Exploring Energy Management on GPUs in Game Architectures

Marcelo Zamith*, Luis Valente†, Mark Joselli‡, José Ricardo Silva Junior§, Esteban Clua§ and Bruno Feijó†
* Universidade Federal Rural do Rio de Janeiro
e-mail: zamith.marcelo@gmail.com
† VisionLab/PUC-Rio
e-mail: lvalente,bfeijo@inf.puc-rio.br
‡ Escola Politécnica Pontifícia Universidade Católica do Paraná - PUCPR
e-mail:mark.joselli@pucpr.br
§ Universidade Federal Fluminense
e-mail: josericardo.jr@gmail.com, esteban@ic.uff.br

***Abstract*—CPUs and GPUs have been evolving rapidly over time regarding their capabilities and processing power. This has opened many new possibilities for interactive and real time systems, such as more sophisticated scene realism, more precise and complex artificial intelligence, and better physical simulations. However, these improvements come at a cost: increase of energy consumption. Energy management in interactive and real time architectures have not been receiving much attention over the years, but this issue is likely to become important in the near future due to the increasing energy demand and consumption required by top-notch game applications (especially regarding the mobile and portable consoles). In this paper we introduce the concept of intelligent energy management for games and interactive systems and address the aforementioned issue through these contributions: 1) an investigation of works related to energy management (in general); 2) implementations of feasibility tests for energy management on GPUs; and 3) a novel game architecture with energy management, using multiple GPUs.**

***Keywords*—energy management, parallel computing, multi-thread, GPGPU, game loop models, real-time systems, multiple GPUs.**

I. Introduction

Computer games have become very sophisticated regarding rendering enhancements, modeling, animations, artificial intelligence, and physics simulations. A condition that contributed to this scenario is that available processing power of multicore CPUs and GPUs has been increasing greatly over the last years. Another condition is the development of sophisticated GPU architectures has led also to the GPGPU paradigm, where real-time applications (as games and simulations) adopt GPUs for general computation, allowing the inclusion of new features in games and real time simulations, such as Monte Carlo Methods [1], artificial intelligence [2], crowd simulation [3], fluid simulation [4], and ray casting [5].

In order to coordinate the correct execution of all game activities, a game architecture employs at its core a structure known as the *game loop*. The game loop is responsible to keep the "heart beat" of a game, and thus providing the illusion that all activities in a game are happening simultaneously. This "illusion" is a peculiarity of interactive real-time applications. We say that these applications have *real-time requirements* because if these applications are not able to process their tasks on time, the user experience will not be good enough — in fact, user experience could be severely impaired, thus breaking the "illusion" that the game provides. The literature presents some works that address the nature of game loop models, such as: Dalmau [6], Valente et al. [7], Dickinson [8], Watte [9], Gabb and Lake [10], Joselli et al [11], and Mönkkönen [12].

As games became more sophisticated, the activities that a game loop coordinates have greater hardware requirements in terms of processing power (regarding CPUs and GPUs). These requirements come at a cost — increased energy consumption, as well as energy waste, high hardware power requirements, and thermal management requirements. Hence, this is an issue that game architectures will have to address in the near future.

Considering non-gaming applications, research on energy consumption and thermal control has been receiving more attention over the last years. In the energy consumption area, an example is the development of techniques as Dynamic Voltage Scaling (DVS), which enables applications to manipulate the processor clock frequency to reduce energy consumption and processor temperature as a consequence ([13], [14], and [15]). In the thermal control area, an example is research on *dynamic thermal management* ([16], [17]), which has become necessary for these new multicore hardware. Thermal management is important because if the system temperature increases too much, the hardware can be severely damaged.

Until recently, custom energy management on GPUs was impossible because the hardware did not provide the means to implement it. This situation started to change with the latest nVidia GPU series (codenamed "Kepler" [18]), which enables developers to implement different thermal management policies through a library named NVML (nVidia Management Library [19]. Previous generations of nVidia GPUs implemented thermal management automatically through the graphics device driver, which works well when the system has only one GPU. However, in a system with more GPUs, there is some energy waste resulting in unnecessary temperature increase. In these cases, the automatic thermal management policy implemented by GPU device drivers usually sets up the GPU clock to the highest possible value whenever the GPU is working. When the GPU is idle, the driver sets the GPU clock to the lowest possible value. Using this policy generates energy waste when the system has two or more GPUs, and some GPU needs

ISSN: 2236-3297

to wait for another one to finish processing before carrying on running tasks. In this case, a good strategy is reducing GPU clocks in order to balance GPU processing times, thus balancing hardware usage and lowering system temperature and energy consumption.

When it comes to game architectures, we have not been able to find works that address energy management in games. In order to help in filling this gap, in this paper we provide these contributions: 1) an investigation of works related to energy management (in general); 2) implementations of feasibility tests for energy management on GPUs; and 3) a novel parallel game architecture that uses multiple GPUs and applies energy management .

We start by investigating how energy management has been applied to general applications (non-games). These energy management strategies are concerned only with CPUs.

We provide feasibility tests to study how energy management works on recent GPUs. These tests serve as a proof of concept, since we wanted to know if it was worthy to explore energy management on GPUs and integrate it in a game architecture.

Our novel game architecture consists of a game loop with integrated GPU thermal management. The "game loop" represents the central core of game, being a structure responsible for coordinating the execution of various game tasks (rendering, receiving input, physics simulations, game logic, among others). Our architecture integrates multiple GPUs as resources in the game loop, making it possible to use GPUs for rendering and GPGPU activities. Our architecture is based on heuristics that make it possible to implement different GPU energy management strategies. We want to be able to achieve an acceptable FPS (frames per second) rate for the application, while providing automatic energy management. For example, an acceptable FPS application rate is about 30 frames per second. We provide two specific test cases for this architecture. The first one addresses a physics problem and the second one regards adaptative Bézier tessellations.

This work is organized as follows: Section II presents an overview on game loop models and related works, as well as works related to energy management. Section III presents our game loop architecture. Section IV discusses the two test cases. Finally, Section V presents the conclusions.

II. Related Work

This section provides an overview on game loop architectures and related works. We also provide related works to energy management in general. The works related to energy management concern non-games applications and focus on CPU energy management. We have not been able to find works that discussed game architectures that applied energy management schemes, although the topic is very important for games and virtual simulations that run in mobile phones.

A. Game Loop Architectures

Games and real time simulation processes tasks typically fall into three general stages: data acquisition, data processing, and presentation. Data acquisition means gathering data from available input devices as mice, joysticks, keyboards, touch screens, and motion sensors. The data processing part refers to interpreting user input, applying simulation rules (the simulation logic), physics simulation, artificial intelligence simulation, and related tasks. The presentation refers to providing feedback to the user about the current simulation state, through images and audio.

This subsection provides an overview on various ways to organize the execution of these tasks. This overview presents basic game loop models (Subsection II-A1), game loop models that use parallel computing (Subsection II-A2 and II-A2a), and game loop models for mobile devices that use cloud services (Subsection II-A2b).

1) Basic Game Loop Models: The simplest possible game loop model corresponds to an architecture with three steps: gathering player input, game update and rendering. In this model, the game runs the three tasks sequentially, as fast as possible. This model is useful for platforms with fixed hardware configuration. A drawback of this model is that when running the game or simulation in machines with different configurations, the user will perceive the game as running faster in more powerful machines, and running slower in less powerful machines. Tasks are organized sequentially and there is no parallelism. This model is known as the "Simple Coupled Model". Figure 1 depicts this scheme.

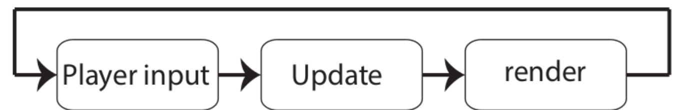

Figure 1. Simple Coupled Model.

A strategy applied in the past ([20],[21]) to solve the main drawback of the Simple Coupled Model was to force a synchronization step at the end of the game loop. For example, the game loop could try to enforce an specific update rate (e.g. 30 frames per second). However, this strategy fails if the game is unable to run all the tasks before the enforced deadline, thus creating lags and disrupting the user experience.

Another strategy to solve the main drawback of the Simple Coupled Model is to make the simulation stages independent of the host machine hardware capacity (i.e. uncoupling these stages). Examples of uncoupling the simulation stages (e.g. game logic update, physics, artificial intelligence) are defining a simulation time unit for some stages (like physics) and forcing some specific stages to run at fixed frequency (like game logic).

The game loop models that apply these uncoupling strategies are known as "uncoupled models". The literature presents Single-Threaded Uncoupled Models ([7], [8]) and Multi-Threaded Uncoupled Models ([7], [10], [12]).

The Single-Threaded Uncoupled Model ([20], [21]) uses the time elapsed between consecutive game loop executions as an input to the update stages, thus defining a simulation time unit. As a consequence, the simulations runs correctly (at the same "speed") in machines with different capacities. A powerful machine would run the loop more often (using a lower delta value), which means it would provide a better experience to the user (more smooth animations, for example). A machine

ISSN: 2236-3297

with low processing power would still provide the correct result, even if this means less presentation quality. The main difference of Single-Threaded and Multi-Threaded Uncoupled Models is that the single-threaded ones avoids dealing with issues related to concurrency.

The Single-Threaded Uncoupled Model tries to considering the elapsed time in the main loop of the game execution by feeding the update stage with a time parameter, as Figure 2 presents this scheme. Existing engines such as Cocos2D [22] adopts this model.

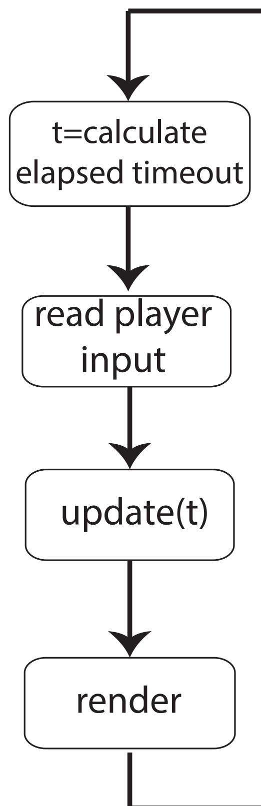

Figure 2. Single-Thread Uncoupled Model.

Although these are working solutions, time measuring may greatly vary in different hardware devices due to many reasons (such as process load), making it difficult to be reproduced faithfully. For example, a network module implementation and program debugging [8] may be easier to implement if the loop uses a deterministic model. Another issue is that running some simulations too frequently, like AI and the game logic, may not necessarily correspond to better results.

In order to address these issues, some researchers proposed Fixed-Frequency Uncoupled Models ([7], [20], [21]). These models feature another update stage that runs at fixed frequency, besides the time-based one. The work by Dalmau [6] presents a similar model, although not naming it explicitly. Those works describe the model using a single-thread approach. Figure 3 llustrates the Fixed-Frequency Uncoupled Model, where the "run game logic" stage runs at a fixed frequency and the "update" stage runs as fast as possible.

An interesting consequence regarding the Fixed-Frequency

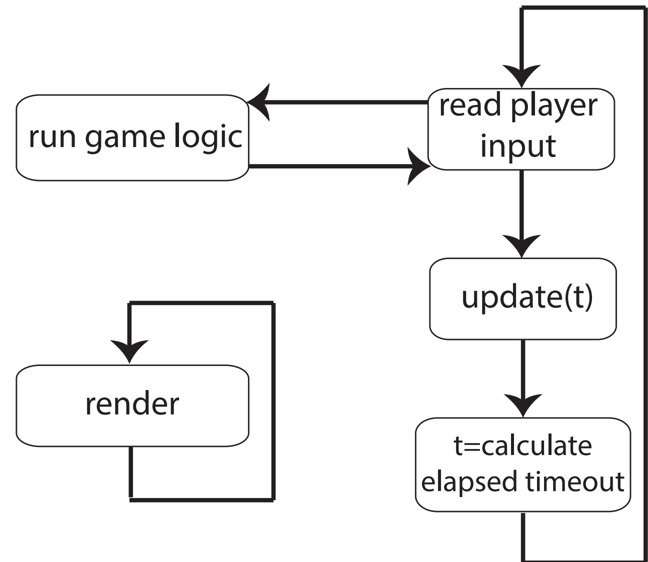

Figure 3. Fixed-Frequency Uncoupled Model.

Uncoupled Model is that it makes the game execution deterministic, which makes it possible to attain reproducibility. There are some game functionalities that can take advantage from game execution determinism, such as replay feature, program debugging, and network module implementation [8].

2) Parallelism in Game Loops: The advances in CPU and GPU architectures brought into view the issue of parallelism. The earlier single-threaded models were more useful at the time when multicore CPUs and GPGPU programming were not widely available or used. More recent game loop models try to take advantage of these resources, by parallelizing game tasks.

However, dealing with concurrent programming introduces another set of problems, such as data sharing, data synchronization, and deadlocks. Also, as Gabb and Lake [10] state, not all tasks can be fully parallelized due to dependencies among them. For example, the system is unable to render a character in the correct state before computing the logic and updating the overall state. Hence, serial tasks represent a bottleneck to parallelizing simulation computation. The remaining of this subsection explores game loop models that apply parallelism using CPUs and GPGPU. In particular, the architectures that use GPGPU are the basis upon which we developed the architecture that includes energy management (Section III).

a) Multi-thread Models for CPUs: Rhalibi et al. [23] present a different approach for real-time loops by taking into consideration dependencies among game related tasks. Their model divides the loop steps into three concurrent threads, creating a cyclic-dependency graph to organize the ordering in game related processing. Each thread divides the rendering and update tasks according to their dependency.

Mönkkönen [12] presents multi-thread game loop models that are grouped into two categories: function parallel models and data parallel models. The first category correspond to models that present concurrent tasks, while the second one concerns models that try to process data entirely in parallel, if

possible. As an example (first category), Mönkkönen proposed the Asynchronous Function Parallel Model, which does not wait for task completion to perform its job. The Asynchronous Function Parallel Model runs the render stage using the last completed game state, even if the update stage is still computing the next one. As an example related to the second category, there is the Synchronous Function Parallel Model [12], which processes the game physics in a separated thread while the main thread process the characters animations. Figure 4 depicts this approach.

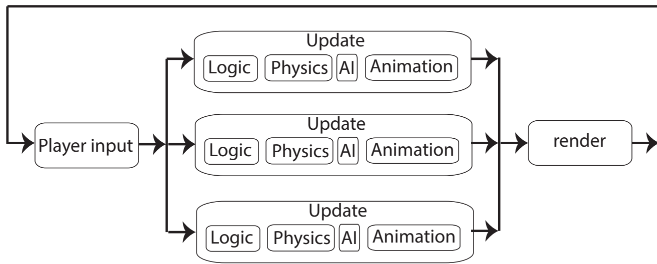

Figure 4. Data Parallel Model.

The Synchronous Function Parallel Model [12] proposes to allocate a thread to all tasks that are (theoretically) independent of each other. For example, performing Physics simulation while calculating animation. Figure 5 illustrates this model. Mönkkönen states that this model is limited by the amount of available processing cores, and the parallel tasks should have little dependency on each other.

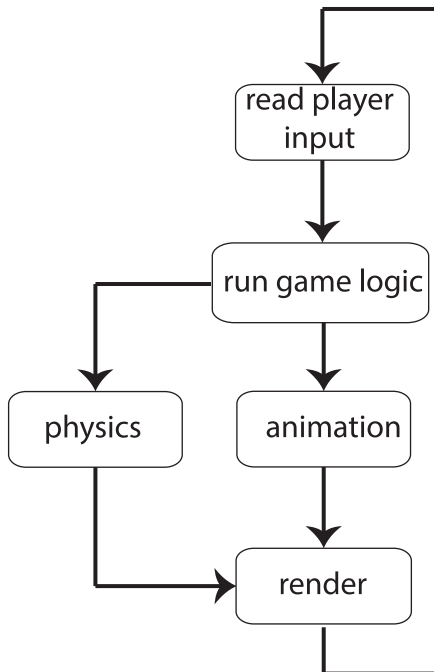

Figure 5. Synchronous Function Parallel Model.

The Asynchronous Function Parallel model [12] uses a dedicated thread to process specific kinds of tasks (such as AI) or tasks that have high interdependencies. In this model, each thread runs in its own loop. The model is deemed as asychronous because tasks do not wait for the completion of others to perform their job. Instead, the tasks use the latest computed result to continue processing. For example, the rendering task would use the latest completed physics information to draw the objects. While this measure decreases the dependency among tasks, the task execution should be carefully scheduled for this scheme to work nicely. Unfortunately, this is often out of the scope of the application. Also, serial parts of the application (like rendering) may limit the performance of parallel tasks [10].

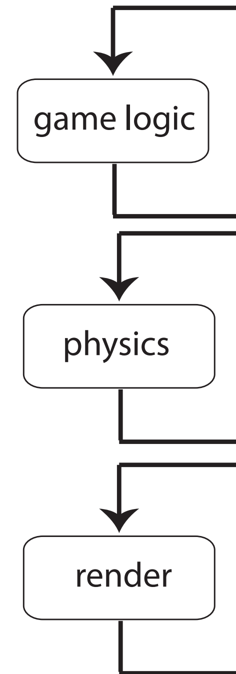

Figure 6. Asynchronous Function Parallel Model.

b) Game Loop Models using GPGPU: The models that this subsection presents use the GPUs as extra co-processors for general computation. Usually, this strategy applies to processing tasks as math problems and physics simulations.

The first game loop model that integrated the GPU for GPGPU tasks used GPUs as math co-processors in real-time applications (as games and physics simulations) [24]. The model by [24] corresponds to a simple coupled model that includes a dedicated GPGPU stage. Figure 7 illustrates this scheme.

Zamith et. al [25], [26] improved the previous game loop model by adding these functionalities: 1) the GPGPU stage is responsible for computing game physics; 2) the GPGPU stage and the other stages are uncoupled, running in separate threads. The idea of using a Multi-Thread Uncoupled Model is based on [10]. The architecture by Zamith et. al [25], [26] implements a static load balancing scheme, using a Lua script to allocate tasks on processors. Figure 8 presents the Multi-Thread Uncoupled Model with a GPGPU stage.

Joselli et. al [11] present a Multi-Thread Uncoupled Model with an automatic load balancing scheme that also integrate GPGPU. The automatic load balancing approach uses heuristics to define task allocation on processors (considering

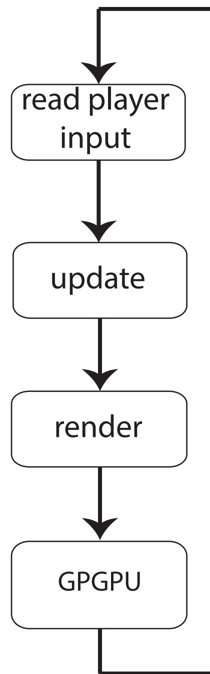

Figure 7. Simple Coupled Model with an GPGPU stage.

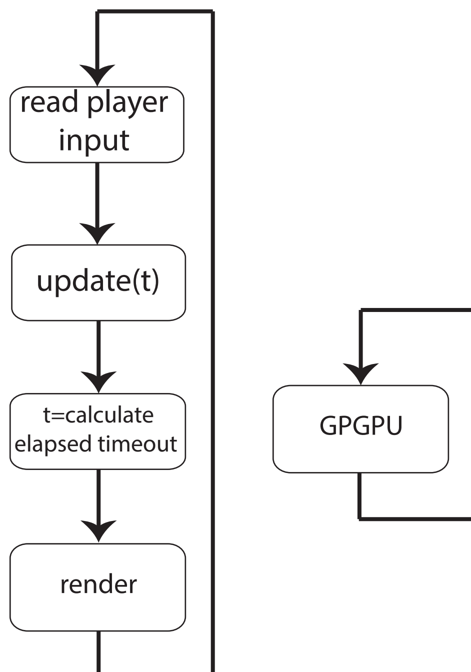

Figure 8. Multi-Thread Uncoupled with GPGPU.

hardware with multicore CPUs and programmable GPUs). This load balancing scheme is able to work dynamically, moving tasks between processors during the application lifetime to guarantee task load balance.

3) Game Loop Models for Mobile Devices that Use Cloud Services: Joselli et. al [27] present a Multi-Thread Uncoupled Model for mobile devices that uses cloud computing. The general idea that [27] proposed is to have a game application to use remote computers to process services that have higher processing power requirements, such as image recognition and speech recognition. As a consequence, the game application will be able to run on mobile devices that do not have the processing power required to process the services that the cloud provides.

The architecture by Joselli et. al [27] provides modules to access cloud services (image and speech recognition), networking, social networks, input, rendering, AI processing, and publishing player achievements to social networks. Figure9 illustrates this model.

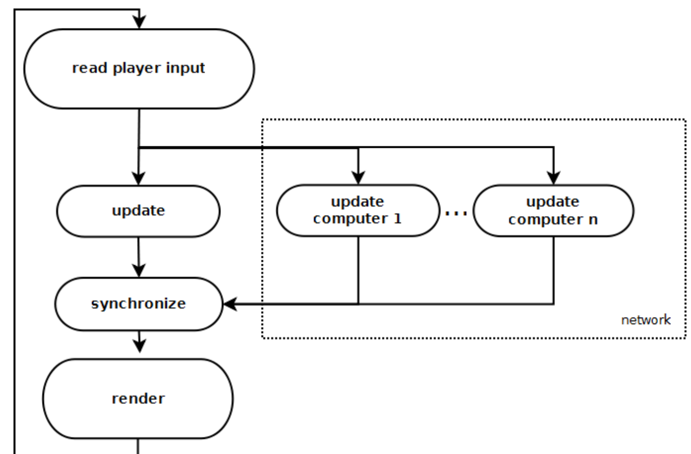

Figure 9. The Cluster Architecture.

B. Energy Management and Thermal Control

There are several works in the literature that explore Dynamic Voltage Scaling (DVS) techniques to implement energy management as a result of reducing temperature. DVS is a technique that enables to reduce processor temperature by manipulating processor clock frequency through software [13] [14], [15].

Applications that use DVS are real-time applications that solve an optimization problem — to maximize CPU idle time while minimizing energy consumption and maintaining real-time requirements. In order to accomplish this goal, the algorithm applies a policy and reduces or raises the processor clock through DVS. The kinds of tasks that exist in these applications are usually recurrent – they are processed several times during the application lifetime. As a result, it is possible to analyse task behavior according to clock frequency changes.

Our game loop architecture employs an idea similar to DVS through nVidia's NVML library. The main difference is that our architecture applies the idea to GPUs, while works related to DVS apply the idea to CPUs only. This section explores some of these works.

Trevor et. al [28] analyse four algorithms related to energy management and thermal control that employ DVS techniques: FLAT, COPT, PAST, and AVG. These algorithms use different policies to change processor clock as means to optimize energy consumption. Trevor et. al [28] wanted to compare

the algorithms to learn about how much energy they could save. The FLAT algorithm defines a fixed value for voltage to use during the entire application. The COPT algorithm takes advantage of a task performance history to learn about how much energy a task usually requires, in order to adjust the processor clock to meet the energy requirement. The PAST algorithm uses the last CPU idle time value as basis to calculate a CPU clock value that is able to improve energy consumption and keep the task real-time requirements. The AVG algorithm is similar to the PAST algorithm. The main different is that the AVG algorithm uses an average of all CPU idle times as basis to calculate the new CPU clock value.

Kim et. al [29] propose a DVS optimization algorithm. This algorithm aims at minimizing the time spent computing periodic tasks. In this algorithm, each task has a priority and a deadline. The tasks are organized in a queue data structure that defines the task priorities. When a task reaches its deadline, the algorithm moves the task to the end of the queue. While a task is running, the algorithm analyses the task to define the appropriate processor voltage value to use in order to achieve the optimization goal.

Another work by Kim et. al [30] proposes another DVS optimization algorithm, focused on preemptive task control. In their model, the tasks have different priority levels. This algorithm changes processor voltage values to achieve two goals: 1) to reduce the elapsed time of a lower-priority task before it is time to process a higher priority task; 2) to delay running a higher-priority task so as a lower-priority completes execution without being preempted.

Xiaobo et. al [31] developed a DVS algorithm to manage processor energy consumption in a system that has a memory energy management system controlled by hardware. This memory management system is independent and cannot be controlled through software. Xiaobo et. al [31] consider their results satisfactory as they are able to raise the CPU processor clock and still have benefits in total energy consumption, because their approach takes into consideration the effects of the independent memory management system on energy consumption.

Zhang et. al [32] propose a DVS algorithm composed by two parts: The first part is pre-computed (off-line) while the second one runs on-line. In the first part, the algorithm analyzes a log file containing task information to learn about the average task running time and task priority. In the second part (on-line), the algorithm uses the task information to calculate the clock frequency in order to minimize energy consumption while meeting the task real-time requirements.

III. The Proposed Game Loop

This work proposes a parallel game loop architecture for one or multiple GPUs that provides a module to manage GPU energy consumption. This work extends the architecture first proposed by Joselli et. al [33], which yields an efficient automatic load balancing scheme for game tasks among GPUs.

Before going further, it is necessary to define two concepts: 1) a “task” is something that the game needs to process using either the CPU or the GPU. For example, processing artificial intelligence, physics, and rendering; 2) an “idle state” in the CPU or GPU happens when there is nothing to process. In this state, a task does not exist.

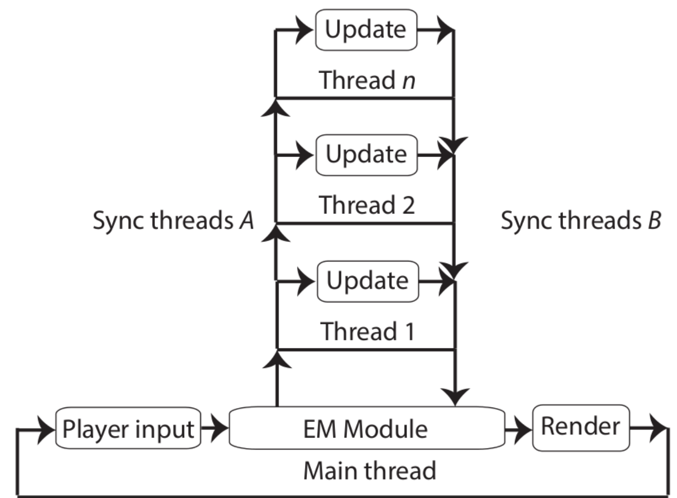

Figure 10. Parallel game loop model with multiple GPUs.

Some game related tasks (as reading input devices, playing audio, and applying force feedback effects on joysticks) can be performed only by CPUs. Other kinds of tasks (as intense mathematics calculations), can be performed either on CPUs or GPUs. Finally, some tasks are processed only by GPUs, as rendering.

The proposed architecture comprises one main thread and several secondary threads. The main thread is responsible for reading player input, running the Energy Manager, and rendering. The secondary threads correspond to behavioral tasks (as tasks related to AI and Physics). Some update tasks run on GPU (GPGPU task). In order to guarantee data consistency for rendering, the architecture uses a synchronization scheme based on semaphores. Figure 10 illustrates the architecture.

The architecture assigns a CPU thread to each GPGPU task. As GPGPU tasks are not able to access main memory (due to GPU hardware limitations), the CPU thread stores a copy of some game data (as player input and physics simulation data) to share with the GPGPU task. Figure 10 also illustrates this one-to-one relationship. The reader should refer to [26] for more details on this approach.

The Energy Manager (EM) represents the core of the proposed architecture. The EM is responsible for monitoring the GPU temperatures and adjusting GPU clocks to avoid wasting energy, managing all the different tasks, and synchronizing all threads and data exchange. The EM adjusts GPU clock frequencies (processor and memory access) and synchronizes all threads. The EM itself is also a task run by the CPU.

The EM synchronizes all threads (including the main one) each time the game loop runs to share player input data among the threads (*Sync threads A* in Figure 10). The EM performs another synchronization step to gather data from each GPGPU task thread for the render task, so the rendering process happens correctly and consistently (*Sync threads B* in Figure 10).

When a task that requires heavy GPU processing is running (*e.g.* rendering), the EM observers energy consumption of other GPUs (dedicated to other tasks) and reduces the clocks of these GPUs when it detects that these GPUs are idle. Otherwise, if the EM detects that a GPU is not running a task at that cycle, it adjusts the GPU clocks (memory access and processor) to obtain better performance.

In parallel architectures, a thread could finish processing before another one. Consequently, it is possible that some threads enter the idle state while others do not. A strategy to solve this problem is to keep all threads occupied through a load balancing scheme. The architecture provides this strategy as it is an extension of Joselli et. al [11].

Typically, GPUs are fast enough to process all their game tasks and wait for game rendering. If the graphics card driver controls the GPU temperatures automatically, it adjusts the GPU clocks (memory access and processor) to the highest possible value when they are active. Therefore, a good strategy is to apply an energy management policy that reduces GPU clocks (and consequently, system temperature).

The EM employs heuristics to decide when it should change the GPU clock frequency (memory access and processor). Our architecture provides a default heuristic to accomplish this task. However, it is possible to define other heuristics through Lua scripts, if desired. This makes it possible to test different energy management heuristics without rebuilding the application.

Next subsection details the default heuristic that the Energy Manager applies.

A. The Default Energy Manager Heuristic

The heuristic accepts two input parameters, each one being a high-precision floating point value (double). The first one is a double value representing rendering elapsed time. The second one is a double value representing the GPU task elapsed time, for a given thread i. The heuristic produces one output parameter, an integer number with three possible values: -1, which informs that the EM should reduce the GPU clock frequency; 1, which informs the EM to increase the GPU clock frequency and 0, which informs the EM to maintain the current GPU clock frequency.

The heuristic compares performance of a single GPU (running a GPU task) with the rendering GPU. If a GPU is faster than the rendering GPU, the heuristic determines that the GPU should have its clock reduced. If the GPU is slower than the rendering pipeline, the heuristic determines that the GPU should have its clock increased. Otherwise, the heuristic does nothing.

The architecture needs to run the heuristic periodically to evaluate the system in order to perform energy management. Games are dynamic applications with tasks that game tasks can be processed very quickly (e.g. a simple particle system). If the architecture fails to run the heuristic in reasonable time, energy management can become inefficient. Thus, we empirically chose to run the heuristic every 10 frames in the beginning. Choosing an interval much higher than this value could make the heuristic miss valuable information about task performance. By the other way, using values much lower than 10 (like at every frame) could be costly and become a burden to the application.

The EM is responsible for running the heuristic for all GPUs (running GPU tasks) in the system. The metric that the heuristic uses to compare performance is the mean elapsed time for the past 10 frames (for both GPU and render tasks). As the GPU configurations could possibly change, it is necessary to wait some time for the new configurations to take effect. Hence, the heuristic waits for 10 frames before running again. Algorithm 1 presents the heuristic pseudocode.

Algorithm 1 Energy Manager Heuristic Algorithm

```
if frameCount == 10 then
  meanElapsedTimeRender = getMeanRenderElapsedTime(frameCount)
  meanGPGPUelapsedTime = getMeanGPGPUElapsedTime(frameCount)
  if GPGPUelapsedTime < elapsedTimeRender then
    return -1
  else
    if GPGPUelapsedTime > elapsedTimeRender then
      return 1
    else
      return 0
    end if
  end if
else
  if frameCount == 20 then
    frameCount = 0
  end if
end if
```

IV. Tests and Results

This section describes the tests we have performed to investigate the feasibility of our architecture. We have conducted two tests.

The first test is a benchmarking to learn about the new functionality that the Kepler GPU series offer related to energy management. We were interested to know if following this path would be convenient.

The second test solves a physics simulation on the GPU using our architecture. This physics simulation corresponds to an explosion (sound wave propagation) that travels through an environment with large obstacles, a problem that requires high processing power.

The test platform is an Intel i7 with four physical cores of 3.60 GHz, 8GB RAM memory and two GPUs: nVidia K20 and nVidia GTX480. The GTX480 runs the render task and the K20 runs the GPGPU task.

Both tests use NVML library API calls [19] to change memory access and clock frequencies in the K20 GPU. This API works by providing both values in one call, as a value-pair. Currently, there is a hardware limitation as the possible values to use are limited. The NVML offers functions to query the possible value-pairs according to the GPU model. For example, currently the K20 GPU accepts the following value-pairs: $2,600/758$, $2,600/705$, $2,600/666$, $2,600/640$, $2,600/614$ and $314/314$ (memory/processor clock frequency).

The remaining of this section discusses the two tests.

A. Test: Benchmarking

We implemented a benchmark to answer the following question: If we changed the GPU clocks, what would be the performance gain or loss considering the nature of the task. In this case we consider tasks that require more memory accesses versus tasks that require more processing power. We were interested in learning about GPU thermal and energy consumption with different kinds of tasks.

Our benchmark is based on four applications that ship with the nVidia SDK [34]. We extended these applications in two ways: i) by implementing functionality to record a log file containing the temperature and clock values at each step; and ii) by having the applications accept parameters that indicate the memory and processor clock values to use. Nowadays, these applications work only with the K20 GPU, as changing memory and processor clocks is a functionality first introduced in the Kepler GPU series. New mobile GPU chips are being announced with this kind of technology.

The selected applications are: *cdpAdvancedQuicksort*, *cdpLUDecomposition*, *radixSortThrust*, and *cdpBezierTessellation*. The *cdpAdvancedQuicksort* application implements a parallel quick sort algorithm. The *cdpLUDecomposition* application runs a parallel LU decomposition. The *radixSortThrust* application implements a parallel version of the radix sort algorithm. The *cdpBezierTessellation* application dynamically generates a tessellation pattern of Bézier surfaces, which makes it possible to perform tessellation of complex models in real time [35].

We selected these specific applications because:

- they demand high computation power;
- the sorting algorithms demand more memory access than GPU processing;
- the LU decomposition as well as the bezier tessellation require more GPU processing than memory access;
- these specific applications are simpler to extend than other ones that exist in the nVidia SDK;

The original applications are able to run the algorithms with different domain sizes. For our benchmark, we selected the same domain size for all applications (8,192 elements) because this is the maximum value that the *cdpLUDecomposition* application accepts. (the other three applications accept higher values).

The tests consisted in running each application 1,000 times, using different clock frequencies. The tests recorded the GPU temperature and elapsed times (memory transfer and GPU processing) for each clock frequency. We decided to execute each application 1,000 times because the processor *temperature does not change immediately after changing clock frequency*. This is a behavior that Hefner and Blackburn [36] suggested and we were able to confirm it. However, changing clock frequencies affects immediately the elapsed times of memory transfer and GPU processing.

1) Results: Table I presents the benchmark results. Column *memory/GPU* indicates the clock frequencies of memory access and GPU processor, respectively. Columns *memory* and *GPU* represent the elapsed time of memory transfer and GPU processing, in milliseconds. All values in Table I represent the averages of all 1,000 runs.

The test results in Table I demonstrate that the applications that require more GPU processing (*cdpLUDecomposition* and *cdpBezierTessellation*) are the most affected when the GPU processor clock changes. The results also demonstrate that the applications that require more memory access (as the sorting algorithms) are more impacted by changes in memory access clock than in changes in GPU processor clock. The other two applications (*cdpLUDecomposition* and *cdpBezierTessellation*) that require more processing power than memory access are more influenced by changes in GPU processor clock.

We classify these tests in three groups: tests related to changing the memory access clock, tests related to changing the GPU processor clock, and test related to GPU temperature behavior. The next subsections analyse these tests.

2) Tests Related to Memory Access Clock: Figure 11 displays memory access elapsed times of sorting algorithms (*cdpAdvancedQuicksort*, *radixSortThrust*) when memory access clock changes. Figure 11 illustrates that changing memory clock does not impact memory access performance for these applications.

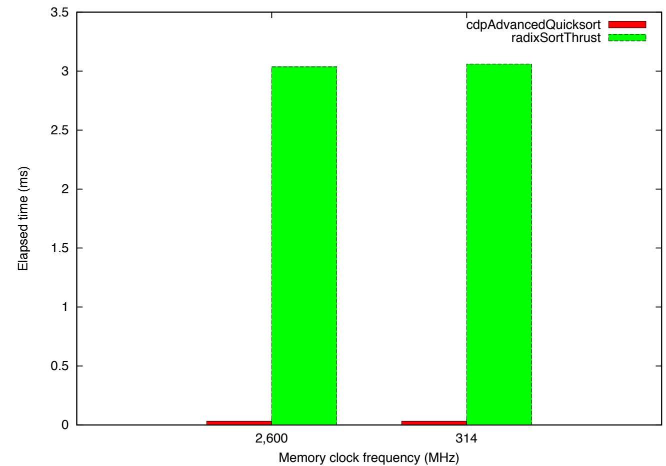

Figure 11. Memory access elapsed time of *cdpAdvancedQuicksort* and *radixSortThrust*.

Figure 12 display the results about memory access regarding the *cdpLUDecomposition* application. This figure illustrates that despite the $2,600Mhz$ clock frequency being significantly higher than the $314Mhz$ frequency, the performance gain (1.07) was not also significantly higher.

Figure 12 illustrates the elapsed time to run the *cdpBezierTessellation* application using different memory access clock frequencies. By changing the memory access clock frequencies, we were able to attain a performance increase of approximately 11%. However, this performance increase came at the cost of increasing the memory access clock in approximately 731%.

3) Tests Related to GPU Processor Clock: Figure 13 illustrates the GPU processing elapsed time using different clock values,

ISSN: 2236-3297

TABLE I. APPLICATION BENCHMARKING PERFORMANCE

	cdpAdvancedQuicksort		*cdpLUDecomposition*		*radixSortThrust*		*cdpBezierTessellation*	
memory/GPU	memory	GPU	memory	GPU	memory	GPU	memory	GPU
2,600/768	0.032	3.003	0.030	8,020.080	3.036	80.845	0.097	3.051
2,600/705	0.032	3.005	0.030	12,147.500	3.026	81.123	0.097	3.073
2,600/666	0.032	3.013	0.030	16,315.200	3.014	80.397	0.097	3.086
2,600/640	0.032	3.010	0.030	20,500.800	3.032	80.453	0.097	3.091
2,600/614	0.032	3.048	0.030	24,715.400	3.038	81.583	0.097	3.103
314/314	0.032	3.079	0.032	28,861.500	3.060	80.233	0.108	3.118

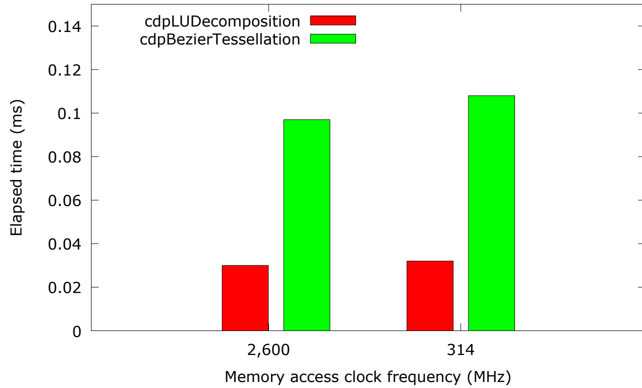

Figure 12. Memory access elapsed time.

for these applications: textitcdpAdvancedQuicksort, *radixSortThrust*. It is possible to notice in Figure 13 that the GPU processing time is almost constant regardless of clock value.

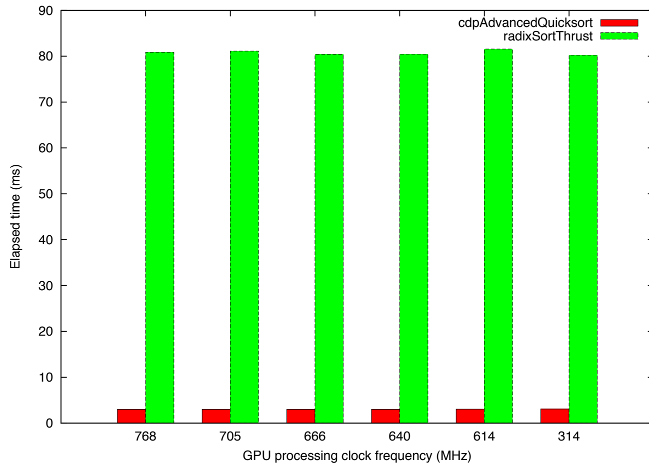

Figure 13. GPU processing elapsed time of *cdpAdvancedQuicksort* and *radixSortThrust*.

Changing the GPU processor clock frequency directly affects the performance of this application. We expected this behavior as *cdpLUDecomposition* requires high computational power. Figure 14 illustrates that the *cdpLUDecomposition* performance changes linearly as processor clock changes.

Figure 15 illustrates how different GPU clock frequencies influence the performance of *cdpBezierTessellation*. We were

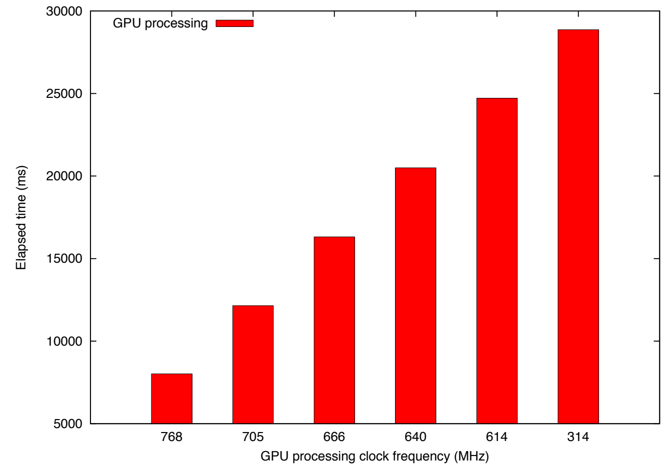

Figure 14. GPU processing elapsed time (*cdpLUDecomposition*).

able to improve the performance of this application in 11

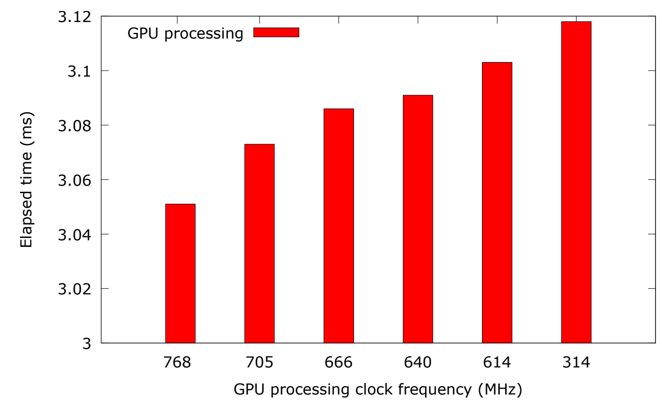

Figure 15. GPU processing elapsed time of *cdpBezierTessellation*.

4) GPU Temperature Behavior: In the GPU temperature behavior test, we wanted to observe how the temperature would change when we ran the benchmark applications. We were particularly interested in learning about how much time it would take for the GPU hardware temperature to rise and stabilize at the peak value. We wanted to use this information to help in evaluating if it is feasible to use temperature variation as data for managing energy consumption. If the time elapsed to reach the peak temperature is too long, using temperature variation to manage energy consumption would be infeasible,

as at the time the peak temperature is reached, the system would already have consumed too much energy.

The starting condition for this test was the GPU temperature at 30 degrees Celsius. This was the temperature in an idle state, just after powering up the system. This test used these values for the memory access and processor clocks, respectively: 2,600 and 768. These values are the maximum supported values for the test hardware. The test consisted of a batch of 1,000 runs for each benchmarking application. Figure 16 illustrates a chart with temperature variations for all four applications.

Figure 16 illustrates that the GPU temperature takes almost 10 milliseconds in order to achieve the maximum temperature. For example, the *cdpAdvancedQuicksort* was the slowest one to reach temperature stabilization (in 7.5 milliseconds) ending at 45 degrees Celsius, while the *radixSortThrust* application was the fastest to reach temperature stabilization (2 milliseconds), ending at 47 degrees Celsius. The *cdpLUDecomposition* reached temperature stabilization in 3 milliseconds, ending at 50 degrees Celsius. Finally, *cdpBezierTessellation* achieved the peak temperature (45 degree Celsius) after 2.8 milliseconds. These results illustrates that the temperature does not take much time to reach the peak value, which makes it feasible to use temperature variation as input for an energy management scheme.

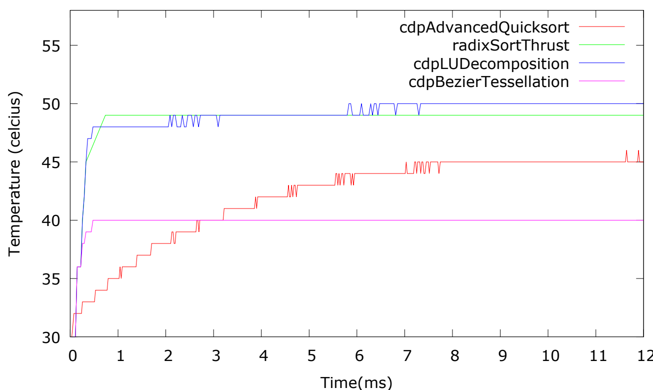

Figure 16. GPU temperature as a function of time.

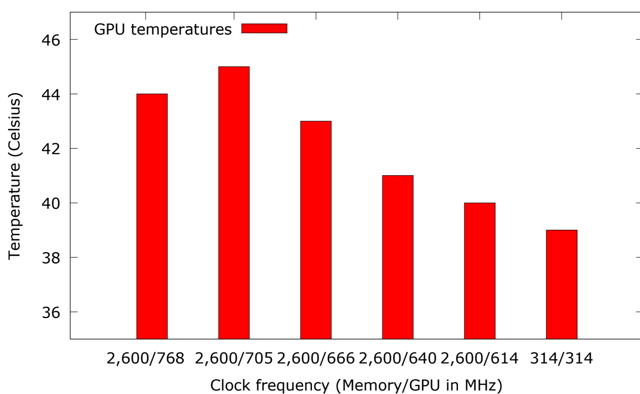

Figure 17. Temperature of GPU in *cdpBezierTessellation* benchmark.

B. Test: The Physics Simulation - Shock-wave Explosion

The second test corresponds to a shock wave simulation that models how a shock wave (originated from an explosion) travels through an environment with large obstacles, such as a city with tall buildings. The application used the resulting amplitude field to render the propagation of a shock-wave-like effect at each frame step. Although this example is not a game, the shock-wave simulation requires solving a physics model in real-time, which is a common feature found in current games.

Solving these kinds of simulations through an analytical solution becomes impossible depending on the medium selected for the simulation. This is the case of our test. In these cases it is necessary to use approximative techniques to solve the simulation. In this regard, we selected Finite Difference Methods (FDM) [37]. Generally speaking, it is not possible to solve FDMs in real-time due to high computing demands. However, due to the high processing power of GPUs, processing FDMs in real-time becomes possible. The reader should refer to [37] for implementations details about solving FDM on GPUs (that also applies to the physics simulation that this test solves).

1) Test Scene Description: The outdoor environment was represented by a lattice with larger cells, using Reynold's boundary condition [38] to simulate domain continuity. The buildings were represented by zeroing the velocity of the cells that intercepted the visual models at the ground level. The kernel parameters for this experiment are: $\Delta h = 1.0$ meter, $\Delta t = 0.0033$ seconds, and the domain was variable from 128×128 points to 4096×4096, doubling the square size. As $\Delta h = 1.0$, the scene follows the same proportion. In other words, for each simulation the test doubles the square size, starting from 128×128 meters to 4096×4096 meters. Domain sizes larger than 4096×4096 are infeasible to process using our test hardware.

Figure 18 display the test scene geometry and the evolution of the shock-wave propagation effect in time. The wave propagation starts in (A) and ends in (D). The buildings interfere in the wave propagation. Parts (A) and (B) omit buildings to help in visualizing wave reflection. Parts (C) and (D) present the complete scene.

2) Results: Figure 19 illustrates that changing clocks in GPUs responsible for the GPGPU tasks does not influence the rendering elapsed time. This result suggests that the rendering task is the heaviest one in our example.

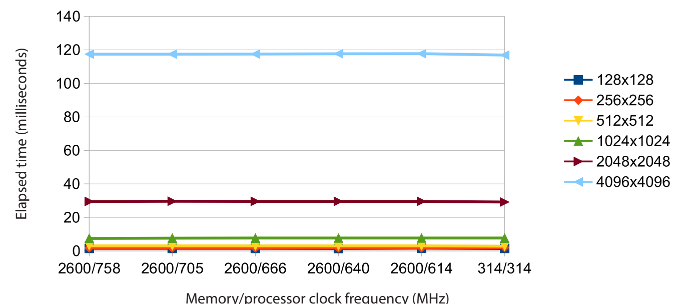

Figure 19. Render task elapsed time.

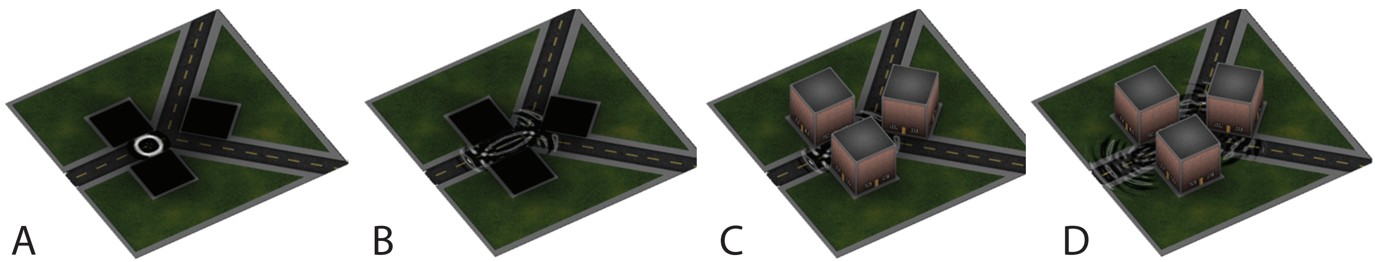

Figure 18. Test scene geometry and the evolution of wave propagation effect in time. Parts (A) and (B) omit buildings to help in visualizing wave reflection. Parts (C) and (D) present the complete scene.

Figure 20 illustrates a pattern: when the processor clock value changes (while keeping the same memory clock value and domain size), the elapsed time does not change significantly. Changes in elapsed time are significant only when the memory access clock also changes.

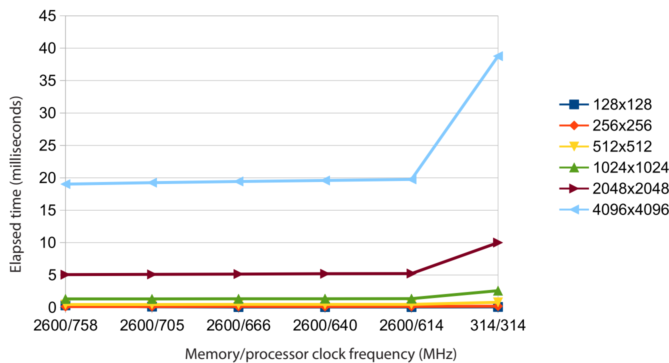

Figure 20. Update task elapsed time.

Figure 21 illustrates a similar pattern as Figure 20. For a given domain size, the temperature does not change significantly when the processor clock value changes (while keeping the same memory access clock). The GPU temperature changes more significantly when the application varies the GPU memory access clock.

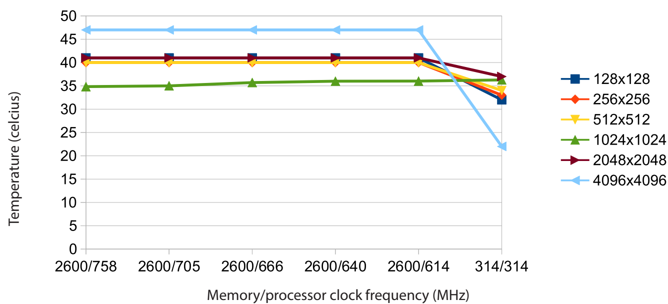

Figure 21. The GPU temperature.

When using $2,048 \times 2,048$ as the domain size, the test performance is 33 FPS, which we consider a reasonable FPS rate. On the other hand, the test performance is worse when using a larger domain $(4,096 \times 4,096)$, resulting in approximately $8,3$ FPS. Figure 21 illustrates these results (the FPS is calculated as $\frac{1}{X}$ where X is the time in seconds that Figure 21 represents).

These results demonstrate that there is energy consumption reduction while not hindering application performance. However, we were not able to know the specific amount of saved energy as the NVML library does not provide an API to query this information.

V. Conclusion and Future Works

The rise in processing power regarding current multicore CPUs and programmable GPUs makes it possible to have more sophisticated games and real-time simulations. However, such hardware requires more energy to operate, thus elevating energy consumption. In cases where energy is a scarce resource, this issue is highly important (e.g. for devices that use batteries, such as mobile phones).

In current GPUs, the driver is responsible for managing the clock frequencies. We observed that the driver usually raises the GPU clock frequencies to the highest possible values when it perceives that the processing load increases. However, the driver usually maintains these high clock frequency values for the lifetime of the application even if the processing load lowers. This default behavior opens up the possibility to energy waste.

As new GPUs (the Kepler GPU series) enable indirect energy management through changing GPU clocks, we considered that using this functionality for energy management in games and interactive applications could be a promising idea. We first approached the idea by conducting a benchmark test to explore this functionality. We believed the test results were interesting enough to start exploring this functionality in a game architecture.

The benchmark results suggest that applications have unique behaviors regarding GPU processing. As examples, some tasks in applications require more memory access, while other tasks demand more processing power. In this sense, an energy management strategy must take into account the nature of tasks. In case of games, a challenge is to develop strategies that change clock frequencies dynamically in a simple way, while keeping real-time and interactivity requirements.

The new GPUs (Kepler architecture) do not provide direct ways to measure energy consumption. Due to this issue, we use an indirect way to know if energy consumption has risen or lowered — monitoring temperature variation. Using temperature variation is feasible because the time it takes

for the GPU hardware to reach and stabilize at the peak temperature is not very long.

Although its possible to affect the behavior of new GPUs (Kepler series) programmatically, the current API has limitations. For example, the current API requires the developer to provide a pair of values (memory clock, processor clock) to change the GPU behavior. In this regard, it is not possible to change each variable separately. As a consequence, we are unable to analyse the impact of each variable in isolation. Another problem is that the new GPUs accept only a set of predefined clock values.

In this sense, this paper proposed a scheme to manage GPU energy consumption in games through a multi-thread game loop model. Although we have not tested the proposed architecture with a real game, we tested the architecture with a game-related task (a physics simulation) that has high processing demands, while being able to keep real-time requirements. Our architecture also makes it possible to define heuristics through Lua scripts. This enables testing different energy management heuristics without rebuilding the application.

Research on energy management for games is scarce. We were not able to find works related to this topic in the literature. Additionally, the Kepler GPU series represent the first generation of GPUs that enables energy management. In this sense, our work represents a first attempt at proposing a solution for this area. With this being said, research on energy management for games is a novel area that has a long and promising road ahead, which requires further investigation. It is well known that hardware assemblers are focusing lot of effort for increasing and optimizing energy management. We believe that intelligent real time management of energy will become an important issue for future games and interactive applications.

Acknowledgements

The authors gratefully acknowledge CNPq, CAPES, FINEP and FAPERJ for the financial support of this work.

References

[1] A. G. Anderson, W. A. G. III, and P. Schrder, "Quantum monte carlo on graphical processing units," *Computer Physics Communications*, vol. 177, no. 3, pp. 298 – 306, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465507001993

[2] T. Rudomín, E. Millán, and B. Hernández, "Fragment shaders for agent animation using finite state machines." *Simulation Modelling Practice and Theory 13(8)*, pp. 741–751, 2005.

[3] M. Joselli, E. B. Passos, M. Zamith, E. Clua, A. Montenegro, and B. Feijó, "A neighborhood grid data structure for massive 3d crowd simulation on gpu," in *Games and Digital Entertainment (SBGAMES), 2009 VIII Brazilian Symposium on*. IEEE, 2009, pp. 121–131.

[4] S. R. J. Junior, E. Clua, A. Montenegro, M. Lage, A. M. Dreux, M. Joselli, P. Pagliosa, and C. L. Kuryla, "A heterogeneous system based on gpu and multi-core cpu for real-time fluid and rigid body simulation," *International Journal of Computational Fluid Dynamics*, vol. 26, no. 3, pp. 193–204, 2012.

[5] E. Gobbetti, F. Marton, and J. A. I. Guitián, "A single-pass gpu ray casting framework for interactive out-of-core rendering of massive volumetric datasets," *The Visual Computer*, vol. 24, no. 7-9, pp. 797–806, 2008.

[6] D. S. C. Dalmau, *Core Techniques and Algorithms in Game Programming*. New Riders Publishing, 2003.

[7] L. Valente, A. Conci, and B. Feijó, "Real time game loop models for single-player computer games," in *Proceedings of the IV Brazilian Symposium on Computer Games and Digital Entertainment*, 2005, pp. 89–99.

[8] P. Dickinson, "Instant replay: Building a game engine with reproducible behavior," Available at `http://www.gamasutra.com/features/20010713/dickinson_01.htm/` , 2001.

[9] J. Watte, "Canonical game loop," Available at `www.mindcontrol.org/~hplus/graphics/game_loop.html/` , 2005.

[10] H. Gabb and A. Lake, "Threading 3d game engine basics," Available at `http://www.gamasutra.com/features/20051117/gabb_01.shtml/` , 2005.

[11] M. Joselli, M. Zamith, E. Clua, R. Leal-Toledo, A. Montenegro, L. Valente, B. Feijo, and P. Pagliosa, "An architeture with automatic load balancing for real-time simulation and visualization systems," *JCIS - Journal of Computational Interdisciplinary Sciences*, pp. 207–224, 2010.

[12] V. Mönkkönen, "Multithreaded game engine architectures," Available at `http://www.gamasutra.com/features/20060906/monkkonen_01.shtml` , 2006.

[13] A. P. Chandrakasan and R. W. Brodersen, *Low-power CMOS design*. IEEE press Piscataway, 1998.

[14] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5. ACM, 2001, pp. 89–102.

[15] K. Roy and S. C. Prasad, *Low-power CMOS VLSI circuit design*. John Wiley & Sons, 2009.

[16] J. Donald and M. Martonosi, "Techniques for multicore thermal management: Classification and new exploration," *SIGARCH Comput. Archit. News*, vol. 34, no. 2, pp. 78–88, May 2006. [Online]. Available: http://doi.acm.org/10.1145/1150019.1136493

[17] C. Isci, A. Buyuktosunoglu, C.-Y. Chen, P. Bose, and M. Martonosi, "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget," in *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, 2006, pp. 347–358.

[18] nVidia, "Kepler - the world's fastest, most efficient hpc architecture," Avalible at: `http://www.nvidia.com/object/nvidia-kepler.html`, 2013, 23/04/2013.

[19] nVidia NVML, "nvidia management library," Avalible at: `https://developer.nvidia.com/nvidia-management-library-nvml`, 2013, 23/04/2013.

[20] A. LaMothe, *Tricks of the 3D game programming gurus: advanced 3D graphics and rasterization*. Sams, 2003, vol. 2.

[21] A. Rollings and E. Adams, *Andrew Rollings and Ernest Adams on game design*. New Riders, 2003.

[22] COCOS2D, "Cocos2d," Avalible at: `http://www.cocos2d-iphone.org/games/`, 2011, 30/09/2011.

[23] A. E. Rhalibi, S. Costa, and D. England, "Game engineering for a multiprocessor architecture," in *DIGRA Conf.*, 2005.

[24] M. Zamith, E. Clua, A. Conci, and A. Montenegro, “Parallel processing between gpu and cpu: Concepts in a game architecture,” in *Computer Graphics, Imaging and Visualisation, 2007. CGIV ’07*, 2007, pp. 115–120.

[25] M. Zamith, E. Clua, P. Pagliosa, A. Conci, A. Montenegro, and L. Valente, “The gpu used as a math co-processor in real time applications,” *Proceedings of the VI Brazilian Symposium on Computer Games and Digital Entertainment*, pp. 37–43, 2007.

[26] M. P. M. Zamith, E. W. G. Clua, A. Conci, A. Montenegro, R. C. P. Leal-Toledo, P. A. Pagliosa, L. Valente, and B. Feijó, “A game loop architecture for the gpu used as a math coprocessor in real-time applications,” *Comput. Entertain.*, vol. 6, no. 3, pp. 1–19, 2008.

[27] M. Joselli, M. Zamith, J. Silva, E. W. G. Clua, B. Feijó, R. LEAL, L. Valente, and E. Soluri, “An architecture for mobile games with cloud computing module,” in *SBGames*. SBC, 2012.

[28] T. Pering, T. Burd, and R. Brodersen, “The simulation and evaluation of dynamic voltage scaling algorithms,” in *Proceedings of the 1998 international symposium on Low power electronics and design*. ACM, 1998, pp. 76–81.

[29] W. Kim, J. Kim, and S. L. Min, “Dynamic voltage scaling algorithm for fixed-priority real-time systems using work-demand analysis,” in *Proceedings of the 2003 international symposium on Low power electronics and design*. ACM, 2003, pp. 396–401.

[30] ——, “Preemption-aware dynamic voltage scaling in hard real-time systems,” in *Proceedings of the 2004 international symposium on Low power electronics and design*. ACM, 2004, pp. 393–398.

[31] X. Fan, C. S. Ellis, and A. R. Lebeck, “The synergy between power-aware memory systems and processor voltage scaling,” in *Power-Aware Computer Systems*. Springer, 2005, pp. 164–179.

[32] Y. Zhang, Z. Lu, J. Lach, K. Skadron, and M. R. Stan, “Optimal procrastinating voltage scheduling for hard real-time systems,” in *Proceedings of the 42nd annual Design Automation Conference*. ACM, 2005, pp. 905–908.

[33] M. Joselli, M. Zamith, E. W. G. Clua, A. Montenegro, R. C. P. Leal-Toledo, L. Valente, and B. Feijó, “An architecture with automatic load balancing and distribution for digital games,” in *Games and Digital Entertainment (SBGAMES), 2010 Brazilian Symposium on*. IEEE, 2010, pp. 59–70.

[34] nVidia GPU Computing SDK, “Gpu computing sdk,” Avalible at: `https://developer.nvidia.com/gpu-computing-sdk`, 2013, 22/07/2013.

[35] nVidia, “Cuda samples,” Avalible at: `http://docs.nvidia.com/cuda/cuda-samples/`, 2012, 25/02/2014.

[36] A. R. Hefner and D. L. Blackburn, “Simulating the dynamic electrothermal behavior of power electronic circuits and systems,” in *Computers in Power Electronics, 1992., IEEE Workshop on*. IEEE, 1992, pp. 143–151.

[37] M. Zamith, E. Passos, D. Brando, A. Montenegro, E. Clua, M. Kischinhevsky, and R. Leal-Toledo, “Sound wave propagation applied in games,” in *Games and Digital Entertainment (SBGAMES), 2010 Brazilian Symposium on*, 2010, pp. 211–219.

[38] A. Reynolds, “Boundary condition for the numerical solution of wave propagation problems,” *Geophysics*, vol. 43, no. 1, pp. 1099–1110, 1978.