A Game Architecture Based on Multiple GPUs With Energy Management

Marcelo Zamith*, Luis Valente†, Mark Joselli‡, José Silva Junior§ Esteban Clua§ and Bruno Feijó†
*Federal University of Viçosa
Email: zamith.marcelo@gmail.com
†VisionLab/PUC-Rio
Email: lvalente,bfeijo@inf.puc-rio.br
‡Escola Politécnica
Pontifícia Universidade Católica do Paraná - PUCPR
Email: mark.joselli@pucpr.br
§UFF - IC - Medialab
Email: josericardo.jr@gmail.com, esteban@ic.uff.br

***Abstract*—The availability of multicore CPUs and programmable GPUs have risen the provision of processing power for applications. In case of games, this means increased scene realism and more sophisticated artificial intelligence and physics simulations, for example. However, using more power raises energy consumption and system temperature. Therefore, energy consumption and thermal management are research fields that have been receiving increased attention over the last years. This work proposes a multi-thread game architecture based on the GPGPU paradigm to make use of available hardware while providing energy consumption and thermal control management for multiple GPU processors.**

***Index Terms*—thermal management, parallel computing, multi-thread, GPGPU, game loop models, real-time systems, multiple GPUs.**

I. Introduction

Realism level in games has been increasing over the years not only due to modeling and rendering enhancements, but also by improvements in other areas as animation, artificial intelligence, and physics simulation. The rise in processing power has made it possible to implement all those enhancements. Currently, this processing power is available through multicore processors and GPUs.

The development of sophisticated GPU architectures has led also to the GPGPU paradigm, where real-time applications (as games and simulations) adopt GPUs for rendering and general computation. For example, nVidia GeForce 8 series GPUs started to provide processing power for massive parallel mathematics and physics problems. Examples of using the GPU to process these kinds of tasks are Monte Carlo [1], artificial intelligence [2], crowd simulation [3], fluid simulation [4], and ray casting [5].

However, using these multicore processors and GPUs raise energy consumption and chip temperature. For example, this raises concerns as energy waste, high hardware power requirements, and thermal management. As a result, research on energy consumption and thermal control has received 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 ([6], [7], and [8]).

In the thermal control area, an example is research on *dynamic thermal management* [9], [10], 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.

In the GPU area, the latest nVidia GPU series (codenamed ”Kepler” [11]) enables developers to implement different thermal management policies through a library named NVML (nVidia Management Library [12]. 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 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.

Although research on energy management and thermal control is a topic that has been gaining attention, we were not able to find works that combined game architectures with energy management concerns. In order to help in filling this gap, this work proposes a novel parallel game loop architecture with GPU thermal management based on heuristics. Our architecture takes advantage of new GPUs that make it possible for developers to implement different thermal management

policies.

We want to be able to achieve an acceptable FPS (frames per second) rate for the application, while providing automatic energy management. An acceptable FPS application rate is about 30 frames per second [13].

In order to achieve this goal, we designed two test cases. With the the first one, we wanted to explore this new GPU functionality to learn how this functionality works and if it would be valuable to apply it in a game architecture. The second one tests our proposed architecture as a whole.

The work is organized as follows. Section II presents works related to game loops and 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

The availability of multicore CPUs and programmable GPUs have risen the provision of processing power for applications. For example, for games this means increased scene realism and more sophisticated artificial intelligence and physics simulations. However, using more power raises energy consumption and system temperature.

Our work proposes a multi-thread game loop architecture that addresses both concerns by: 1) integrating GPUs as a resource in game loops; and 2) providing an energy management scheme[a]. This section presents works related to both areas, in separate subsections.

This section provides works related to game loop architectures and energy management. However, we were not able to find works that combined game architectures with energy management.

A. Game Loop Architectures

Games and some visual simulations provide the illusion that everything is happening at once. 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”.

A game processes tasks that fall into three general groups: 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.

During the game lifetime, the game tasks runs periodically. A model that organizes how these game tasks run is known as a *game loop model*. The literature presents some works that address this subject, such as: Dalmau [14], Valente et al. [13], Dickinson [15], Watte [16], Gabb and Lake [17], Joselli et al [18], and Mönkkönen [19].

The simplest possible game loop model corresponds to an architecture with three steps: reading player input, game update and render. In this model, the game runs the three tasks sequentially. As a consequence, the simulation is perceived as running faster in more powerful machines, and running slower in slower machines. Dalmau [14] provides an example about this game loop model.

A strategy to solve hardware architecture dependency is to use uncoupled models, which uncouples the rendering and update stages. These uncoupled models can be single-threaded ([13], [14]) or multi-threaded ([13], [17], [19]).

The Multi-thread Uncoupled Model and the Single-thread Uncoupled Model use a time parameter to adjust the game loop frequency [13]. By using these models, the application is able to adjust its execution with time, so the game runs in a similar way in different machines while maintaining interactivity. More powerful machines will be able to run the game more smoothly, while less powerful ones will still be able to provide some experience to the user.

Nowadays, multicore processors are common in desktops, mobile devices and video game consoles. Current game loops must consider this fact in order to maximize hardware usage. This includes parallelizing tasks into multiple threads. However, dealing with concurrent programming introduces another set of problems, such as data sharing, data synchronization, and deadlocks. Also, as Gabb and Lake [17] state, that not all tasks can be fully parallelized due to dependencies among them. For example, the game is unable to render a character in the correct state before computing the game logic and updating the overall game state. Hence, serial tasks represent a bottleneck to parallelizing simulation computation.

Rhalibi et al. [20] 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 [19] 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 complete game state, even if the update stage is still computing the new one. As an example related to the second category, there is the Synchronous Function Parallel Model [19], which processes the game physics in a separated thread while the main thread process the characters animations.

The first work that integrated GPGPU into a game loop model was Zamith et. al [21]. Zamith et. al proposed using GPUs as math co-processors in real-time applications (as games and physics simulations). The work by Zamith et. al

[21] extended the Single-thread Uncoupled Model [22] by creating a secondary thread responsible for managing the GPU as a math co-processor.

Zamith et. al [23] present another model that integrates GPGPU into game loops, as an architecture where the GPU runs game physics and the CPU runs other tasks that GPUs cannot process (as reading player input or networking). This approach extends the Multi-thread Uncoupled Model [17] by defining a manager that is responsible for distributing tasks between the GPU and the CPU. The work by Zamith et. al [23] implements a static load balancing scheme, using a Lua script to allocate tasks on processors.

Joselli et. al [18] present a Multi-thread Uncoupled Model with an automatic load balancing scheme that also integrate GPGPU. The automatic load balancing scheme uses heuristics to define task allocation on processors (considering 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.

Joselli et. al [24] present a Multi-thread Uncoupled Model for mobile devices that uses cloud computing. This architecture enables the game application to use cloud services for image recognition and speech recognition, for example. This architecture provides modules to access cloud services, networking, social networks, input, rendering, AI processing, and publishing player achievements to social networks.

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 [6] [7], [8].

Applications that use DVS are real-time applications that solve an optimization problem — to maximize CPU idle time while minimizing energy consumption and meeting 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, and works related to DVS apply the idea to CPUs only. This section explores some of these works.

Trevor et. al [25] analyses 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 [25] 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 [26] 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 [27] proposes another DVS optimization algorithm, now 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 [28] 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 [28] 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 [29] 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 multiple GPUs that provides a module to manage GPU energy consumption. This work extends the architecture by Joselli et. al [30], 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

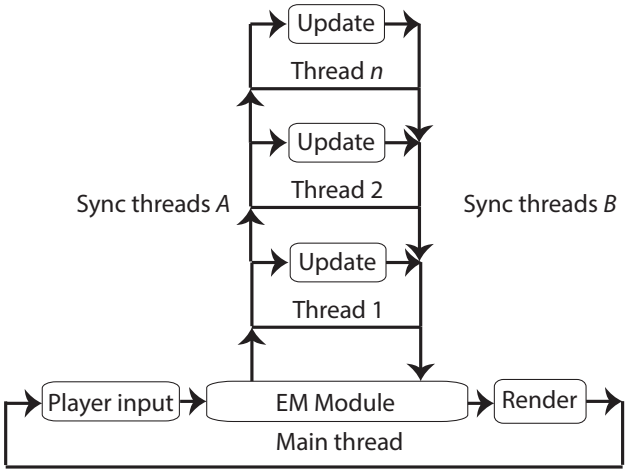

Fig. 1. Parallel game loop model with multiple GPUs.

CPU or GPU happens when there is nothing to process. In this state, a task does not exist.

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 update 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. Fig. 1 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. Fig. 1 also illustrates this one-to-one relationship. The reader should refer to [21] for more details on this approach.

The Energy Manager (EM) represents the core of the 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. Besides, the EM is also a task that the CPU runs.

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 Fig. 1). 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 Fig. 1).

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 on time, 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. 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 [18].

Several times, 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 GPGPU 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 GPGPU 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 GPU, 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 chose to run the heuristic every 10 frames to start with, because we found this to be a reasonable value. If we used an interval much higher

than that, the heuristic could miss valuable information about task performance. If we used a value much lower than that (like at every frame), running the heuristic that often could become a burden to the game.

The EM is responsible for running the heuristic for all GPUs (running GPGPU 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 GPGPU 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 = getMeanRenderElapsed-
  Time(frameCount)
  meanGPGPUelapsedTime = getMeanGPGPUElapsed-
  Time(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 on energy management. We were interested to know if following this path would be interesting.

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 [12] 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 1: Benchmarking

We implemented a benchmark to answer this question: If we changed the GPU clocks, what would be the performance gain or loss considering the nature of the task? By nature of the task, we mean 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.

The benchmark applications are based on three applications that ship with the nVidia SDK [31]: *cdpAdvancedQuicksort*, *cdpLUDecomposition*, *radixSortThrust*. 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.

We selected these specific applications because:

- they demand high computation power;
- the sorting algorithms demand more memory access than GPU processing;
- the LU decomposition requires more GPU processing than memory access;
- these specific applications are simpler to extend than other ones that exist in the nVidia SDK;
- the *cdpAdvancedQuicksort* and *cdpLUDecomposition* algorithms use a new GPU functionality — dynamically parallelism [32] — and we were interested to learn how would affect the task performance.

Dynamically parallelism [32] is a new functionality available in the Kepler GPU series that makes it possible for a GPU kernel to spawn other kernels. In previous GPU architectures, spawning GPU kernels was only possible through the CPU. Fig. 2 illustrates the dynamically parallelism scheme, where the arrows indicate the act of invoking a kernel.

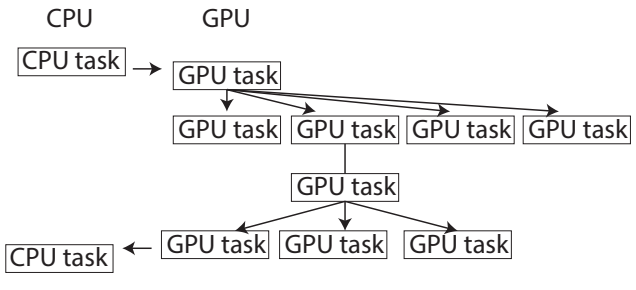

Fig. 2. Dynamic parallelism in Kepler GPU series.

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.

The original applications are able to run the algorithms with different domain sizes. We selected the same domain size for all of them: $8,192$ elements. We selected this value for all applications because this was the maximum value that the *cdpLUDecomposition* application accepted (the other ones accepted 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 [33] 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 results in Table I demonstrates that the application that requires more GPU processing (*cdpLUDecomposition*) is the most affected when the GPU processor clock changes.

Fig. 3 displays memory access elapsed times of sorting algorithms (*cdpAdvancedQuicksort*, *radixSortThrust*) when memory access clock changes Fig. 3 illustrates that changing memory clock does not impact memory access performance for these applications.

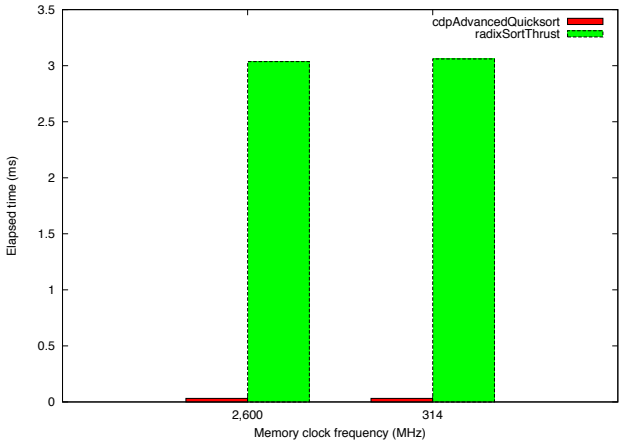

Fig. 3. Memory access elapsed time of *cdpAdvancedQuicksort* and *radixSortThrust*.

Fig. 4 illustrates the GPU processing elapsed time using different clock values, for these applications: textitcdpAdvancedQuicksort, *radixSortThrust*. It is possible to notice in Fig. 4 that the GPU processing time is almost constant regardless of clock value.

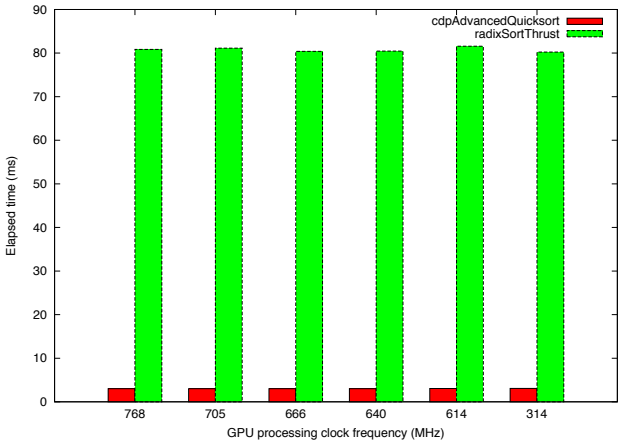

Fig. 4. GPU processing elapsed time of *cdpAdvancedQuicksort* and *radixSortThrust*.

Fig. 5 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 speedup (1.07) is not significative.

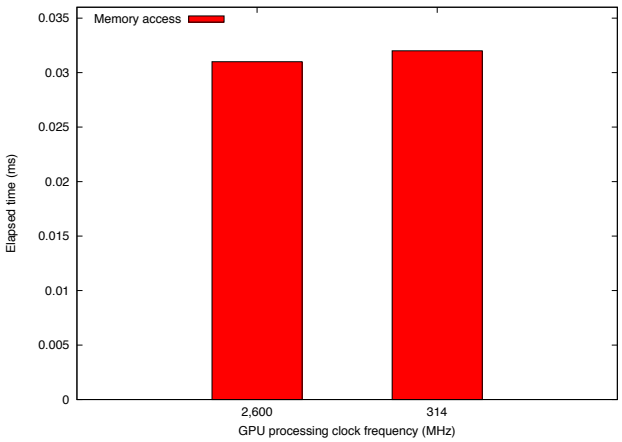

Fig. 5. GPU processing elapsed time (*cdpLUDecomposition*).

Changing the GPU processor clock frequency directly affects the performance of this application. We expected this behavior as *cdpLUDecomposition* requires high computational power. Fig. 6 illustrates that the *cdpLUDecomposition* performance changes linearly as processor clock changes.

Fig. 7 illustrates a chart with temperature variations for all three applications, considering a batch of $1,000$ runs for each application. In order to prevent a test batch from interfering with another batch, the three batches used the same initial condition: the GPU temperature at (37 degrees celsius). This

TABLE I
APPLICATION BENCHMARKING PERFORMANCE

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

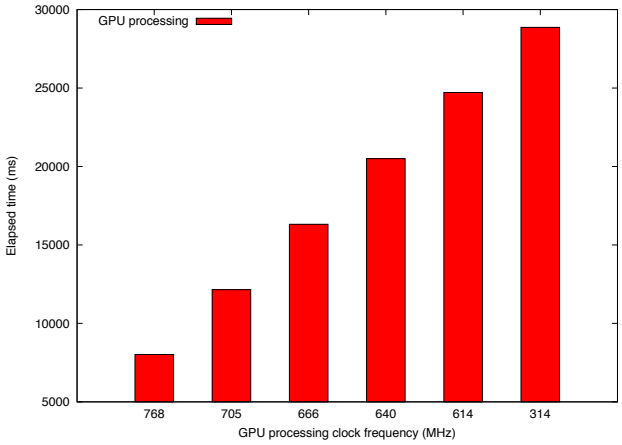

Fig. 6. GPU processing elapsed time (*cdpLUDecomposition*).

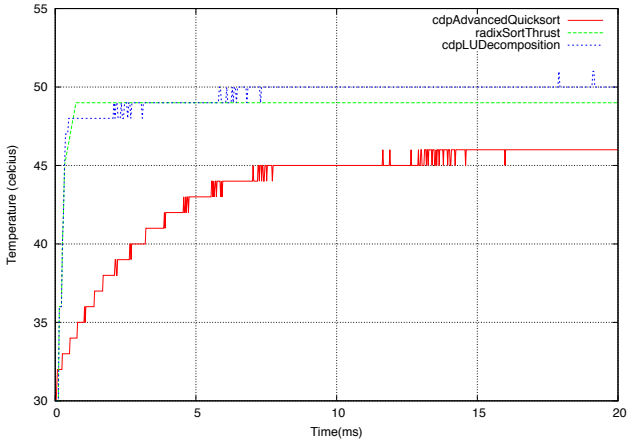

Fig. 7. GPU temperature as a function of time.

is the temperature we measured when the GPU was in an idle state, after just powering up the system.

Fig. 7 illustrates that the temperature rises over time and then stabilizes at some point. 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. Finally, the *cdpLUDecomposition* reached temperature stabilization in 3 milliseconds, ending at 50 degrees celsius.

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 benchmarking applications *cdpAdvancedQuicksort* and *cdpLUDecomposition* use dynamic parallelism, a new feature of Kepler GPU series that enables a GPU kernel to invoke other GPU kernels. We were interested in learning about how this feature would affect our tests. However, after running the tests we did not have strong evidences to affirm that this feature affected or not affected the test performance.

B. The Physics Model: 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 block 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) [34]. 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 [34] 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 Reynolds boundary condition [35] 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 $128\times$ 128 meters to $4096\times$ 4096 meters. Domain sizes larger than 4096×4096 are infeasible to process using our test hardware.

Fig. 8 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: Fig. 9 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.

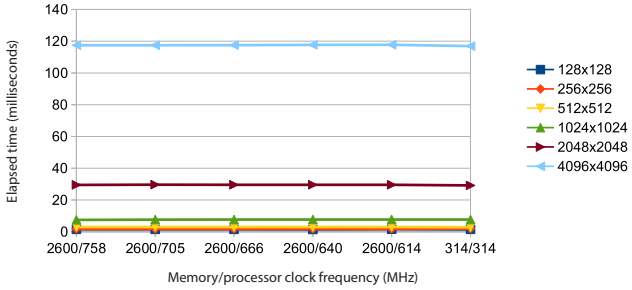

Fig. 9. Render task elapsed time.

Fig. 10 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.

Fig. 11 illustrates a similar pattern as Fig. 10. 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.

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. Fig. 11 illustrates these results (the FPS is calculated as $\frac{1}{X}$ where X is the time in seconds that Fig. 11 represents).

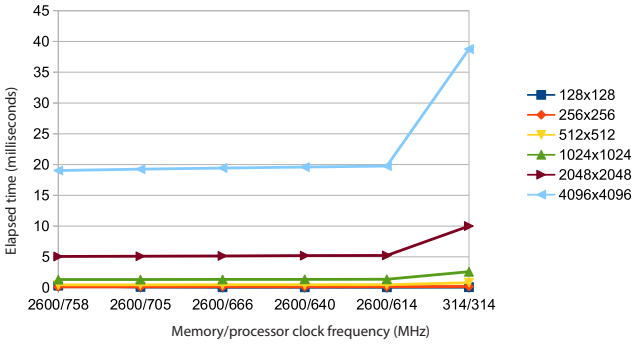

Fig. 10. Update task elapsed time.

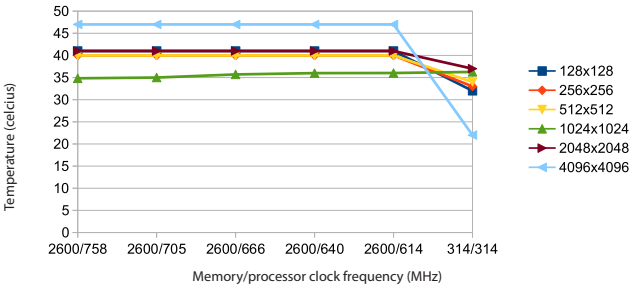

Fig. 11. The GPU temperature.

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, these hardware require more energy to operate, thus elevating energy consumption.

In systems with multiple GPUs, the energy consumption problem could become worse if we consider the GPUs running at full speed while the application do not use the full processing power available in the system. These situations open 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 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.

In this sense, this paper proposed a scheme to manage GPU energy consumption in games through a multi-thread game

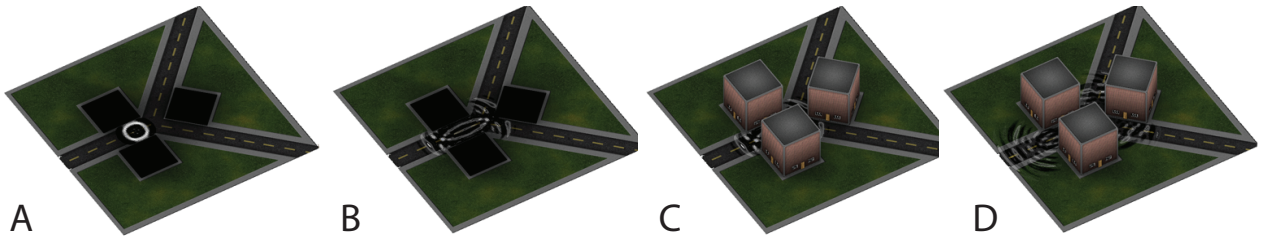

Fig. 8. Test scene geometry and the evolution of wave propagation effect in time. Parts (A) and (B) omit buildings to help in visualizing wave reflection. Parts (C) and (D) present the complete scene.

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 regard, our work represents a first and modest attempt at proposing a solution for this area. e research on energy management for games is a novel area that has a long and promising road ahead, which requires further investigation. A possible future work regards investigating how applying energy management policies could affect overall quality in games, as these policies can reduce available processing power in the system.

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

[7] 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.

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

[9] 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

[10] 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.

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

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

[13] 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.

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

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

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

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

[18] 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.

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

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] ——, "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.

[28] 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.

[29] 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.

[30] 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.

[31] nVidia GPU Computing SDK, "Gpu computing sdk," Avaliable at: `https://developer.nvidia.com/gpu-computing-sdk`, 2013, 22/07/2013.

[32] nVidia Dynamic Parallelism, "Dynamic parallelism in cuda," Avaliable at: `http://developer.download.nvidia.com/assets/cuda/docs/TechBrief_Dynamic_Parallelism_in_CUDA_v2.pdf`, 2013, 16/07/2013.

[33] 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.

[34] 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.

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