D5.5.2– Architectural Techniques to exploit SLACK & ACCURACY trade-offs

Document Information

Contract Number	288653
Project Website	lpgpu.org
Contractual Deadline	01-02-2013
Nature	Prototype (P)
Dissemination level	Public (PU)
Authors	Stefanos Kaxiras (UU), Georgios Keramidas (Think-S), Konstantinos Koukos (UU)
Contributors	Ben Juurlink (TUB)
Reviewers	Iakovos Stamoulis (Think-S)

Notices:

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement No. 288653.

Contents

1 Slack

1.1 Abstract

In this work we are (a) exploring memory slack for the state-of-the-art many-core CPUs and GPUs, (b) present techniques to eliminate slack, and (c) explore the architectural parameters to improve power efficiency. Dynamic Voltage-Frequency Scaling (DVFS) is one of the most beneficial techniques for CPU's to improve power efficiency. The end of Dennard scaling however, in which as technology advances the available voltage range shrinks, is threatening the effectiveness of DVFS. This is very common in GPUs today and will become a severe limitation for many-cores in the near future. In this report we are analysing the impact of core DVFS for different memory frequencies into state of the art GPUs. Because of the limitations imposed by either the programming models or the hardware itself we could not apply DVFS on embedded low power GPUs. Therefore we swift our attention to general purpose multi-cores and demonstrate significant energy benefits from our proposed execution scheme. For the GPU evaluation part we are using the NVIDIA-CUDA toolkit and some custom micro-benchmarks. Our analysis shows that DVFS can give significant energy benefit at architectures with restricted memory bandwidth, such as embedded or mobile GPUs (although this is restricted to simulated runs only due to limitations). Finally our work (a) proposes and evaluates a novel execution scheme for general purpose many-cores, and (b) investigates and intriguing future direction and reveal that energy inefficiencies of GPUs are not related with memory slack but with the mechanisms used to hide slack which seems to compromise applications locality.

1.2 Description of Task

This task involves the following work:

- Modeling and estimation of slack. Modeling of memory access behavior slack and load balance slack in GPUs and the stalls it can create. These models will be used to predict performance, energy, and optimal voltage frequency settings (DVFS) to maximize power efficiency in WP5.

- Design and develop techniques for exploiting load balancing slack: Develop dynamic (run-time) and static (profile-based) techniques necessary to DVFS or power down idle cores in load-imbalance situations.

- Design and development of techniques for exploiting memory access slack: Develop run-time mechanisms for GPU DVFS to determine optimal power-efficiency operational points according to memory behavior.

1.3 Introduction

Memory slack is defined as the time the execution units are stalled due to off-chip misses. During this stall time the executions units keeps consuming power executing NOP instructions which is traditionally called bubble in the pipeline. This source of inefficiency is handled in general purpose multi-cores and many-cores with DVFS. Because off-chip access latency is inelastic to core DVFS we can expect that reducing the core frequency when we are stalled to memory (DRAM) may increase execution time which overlaps with the stall therefore performance is not damaged and energy benefits occur from the reduced frequency. This case is shown in Figure 1 diagrams (a), (b), and (c) for a general purpose micro-architecture. In a coupled execution as shown in these diagrams we can only expect to exploit a part of the available slack with a tradeoff in performance. This is because we cannot have instant DVFS and therefore we need to adjust a global frequency for a long instructions interval.

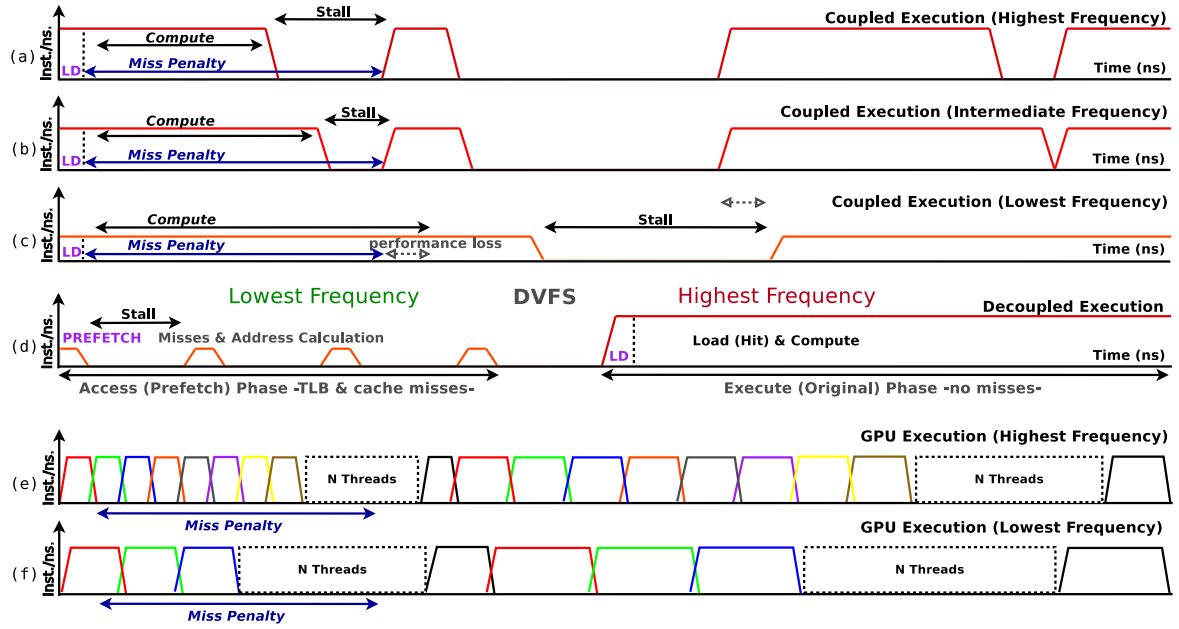


Figure 1: Example of coupled execution under different core frequencies (a,b and c), decoupled execution (d) and GPU executions (e and f). Figure from [7].

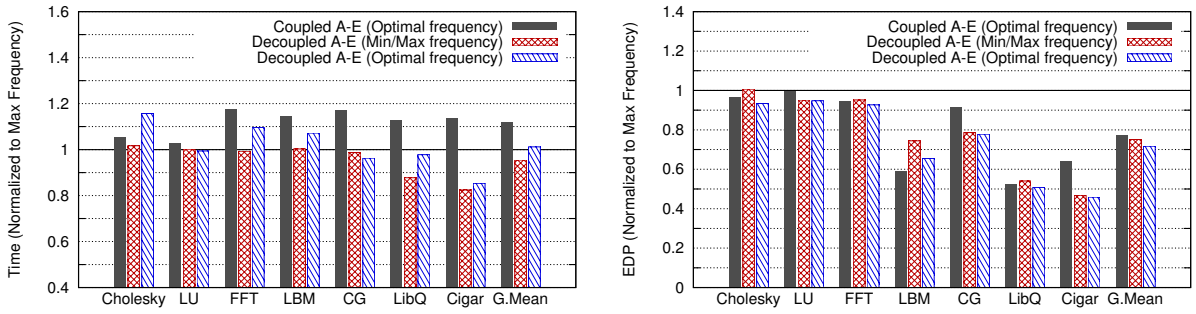
Diagram (d) shows how a decoupled execution can improve energy efficiency without sacrificing performance. Instead of trying to adjust DVFS to the program demands, it adjusts program demands to DVFS granularity by splitting the program into 2 phases and DVFS each phase separately. For the access phase in which we prefetch the data to warm up the cache we are using the lowest frequency. For the execute phase which is scheduled to run immediately after the access phase with the data already in the cache we adjust the frequency to the highest. This role distinction between totally memory bound and compute bound in program phases, allows us to maintain optimal performance because performance is affected by the execute phase and at the same time, save as much energy as possible by running the access phase at the lowest frequency. Diagrams (e) and (f) show GPU executions under different frequencies. Because GPUs feature hardware implementation to context switch between threads with 1 cycle overhead and support thousands of outstanding threads, they can hide all slack despite the long stalls per thread. This mechanism is a source of power inefficiency because it compromises locality as shown later on in this report.

1.4 Work Performed

The work performed involves the creation of (1) a novel general purpose execution scheme that can significantly improve power efficiency on many-cores, (2) slack analysis on state-of-the-art high end GPUs, (3) simulations of DVFS capable embedded GPUs, (4) hardware infrastructure to measure power consumption of the components at runtime, and (5) investigation of the power inefficiencies of state-of-the-art GPUs.

1.4.1 Decoupled Access - Execute

In this work we demonstrate techniques to improve power efficiency in general purpose many-cores. The methodology created in this work is not tied to a specific programming model or architecture. The work is described in [7], while in this report we only present the key idea, how is this work related with the LPGPU project and how it can affect future GPU architectures. Due to the end of Dennard scaling, voltage scaling is expected to shrink in future nodes, limiting the energy savings provided by DVFS. This is very common in all latency-sensitive state-of-the-art many-cores today where slack is the main source of inefficiency. Therefore we need to swift our attention on how to improve energy efficiency without hurting performance. Decoupled execution is very promising for that due to its ability to adjust application behavior to DVFS granularity. This work advances our previous work of [11] and [12] by focusing on maintaining the highest possible performance while at the same time, optimize energy and increase the effectiveness of DVFS. The underlying insight of our work is that by decoupling access and execute we can take advantage of the memory-bound nature of the access phase and the compute-bound nature of the execute phase to optimize power efficiency, while maintaining good performance.



(a) Normalized execution time

(b) Normalized EDP

Figure 2: Time and EDP for coupled and decoupled executions on Intel Sandybridge using 4 threads, normalized to coupled execution at maximum frequency. Figure from [7].

To demonstrate this we built a task based parallel execution infrastructure consisting of: (1) a minimalistic task-based runtime system to orchestrate the execution, (2) power models to predict optimal voltage-frequency selection at runtime, (3) a modeling infrastructure based on hardware measurements to simulate zero-latency, per-core DVFS, and (4) a hardware measurement infrastructure to verify our models accuracy. Based on real hardware measurements we project that the combination of decoupled access-execute and DVFS has the potential to improve EDP by 25% without hurting performance. On memory-bound applications we significantly improve performance due to increased MLP in the access phase and ILP in the execute phase. Figure 2 shows performance

(2a) and EDP (2b) improvement achieved by decoupling. We employ two different policies and test them against a normal *Coupled A-E* execution. The naive approach in which we assume that no knowledge is required in the execution and we just use lowest frequency for the access and highest for the execute phase of each task (*Decoupled A-E min/max*). Our second policy named *optimalEDP* tries to predict the frequency that is optimal (in most of the cases is very close to min/max) using an IPC-based power model. In fig. (2a) we observe that our *min/max* policy does not reduce performance in any of the applications tested. On the contrary in some memory bound applications by improving prefetch accuracy and memory level parallelism (MLP) it manages to significantly improve performance over the hardware prefetcher. For the *optimal EDP* policy we trade-off some performance to achieve optimal EDP. We observe that on average it can slightly affect performance. In fig. (2b) we observe that the performance benefits achieved by our policies comes at the same EDP or even lower compared to a coupled execution. This means that the execution method itself manage to exploit memory slack on a much more efficient way that standard DVFS methods.

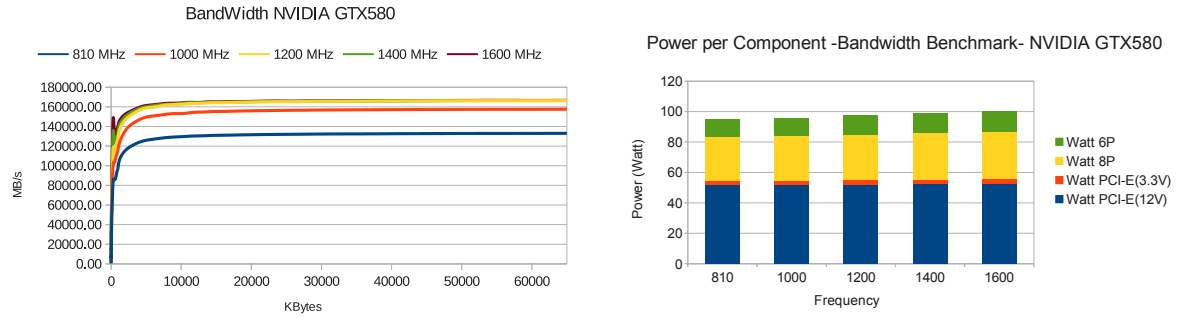
1.4.2 DVFS state-of-the-art GPUs

Modern GPU architectures can be classified as high-end (discrete) and embedded or mobile GPUs. High-end GPUs targets gaming market and scientific community where they can be used as accelerators for parallel scientific workloads. These GPUs are characterized by the enormous memory bandwidth they have which is typically 15-20 times higher than the main memory. Architecturally they do not feature memory coherency, branch prediction or any other sophisticated components used in CPU's. Even caches are newly introduced into such architectures. The design mainly focuses on performance and uses power constraints only to the degree of working TDP. These cards typically exceed 250W operational power. On the other hand embedded or mobile GPUs are designed with respect to power consumption either because they are on the same package with a CPU and can afford only a portion of the die TDP or because of battery lifetime. Our work is extended on both classes of GPUs. The benefits of GPUs over general purpose processors comes is the enormously higher memory bandwidth (typically 10-20x) and the ability to context switch with 1 cycle overhead. None of them is for free because the trade-off for the high bandwidth offered by the wide GDDR buses is power consumption while context switching mechanisms reduces locality to a few registers per thread.

In this paragraph we are discussing the power consumption measurements taken from a state of the art discrete GPU (NVIDIA fermi - GTX580). The card power consumption is 90W idle (without driving a screen) and goes up to 250W when running CUDA scientific workloads. It is roughly impossible to measure the power consumption of each component inside the GPU without interference to the voltage regulators. Each modern GPU is supplied by 4 main resources as listed:

- PCI-Express (12V input)
- PCI-Express (3.3V input)
- 8-Pin PSU adapter
- 6-Pin PSU adapter

As shown in Figure 3b almost 80% of the power comes from 12V PCI-E and 8-Pin PSU adapter. Each cluster of bars in the graph shows the power for different core frequency from 800MHz -



(a) Measured Bandwidth

(b) Measured Power

Figure 3: Bandwidth and power consumption under different core frequencies on NVIDIA Fermi

1600MHz at a step of 200MHz. The power consumption is load balanced across all available input rails to the card and is increased proportionally. Our study is further complicated by the fact that memory frequency does not only affect the power consumed by the memory itself but can also increase the power consumption of the GPU core. To verify that, we run our benchmark suite at different core and memory frequencies. This effect is caused from the thread scheduler of the card. Doubling the memory frequency almost doubles the bandwidth and that allows the thread scheduler to better utilize the GPU core (increasing both performance and power consumption). The performance impact of reducing frequency is severe and in all cases results to worst Energy-Delay Product (EDP).

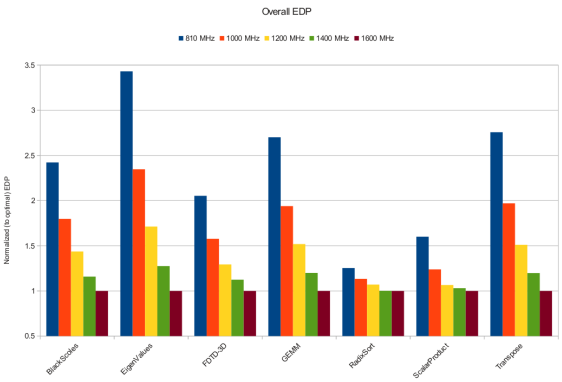
In this section we are discussing the impact of core DVFS to the GPU bandwidth. We observe that only at the lowest core frequency there is a significant bandwidth decrease of 20% as shown in Figure 3a. Further study on power consumption using bandwidth micro-benchmarks, shows that memory is responsible only for a small portion of the total power spent by the GPU. 3b shows the power consumption of a typical bandwidth micro benchmark. Doubling the core frequency increase the power by a maximum of 5% and leads to a maximum of 20% bandwidth improvement. For that benchmark the best EDP is at 1GHz core frequency 4.92% better than running at highest frequency. In the next section we further discuss DVFS impact on real applications.

As noticed on the previous section high end discrete GPUs have an enormous bandwidth compared to main memory. This becomes the main restriction when using traditional DVFS techniques to improve power efficiency. DVFS is a very useful technique to improve power efficiency for applications that are highly memory bound. In section 1.4.2 we give a definition of slack. Many heuristics have been used as a criterion for reducing frequency such as miss rate and IPC in CPUs. These ideas can be used successfully in GPUs when necessary. Figure 4b depicts the EDP for a subset of the CUDA toolkit benchmarks for different core frequencies. We observe that all applications have a best EDP at the highest frequency. There are two reasons for that:

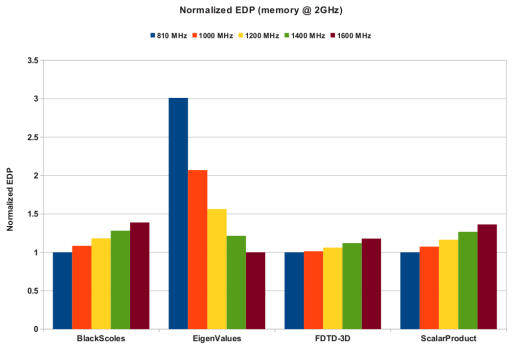
1. The enormous device memory bandwidth on the card (up to 170 GB/s total)
2. The enormous amount of outstanding threads per core (up to 1536 threads)

The NVIDIA Fermi architecture uses giga-thread scheduling engine that allows up to 1536 threads to be scheduled on the same core. When some thread is stalled the scheduler is responsible to schedule

the next immediate ready thread on the core. The L1 access latency is 200 core cycles, while the memory access latency is 800 core cycles [6]. With that configuration it is impossible for some core to become idle even by excluding scheduling overhead. The impact of DVFS for real applications varies from 30% for the case of radix sort till 3.5x for heavily computational bound applications. The observation that the best EDP is observed always at the highest frequency, defines complete lack of slack for high end GPUs. For our study we are using an NVIDIA GTX580, the architecture and specifications of which is described in [14]. For the evaluation workload we are using CUDA SDK benchmarks [15].



(a) EDP under different core frequencies at 4GHz memory



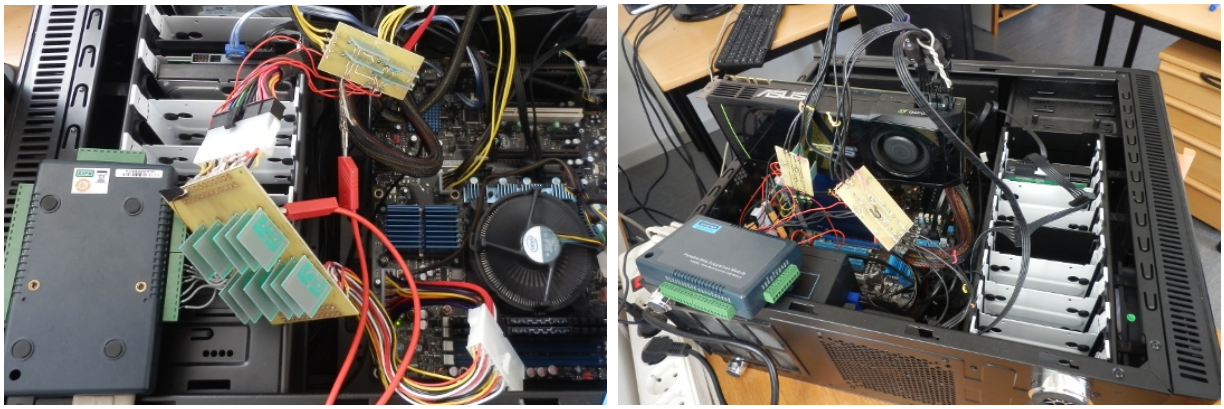
(b) EDP under different core frequencies at 2GHz memory to simulate embedded GPUs

Figure 4: Bandwidth and power consumption under different core frequencies on NVIDIA Fermi

1.4.3 Simulating Future Low-Power GPUs

The conclusions from previous section show that the best EDP is achieved by running applications at the maximum core frequency. Although this observation is orthogonal for discrete cards it does not apply to embedded or mobile GPUs. In the latter the GPU has a restricted memory bandwidth depending on the implementation. Usually it can access the same L3 cache as the processor, have direct access to the main memory or use some dedicated link to the CPU cores. Even in the best case that the selected memory hierarchy is on chip (L3) the total available bandwidth is less than the bandwidth available on discrete cards. In this section we are evaluating the performance and DVFS impact on a memory frequency close to that of the main memory. We are reducing the memory frequency close to that of modern DDR3 modules at 2GHz which is half of the maximum card frequency. At this memory frequency the bandwidth is measured at 90GB/s which is 5x-6x higher than high-end DDR3 memories and even higher than state of the art L3 caches. Figure 4a shows DVFS impact when running at restricted bandwidth. In most of the cases the best EDP is observed when running at the lowest frequency. Eigen-Values application is a proof that a very computational bound application will benefit from higher core speed. Memory bound applications tend to have best EDP close to the lowest frequency resulting to a maximum 40% EDP reduction. With that set-up we emulate less than half of the theoretical throughput 18Gpixels/second at core frequencies varying from 800MHz to 1.6GHz. Compared with a low power GPU such as the ARM mali 450MP which operates at 480 MHz providing roughly 3.8Gpixels/second our set-up can be characterized as efficient to describe in terms of bandwidth and cores many generations of low power GPUs.

1.4.4 Measurement Infrastructure



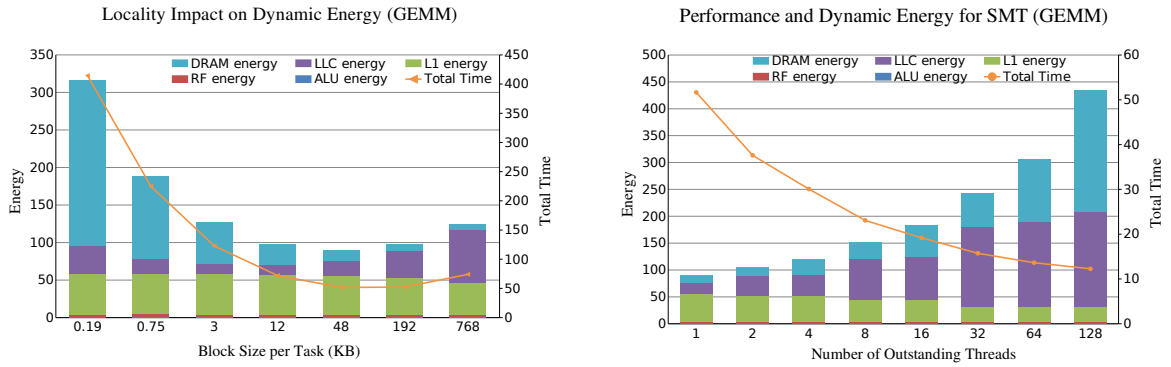
(a) HW Infrastructure to measure IO, CPU and DRAM (b) HW Infrastructure to measure GPU

Figure 5: HW Infrastructure to measure system components

We develop an infrastructure to measure power consumption of the different components comprising the total system. With our setup we can measure the power consumed by the CPU, the main memory and the GPU. To achieve this, we place current sensor boards between each component and the power supply. For each cable that supplies current to a component of interest, we develop a small board that contains a current sensor. These small boards are then placed at the top of a main board. The output of the power supply is connected to the input of the main board, and the output of the main board is connected to the motherboard. By measuring the voltage of each cable and the current that flows through the cable, we are able to determine the power consumed be the corresponding component. Modern GPUs are supplied with power: i) directly from the system power supply through dedicated cables and ii) through PCI-express pins. For the first source of power consumption we use one of the boards described above. There is a total of 6 cables supplying the GPU with power, so we use 6 current sensors. For the power supplied through the PCI express we use a PCI-express extender. This is a board that is placed between the motherboard and the GPU and allows us to have access to each of the PCI-express pins. Two voltage rails (of 12V and 3.3V) supply power to the GPU through PCI-express. By placing current sense resistors along these two voltage rails we are capable of measuring the current and consequently the power consumed in the GPU. With our current setup we are able to measure concurrently the power consumed by the CPU, the main memory and the GPU of two separate systems.

1.4.5 Investigating Power Inefficiencies

In this section we explore the impact of simultaneous multi-threading mechanisms into programs locality and how it can affect performance and energy. For our study we consider a well known-studied linear algebra kernel (GEMM) applying $C+ = A \times B$. The locality of this kernel is analytically described in [8]. Using the energy assumptions for data movement in GPUs from [2] in combinations with the miss rate analysis described in [8] we conclude in the results shown in Fig. 6. Results in Fig. 6a prove that there is an energy optimal point for the kernel which is observed when the task size is very close to L1 size. Figure 6b shows the impact of multi-threading. Although multiple outstanding threads can be used to prefetch data and hide slack from the pipeline they require the dataset to be segmented binomial when doubling the number of outstanding threads as shown in



(a) Block Size impact on Performance and Energy (b) Multi-Threading effect on Performance and Energy

Figure 6: HW Infrastructure to measure system components

Fig. 6b. This compromises locality and increases energy consumption super-linear at a maximum of sub-linear performance improvement. These graphs show only the impact on dynamic power. Taking leakage power into consideration requires detailed simulations and is left for future exploration. We strongly believe that through this exploration we will be able to find an EDP optimal point.

1.5 Executive Summary

In this section we summarize the work done and to which degree we meet DoW expectations. The work done this far referring slack *exceeds initial DoW expectations* in the following ways:

- We found that through decoupled execution we can optimize many algorithms to take advantage of locality and at the same time run at optimal EDP eliminating all slack (in many-cores).
- We discovered that one of the main sources of energy/EDP inefficiencies in state-of-the-art GPUs is the streaming mechanism itself along with programming model which compromises locality.

The contribution of this deliverable is to reveal that the source of power inefficiency on GPU execution models is not slack as we initially assumed but the restricted locality. We quote our initial results of DVFS state-of-the-art GPUs as a proof of concept. Although there are significant opportunities to eliminate slack through DVFS in embedded GPUs (especially when using shared resources such as LLC), there are also severe limitations to verify that in practice; the inability to DVFS embedded GPUs and the very bad support of OpenCL programming model at the first months of the project. For these reasons we swift our attention to optimize the execution of many-cores. Contrary to GPUs many-cores and many-core accelerators such as Xeon Phi provide simpler programming models and large caches compared to GPUs at the same bandwidth capabilities. This part of the project discusses (1) how we can take advantage and fully optimize such architectures using decoupled access-execute and (2) demonstrate some of the energy inefficiencies of state-of-the-art GPUs. This creates a promising direction for future work of this project.

2

Redundancy and Accuracy

2.1 Executive Summary

This section presents the current development state of the work conducted in Task T5.2 (Redundancy) and Task T5.4 (Accuracy) during the first half of the second year of the LPGPU project. Although the purpose of this deliverable, as it was initially defined in the LPGPU work plan, was to discuss the potential techniques to exploit slack and accuracy tradeoffs in GPUs, in the course of the LPGPU project we realized that the concept of redundancy (T5.2) and the concept of accuracy (T5.4) are tightly coupled and cannot be studied separately: there is a significant interdependence between the two concepts resulting in QoS/power trade-offs in calculations and in off-chip memory accesses. Thus, as also mentioned in D5.5.1 [1], we selected to investigate the two concepts in the same framework. In this respect, this deliverable complements D5.5.1 deliverable [1]. In the D5.5.1 deliverable, we presented our preliminary results in the following directions:

- Exploiting redundancy/accuracy in calculations by relying on memorization or work reuse techniques. We named this approach as Value Cache. We choose to apply our methodology to two common functionalities of a graphics processing system and more specifically to typical image color transformations and to image blending transformations. The reason for this choice was that both techniques require a significant amount of complex floating point operations per image pixel.
- Exploiting redundancy/accuracy in off-chip memory accesses. The target was to substitute, to the extent possible, the expensive off-chip memory accesses with on-chip memory accesses. We opted to apply this technique to a resource (in terms of power, time and I/O bandwidth) consuming memory operation which is the writing/storing of a rendered image to the so-called framebuffer.

In general, in this document we present the progress performed in those two directions. More specifically, the contributions of this section are as follows:

- Structural similarity factor [14]: We re-evaluate our proposals (presented in D5.5.1) using a more appropriate error metric which is able to take into account the human perception.
- We define a new set of realistic benchmarks using the tools provided by the Attila framework [1]. While our previous analysis pertained to small, GUI-based animations for benchmarking our proposals, we created a new set of high quality OpenGL-based benchmarks representative of state-of-art computer games.
- Value Cache: In the current state we are porting the value cache approach in the Attila simulator [1] and more specifically in the OpenGL fragment shaders of the simulated GPU. The reasons for selecting the Attila simulator will be explained later in this deliverable.

- Reduction of framebuffer activity: We created demos that investigate the dynamic behavior of the selective update scheme presented in the previous deliverable. Our results indicated that the selective framebuffer update scheme is effective only when the rendering operations are idle or not fully utilized. As a result and in order to mitigate to the extent possible the costly (in terms of power, time and I/O bandwidth) off-chip memory transactions, we implement and evaluate a new lightweight framebuffer compression scheme which is able to offer a significant reduction in off-chip bandwidth. The new framebuffer compression technique is carefully designed to work on top of the selective update scheme reaping the benefits of both approaches.

In the rest of this section, we present our new results in each those directions, we define the current status of the Redundancy (T5.2) and Accuracy (T5.4) tasks, and we draw specific directions for future work.

2.2 Relevance of the tasks to the project

The target in Task T5.2 is to identify and exploit redundancy, either in calculations or in (off-chip) memory accesses, in various levels of a graphics processing system. The target in Task T5.4 is to devise informed performance or power efficient policies in which controllable errors are allowed to occur. More specifically, our goal is to dynamically decrease the accuracy of the computations or the memory accesses (e.g., by ignoring few low order bits) without significantly hurting the quality of the rendered images. Of course, by decreasing the accuracy of specific operations (calculations or memory accesses) noteworthy power and performance benefits can be reported.

The drawback of the above proposals is that we alleviate the "correctness tax" imposed in traditional computing systems since we may introduce errors in the output screen reducing the quality of the rendered image. The advantage of this technique is that it is possible to further reduce the energy consumption of the rendering process. As a result, a trade-off between QoS (what you loss) and energy consumption (what you get) is coming into the table. Finally, we should mention that the application domain of the LPGPU project is a perfect "fit" towards this "relaxed correctness" direction. This is because in graphics applications, the final output is interpreted by human senses, which are not perfect. This fact obviates the need to produce exactly correct numerical outputs (accuracy).

2.3 Work Performed

2.3.1 Structural similarity factor

Since the concept of accuracy incorporates the notion of controllable errors, the next natural step is to identify the correct metric to quantify the resulting errors. In D5.5.1, we relied our methodology based on the well established root mean squared error or RMS error. While simple to calculate and mathematically convenient, the RMS error is not very well matched to the human visual system. In general, finding the similarity or dissimilarity between two images is a difficult task and various metrics have been proposed. After reviewing the corresponding literature, we choose to use the

popular image structural dissimilarity metric or DSSIM [9] which is able to into account the human perception when comparing two digital images. The DSSIM metric is based on the structural similarity metric (SSIM) which was proposed as an alternative more closely modeling perceived visual quality [9]. SSIM is a function of luminance (l), contrast (c), and structural comparison (s):

$$SSIM(x,y) = f(l(x,y), c(x,y), s(x,y))$$

Luminance, contrast, and structure are defined as functions of mean value, standard deviation and correlation coefficient. More specifically, for a singlechannel image x, let xi, be its N pixels:

$$\mu_x = \frac{1}{N}\sum_{i=1}^{N} x_i$$
$$\sigma_x = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu_x)^2}$$
$$\sigma_{xy} = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu_x)(y_i - \mu_y)$$
$$l(x,y) = \frac{1\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1}$$
$$e(x,y) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_1}$$
$$s(x,y) = \frac{2\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3}$$
$$SSIM(x,y) = l(x,y)^{\alpha} e(x,y)^{\beta} s(x,y)^{\gamma}$$

The purpose of the constants Ci is to avoid numerical instability when calculating SSIM. SSIM is symmetric, non-negative and bounded. The exponents $\alpha, \beta, \gamma > 0$ are used to adjust the impact of each measurement on SSIM. On the opposite side, image dissimilarity (DSSIM) has been defined to keep the advantages of SSIM, but make it more similar to distance measures [9]:

$$D(x,y) = \frac{1}{SSIM(x,y)} - 1$$
$$D(x,y) = l(x,y)^{-\alpha} e(x,y)^{-\beta} s(x,y)^{-\gamma} - 1$$

In image dissimilarity the constants Ci are set to zero leading to:

$$D(x,y) = \left(\frac{1\mu_x\mu_y}{\mu_x^2 + \mu_y^2}\right)^{-\alpha} \left(\frac{2\sigma_x\sigma_y}{\sigma_x^2 + \sigma_y^2}\right)^{-\beta} \left(\frac{2\sigma_{xy}}{\sigma_x\sigma_y}\right)^{-\gamma} - 1$$

As a result, image dissimilarity is non-negative, reflexive, and symmetric dictating to the following properties:

$$D(x,y) \geq 0$$
$$D(x,y) = 0 \leftrightarrow x = y$$
$$D(x,y) = D(y,x)$$

DSSIM, like SSIM, is defined for single-channel (e. g., grayscale) images only. The minimum over all channels has been used when calculating SSIM for multichannel images (e. g., [10]), however for a more realistic and representative comparison, in this work, the maximum of the individual channels DSSIM values is used as representing DSSIM for multichannel images.

Although, the DSSIM metric is very popular metric and widely accepted by the research community, there is not available a comprehensive study to understand or quantify the distance values returned by DSSIM (this is especially true since DSSIM is not bounded). Thus, as a second step and in order to get a full understanding of the range of values returned by DSSIM, we implemented our own version of DSSIM and applied it to the Value Cache simulator presented in the previous deliverable (D.5.5.1).

The concept of Value Cache offers a nice framework to analyze the strength of DSSIM. This is because ignoring a few low order bits in calculations may result in different numerical values (captured by the RMS error) but these distances may not be identified by human eyes. The following two figures compare the DSSIM values and RMS errors returned by the Value Cache simulator for various design points.

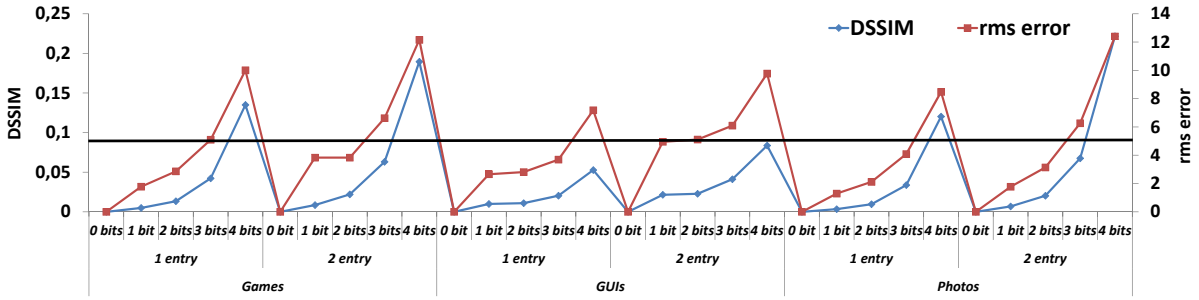


Figure 7: Comparison between DSSIM and RMS error for the Color Transformation function as averaged values for the three image categories.

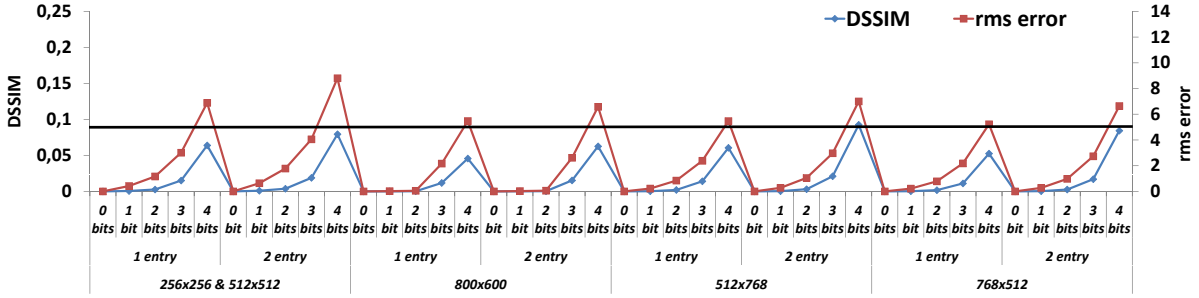


Figure 8: Comparison between DSSIM and RMS error for the Color Blending function as averaged values for various image sizes.

Figure 7 depicts the results for the Color Transformation function while Figure 8 illustrates the corresponding results for the Color Blending function. The benchmark suite (and all the simulation details) used for this comparison can be found in D5.5.1 [5]. In both figures, the blue lines correspond to the resulting DSSIM metric (left vertical axis) and the red lines show the RMS error (right vertical axis). Furthermore, in FIGURE 1, the benchmarks are divided into three categories (Games, GUIs, and photos) according to their type, whereas in Figure 8, the categorization is performed based on the size of the input images. The horizontal axis is further divided into two main parts (tagged as 1-entry and 2-entry) for each benchmark category corresponding to the 1-entry and 2- entry Value

Cache organization respectively. Finally, for the Value Cache configuration the number of low-order bits ignored during the comparison process is also shown. For example, the leftmost points (tagged as "0-bit") show the results when full matches are taken place (i.e., only the effect of redundancy is exploited), whereas the points tagged as "1-bit" correspond to the case when 1-bit is excluded from the matching process (exploiting accuracy) and so forth. In all cases, the averaged value across all the benchmarks (images) belonging to the same category is shown.

As is can be seen in both figures, there is a correlation between the resulting DSSIM and the RMS error. As it was expected, this correlation is not linear. Different RMS errors lead to quite different DSSIM values. Consider for example, the GUI category/2-entry Value Cache in FIGURE 1. While significant RMS errors are reported (above 5%), the corresponding DSSIM is well below 0.05 (for the 0-bit, 1-bit, 2-bit, and 3-bit cases). Another interesting point is in the case of Figure 7/GAMES/2-entry Value Cache, in which the RMS error is almost 4% for the 1-bit and 2-bit cases (meaning that there is a difference of 4% in the numerical values of the corresponding pixels), but those differences cannot be identified by human eyes (the DSSIM is well below 0.023).

In general and after examining the resulting images in all cases (one-to-one comparison in all images), a DSSIM value below 0.03 can be considered as a safe limit when comparing static images (the human eyes cannot understand the resulting differences). When comparing moving images (e.g., the snapshots saved in the framebuffer in a typical vector graphics application), this limit may increase to 0.05 (or even further) without compromising the quality of the end result (moving images). Finally, as Figure 7 indicates ignoring 2-bits in the calculations in the Color Transformation process results always in a reporting DSSIM well below 0.03, while in the case of Color Blending process 3-bits can be ignored revealing the potential of the Value Cache technique explored in the context of the LPGPU project.

2.3.2 Attila simulator

While in the first year of the project, we have evaluated the Value Cache approach (i.e., the Redundancy and the Accuracy Tasks) by relying on a small subset of OpenVG standard [4] using an in-house simulator, we now decide to proceed with a more realistic simulation environment. Even if there are many GP-GPU simulators available (GPU simulators running general purpose programs), this is not the case for real graphics applications. After reviewing the available simulators in the area, we have found that there is only one GPU architectural-level simulator (called Attila [1]) capable to work at the OpenGL level. The reason for choosing Attila was mainly driven by the fact that the concept of Redundancy and Accuracy can be better exploited in real graphics applications and not in general purpose scientific codes running on GPUs (e.g., the gp-gpusim simulator). No other available OpenGL-based simulator exists in the area. Two other popular architectural simulators (Multi2sim [13] and Macsim [6]) planning to support OpenGL in their future releases, but their current status does not include such support.

Attila simulator [1] is a generic microarchitecture, cycle-level, execution-driven simulator that closely tracks todays GPUs without being an exact replica of any particular product. The microarchitecture of the simulator is versatile and highly configurable and can be used to evaluate multiple configurations: either high-end desktop GPUs or even embedded GPUs for mobile systems. In addition, the Attila simulator is accompanied with a rich set of tools for collecting, process, and logging traces from real graphics, verifying and simulating the traces and verifying the simulation results.

Figure 9 shows the complete Attila framework. For more details about this framework, we refer the interested reader to [1].

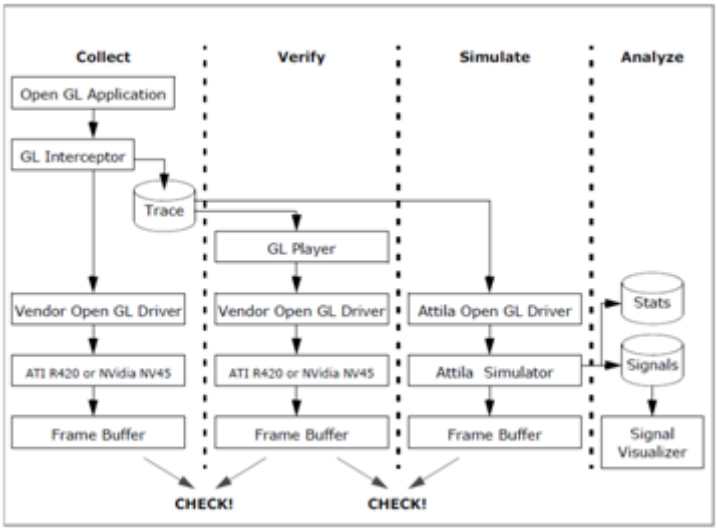


Figure 9: Attila simulation framework.

Furthermore, Attila GPU pipeline can be configured to follow either a non-unified shader mode (with separated vertex and fragment shaders) or a unified pool of shaders. The block diagram of the latter mode of operation can be seen in Figure 10.

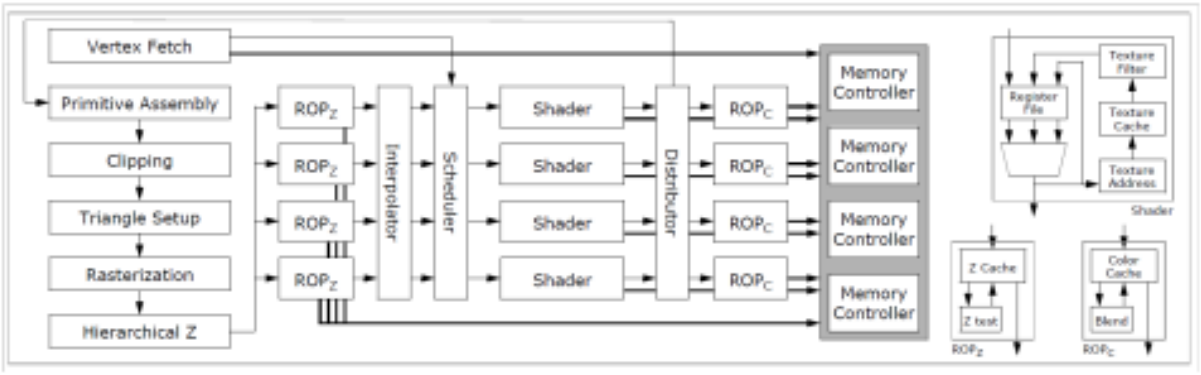


Figure 10: Attila unified architecture.

In addition, Attila framework is able to offer a wide range of output statistics concerning all the events occurring in the underlying micro-architecture, including low-level resource utilization of all pipeline stages (shown in 10), cache hit and miss ratios, memory bandwidth usage etc. At the current developing phase we are porting the Value Cache implementation (developed during the first year of the project) in the Attila framework and more specifically into the fragment shaders. Since Attila is a cycle-level simulator extended with timing characteristics, it is possible to extract not only the input/output (coverage and accuracy) of the Value Cache, but also the performance benefits provided by the proposed Value Cache technique.

A limitation of the current version of the Attila framework is that it supports a relatively obsolete version of the OpenGL API (the current version resembles the OpenGL version 2.0) and there is no support for the OpenGL ES standard. However, in the context of the Value Cache approach, those

limitations are not considered as critical. Finally, the Attila framework comes also with a wide set of OpenGL traces captured by the modern games which is the subject of the next subsection.

2.3.3 Benchmark suite

The Attila OpenGL library implements a memory abstraction component for the different OpenGL objects: programs, textures, and vertex buffers. In addition, a tool called GLInterceptor (see Figure 9) replaces the OpenGLlibrary and records all OpenGL commands issued by the application with all their parameter values, associated texture, and vertex buffers data. This information is stored in an output file; a trace file ready to be fed to the simulator. Another tool, called GLPlayer, can be used to reproduce and validate the captured trace mainly for verification purposes. Using those two tools we are able to capture the behavior of real-life applications and also to check the quality of the rendered frames when they are stored to the framebuffer (the latter feature is very important when we reduce the accuracy of the calculations). In addition, the Attila website contains a rich set of captured graphics traces from modern OpenGL games like Doom 3, Quake 4, Crysis etc. Using the GLPlayer we were able to extract each snapshot (or framebuffer instance) and record those images for further processing. All the traces were captured assuming Anisotropic Filtering.

Those new traces helped us to extent our benchmark toolbox using real-life applications in addition to the small, GUI-based applications provided by the Qt framework [3] (as we did in the first year of the LPGPU project). Table 1 depicts the benchmarks (and a small description) that will be used to evaluate the Accuracy and the Redundancy concepts from now on in the LPGPU project. To this end, the benchmark suite consists of three OpenGL games and two applications typical of modern graphical user interfaces such as those on embedded devices or tablets.

2.3.4 Selective framebuffer update and framebuffer compression

In D.5.5.1 we have described a selective framebuffer update scheme for tilebased rendering graphics systems. The target was to minimize to the extent possible the (redundant) off-chip memory accesses. Off-chip memory accesses are responsible for a large portion of the total power budget. According to [2], transferring data between the processing chip and the main memory approximately consumes a factor of 40x more energy than accesses to the last level system cache, which is 2 to 3 orders of magnitude more than the energy dissipated by floating point operations and 16-bit MAC operations.

Figure 11 depicts schematically a high-level diagram of a typical graphics processing system. As we can see, the graphics generation logic generates the output frame that is to be displayed on a display device (LCD or a TFT LCD screen). Upon a portion or a whole image is produced by the graphics hardware, it would then normally be written to a framebuffer in the off-chip memory through an interconnection network (write path in Figure 11). In this example and without loss of generality, we assume that the framebuffer is hosted in the system main memory, however different arrangements may be assumed, i.e., the framebuffer can be a separate off-chip memory or can be a separate on-chip memory residing in the display controller.

Sometime later, the framebuffer will be read by the display controller in order to output the generated frame (read path in Figure 11) For the sake of completeness, Figure 11 contains also a

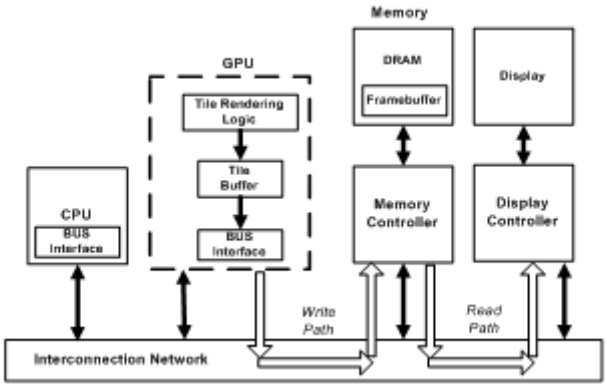


Figure 11: A typical graphics processing system.

host CPU.

Bandwidth and power benefits A typical scene is composed by many triangles. As each triangle covers a number of pixels, the number of fragments to be written to the display memory can be large. For instance, a scene may be composed of 1,000,000 triangles, each of which may cover 50 pixels. If the scene is rendered 60 times per second, 300,000,000 fragments must be generated, processed and sent to the framebuffer every second. If each such fragment carries about 5 bytes of data, 1.5 GBytes of data must be processed and stored every second. Further, many applications require to arithmetically blend newly rendered fragments with the contents of the framebuffer, doubling the data that must be transferred to and from the framebuffer.

Based on the power and area estimates provided in [2] and considering first order effect of the framebuffer and ignoring, for the purposes of this example, the display control power, the power consumed by the LCD panel, the video output power consumption, etc., a 32-bit mobile DDR-SDRAM transfer consumes about 1nJ per 32-bit transfer. Thus assuming a graphics processor frame output rate of 60 Hz and considering first order effects only, graphics processor framebuffer accesses consume about $(1920x1080x4)x(1nJ/4)x60x2 = 250mW$ and 948 MB/s for HD graphics and $(1024x768x4)x(1nJ/4)x60x2 = 94mW$ and 377 MB/s for 1024x768 resolution displays.

So it is clear that if someone is able to eliminate 40% of the framebuffer traffic (either from the traffic generated from the GPU and directed to the framebuffer or the traffic required to send the contents of the framebuffer to the display controller) that would save about 100mW and 379MB/s for HD composition framebuffer and 38mW and 151MB/s for 1024x768 graphics. Therefore, the power savings, by exploiting the redundancy between subsequent frames, can be significantly high which actually proves that there is ample room for optimization.

Selective framebuffer update In D.5.5.1 we have proposed and evaluated a selective framebuffer update scheme for tile-based rendering graphics systems. Figure 12 shows a high level view of our previous approach.

As Figure 12 indicates, the framebuffer update process is modified by the use of a specialized hardware unit called redundancy prediction unit or RPU. The role of the RPU is to predict if the new generated block is identical or almost identical with the block already stored in the framebuffer.

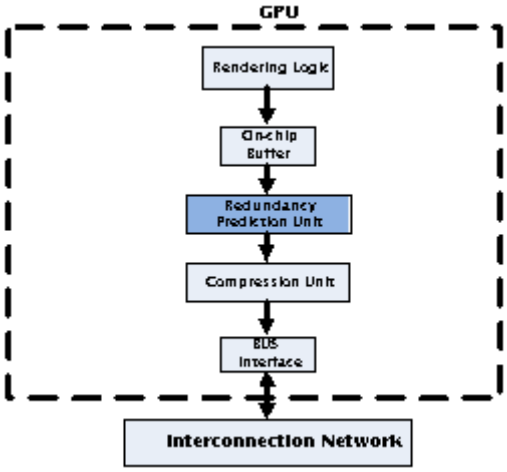


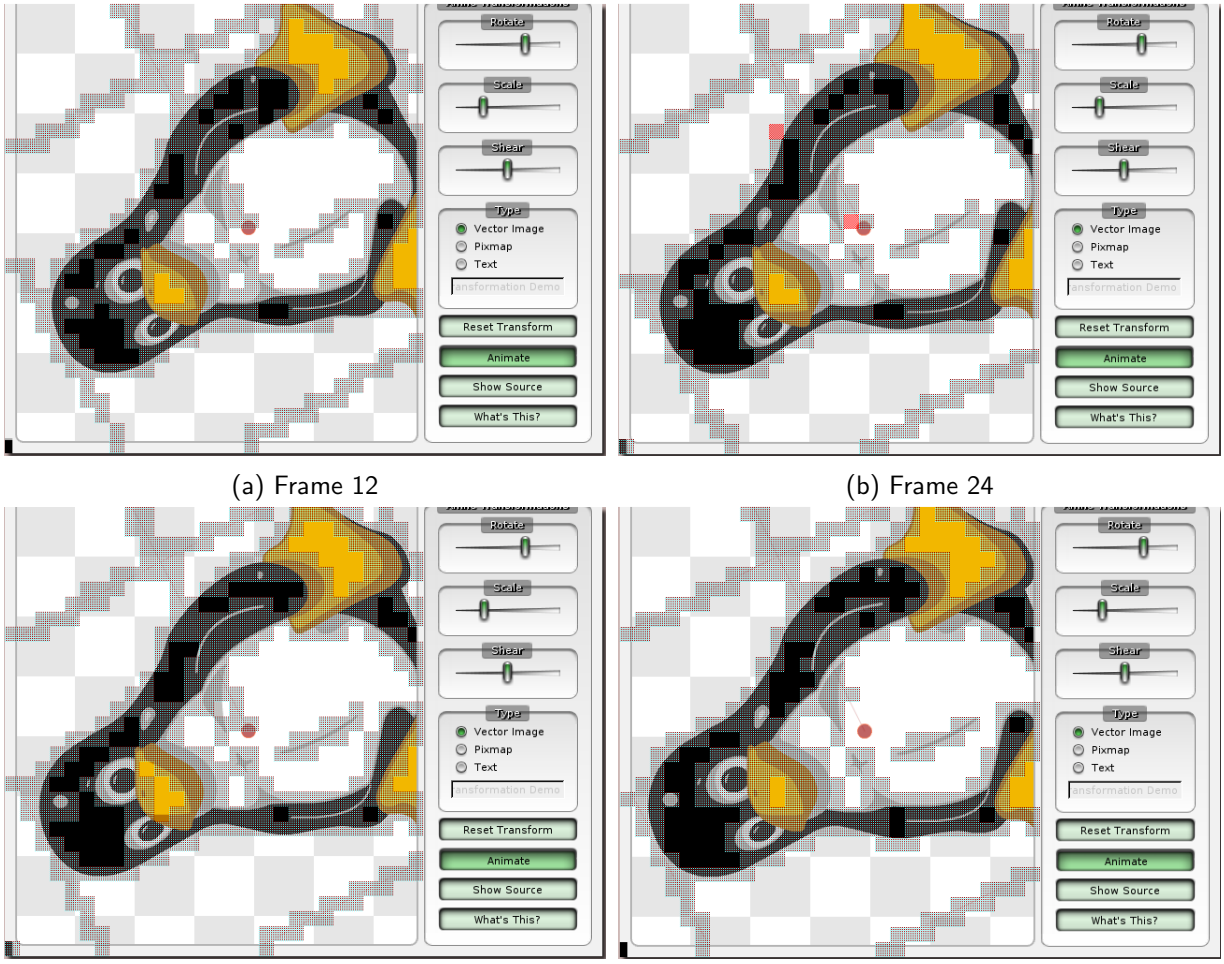
Figure 12: Proposed scheme.

More details can be found in [5]. D.5.5.1 contains a comprehensive study of the efficiency of the above scheme. However, that study was pertained to GUI-based animations provided by the Qt framework [3].

As a second step, we study the performance of RPU module using the traces provided by the Attila framework. In addition, in order to get a full understanding of the run time behavior of the underlying selective update scheme, we created demos using the benchmarks described in Table 1. Figure 13 and Figure 14 show four snapshots (frames) of the execution of the Affine (extracted from Qt framework) and the Quake 4 (extracted from the Attila framework) benchmarks, respectively. The corresponding frame number is depicted below the captured snapshots. The Animated Tiles benchmark has similar behavior to the Affine benchmark, while the Doom 3 and Prey Guru 5 resemble the behavior of Quake 4. In all cases, a 16x16 pixels tile rendering scheme is assumed (different tile organizations produce similar results).

In both figures, the dashed (shaded) parts of the images mark the frame portions that have been updated during the generation of the new frame (or the portions of the images that the RPU unit identified that are not identical with the previous frame), while the not-dashed parts indicate that those parts of the images remain the same (or the portions of the images that the RPU unit predict that have not changed with respect to the previous frame). As we can see, while the selective update scheme work very efficient in the case of the Affine benchmark, it falls to provide significant BW savings in the more realistic OpenGL-based Quake 4 benchmark. As it is further depicted in FIGURE 9, only 6.53% bandwidth reduction is achieved for the Quake 4 benchmark, while no significant bandwidth reduction is reported in the remaining OpenGL benchmarks rendering the selective framebuffer update scheme as not efficient in the case of dynamic and highly demanding applications (like the ones studied in this deliverable).

Furthermore, the proposed selective update scheme is able to offer bandwidth and power savings only in the framebuffer update traffic (write path in FIGURE 5). The traffic generated by the display controller in order to read the contexts of the framebuffer remains untouched (read path in FIGURE 5). As a result, a different approach is required which should be able to attack both sides of the problem, which is the subject of the next section.



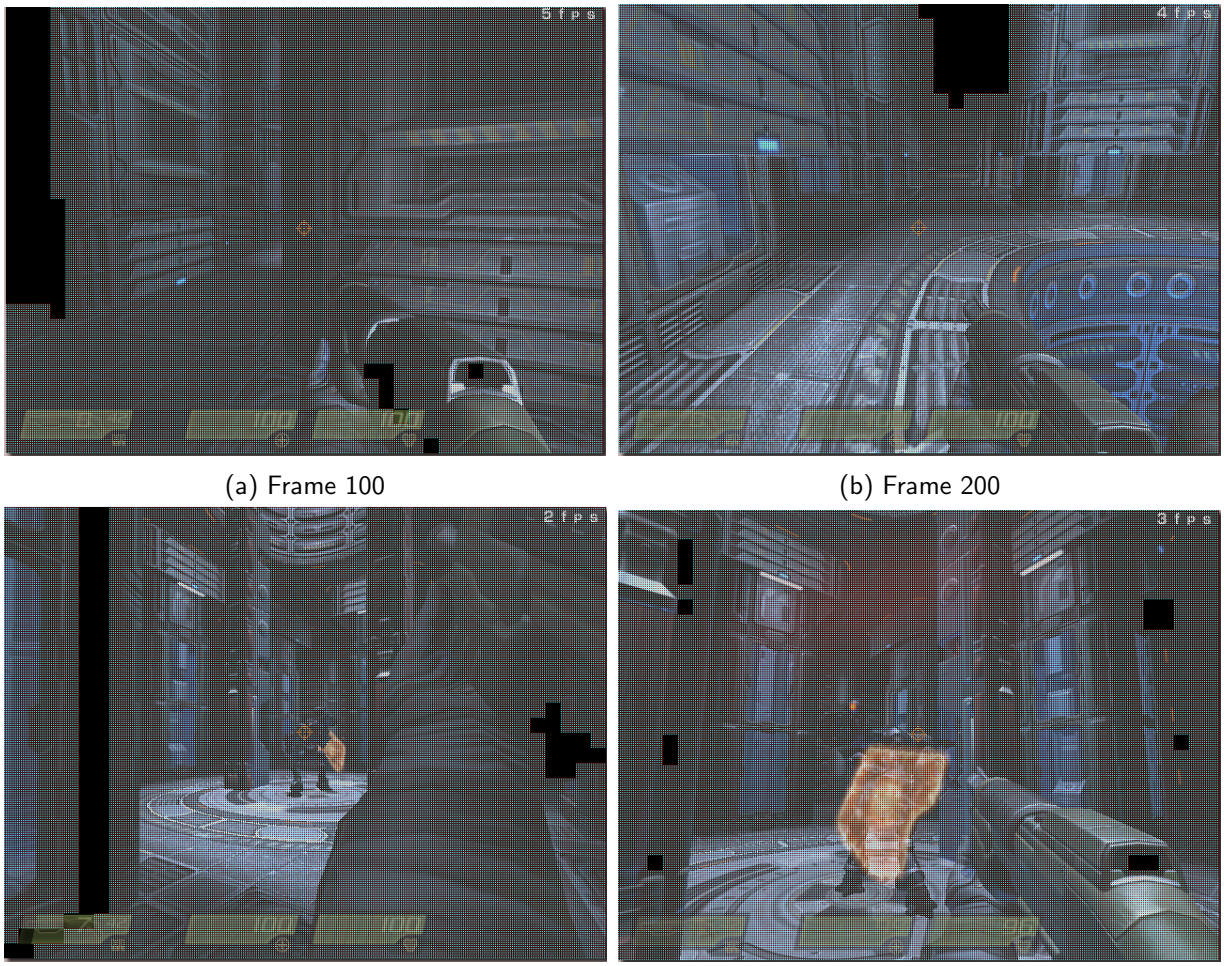
(a) Frame 12 (b) Frame 24

(c) Frame 36 (d) Frame 48

Figure 13: Run time behavior of the framebuffer selective update scheme for the Affine benchmark.

Framebuffer compression In order to mitigate the costly (in terms of power, time and I/O bandwidth) offchip memory transactions, we implement and evaluate a new lightweight framebuffer compression scheme which is able to offer a significant reduction in off-chip bandwidth. The new framebuffer compression technique, tailored to the special characteristics of a tile-based rendering system, is carefully designed to work on top of the selective update scheme reaping the benefits of both approaches. The proposed compression scheme can be configured to work either in a lossless or in a lossy compression manner at run time (adaptive compression). More specifically, the adaptive compression technique is able to issue informed compression decisions (limiting the number of errors introduced during the lossy compression process or revert back to a lossless compression technique if it is deemed necessary) trading the tile compression ratio with the quality of the output color data (adaptive compression with controllable errors). A high-level diagram of the proposed approach is depicted in Figure 16.

Compared to a vanilla tile-based rendering scheme, the scheme presented in Figure 16 relies on three new hardware units, namely Sampling Unit (SU), Redundancy Prediction Unit (RPU), and Compression Unit (CU). In the rest of this section we will first explain the functionality of each of these three units and we will show how these units can work together to reduce the bandwidth needed to access (via read or write transactions) the color data stored in the framebuffer.



(a) Frame 100 (b) Frame 200

(c) Frame 300 (d) Frame 400

Figure 14: Run time behavior of the framebuffer selective update scheme for the Quake 4 benchmark.

Framebuffer compression The cornerstone of the proposed compression process is that it is carefully performed so as to minimize the visual artifacts when the color data is decompressed. The compression process is driven by the sampling unit which enables informed compression decisions with controllable errors so as the output color data can be reliably decompressed to produce the original color data with minimal or no errors.

In general, the role of the SU is to capture and store at run time the color characteristics of the pixels of a rendered tile (in the form of a signature). This process is performed at run time i.e., when the tile rendering logic generates (produces) the color data of each pixel belonging to an under-rendering tile. Upon the rendering of a new tile is finished, the information gathered by the SU can be used for two purposes. First, the current instance of the information stored in the SU can act as a representation of the color data residing in the newly generated tile i.e., it may serve as a signature of the newly generated tile. At this point, we have to mention that in contrast to the redundancy elimination technique presented in D.5.5.1, in our current new implementation the representation of the tile color data is not based on hash values or signatures from image tiles (e.g., using checksums, CRC, MD5, or SHA-1 standards).

Second, the contents of the SU can be used not only as a representative icon of the color data

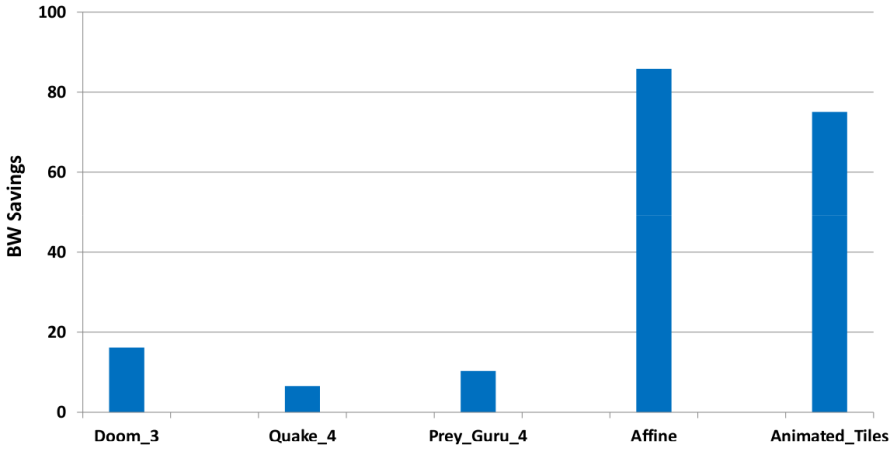


Figure 15: Bandwidth savings in framebuffer write (update) traffic (traffic from GPU to framebuffer).

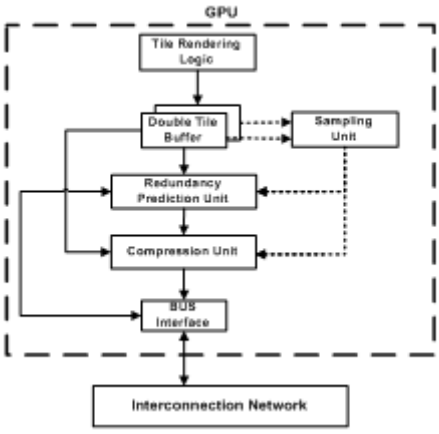


Figure 16: Selective framebuffer update and framebuffer compression scheme.

present in a new generated tile. The SU information can be also utilized to assist the compression unit to configure the lossless compression process in a way that the compression ratio of the tile is maximized. In case of a lossy compression scheme, the SU information can be used to control the amount of data lost during the compression process, thus keeping the error due to the lossy compression process as minimal as possible.

Figure 17 depicts an abstract implementation of the SU. As we can see, the implementation is divided into four identical parts (one for each color plane). Each part consists of a reference color channel and a storage array. Assuming a HD graphics system using 8-bits RGBA color planes, the most frugal version of the SU array can be as simple as an 8-entry bit array (one for each channel). A specific implementation can be seen in the following figure.

Thus, based on the SU implementation shown in Figure 18, upon a new pixel is generated the following process is taking place. First, the arithmetic distance or delta (e.g., using 2s complement logic) between the newly generated pixel and reference pixel in a per-channel basis is created. Of course, the maximum range of this arithmetic distance (difference) is 32-bits or 8-bits per-channel. The extracted differences are then mapped into the four bit-vectors. A working example of this procedure is show in Figure 19, assuming for simplicity one color channel.

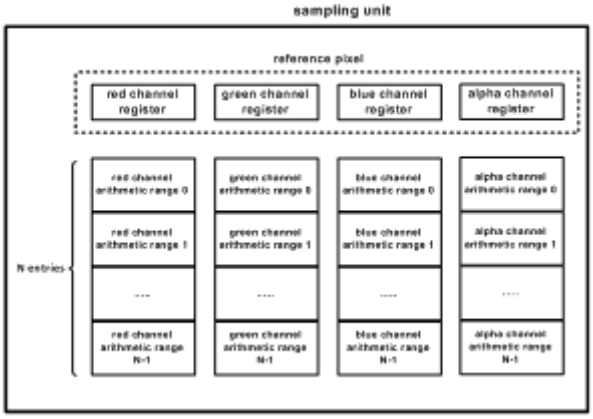


Figure 17: Sampling Unit implementation.

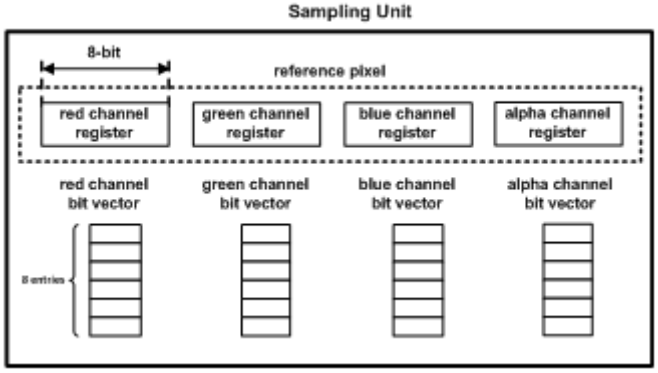


Figure 18: Sampling Unit implemented with bit-vector arrays assuming 8-bit RGBA color planes.

So as Figure 19 indicates, the SU acts as a superset vector of the 1s appearing in the arithmetic distances (deltas) between every new produced color and the reference color. Since this process includes only a simple comparison and a mapping of the resulting differences into the bit-vector, it can be performed at run time without introducing any additional latency in the system. In terms of storage overheads, it requires 4x8-bit bit vectors and a 32- bit register to be used as reference pixel (64-bits in total which is negligible). In addition, we have to mention in this point, that the information (arithmetic differences) captured by the SU is stored in a per-channel and not in a per pixel basis. This eventually means that the upcoming compression process (either lossy or lossless) will be performed in a per-channel basis, in contrast to all related approaches which attempt to compress color data in a per-pixel basis.

Apart from the SU configuration shown in Figure 19, other more sophisticated configurations are also possible. One such configuration is shown in Figure 20. The main difference with the SU configuration in Figure 19 is that each bit-entry is replaced with a counter (8 x 8-bit counters in total for a 16x16 tile system assuming 32-bits RGBA colors). The goal of those counters is to measure the summary of the 1s appearing in the resulting arithmetic differences. For example, when the rendering of a generated tile is completed, if the MSB counter contains a value equal to N (see Figure 20), this means that in N pixels, the MSB of the arithmetic differences is equal to 1. All the other counters work in the same manner but for different bit positions of the arithmetic distances. Having this counter-based information it is now possible to compress the new generate tile in an

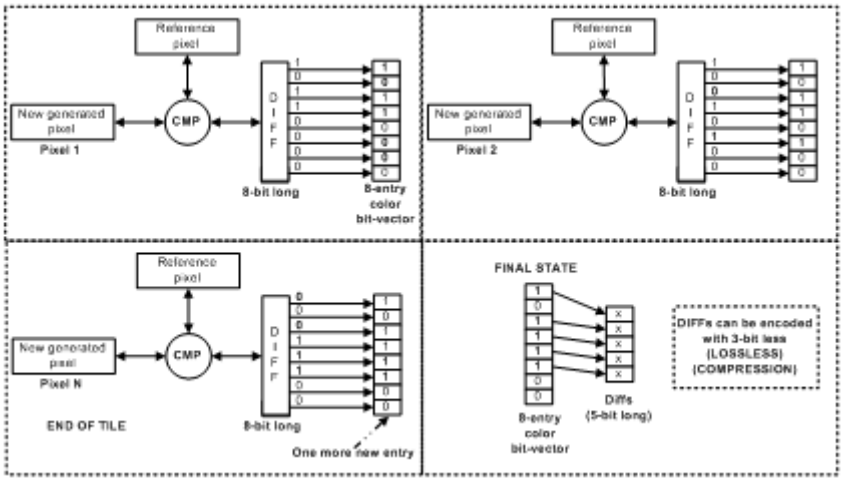


Figure 19: Working example illustrating the Sampling Unit driving the lossless compression process.

adaptive lossy manner, while keeping the amount of color data lost during the lossy compression process minimal (in other words, the compression errors are controllable). For example, as it is shown in the right part of Figure 20, if we choose to encode the arithmetic differences as 3-bits values, then the resulting error is limited to 10 pixels (controllable error).

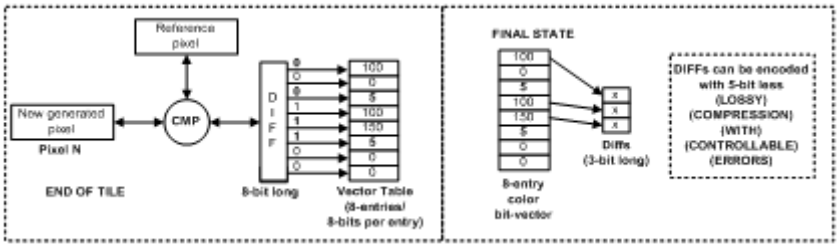


Figure 20: Working example illustrating the Sampling Unit driving the lossy compression process with controllable errors.

Finally, we have to mention how the reference pixel is selected. This is a critical design issue since all the other color values in the tile will be expressed as the arithmetic difference (e.g., using 2s complement logic) between each color value and the reference color value on a per tile basis. In general, the reference pixel (point) can be any pixel belonging in the tile under compression or it can be a pixel that does not belong to the tile under compression. For example, the reference pixel can be the first pixel of the tile (the pixel residing in the upper leftmost part of the tile) or it can be the most frequent pixel in the tile (the pixel with the larger occurrence in the tile) independently of the pixel coordinates. In our implementation, we consider for simplicity as reference pixel, the first rendered pixel in the tile independently of the pixel coordinates. The reference pixel is also stored in the framebuffer, so the color data can be safely decompressed.

Let now proceed with the description of the second main hardware unit required by our design, called Redundancy Prediction Unit or RPU (shown in Figure 20). This unit works in a similar manner with the one described in deliverable D.5.5.1 [5]. The only difference is that the signature of the tile color data is not longer a hash value (using for example a suitable algorithm from the CRC family), but the contents of the counter-based SU implementation. In this way, the SU contents serve two purposes at the same time (driving the compression process and acting as a signature of the tile color data). So when a new tile is generated by the graphics processing system, the counter-based

SU contents are also created. The next step is to fetch from the framebuffer the SU contents of the tile with the same coordinates that is generated and stored in the framebuffer during a previous generation of the tile. Upon the SU contents stored in the framebuffer arrive, they are inserted in the buffer (shown in Figure 20) and they are compared with the new generated SU contents (via the comparator shown in Figure 20).

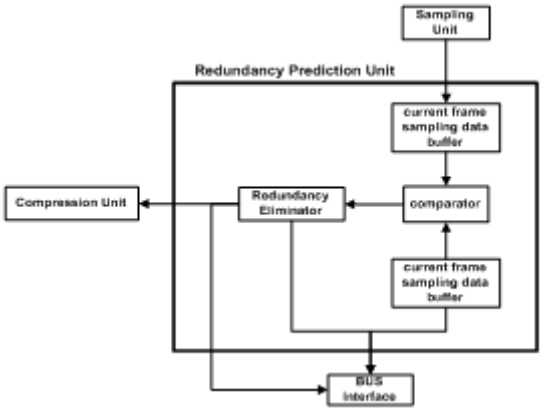


Figure 21: Redundancy Prediction Unit.

If a match occurs then the redundancy eliminator hardware decides not to store the new generated tile in the framebuffer (the contents of the on-chip buffer are immediately thrown away). In case of a mismatch, the new tile is transferred to the framebuffer (after it is compressed), as it would normally be happened in a typical graphic rendering system. If we assume that the size of a tile is equal to 16x16 pixels (16 x 16 x 4 = 1024 bytes) and the size of the SU array is 8 bytes (8 entries and each entry is 8-bits long), then a match will save 1016 bytes from storing them to framebuffer, while a mismatch will increase the traffic in the framebuffer by only 16 bytes (eight bytes to read the old SU contents and another eight bytes to store the new SU contents) which is insignificant (assuming that the compression process is skipped for the purposes of this example).

Of course and as it was explained in D.5.5.1, there always the possibility that two tiles with different contents may have the same SU contents, thus if the comparator indicates that two tiles have the same SU contents, then the tiles may be similar, we call them true similar tiles, or totally different (still with the same SU contents), we call them false similar tiles. As it is shown in the Result Section, this possibility is very low (well below 0.2%).

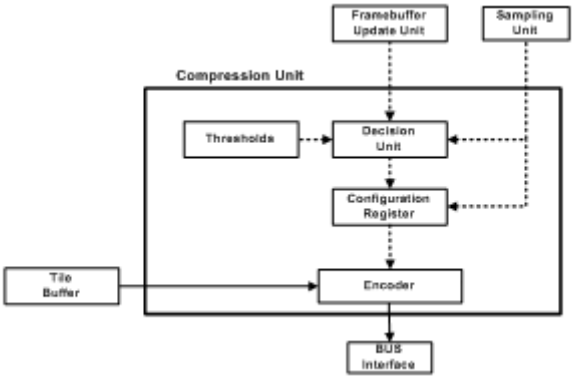


Figure 22: Compression Unit.

As mentioned, if the RPU hardware module indicates a mismatch, then the new generated tile (along with the SU contents) is forwarded to the compression unit (CU) for further processing before it is finally stored to the framebuffer. A sketch of the CU can be seen in Figure 20. The main role of the CU is to operate on the SU contents and accordingly drive the compression process. A dedicated hardware module, called Decision Unit (Figure 20) is responsible for this task. The output of the Decision Unit is stored to a special purpose register, called Configuration Register, and it is also forwarded to the Encoder Unit which is responsible to perform the final compression process according to the information stored in the Configuration Register. For the underlying compression algorithm, we selected to rely on a differential or delta compression scheme, so the critical design parameter at this point is how to define the size (length) of the resulting arithmetic distances (deltas).

In the case that the SU arrays are configured as bit-vectors, the role of the Decision Unit is quite straightforward. As it depicted in FIGURE 13 (bottom right part) the Decision Unit simply indicates the bit positions that can be ignored during the generation of the arithmetic distances (e.g., using 2s complement logic). Of course, in this case, a lossless compression process is implied. Note that the compression is applied in a color plane basis, which means that different compression ratios may be achieved for the different tile color channels. In other words, the Configuration Register contains information in a per-color basis (4 x 8-bits registers in total). As it also depicted in Figure 23, the Configuration Registers are also stored in the framebuffer, so as the compressed color data can be safely decompressed by the display controller.

However, the situation becomes more interesting when the SU arrays are configured as counter vectors (each SU entry is 8-bit long assuming a 16x16 pixels with 32-bit RGBA depth color tile-based rendering system). In this scenario, the Decision Unit is responsible to control the aggressiveness of the lossy compression process or revert back to a lossless one if it is deemed necessary (adaptive framebuffer compression). In general the target of any lossy compression scheme is to minimize the resulting errors (amount of information lost during the lossy compression encoding), while increasing the compression ratio to the extent possible (for power and memory bandwidth savings). In the case of image compression, if the lossy compression is not carefully performed, the quality image will be significantly reduced and most importantly visual artifacts will occur. As a result, the amount of resulting errors must be totally controllable and the lossy compression scheme must be able to adapt its aggressiveness (or compression errors) according to predefined error rates.

The information gathered by the counter-based SU arrays serves exactly this purpose. Consider the example shown in FIGURE 14, when the tile generation is finished, the Decision Unit is able to provide informed compression decisions (controlling the aggressiveness of the lossy compression process), since the resulting number of errors can be easily extracted from the information stored in the SU arrays. The question which arises now is how the error rate level is defined. This can be the subject of various system parameters taking into account the application dynamic behavior (e.g., refresh or changing rate of the rendered images), the user requirements (e.g., quality of service requirements set by the user), and/or the run time conditions of the rendering hardware (e.g., the remaining battery life of a portable device or the lighting conditions of the environment in which the graphic device operates). For the purposes of this deliverable the resulting error rate is predefined to 0.04 using the DSSIM metric (which means that Decision Unit tries to increase the compression ratio without increasing the dissimilarity factor by more than 0.04 on average). If the Decision Unit realizes that reducing the delta sizes will exceed the predefined threshold, then the compression process automatically revert back to a lossless scheme or even a no-compression decision is issued (in this case the compression process is skipped, and the tile is transferred uncompressed to the framebuffer).

A final touch in our approach that is orthogonal to all previous steps is that the compressed data (either in a lossy or a lossless manner) are further processed by a simple, single pass, lossless, run length encoding (i.e., an extra layer of compression is introduced before the color data are finally sent to the framebuffer in memory). This run length encoding mechanism aims to capture and encode multiple color deltas that are similar as a single entry. More specifically, the encoder (shown in Figure 20) scans sequentially the resulting deltas and compares the new deltas with the previous ones and increments a counter if they are equal. The target is to substitute one or (preferably) multiple deltas with a single delta value and a count corresponding to the occurrences of the same delta value in the data stream. Again, this run length encoding technique is configured to operate at color plane level rather on the pixel level.

The only parameter that should be defined in the design of the run length mechanism is the size of the length counters (number of bits required for the length counters). Of course, this parameter is application dependent. We experimentally found that configuring the length counters to be 3-bits long provided the best compression ratio (on average) for the benchmarks listed in TABLE 1.

Finally, in Figure 23, it is also depicted how the framebuffer should be organized to support the above mentioned framebuffer compression and redundancy elimination schemes.

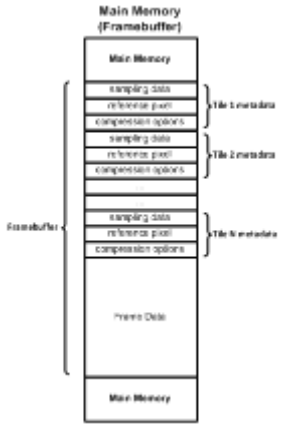


Figure 23: The proposed framebuffer organization.

Evaluation results This subsection provides our evaluation results for the selective framebuffer update and framebuffer compression schemes. As noted, we performed our experiments using the benchmarks listed in TABLE 1. In all cases, we provide our results assuming a 16x16 tile based rendering system with 32-bits RGBA HD colors (our results for a 32x32 tile system are fairly similar). We have also experimented with tiles of different sizes. However we ended up with those two tile organizations for the following reasons. First, for smaller tile organizations, the size of the counter-based SU arrays are prohibitively high (30% overhead in a 4x4 tile system, while a 3.51% (1.07%) in a 16x16 (32x32) tile organization). Second, larger tiles are proved to be ineffective due their large on-chip storage requirements and due to their algorithmic complexity.

Figure 24 quantifies the quality of the predictions done by the RPU module. For this measurement, no specific type of compression is assumed, because the type of compression (lossy, more lossy, or lossless) does not influence the RPU predictions (if the RPU predicts that a new generated tile is similar to the tile stored already to the framebuffer, then the whole process is immediately stopped, the contents of the on-chip buffers are thrown away and a new tile is inserted in the rendering

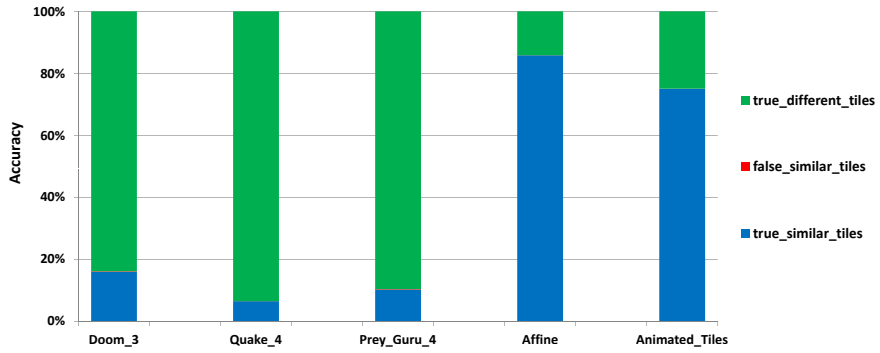


Figure 24: Prediction accuracy of the framebuffer selective update scheme.

pipeline). There are five stacked bars in Figure 24; one for each studied scene. Each bar is divided into three parts. The blue part (tagged as true_similar_tiles) corresponds to the cases in which the comparisons of the respective signatures (the signature of the newly generated tile and the signature produced during a previous generation of the tile) indicates a match and the respective tiles (new and old one) are indeed similar (true positive answer). In this case, the generated tile is not written to the framebuffer. The red part (tagged as false_similar_tiles) corresponds to the cases in which the comparison indicates a match and the respective tiles are not similar (false positive answer). Again, the generated tile is not written to the framebuffer, although in this case this decision is not correct (the two tiles differ). Finally, the green part of the bars (tagged as true_different_tiles) corresponds to the cases in which a mismatch is reported (true negative answer). In this case, the generated tile is written to the framebuffer. In other words, the blue and the green parts of the bars correspond to the correct decisions indicated by the comparison process (the tiles are indeed similar or not similar), whereas the red parts show the incorrect or false decisions (the tiles are not similar, while the signatures are similar).

Two main conclusions can be drawn from Figure 24. First, using the SU contents to act as a representation (signature) of a tile is proved to be very effective leading to prediction errors below 0.2% in all benchmarks (the red part of the bars is actually not visible). Second, as shown also in FIGUREs 7 and 8 (demos), the potential in terms of bandwidth savings of the selective framebuffer update scheme is significantly reduced in the high-quality OpenGL games (first three bars). Only 6.52% savings in framebuffer update traffic is achieved in Quake 4, 16.02% in Doom 3, and 10,24% in Prey Guru 4. In contrast, in the GUI-based animations the corresponding savings are significantly high (85.82% in Affine and 75.02% in Animates Tiles) replicating the results of FIGUREs 7 and 8. Similar conclusions can be sketched from Figure 25.

Figure 25 shows the resulting bandwidth savings achieved by a wide range of configurations of the mechanisms proposed in this deliverable. Note that the results shown in Figure 25 are normalized to framebuffer update traffic (traffic from the graphics rendering system to the framebuffer). For completeness, we also depict in Figure 26, the bandwidth savings reported in the framebuffer read traffic (from the framebuffer to the display controller). There are seven bars attached to each benchmark in Figure 25. Starting from left to right, the bars correspond to the following cases: i) the first (light blue) bar depicts the benefits achieved by the standalone run length encoding technique, ii) the second (red) bar shows the performance of the standalone selective framebuffer update scheme, iii) the third (yellow) bar shows the performance when the run length compression and the selective update schemes are applied at the same time, iv) the fourth (purple) bar shows the additive benefits of the lossless compression scheme combined with the run length encoding, v)

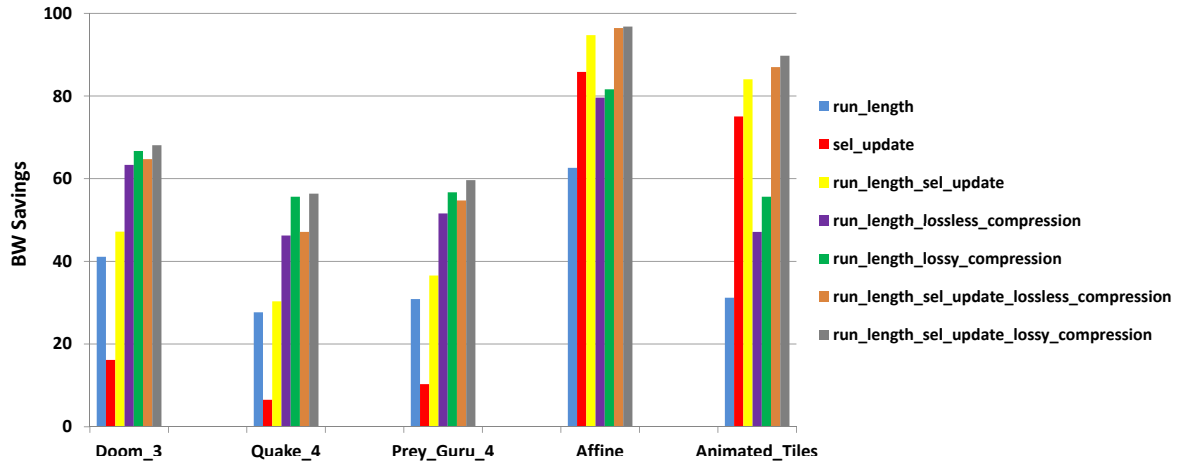


Figure 25: Bandwidth savings in framebuffer write (update) traffic (traffic from GPU to framebuffer).

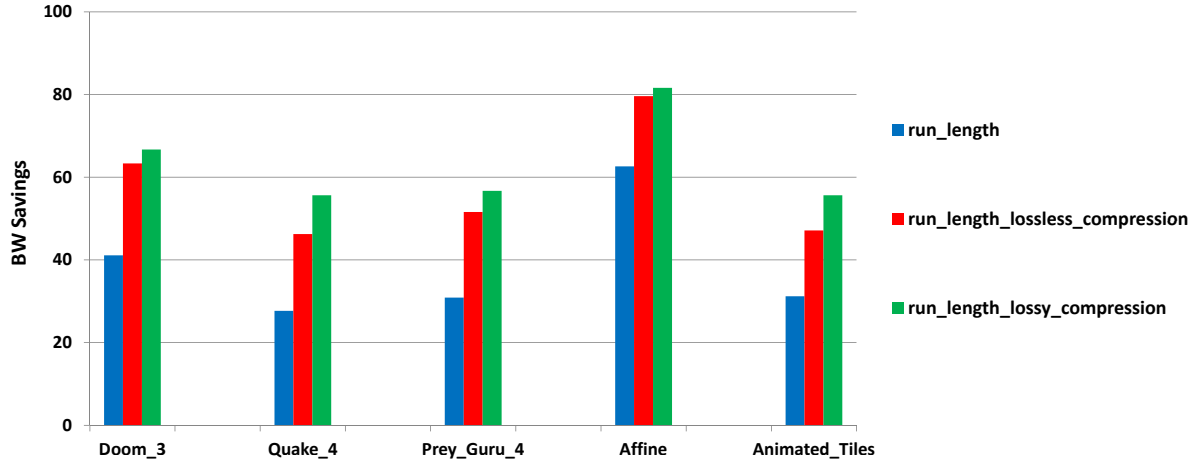


Figure 26: Bandwidth savings in framebuffer read traffic (traffic from framebuffer to LCD controller/display).

the fifth (green) bar corresponds to the case which the run length encoding is combined with the lossy compression scheme, vi) the sixth (brown) bar illustrates the performance when the run length encoding, the selective framebuffer update scheme, and the lossless compression are utilized at the same time, and vii) finally the seventh (grey) bar is similar to the previous case, but now a lossy compression scheme is enforced.

As we can see from Figure 25, the resulting bandwidth savings are application dependent. As noted, the selective update scheme is only effective in the two GUI-based applications (the two applications in the right part of the graph). The standalone run length mechanism, although very simple to implement, is not able to offer significant bandwidth savings (except in the case of the Affine benchmark). In general, it is obvious that in all cases, a combination of techniques is required in order to achieve the best possible result. The rightmost bars (run length plus selective update plus lossy compression) is the one that produces the higher bandwidth savings: 68.08in Doom 3, 56.38% in Quake 4, 59.65% in Prey Guru 4, 96.79% in Affine, and 89.75% in Animated Tiles. Furthermore, Figure 26 illustrates the bandwidth savings in the read framebuffer traffic. In this case, only the various compression schemes are analyzed (the selective update scheme is not applicable

in this case). Again, the higher bandwidth savings are reported when the lossy compression scheme is combined with the run length encoding mechanism: 66.71% in Doom 3, 55.63% in Quake 4, 56.72% in Prey Guru 4, 81.61% in Affine, and 55.63% in Animated Tiles.

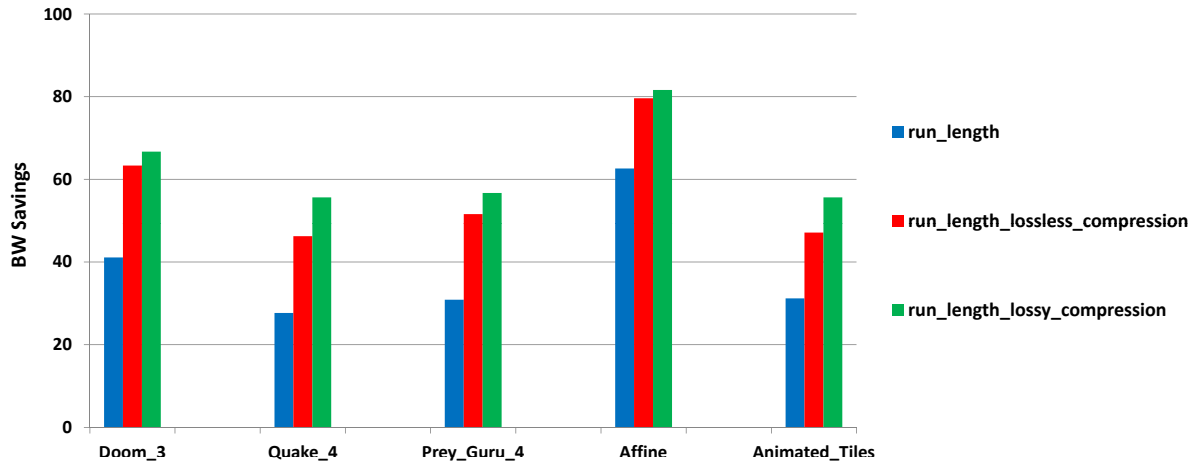


Figure 27: Average and maximum structural dissimilarity (DSSIM) reported by the selective update scheme and the proposed tile-based lossy compression technique.

Finally, Figure 27 presents the error introduced in the image quality by relying in the structural dissimilarity (DSSIM) metric. Of course, a possible reduction in the image quality can be only experienced in two cases: in the selective framebuffer update scheme (due to the false positive predictions) and in the case of the lossy compression technique. For those two cases, Figure 27 shows the average and the maximum errors. All the resulting errors are computed by comparing the outputs (framebuffer instances) of the corresponding mechanisms with the outputs in which no optimizations were applied. As we can see, in all cases, the average (or even the maximum) reported errors are well below 0.03 which is considered as a safe limit even in the case of comparing static images (see the Structural similarity factor section).

2.4 Discussion

In this section we presented the work conducted in Task T5.2 (Redundancy) and Task T5.4 (Accuracy) during the first half of the second year of the project as defined in the LPGPU work plan. From the start of the LPGPU project, we try to attack these two concepts from two different angles. The target of the first one is to override complex arithmetic calculations via simple value memorization techniques (also called the Value Cache approach). The target of the second one is to reduce the resource greedy (in terms of time, power and bandwidth) off-chip framebuffer memory accesses. The latter approach is fully described in this deliverable and our results reveal that significant bandwidth savings can be achieved using the techniques presented in this deliverable. In that respect, the work targeting the reduction of the framebuffer activity can be considered as completed.

In the rest of the project, we will concentrate on the Value Cache idea. In our current developing stage, we are porting the value cache approach in the Attila simulator [1] and more specifically in the OpenGL fragment shaders of the simulated GPU. Our first results are promising, but a significant amount of work is needed to understand the underlying behavior of the system and create the

appropriate mechanisms to architect the Value Cache approach. In addition, working with the Attila simulator offered to us one more opportunity: to get the full picture of the real bottlenecks of a working graphics processing system. For example, we have found that based on the input application, either the OpenGL fragment shading part or the OpenGL vertex shading part can be the bottleneck of the system. This changing behavior may be exploited for further optimizations. For example, the shading part that is not the current system bottleneck can be clocked down (via DVFS) or in other words there is available slack at the OpenGL level that can be utilized to increase the power efficiency of the target system especially in the case of multicore GPUs.

Table 1: Benchmark suite for evaluating the Accuracy and Redundancy Tasks.

ID	Description	Number of snapshots	Size in pixels	Sample
Scene 1	Game: Doom 3 (OpenGL)	2398	640x480	
Scene 2	Game: Quake 4 (OpenGL)	2270	640x480	
Scene 3	Game: Prey Guru 5 (OpenGL)	2364	640x480	
Scene 4	Affine: cartoon-like flash based applications.	101	640x480	
Scene 5	Animated tiles: animated menu screen.	98	640x480	

References

[1] Attila framework, 2012. http://attila.ac.upc.edu/.

[2] Bill Dally. Gpu computing to exascale and beyond, 2010. http://www.nvidia.com/content/PDF/sc_2010/theater/Dally_SC10.pdf.

[3] Digia. Qt framework, 2013. qt.digia.com.

[4] Khronos Group Inc. Openvg specification, 2008. http://www.khronos.org/registry/vg/specs/openvg-1.1.pdf.

[5] Stefanos Kaxiras, Georgios Keramidas, Konstantinos Koukos, and Iakovos Stamoulis. D5.5.1 preliminary report on architectural techniques for power efficiency, 2012. http://www.lpgpu.org/wp/.

[6] Hyesoon Kim, Jaekyu Lee, Nagesh B. Lakshminarayana, Jieun Lim, and Tri Pho. Macsim simulator. http://code.google.com/p/macsim/.

[7] Konstantinos Koukos, David Black-schaffer, Vasileios Spiliopoulos, and Stefanos Kaxiras. Towards More Efficient Execution : A Decoupled Access-Execute Approach Categories and Subject Descriptors. In *ICS'13, June 10 - 14 2013*, Eugene, OR, USA, 2013. ACM 978-1-4503-2130-3/13/06.

[8] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems - ASPLOS-IV*, (Asplos Iv):63–74, 1991.

[9] A. Loza, L. Mihaylova, N. Canagarajah, and David Bull. Structural similarity-based object tracking in video sequences. In *Information Fusion, 2006 9th International Conference on*, pages 1–6, 2006.

[10] D. Nowroth, I. Polian, and B. Becker. A study of cognitive resilience in a jpeg compressor. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 32–41, 2008.

[11] V. Spiliopoulos, S. Kaxiras, and G. Keramidas. Green governors: A framework for continuously adaptive dvfs. In *Proceedings of the 2011 International Green Computing Conference and Workshops*, IGCC '11, pages 1–8, Washington, DC, USA, 2011. IEEE Computer Society.

[12] V. Spiliopoulos, A. Sembrant, and S. Kaxiras. Power-sleuth: A tool for investigating your program's power behavior. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, pages 241 –250, aug. 2012.

[13] Multi2Sim Developer Team. Multi2sim simulator. http://www.multi2sim.org/.

[14] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *Image Processing, IEEE Transactions on*, 13(4):600–612, 2004.