine the actual classification of the task. In each E-cycle, the classification put out by both methods is stored into its respective window. Every time a new value is stored, the oldest value is discarded. The final task of the MUD component at each E-cycle is the analysis of the window to determine the actual GPU task classification. The analysis consists of a poll where each element in each window contributes with a vote for the final result.

\[
M_{Rate_{thresh}} = \frac{\text{Relative Memory Perf}}{\text{Relative GPU Perf}} \times \text{Default Bandwidth} \tag{5}
\]

Probably, the most straightforward way of determining if the memory’s frequency can be lowered is to compare its current data rate and compare it to the maximum possible data rate at the lower frequency. Putting it in form of a question, can the memory at the lower clock frequency sustain the data rate currently requested by the GPU? To answer this question, at each E-cycle the current memory data rate is calculated. The maximum theoretical memory data rate is obtained using the `cupiDeviceGetAttribute` function in CUPTI which can be used to calculate the data rate threshold value at which the GPU task is considered memory bound. However, the threshold cannot be only defined by memory performance. Because clock frequencies are set in pairs of Memory and GPU frequencies, when lowering the memory frequency, the decrease in memory requests caused by lowering the GPU frequency also has to be taken into account. For example, lowering memory from 2600 MHz to 324 MHz lowers memory bandwidth to \(\approx 12.5\%\) of the default value. However, the associated decrease in GPU frequency from 705 MHz to 324 MHz can be assumed to also decrease memory requests in about \(46\%\). The actual threshold value can, therefore, be obtained by dividing the relative memory performance by the relative GPU performance and multiplying the result by the default memory bandwidth, using equation (5).

An example of ART classification can be observed in figure 3. For a given constant clock frequency, anytime the data rate is above the horizontal line, ART considers the task to be memory bound.

**Fig. 3. Example of ART classification**

ART is a straightforward method but, because only the data rate is analyzed, some factors can be overlooked. Latency is not taken into account, either from memory or from the GPU. If a task is being limited by memory latency, a consequential low data rate may wrongly lead ART to detect it as compute bound. Also, lowering the GPU frequency may not lead to a directly proportional decrease in memory transaction requests due to latencies internal to the GPU. Therefore, another method besides ART is needed to improve the detection of the GPU task’s memory profile. FV is based on the principle that, in a compute bound task and for a constant memory frequency, a change in the GPU frequency results in a proportional change in data rate. If the memory channel is not saturated, any change in the rate of memory transaction requests by the GPU should result in a proportional change of the actual rate of memory transactions. If the memory channel is saturated, a change in the rate of memory transaction requests should result in a lesser change or in no change at all in the actual rate of memory transactions. This principle is illustrated in figure 4, where figure 4a exemplifies a case where the task is compute bound, as a change in GPU frequency causes the same proportional change in the memory transactions rate. On the other hand, figure 4b exemplifies a case where the change in the memory transactions rate is not proportional to the change in GPU frequency.

The GPU frequencies used for FV should be close enough so that they do not degrade performance significantly and they should be separate enough to account for small errors in event counts, or other factors. For the K20c GPU, the chosen frequencies were the default 705 MHz and 640 MHz.

In the operation of FV, the GPU is running at each of the frequencies alternately at each E-cycle. In each E-cycle, the memory transaction rate is calculated and is compared with that of the previous E-cycle. If the ratio of memory transaction rates is close to the ratio of the respective E-cycle GPU frequency, the task is considered compute bound, or memory bound if otherwise. Because FV is used with ART at the same time, the threshold value for ART also has to change at each E-cycle according to the corresponding GPU frequency.

**B. Dynamic Frequency Scaling**

While the purpose of the MUD component is to detect the current state of the GPU task, the purpose of the DFS component is to act upon the results of MUD. The basic functionality of DFS is: set the GPU to a low power idle state when it is detected that no task is being executed; when a task is being executed monitor the output of MUD; if
MUD detects a NOT_MEM_BOUND situation, enter a low frequency period such that energy usage is reduced; at anytime if the task is finished return to the lower power state. When on the Idle state, it is necessary to directly set the GPU and memory frequencies to the lowest setting (both at 324 MHz) because neither the driver nor the GPU firmware do it automatically. During the low frequency period, both the GPU Streaming Multiprocessor (SMX) and memory frequencies are also lowered to 324 MHz, reducing energy usage by taking advantage of the NOT_MEM_BOUND nature of the GPU task. In the low frequency period, task classification is not executed other than determining if the GPU is idle.

Ideally, DFS should keep receiving classifications and only leave the low frequency state when the task is no longer NOT_MEM_BOUND. However, the principle of testing the current data rate that ART does is not suitable to determine if frequency should be raised. FV could be used but the purpose of energy reduction would be defeated.

In short, DFS is the actuator that solves the energy reduction issue. It uses the classifications given by MUD and dynamically manages the GPU and memory frequencies. At the simplest level, operation of this component can be described as a Finite State Machine (FSM) with the following states, interacting as presented in figure 5:

**GPU_IDLE**

Anytime memory activity falls below 5000 transactions per second, it is considered that the GPU is not being used. In this case, the FSM state will always transition to GPU_IDLE. While in GPU_IDLE, GPU and memory frequencies are set to 324 MHz to reduce energy usage. When memory activity rises above 5000 transactions per second, the FSM transitions to GPU_NOMINAL. During this state, the MUD component is idle.

**GPU_NOMINAL**

While on this state, MUD is active and obtaining classifications. This state is considered “nominal” because the GPU and memory frequencies are set as required by FV and represent the default operation of the GPU. The FSM remains in this state while memory activity is above 5000 transactions and MUD classifies the GPU task as MEMORY_BOUND. State transition to GPU_IDLE occurs when memory activity falls below 5000 transactions. State transition to GPU_LOW_FREQUENCY occurs when MUD classifies the GPU task as NOT_MEM_BOUND.

**GPU_LOW_FREQUENCY**

This state represents the energy saving action of the DFS component and corresponds to the implementation of the low frequency period. While on this state, both GPU and memory frequencies are set to 324 MHz, and MUD is not active. From this state, the FSM can transition to GPU_IDLE if the memory activity falls below 5000 transactions per second or it can transition to GPU_NOMINAL if the low frequency period is over. To detect when the low frequency period is over, on the transition to this state, an int variable is set to the period length in number of E-cycles. Each E-cycle, this variable is decremented and when it reaches 0, the FSM state changes to GPU_NOMINAL.

The proposed DFS procedure, as it was described, has two inherent limitations. The duration of period during which the GPU is operating at lower frequency (GPU_LOW_FREQUENCY state) is static. Enabling the dynamic adjustment of this state’s duration allows the DFS procedure to automatically optimize to the program being executed. There is also no way to mitigate the execution time penalty resulting from the GPU_LOW_FREQUENCY states at the expense of a lesser energy usage reduction which could be useful, for example, in real-time applications. The DFS procedure was extended with mechanisms that counter these limitations.

As already mentioned, the purpose of dynamically adjusting the GPU_LOW_FREQUENCY state’s duration is to optimize
it to the particular program being executed at the time with minimal effort from the user. It assumes that any particular program will execute the same GPU kernel throughout its lifetime with small changes in data size. Each time a low frequency period ends, a measure of energy efficiency is calculated for both the current period and the preceding period. The average for each period is compared and if efficiency was higher in the low frequency period, its length is incremented by one E-cycle and decremented otherwise.

To allow the trade-off between energy and performance, the DFS FSM can transition to a new GPU LOCK state when exiting the GPU LOW FREQUENCY state and remain in that state for a predetermined number of E-cycles. The GPU operation of the GPU LOCK state is equivalent to GPU NOMINAL with the exception that MUD classifications are ignored, which means that while on this state, a transition to GPU LOW FREQUENCY cannot happen. By limiting the amount of time that the FSM is at the GPU LOW FREQUENCY state, a reduction in performance loss is expected along with an increase in energy usage.

To implement the extended functionality, two extra FSM states are introduced as presented by figure 6.

**GPU NOMINAL WAIT**

This is an intermediate state to which the FSM will go to when it is in the GPU LOW FREQUENCY state and the low frequency period ends but the GPU power sensor did not output a new power value during the GPU LOW FREQUENCY state. This state is auxiliary to the dynamic adjustment for the low frequency period length and its only purpose is to obtain a power reading when one does not exist. This can happen, for example, when the low power period is 300 ms but the output period of new power values is 1 second. During this state operation is similar to nominal.

**GPU NOMINAL LOCK**

This state implements the lock period. GPU operation is similar to GPU NOMINAL but the transition to other states is similar to GPU LOW FREQUENCY in that it will transition to GPU NOMINAL when the target number of E-cycles passes but will transition to GPU IDLE if the GPU becomes inactive.

V. EXPERIMENTAL RESULTS

To evaluate the proposed technique, a performance analysis was performed. The MUD component was evaluated on its capacity to generate correct classifications and the DFS procedure was evaluated in conjunction with MUD on the achieved energy reductions. To reach a valid evaluation, a set of benchmarks that covers a diverse ensemble of computational problems has to be used. Containing 12 different benchmarks, 4 of them classified as compute bound and the remainder as memory bound, the benchmark set used in chapter III covers this requirement.

To establish a base for comparison, the total energy usage and execution time of the benchmark set were registered for the GPU with the default and lowest clock settings, and with only the MUD component active (GPU frequencies alternating as necessary for FV). Table V presents the obtained results. Simply running the benchmarks at the lowest frequency not only does not reduce the energy usage but increases it with a large penalty in execution time. The execution of the benchmark set at the constant GPU core frequency of 705 MHz yielded an approximated 5% energy usage increase when compared to execution with the MUD component active, which is within the error margin of NVML.

<table>
<thead>
<tr>
<th>Clock Frequencies</th>
<th>Energy Usage [J]</th>
<th>Execution time[s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>705 MHz &amp; 2600 MHz</td>
<td>46532.2</td>
<td>436.3</td>
</tr>
<tr>
<td>324 MHz &amp; 324 MHz</td>
<td>49147.1</td>
<td>1091.3</td>
</tr>
<tr>
<td>MUD active</td>
<td>44125.3</td>
<td>418.9</td>
</tr>
</tbody>
</table>

The period length can have several impacts on the energy reduction performance. The detection analysis revealed that the percentage of correct classifications was higher for the extreme values of 10 and 2000 milliseconds and practically constant for the values in-between. Observing the values in table VI, the two best performers in terms of correct percentage of classifications, 10 and 2000 milliseconds actually achieved the worst Energy usages. The execution times can give an indication of what caused the bad performance. In the case of the 2000 ms period length, a long execution time means that the GPU spent too much time operating at the lowest clock frequencies which would be expected because the duration of the low frequency state is given in number of E-cycles. In this case, an E-cycle duration of 1000 milliseconds means that every time a low frequency period is triggered, it lasts for 7 seconds under the default settings, which is a larger period than some of the benchmarks. On the other hand, in the case of the 10 ms period length, the short execution time indicates that the DFS remained too much cycles at the GPU NOMINAL state. A 10 ms period length implies 5 ms E-cycles that causes the low frequency period to be of only 35 ms, which means that the DFS could be constantly switching states. As for the remaining values, the period length of 50 ms achieved less 12.7% energy compared with 150 ms and, compared to the default clock setting, a reduction of 22.9%.

The time-related parameters provided information on how the robustness and delays of the MUD decisions along with
the duration of the low frequency cycle affects the usage of energy. The MUD analysis suggested that ART and FV were individually better at identifying compute bound and memory bound situations, respectively and that completely ignoring FV would produce the best results. Table VII shows that together, they achieve the lowest energy usage and individually.

Table VIII presents some of the attempts made by fine-tuning several parameters according to the acquired knowledge of the system’s behavior. From the top to bottom, the first test was the best overall with the second achieving a similar energy usage with an improved execution time. The top performer in energy usage achieved 76% of energy usage when compared to using the default frequencies at the cost of increasing execution time to 146%.

The main disadvantages of this solution are that, for the moment, only the Tesla family of GPUs fulfills the requirements for the technique to be applied, and that optimization is limited with only two available memory clock frequencies.

VI. Conclusions

Power usage in IT infrastructures has become a significant problem. Due to their increasing penetration rate in servers and workstations, GPGPUs has become a suitable target for energy optimization techniques. However, the currently available techniques used to optimize power usage in GPUs have several limitations. The usage of external discrete power sensors severely affects the real-world usability of the techniques that use them. Offline or iterative techniques are also usually tied to specific tasks such as GPU library functions or kernels. Because of this, these techniques are limited in the granularity of their optimization and can have portability issues.

In this dissertation, an online GPU energy reduction technique was proposed. Online energy reduction is achieved with DFS assisted by real-time monitoring of the physical memory channel. Memory monitoring is performed with a combination of two techniques. ART checks if the current data rate is possible with a reduction in clock frequencies, and FV tests the memory channel saturation making variations in GPU frequency and analyzing if the same variation occurred in the data rate. The DFS component lowers the frequency of both memory and GPU when either no task is running or when it is detected that the execution of the GPU task is not being limited by the memory’s performance.

Compared with the existing techniques, the achieved solution has several advantages. By not using additional discrete hardware for energy measuring, implementation in other systems only requires that the GPU exposes performance counters and that it allows clock frequencies to be set. Because the basis for DFS decisions are performance metrics based on event counts and GPU parameters, energy usage reduction is not tied to a particular library of GPU functions. This way, the afore mentioned granularity problems are avoided as well as the problems that come with the ability for the GPU to autonomously generate workloads. On the tested set of benchmarks it was possible to achieve energy usage reductions of up to 24% when compared with running the benchmark at the default GPU clock settings and with a performance penalty of 45%. This result was also better than setting the GPU to the lowest clock settings possible which caused an increase in energy usage. Although other real-time energy reduction techniques could not be found in order to establish a comparison term, this result can be compared to some extent with that of GreenGPU [6]. GreenGPU claims up to 21% energy savings in only one particular benchmark while the presented technique achieved higher energy reductions in a more extensive benchmark set.

This solution allows the user to have flexibility in fine tuning the operational parameters. Although the mechanism to autonomously adjust the number of low frequency E-cycles failed to optimize for the tested cases, its proper operation could be a matter of getting more frequent power reports from NVML. The E-cycle lock mechanism successfully achieved its goal of balancing energy savings and performance degradation.

The main disadvantages of this solution are that, for the moment, only the Tesla family of GPUs fulfills the requirements for the technique to be applied, and that optimization is limited with only two available memory clock frequencies.

References


