D5.5.4– Characterization of Redundancy and Definition of Work Reuse

Document Information

Contract Number	288653
Project Website	lpgpu.org
Contractual Deadline	31-08-2013
Nature	Prototype (P)
Authors	Stefanos Kaxiras (UU), Georgios Keramidas (Think-S), Konstantinos Koukos (UU), Iakovos Stamoulis (Think-S)
Contributors	TUB, Codeplay
Reviewers	Paul Keir (Codeplay)

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 Task Description

This task involves the following work:

- Establishing the relation of *Quality of Service* (QoS) and energy to accuracy.
- Design and development of techniques to dynamically decrease accuracy (e.g., ignore low order bits in computations). Deliberately ignoring a few low order bits in calculations where the application allows it (in terms of impact to QoS) reduces energy requirements. We will implement techniques to allow change of accuracy, dynamically (run-time) or under application control.
- Design and development of techniques to tolerate transient errors in computations. Reducing the supply voltage beyond the point required by the operational frequency can yield significant power reductions but at the expense of potential (timing) errors. Such errors sacrifice some of the accuracy of computations. However, if such errors are not important to the resulting quality of service then this is a promising approach to increase power efficiency. In cases where absence of any error is necessary for the correct operation of the GPU, error detection and correction techniques will be applied.

2

Relevance to the Project

Input dependencies: T4.2: GPU power simulator, T4.3: high-level macro-based simulator.
Output dependencies: T7.2: Evaluation and Optimization Techniques to exploit Redundancy in GPUs

Description: While it is generally considered that GPUs are not power-inefficient in that they have tremendous performance and it is simply a matter of paying (in power) for what you get (in calculations), we believe that there is a great opportunity to change this misconception by examining closely redundancy in calculations (and other operations, especially memory accesses). We have indication that redundancy exists among frames (potentially significantly more than in a single frame) in many graphics applications. Consider for example that to render two nearly identical frames only a small part of the calculations could actually differ. Our goal is to exploit this redundancy, eliminate it completely or use techniques such as work-reuse, memoisation storing the results of calculations to avoid re-calculation, caching, and even result prediction (value prediction), thus reducing the amount of work (calculations and data movement) per frame (in a pipelined fashion). Thus the same performance (in terms of frame throughput but maybe not individual frame latency) could be achieved at much lower operating frequencies (DVFS), yielding significant power benefits both from lower dynamic energy (lower f) and lower switching activity for the same number of rendered frames.

In this task we plan to develop techniques to exploit redundancy at the following levels:

- OpenGL level
- CUDA level
- thread level
- procedure/function level
- hardware (floating point operation) level

At each level we will implement a work-reuse mechanism designed to reduce energy consumption by eliminating redundant computation or data movement. While for the upper levels work-reuse mechanisms are software optimizations, for the lower levels (thread/function/operation) such approaches require hardware support. Frame pipelining: To expose more redundancy to the work-reuse mechanisms it is necessary to deal with more than one frame at a time. Architecturally this means that the operational mode of the GPU or GP-GPUs is a frame pipeline, where instead of completing one frame at a time, small parts of multiple consecutive frames are processed simultaneously. The architecture will also be implemented as part of the simulator in WP4. Memory access is a significant source of power inefficiency in GPUs simply because there isnt enough on-chip cache. This increases the accesses to the external memory as well as the data transfer inside the GPU. Redundancy in

the memory access behavior and data movement will be exploited by caching and intelligent data location techniques. Computation Redundancy can also be increased by ignoring low order bits in the results. Since this has the potential to affect quality of service we examine this in more depth in Task 5.4.

Task 5.2 consists of 3 steps:

- Estimation of the potential redundancy at various levels in graphics applications.
- Design and development of work reuse techniques at various levels to exploit redundancy.
- Design and development of frame pipelining architectures to increase the potential for redundant work.

Role of beneficiaries:

Uppsala will lead the research to assess redundancy in GPUs and also develop frame pipelining. TUB will assist with the exploration using simulation. ThinkS will participate in the design of work reuse techniques to exploit redundancy. Codeplay will assist in the development of frame pipelining from the application side.

3

Work Done

3.1 Introduction

This section presents the latest results of the work conducted in Task T5.2 (Redundancy) and Task T5.4 (Accuracy) during the second half of the second year of the LPGPU project. Although the purpose of the present deliverable, as it was initially defined in the LPGPU work plan, was to provide a characterization of possible techniques to exploit redundancy and to provide a definition of the work reuse techniques in GPUs, the concept of accuracy is still under exploration. The reason for this is that, as also noted in deliverables D5.5.1 [7] and D5.5.2 [8], 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. In other words, there is a significant interdependence between the two concepts resulting in QoS/power trade-offs in calculations and in off-chip memory accesses. Thus, we selected to continue our investigation in those two concepts in the same framework. In this aspect, this deliverable complements D5.5.1 and D5.5.2.

With respect to the said concepts, from the very beginning of the LPGPU project, in WP5 we concentrate on two specific scenarios. The first scenario was to exploit redundancy/accuracy in off-chip memory accesses and in particularly our effort was focused on optimizing the most consuming (in terms of power, time and I/O bandwidth) memory operation in all GPUs which is the writing and reading of a rendered image to/from the so-called frame-buffer. The second scenario was to exploit memorization or work reuse techniques to eliminate the complex arithmetic operations which typically exists during the processing of three-dimensional graphics data. We called this approach *value cache*. This deliverable is divided into two main parts showing our latest results in those two directions. More specifically, the contribution of our work is the following:

- Reduction of frame-buffer activity: In D5.5.2, we have illustrated two schemes for reducing the costly writing and reading memory operations to/from the frame-buffer. Both schemes are implemented as part of our work in the LPGPU project. While in the D5.5.2, the evaluation of both schemes was performed in terms of off-chip bandwidth savings, in the present report, we also provide power/energy number savings.

- Value Cache: we have ported part of the Value Cache simulator developed during the first year of the project in the Attila [1] simulator. More specifically, we opted to explore the concept of redundancy and accuracy in the fragment shading processing part of the Attila simulator. We analyzed the fragment shading codes of state-of-art OpenGL-based computer games in order to indentify the most frequently executing and more complex graphics instructions. We apply the concept of redundancy and accuracy in the vector MUL (multiply) instruction and we present our evaluation results.

3.2 Reduction of Framebuffer Traffic

In D5.5.2 we have analytically presented our proposals for reducing the read/write activity to the frame-buffer. The provided schemes fully exploit the concept of redundancy and accuracy. Two techniques were proposed: a selective frame-buffer update techniques and a set of lightweight compression techniques dedicated for graphics processing systems utilizing the tile-based rendering approach. The proposed compression techniques are adaptive (i.e., they are able to adapt the compression ratio based on the special characteristics of the executing graphics workloads) and they can be configured to work either in a lossless or a lossy fashion. An important characteristic is that both schemes are designed to work either as standalone techniques or to work together (i.e., are orthogonal to each other) further increasing the resulting benefits. In the latter case, the two said schemes share common structures and implementation logic.

All the implementation details as well as our evaluation results are provided in D5.5.2. For completeness, in the present deliverable we provide some additional evaluation results for assessing the effectiveness of our proposals in terms of energy/power savings; only bandwidth savings were provided in D5.5.2. Before presenting our new evaluation results, we want to highlight the value of the said frame-buffer optimization schemes especially in low-end and ultra low power mobile devices.

Consider for example, a typical scheme consisting of 1,000,000 triangles where each triangle may cover 50 pixels (the provided numbers are given after analyzing representative OpenGL-based computer games). If each pixel carries about 4 bytes of data, 1.2 GBytes of data must be processed and transferred to the frame-buffer per second. Moreover, if we take into account that the same amount of data must be also transferred (via read transactions) in an equal or similar frame rate from the frame-buffer to the display controller which is responsible to display the generated frame on the display device. Furthermore, in many cases blending operations are also required which results in almost doubling the data that must be transferred from the graphics processing unit to the frame-buffer.

In addition, if we consider that depth color related data (to calculate the pixel visibility), vertex related data and texture data must also travel in the on-chip interconnection medium, then it becomes obvious that any bandwidth savings technique is of paramount importance especially in devices characterized by scarce resources like the state-of-art smartphones.

In our effort to interpret the benefits of our frame-buffer activity reduction schemes in terms of power/energy savings, we tried to analyze the power figures of a typical tablet device. Our analysis revealed that based on the battery characteristics (voltage level, ampere-hour level, and average battery life), a typical tablet device consumes approximately 11.5 Watt during operation. In addition, given that the battery capacity increases at a modest pace of 5-10% per year [11] combined with the fact that new processing capabilities and new features are included in every smartphone or tablet generation, then if someone wants to keep the same operating time per battery charge, the average power consumption is expected to be kept at the said level.

Not surprisingly, a recent study based on real usage statistics revealed that most of the power today (above the half of the named power level) is consumed by the communications subsystem and the screen [5] and this trend is expected to significantly grow in the future. Furthermore, a state-of-the-art embedded processor (e.g., A9 ARM processor) consumes about 200-500 mWatts [2], while a modern embedded GPU system consumes about 40-100 mWatts [3].

At this point, we can note the following observations. First, the above estimates were extracted by trying to reverse engineer the power figures provided in the websites of major players in the field of mobile devices. Although, it is well known that trying to correlate power numbers provided by different vendors is a tricky process, in the context of this work our effort was not concentrated on identifying highly accurate numbers, but to get a picture of the power distribution in a smartphone or a tablet device. Second, the gpgpu-pow simulator, developed as part of the LPGPU project, could not be used for performing the said characterization, because gpgpu-pow can only provide power numbers for a GPU running general purpose applications (e.g., based on OpenCL), thus ignoring the power consumed by major graphics related components or operations, like the color and depth buffer operations, the texture mapping unit etc.

So, with respect to our frame-buffer traffic optimization techniques and considering only first order effect of the frame-buffer 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 state-of-art GDDR5 memory (a low power DDR3-like memory dedicated for mobile graphics devices) consumes about 15 picoJoules per bit transfer [4], [6]. Thus assuming a graphics processor frame output rate of 60 Hz and considering first order effects only, graphics processor frame-buffer accesses consume about (1920x1080x32) x (15pJ) x 60 x 2 = 119 mW and 949MB/s for HD graphics and (1024x768x32) x (15pJ) x 60 x 2 = 45 mW and 360 MB/s for 1024x768 resolution displays. Note that the factor x2 in those equations exists due to the fact that the frame-buffer data must be transferred from the GPU to the frame-buffer via write transactions and from the frame-buffer to the display controller via read transactions.

So it is clear that if someone is able to eliminate 80% of the frame-buffer traffic (as we propose in the context of the LPGPU project) that would save about 96mW and 759MB/s for HD composition frame-buffer and 36mW and 288MB/s for 1024x768 graphics. Therefore, the power savings, by exploiting the redundancy between subsequent frames, can be significantly high. Moreover, given the power figures presented earlier, it is obvious that the power spent in writing/reading the frame-buffer is equal or almost equal to the power consumed by the whole GPU subsystem especially for HD displays.

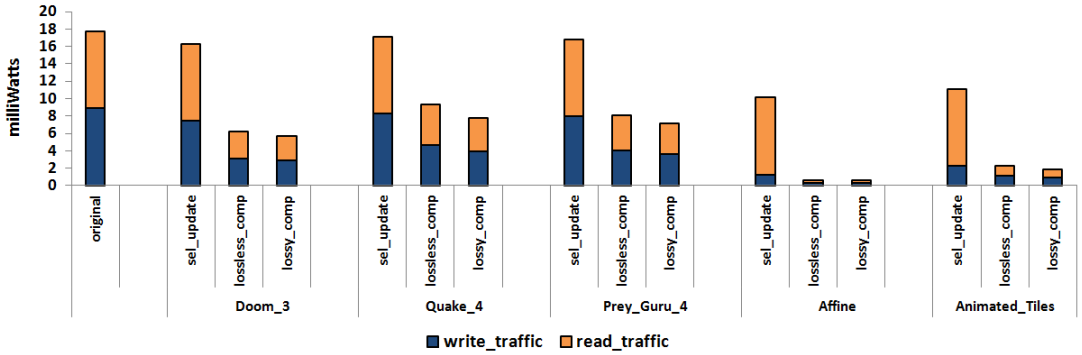


Figure 1: Power consumption (absolute numbers) for the proposed frame-buffer traffic reduction schemes.

Figure 1 shows the resulting power savings achieved by the proposed selective frame-buffer update and frame-buffer compression schemes. The statistics in Figure 1 are gathered assuming a 16x16 tile based rendering system with 32-bits RGBA colors (8-bit color planes). We also assume that the frame-buffer is located in a GDDR5 memory that consumes 15 picoJoules per bit transfer [4], [6] and we consider only the first order effect of the frame-buffer and we ignore, for the purposes of this

example, the display control power, the power consumed by the LCD panel, the video output power consumption, etc. Finally, the graphics processor frame output rate as well as the LCD refresh rate is set to 60 Hz. For more details about the simulation framework and the benchmark suite used in this study, we refer the interested reader to D5.5.2.

The vertical axis in Figure 1 corresponds to the estimated power consumption (in milliWatts) required to perform the writing (updating) and reading frame-buffer operations. Each bar in Figure 1 is divided into two parts. The bottom (blue) part of the bars shows the power consumed during the write operations (traffic from the graphics processing system to the frame-buffer), while the top (orange) part depicts the power consumed during the read operations (traffic from the frame-buffer to the display system). In addition, the leftmost bar illustrates the power spent in a conventional system without any frame-buffer optimization techniques (note that the leftmost bar is benchmark independent; in a conventional system, the power consumed by the frame-buffer is mainly defined by the color depth, the frame rate, and the screen resolution). The remaining bars are divided in groups of three bars and each group corresponds to a specific benchmark. Finally, in each group, the left bar (tagged as sel_update) shows the power consumption when only the selective frame-buffer update scheme is employed; the bar in the middle (tagged as lossless_comp) corresponds to the case in which the selective update technique is combined with the proposed lossless compression scheme, whereas the right bar (tagged as lossy_comp) depicts the additive benefits of the selective update scheme and the proposed lossy compression technique.

Given the above system configuration and as indicated by the leftmost bar in Figure 1, the total power consumed by the frame-buffer related operations is (640 x 480 x 32) x (15pJ) x 60 x 2 = 17.69 mW (note that all the selected benchmarks were extracted assuming 640 x 480 resolution displays). So, as we can see, the standalone selective frame-buffer update scheme is able to report significant power savings (more than 37%) in the two GUI-based benchmarks (the two benchmarks in the right part of the graph), while its effectiveness in the dynamic OpenGL games is rather limited. More specifically, the power savings of the selective update scheme are 8.09% in Doom_3, 3.27% in Quake_4, and 5.16% in Prey_Guru_4). Note that the selective update scheme is only employed during the frame-buffer write transactions, while the proposed compression schemes are able to optimize both the read and the write transactions.

As also shown in Figure 1, the proposed lossless compression scheme combined with the selective update scheme manages to reduce the power spent in the frame-buffer activities by 11.45 mW in Doom_3 (out of 17.69 mW), 8.33 mW in Quake_4, 9.68 mW in Prey_Guru_4, 17.07 mW in Affine, and 15.39 mW in Animated_Tiles. Furthermore, if the compression process is configured to be lossy, then additional power benefits are achieved. In total, the averaged power savings over all the selected benchmarks are 11.7% for the standalone selective update scheme and 67.5% when the lossless compression scheme works on top of the selective update scheme. Moreover, the usage of the proposed lossy compression scheme combined with the selective update scheme manages to further increase the power savings to 72.4%.

Finally, we want also to mention that in order to study, understand, and optimize the operation of the proposed schemes, we created a profiling tool to analyze the working (run-time) behavior of the selected benchmarks. A demo showing the effectiveness of the proposed techniques through appropriate animations has already developed and this demo is expected to be presented in the second year review meeting if there is an available time slot.

3.3 Value Cache

As noted in D5.5.2 [8] for the evaluation of the Value Cache (VC) idea, we selected the Attila simulator [1]. Attila is a detailed, cycle-accurate, architectural level simulator and it is capable to work at the OpenGL level. Attila can be considered as a realistic simulation environment, since it contains all the required graphics components of a real graphic processing system. There is no other available simulator in the area that can execute OpenGL code (e.g., gpgpusim, Multi2sim, or Macsim).

Attila is a highly parameterized simulator in the sense that all the logic, processing, and memory components of a typical GPU can be configured prior to execution. The micro-architecture of the simulator is versatile and highly configurable and can be used to evaluate multiple configurations, such as high-end desktop GPUs or embedded GPUs for mobile systems without being an exact replica of any particular product. The instruction set architecture (ISA) can be considered as a typical GPU ISA. In addition, Attila offers the facilities to add new instructions in the simulated micro-architecture. More details about the Attila framework can be found in [8] or in [1].

3.3.1 The Simulated Fragment Processor

Attila's 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. As already mentioned, in the context of this work, we selected to apply the VC idea in the fragment shading operations of the simulated GPU. In addition, the simulated GPU resembles a typical low-end and low-power GPU.

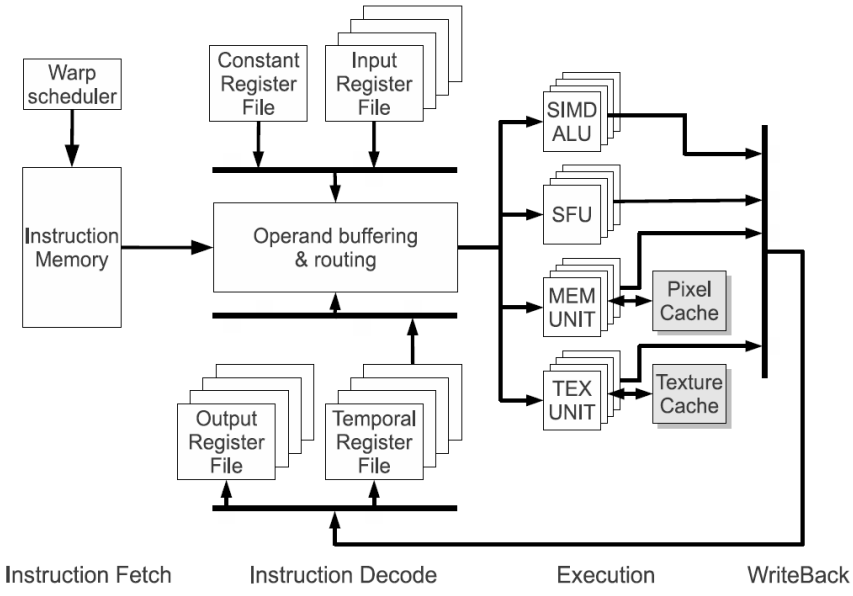


Figure 2: The simulated fragment processor.

Figure 2 illustrates the block diagram of the simulated fragment shading processor. During execution the shading processor operates on a fixed set of inputs and produces a single set of outputs indented to be used by the next stage of the graphics processing pipeline (not shown in Figure 2). In addition, during execution the fragment shading program has access to a small set

of constant parameters typically located in on-chip scratch registers (constant register file), to a another small set of on-chip scratch registers indented to keep the intermediate results (temporal register file), and to a larger set of (typically off-chip) texture maps (texture cache).

The fragment processor consists of a fairly simple in-order four stage pipeline. Typically, a form of SMT is employed where the threads are organized in groups named warps. All threads in a warp are executed in lockstep mode, that is the same instruction is executed by all the threads but each thread operates on a different input fragment. After instructions are fetched, they are decoded and their operands are fetched. Depending on the type of the operand, one of the three different register files are probed, depending on the type of value that needs to be read (i.e., constant, input or temporal register file).

Once all the source operands for the threads in a warp are fetched the instruction is dispatched to the corresponding functional unit. Operand buffering is required since the number of functional units can be smaller than the number of threads in a warp; in this case the dispatch to the functional units takes several cycles. Four types of functional units are included in each fragment processors, namely the SIMD ALU (e.g., vector additions), the Special Functions Unit (e.g., reciprocal operations), the Memory Unit (e.g., load/stores to the color buffer), and the Texture Unit (i.e., compute the color of a texture).

Programmable stages			
8 shaders (unified architecture)			
Non-programmable stages			
Primitive assembly:	1 triangle/cycle	Rasterizer:	4 fragments/cycle
Early X test:	8 in-flight fragments		
Global parameters			
Frequency:	500MHz	Screen Resolution:	640x480
Main Memory			
Latency:	100 cycles	Bandwidth:	4 bytes/cycle (dual channel)
Shader Processor			
Multithreading:	2 warps, 4 threads/warp	Register Size:	16 bytes (4-wide vectors)
Constant Reg. File:	96 registers	Input Reg. File:	64 regs/warp
Output Reg. File:	32 regs/warp	Temporal Reg. File:	48 regs/warp
Functional Units:	4 SIMD ALUs, 4 SFUs	4-stage pipeline:	IF, ID, Exec, WB
Caches			
Vertex Cache:	64b/line, 4-way, 8 KB, 4 in-flight requests, 3 cycles	Pixel and Texture Caches:	64b/line, 2-way, 2 KB, 4 in-flight requests, 2 cycles

Figure 3: GPU simulator parameters.

In the last pipeline stage the results of the functional units are stored in the temporal or in the output register file. There is no forwarding mechanism, so two dependent instructions cannot be executed back-to-back. However, since the warp scheduler uses a Round Robin policy, consecutive instructions usually pertain to different warps. For the sake of completeness, the main parameters of the simulated graphics processing unit are depicted in Figure 3.

3.3.2 Benchmark Suite

The benchmark suite for exploring the concept of redundancy and accuracy in the context of the VC idea consists of four modern OpenGL-based games. The selected games are the following: Doom 3, Quake 4, Unreal Tournament 2004, and Prey Guru 4. More details about those games can be found in [8]. However, we want to mention that in order to limit the simulation times, we randomly select seven frames from the said games (the same frame numbers are selected for all games). The selected frames are: frame 10, 50, 100, 150, 200, 250, and 300.

3.3.3 A Realization of the Value Cache Idea

The concept of redundancy or value reuse has been widely explored in the past by many researchers. However, all the previous approaches tried to employ value reuse techniques at the function level and their target was to increase the performance of the executed applications. In the LPGPU project, our aim is to apply value reuse techniques in order to eliminate the execution of certain (power greedy) instructions or groups of instructions. The application domain of the LPGPU project is a perfect fit towards this direction, since a typical fragment or vertex shading program consists of complex, vector-like and power greedy instructions, like, vector-MAD, reciprocal, square-root and logarithmic calculations.

As mentioned, we selected to apply the value cache idea in the fragment shading operations (called fragments) of the graphics processing pipeline. Fragment shaders typically consist of complex arithmetic operations that may incorporate the geometric and appearance descriptions of the rendered objects and the environment. The target of the fragments is to compute the final color value of a pixel. As we have already shown in [7] and [8], a subset of the fragment shading operations can be performed under small error budgets e.g., by dynamically lowering the precision of specific calculations. Since the final color values generated by the fragment shaders will be interpreted by the human senses, which are not perfect, it is possible to introduce small and controllable errors during the fragment shading operations, if such approach will result in power savings.

The Attila simulator proved very helpful in understanding and estimating the potential of the value reuse approach, especially when it is combined with the concept of accuracy. Since we select to apply the concept of value reuse at the instruction level, as a first step we used Attila to gather information about the most frequently executed instructions in the fragment codes of our benchmark suite. The detailed results of this analysis are presented in the following subsections, however we would like to point out that (at least for the studied instructions) our experimental results revealed that the value reuse property remains highly stable and predictable across a certain number of instructions and most importantly the value reuse property is a function of the executed fragments. In other words, certain fragments exhibit high value reuses, while some other fragments shows medium or even no value reuses, but this behavior remains constant or almost constant during the execution of the same fragments across the frames. Understanding and analyzing this behavior is an important issue and it will be part of our future work.

Based on the above observations, our proposal to fully exploit the concepts of redundancy and accuracy is to insert a specialized functional unit in the data path of the GPU exactly for this purpose. The new special functional unit will operate as a normal functional unit (e.g., like the SFU) and it will be managed by new machine level instructions that will be visible to the system

compiler/assembler. In other words, our proposal is to extend the instruction set architecture (ISA) as well as the data path of the GPU with special decorations to explicitly exploit the property of value reuse (redundancy and accuracy). A high-level diagram of the proposed architecture can be seen in Figure 4.

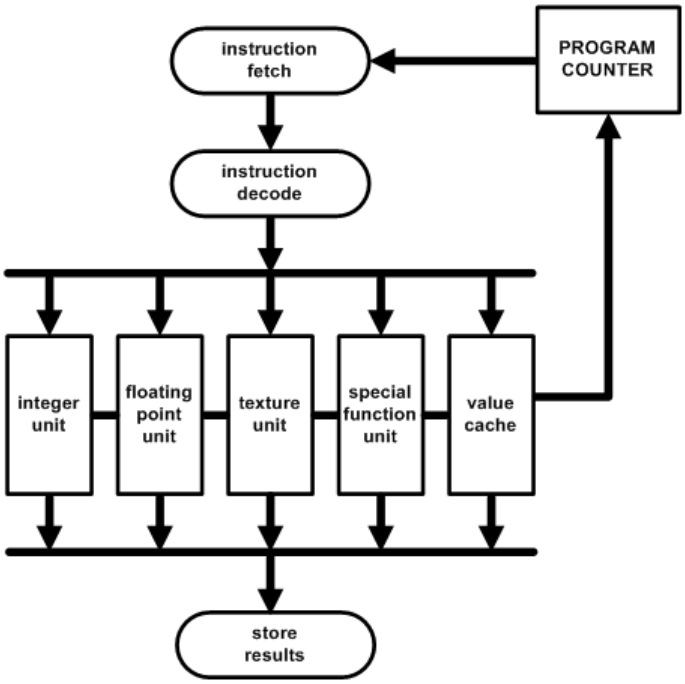


Figure 4: A high-level diagram of a GPU including the proposed value cache functional unit.

As Figure 4 indicates, the data path of the GPU includes a new functional unit called value cache functional unit (VCFU). VCFU consists of a dedicated storage area indented to cache or hold the results of previous computations (namely the value cache) and the associated logic. In general, VCFU operates as a typical functional unit, since it is managed by specific machine-level instructions (part of the GPU ISA) and it has access to the various register files of the fragment processor. More specifically, VCFU is fed by input data located either in the input, constant, or temporary register file; it performs the corresponding operations and register reads as they are uniquely dictated by the corresponding machine-level instructions; it stores the results to the appropriate registers in the output or temporary register file, again as it is defined by the corresponding machine-level instructions.

A unique characteristic of the proposed functional unit is that VCFU can perform targeted branches in the fragment shader code (this functionality is similar to a typical branch predictor in conventional CPUs). Conceptually, VCFU functions as a lookup table which internally maps between a set of output results (e.g., the data itself) and a set of input parameters (e.g., an address or an identifier(s) of the stored output data). Once the data is stored in the value cache, it may be accessed and retrieved while the step-by-step calculation from the initial source input parameters is bypassed (namely a number of instructions may be eliminated; not executed). Thereby, if a value cache match (hit) occurs, the ordering of the to-be-executed instructions of the fragment shader program must be also modified.

In this case, VCFU is responsible to inform the processor PC that a dynamic branching in the fragment shader code must be immediately performed. In other words, the PC of the fragment

processor is required to be notified about the number of eliminated instructions and accordingly perform the above dynamic branching (see Figure 4). The detailed description of this dynamic branching process and the overall operation of the value cache FU is a subject under investigation and it is left for future work.

As mentioned, VCFU will be managed by specialized machine-level instructions (part of the GPU ISA). In general, the primary operations performed by VCFU are the AddEntries and the LookupEntries operation. AddEntries places new results in the value cache and LookupEntries retrieves one or more entries from the value cache, in case of a value cache hit, or produces misses if there is no corresponding entry or entries for the sought input parameters. The exact structure of those instructions is again a subject under investigation.

In order to understand and define the operation and the structure/logic of the proposed value cache functional unit, we rely on the Attila simulator. As a first step, we analyze the instruction mixes of the fragment shading programs of the selected OpenGL-based games. Our next step was to apply the concept of redundancy and accuracy in the vector MUL (multiply) instruction. The following two subsections contain the experimental findings of our analysis.

3.3.4 Fragment Shaders and Instruction Statistics

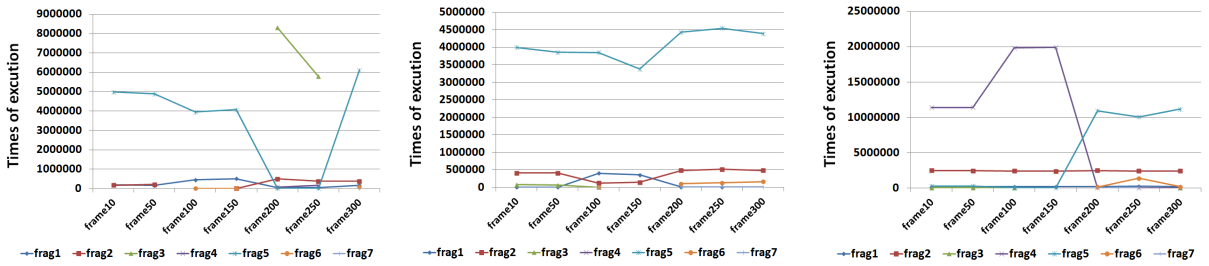
This section presents our profiling results. Our target is to understand the characteristics of the fragments (i.e., their instruction mixes and number of instructions) and many times each fragment is executed across the various frames of the evaluated games.

	Number of fragments	Instructions/fragment (min)	Instructions/fragment (max)	Instructions/fragment (avg)
Quake_4	7	3	27	12
Doom_3	7	3	26	13
Prey	7	4	62	20
UT2004	39	5	32	13

Figure 5: Fragment characteristics.

Figure 5 depicts the gathered statistics. As we can see, three out of four games (Quake_4, Doom_3, and Prey) consist of a fairly small number of fragment programs, while in UT2004 the number of fragments rises to 39. The three rightmost columns in Figure 5 show also the instruction characteristics of the fragments. Before analyzing those characteristics, we have to mention that the Attila ISA has 52 vector and scalar instructions (including typical SFU instructions such as reciprocals, logarithmic and exponential calculations, etc). The Attila ISA can be considered as a representative ISA of a state-of-the-art mobile GPU (ThinkSilicon examined all available ISAs of the mobile GPUs and found that actually small differences exist among the various available products). For more information about the Attila ISA, we refer the reader to [1].

So, as Figure 5 indicates, the complexity of the fragments varies in each benchmark, but almost the same fragment characteristics appear among all the evaluated games. More specifically, there are simple fragments consisting of three to five instructions (e.g., such fragments may perform simple color interpolations consisting of one or two multiplications), while more complex and heavy fragment exists (e.g., such fragments may compute complex equations incorporating geometric and appearance descriptions of the rendered objects).



(a) Quake 4 (b) Doom 3 (c) Prey

Figure 6: Frequency of execution for each fragment per frame.

Figure 6 illustrates the execution frequency of each fragment per frame. The horizontal axis in all graphs of Figure 6 depicts the frame number, while the vertical axis shows the number of times (absolute number) the corresponding fragment is executed. Each lines in the graphs depicts the per-frame execution frequency of the corresponding fragment (for clarity reasons, Figure 6 does not contain the statistics of UT2004 due to its large number of fragments).

We have to mention at this point that the reason for analyzing the behavior of the individual fragments is that in the course of this work we realize (as it will become clear in the next subsection) that the effectiveness of the value cache approach heavily depends on the executed fragments. As can be seen from Figure 4, there are fragments that are constantly executed in each frame (e.g., fragment 1 in Quake_4, fragment 2 in Prey, and fragment 5 in Doom_3), while some other fragments appear for a small amount of frames (e.g., fragment 3 in Quake_4). Note that the evaluated frames in all games are not consecutive. Furthermore, there are fragments that are heavily executed for a number of frames, remain inactive or almost inactive for some frames, and heavily executed again (e.g., fragment 5 in Quake4 and fragment 2 in Doom_3).

The underlying meaning of the statistics presented in Figure 6 is that if the effectiveness of the value cache approach is a function of the executed fragment (as we show in this work), then it is possible to drive the value cache mechanism based on the previous executions of a specific fragment. For example, a possible approach may be to gather the necessary information during the execution of a specific fragment in a specific frame and accordingly activate/deactivate the value cache mechanism during the execution of the same fragment in subsequent (future) frames. More details about this behavior will be presented in the next subsection.

Finally, Figure 7 illustrates the instruction statistics of the evaluated games across all the studied frames. The horizontal axis shows the corresponding instructions. There are only 11 instructions in total. Those instructions are the superset of all the instructions appearing during the execution of the studied fragments/frames/games. We have to mention at this point that the revealed superset of instructions (only 11 instructions in total!) is surprising. Defining an appropriate ISA for a GPU has been the subject of considerable debate. Many GPU companies have included in their ISA complex instructions (resulting in ISA consisting by more than 100 instructions), however our results indicate that only a limited number of instructions are actually frequently executed.

Nevertheless, from the instructions shown in the horizontal axis of Figure 6, the tex instruction is related to texture mapping operations, the mul, mad, dp3, and add instructions are vector instructions operating simultaneously in the four RGBA color planes of the input operands, while the remaining instructions are scalar instructions. The mov instruction is typically invoked to transfer

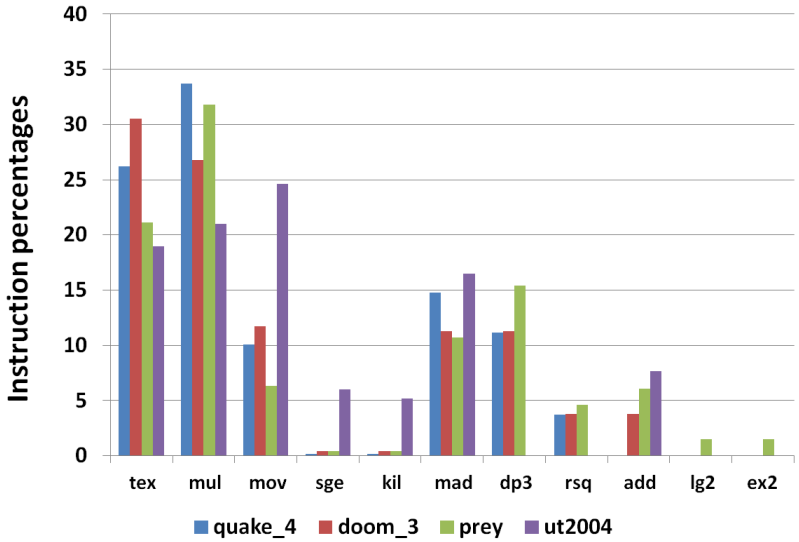


Figure 7: Fragment instruction characteristics.

data (usually RGBA color values) between the temporary to output register file of the fragment shading processor.

The vertical axis in Figure 7 shows the instructions percentages normalized to the total number of executing instructions appearing in each studied game, while the color of each bar in the graph corresponds to a different game. So as Figure 7 illustrates, the mul and tex instructions are the most frequently executed instructions. The popular mad (multiply-add) and dp3 (dot product) instructions corresponds only to 12-13% of the execution. Finally, one other surprising result is that the rsq (reciprocate square root), lg2 (logarithm of 2), and ex2 (exponential) corresponds to only 5% of the execution, or even they do not appear at all in the fragment execution (e.g., the lg2 and ex3 instruction in Quake_4, Doom_3, and UT2004).

Based on the above observations, in the context of this report, we select to study the VC concept in the vector MUL instruction (4 x 32-bit floating point multiplication are performed during the execution of the specific instruction). The next subsection contains our evaluation results, while similar studies for the remaining instructions are left for future work.

3.3.5 Redundancy and Accuracy Statistics

This section presents the evaluation result of the redundant operations in the vector MUL instruction. Similar to [7], we use various value cache sizes, but in contrast to [7], we present our statistics in a per-fragment basis. In case that one fragment contains more than one MUL instruction, then the reported statistics correspond to the averaged results of all the MUL instructions within the fragment. The evaluated value cache sizes are three: 4-entry VC (red part of the bars), 12-entry VC (blue part of the bars), and 20-entry VC (green part of the bars). There are four graphs in Figure 8; one per studied benchmark. In the case of UT2004 (bottom graph), for clarity reasons, we selected to present the fragments that correspond to 86% of the execution.

There are seven groups of bars in each graph and each group is associated to a specific frame

shown in the bottom of the graphs. In addition, each group of bars contains the simulation statistics in a per-fragment basis when only the concept of redundancy is exploited (the corresponding fragments are below the horizontal axis). Finally, each bar depicts the percentages of the eliminated vector MUL instructions in each fragment (number of value cache hit normalized to the total number of MULs in each fragment). In all cases, an LRU replacement policy is assumed in the value cache (our results using a LFU or a FIFO replacement policy are fairly similar). For more information about the value cache mechanism, we refer the reader to [7].

As it can be seen from Figure 8, the number of redundant MUL calculations (coverage) that can be captured by the value cache mechanism is significant in most cases. As expected, the efficiency of the proposed redundancy elimination mechanism depends on the studied benchmark, but most importantly, it also depends on the evaluated fragment. This is apparent in all benchmarks. For example, in the Quake_4/frame 10, there are three executed fragments and the two of them exhibit coverage of almost 100% (fragment 0 and fragment 1), while the third fragment (fragment 4) shows a coverage equal to 12% (which is still significant). This eventually means, that (almost) 100% of the executed MUL instructions of the first two fragments and 12% of the third fragment can be eliminated (not executed) achieving proportional timing and power benefits. The same behavior can be seen in the remaining frames of Quake_4.

Moreover, an important observation is that the reported coverage of each fragment remains fairly stable across the frames. In the case of Quake_4, the minimum and the maximum coverage of the fragments across all the studied frames are as follows: 99.98%-99.99% (fragment 0), 92.11%-99.99% (fragment 1), 8.5%-9.6% (fragment 2), 38.08%-38.36% (fragment 3), 7.97%-32.04% (fragment 4), 31.59%-37.29% (fragment 5). This stable behavior of the fragments (a clear exception is fragment 4) across the frames as well as the fact that specific fragments exhibit specific behavior enables us to reconsider the operation of the value cache mechanism. For example, the proposed mechanism can be applied selectively (not in all fragments in a frame) and this selection may be based on the previous executions of the studied frames. However, more effort is required to further understand and analyze this issue.

Consider the remaining benchmarks, the same trends can be observed in Doom_3 and Prey. In contrast, in the UT2004 benchmark, all fragments in a frame show the same coverage. But even in this case, the behavior of the fragments remains stable between the frames (except fragment 19 in frame 300). On average and for the 4-entries VC, across all the fragments/frames the coverage of the eliminated vector MUL instructions is: 36.79% in Quake_4, 33.69% in Doom_3, 34.96% in Prey, and 24.9% in UT2004.

Finally, as we can also see in Figure 8, increasing the size of the value cache entries by more than 12 entries (blue part of bars) barely produces better results which attest to the fact that a frugal value cache memory array is required. Actually, even the 12-entry configuration improves the results only in a small number of fragment/frames compared to the 4-entries case. In the rest of this report, we will set the value cache size equal to 12-entries.

Having illustrated the potential of the value cache mechanism in eliminating the redundant MUL instructions, our next step is to examine techniques to further optimize the VC performance. More specifically, we examine two techniques. The first technique is to appropriately transform the input arguments (operands) of the instruction under investigation, while the second technique is to examine the effect of ignoring a few low order bits in the input operands of the MUL calculations. Our gathered results are depicted in Figure 9 and Figure 10 respectively.

According to the first technique, our target is to rely on the arithmetic properties of certain calculations. In the case of the MUL instruction, one possible technique is to rely on the interchangeability property, i.e., the input operands of a MUL instruction may be interchanged without affecting the output result. Figure 9 quantifies the effectiveness of this approach.

The red bars in all graphs in Figure 9 show the coverage of the eliminated MUL instructions for a 12-entry VC, when only the concept of redundancy is applied. The blue parts of the bars correspond to the additional benefits when the concept of redundancy is combined with the interchangeability property. Similar to Figure 8, we select to show our gathered statistics in a per-fragment basis. As Figure 9 indicates, enhancing the VC mechanism to take into account the interchangeability property manages to offer meager benefits compared to the vanilla VC mechanism. More specifically, there are only three cases in which an increase in the resulting coverage is reported (4.44% in Quake_4/frame 300/fragment 6, 2.86% in UT2004/frame 250/fragment 19, and 2.02% in UT2004/frame 10/fragment 3).

In contrast to the previous technique, exploiting the concept of accuracy manages to significantly improve the efficiency of the VC mechanism. In the context of this work and in order to not pertain to a specific depth (number of bits) of the color planes, the concept of accuracy is implemented by comparing the relative values of the new input MUL operands to the values already stored in the VC memory array. For example, if the new input MUL operands and the values stored in the VC differ by only a small error threshold (e.g., 2%), then a VC hit is assumed and the output value stored in the VC is immediately retrieved and forwarded to the corresponding output (destination) register of the evaluated MUL instruction.

As also shown in Figure 10, we use two error thresholds: 2% and 4%. Similar to Figure 8, the graphs shown in Figure 10 depict the coverage of the eliminated MUL instructions for the four studied benchmarks. The red parts of the graphs correspond to the reported coverage numbers when the vanilla VC mechanism is used. The blue (green) part of the bars show the additional benefits (increase in coverage) when an error threshold of 2% (4%) is allowed during the VC matching operations. Finally, with respect to the results presented in Figure 8, the graphs of Figure 10 illustrate the gathered statistics in a per-fragment basis.

As Figure 10 illustrates introducing small errors in the VC operation manages to double or even triple the reported statistics in the majority of the fragment programs. Consider the Quake_4 benchmark, on average a 27.63% (for the 2% error threshold) and a 32.06% (for the 4% error threshold) increase in coverage in fragment 4 is achieved compared to the vanilla VC configuration. Most importantly, this increase is almost uniform among the various frames verifying our assumption that the behavior of certain fragments is stable (thereby predictable) across the frames. Similar results can be observed in the majority of the fragments in all studied games. Taking into account all the evaluated fragments/frames in Quake_4, allowing 2% (4%) errors in the VC operation result in an increase in coverage by 27.57% (30.86%) compared to the baseline VC implementation. The results for the other three benchmarks are as follows. In Doom_3, the 2% (4%) error threshold case meliorates the coverage by 27.69% (32.4%), while the corresponding improvements in Prey and in UT2004 are 28.88% (32.14%) and 23.52% (32.11%) respectively.

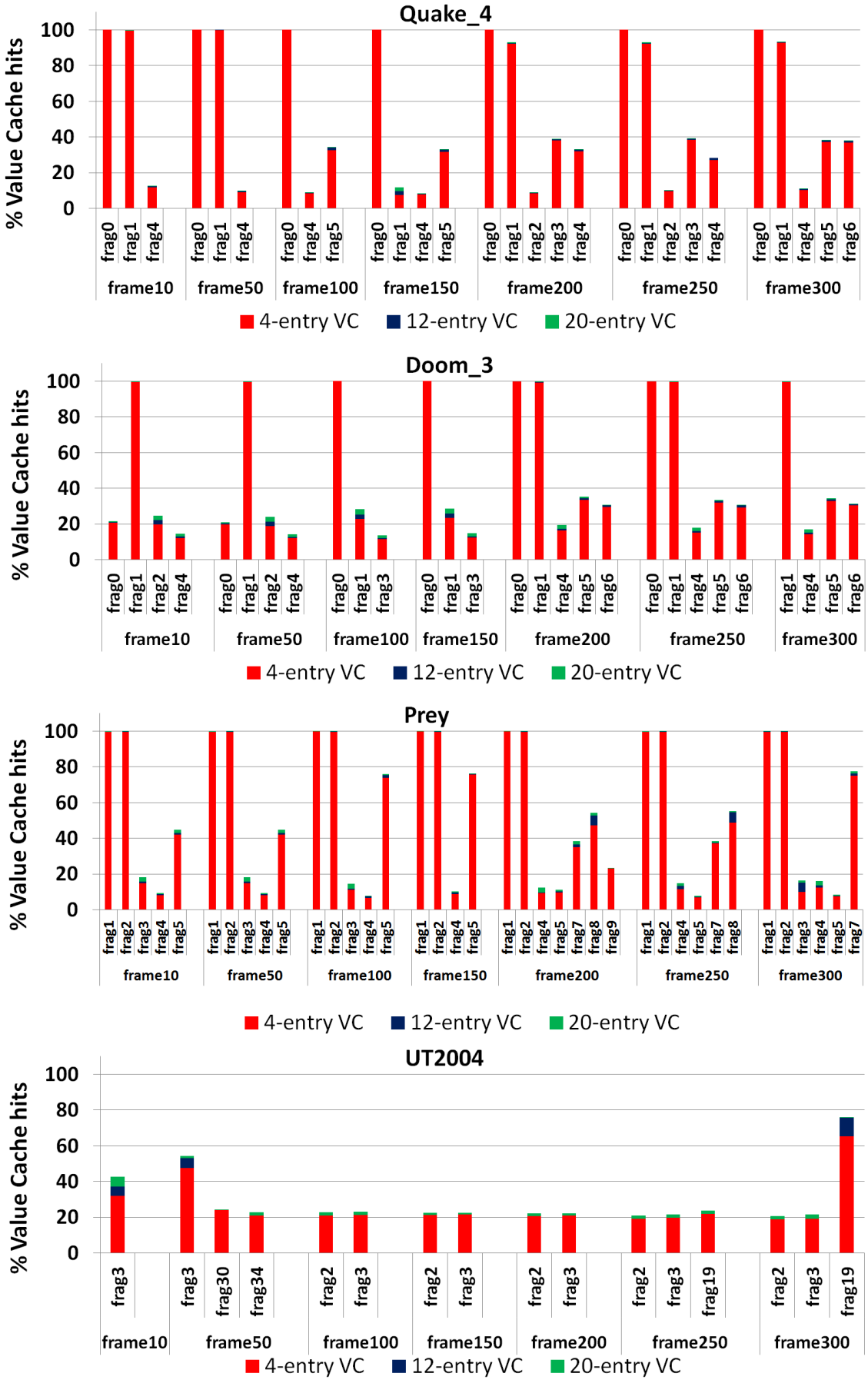


Figure 8: Coverage of MUL redundant operations for various value cache configurations.

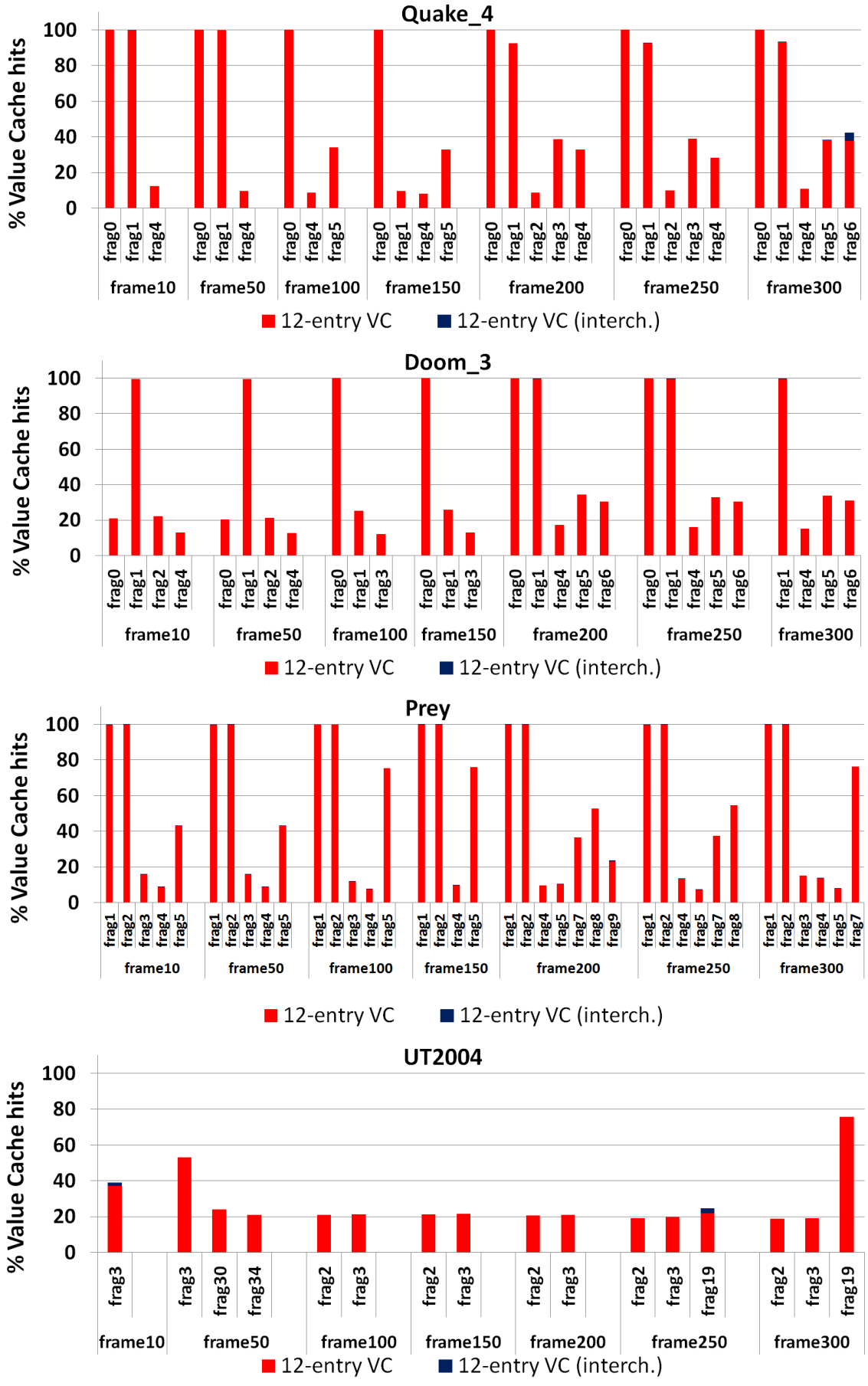


Figure 9: Coverage of MUL redundant operations exploiting the interchangeability property.

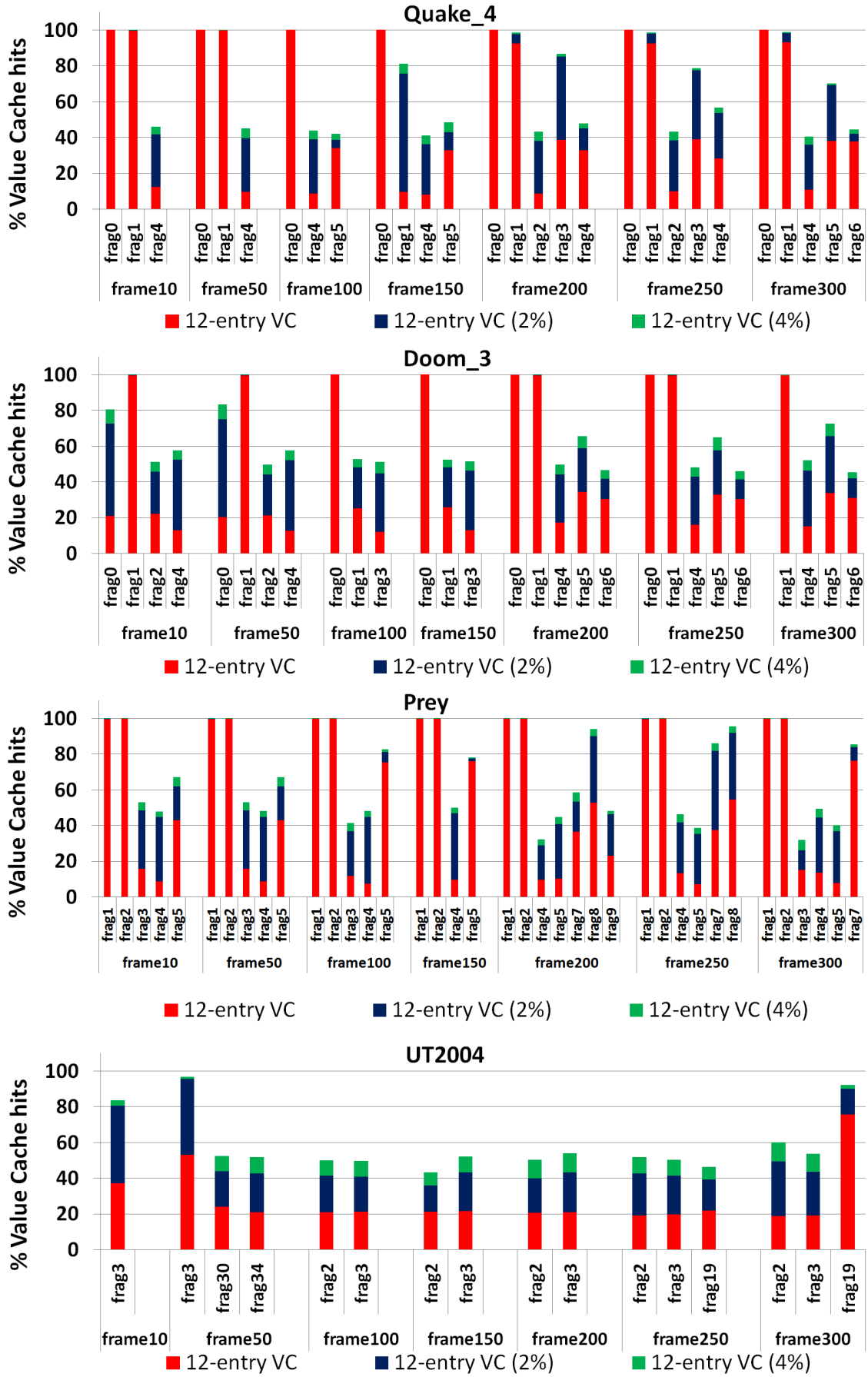


Figure 10: Coverage of the MUL redundant operations exploiting the concept of accuracy.

4

Status

For the redundancy task (T5.2) the following criteria were defined to evaluate the results:

- **Minimum results:** Estimate the potential redundancy of computation (for consecutive frames) at various levels.
- **Expected results:** Develop mechanisms that can eliminate redundant work at multiple levels.
- **Exceeding expectations:** Work-reuse mechanisms can exploit a significant part ($> 90\%$) of the potential redundancy at all levels.

In addition, as mentioned in DoW (Task 5.2), the various levels that initially planned to exploit redundancy were the following: (1) OpenGL level, (2) CUDA level, (3) thread level, (4) procedure/-function level and (5) hardware (floating point operation) level

The work performed in Task 5.2 concentrate on OpenGL and hardware level. The value cache technique is currently viewed as an extra functional unit intended to be part of the data/processing path of a GPU, while the set of frame-buffer traffic elimination techniques try to exploit redundancy and accuracy in tile-based graphics systems (typically programmed in OpenGL language) at the hardware level.

Consider the value cache technique, our results indicated that there is a significant correlation between the redundancy approach and the executing fragment shader programs and this correlation remains fairly stable (thereby predictable) between consecutive frames. However, only the redundancy in the multiply instructions has been evaluated. The fragment shaders can be considered either as threads or procedures/functions invoked for each pixel or for a group of pixels in the rendered frames. As a result, we believe that the present deliverable contains a work reuse mechanism that it is able to exploit the potential redundancy in at least these three levels:

- OpenGL level
- thread or procedure/function level
- hardware level

while the CUDA level redundancy is exploited (overlapping work) in D.5.5.3 and we therefore evaluate the overall outcome of the task as meeting **expected results**.

5

Advances over State of the Art

Consider the frame-buffer traffic elimination techniques, proposed as part of the Task 5.2 and Task 5.4, our IPR investigation revealed that there are two recent patents released by ARM, namely Method and Apparatus for Controlling Display Operations, US2011/007480 and Graphics Processing Systems, US2011/0102446, which discloses a set of schemes to eliminate redundant frame-buffer accesses. We are not aware of any other frame-buffer traffic reduction scheme that exploits the concept of redundancy. Those two patents do explore the idea of redundancy in frame-buffer functionality, but not the idea of accuracy and frame-buffer compression. The results illustrated in this deliverable, as well as in D5.2.2, showed that the proposed redundancy prediction scheme and the proposed compression scheme can be effectively combined (sharing the same hardware memory structures and the majority of the control logic) significantly increasing the resulting benefits. In terms of power savings, the combined approach (redundancy elimination scheme plus lossy compression scheme) was able to reduce the power consumed by the frame-buffer by 72.4% while the standalone redundancy elimination technique offered only 11.7% power savings.

Our investigation during the value cache development showed that there are three recent patent applications. Those are Input vector analysis for memorization estimation, Mar. 21, 2013, US2013 / 0073837, Selecting functions for memorization analysis, Mar. 21, 2013, US2013 / 0074057, and Method of operating a computing device to perform memorization, Dec. 8, 2011, US2011 / 0302371. Those patents exploit value reuse techniques at the boundaries of the application source code functions and they try to optimize only the performance paybacks of value memorization. In contrast to those patents, the value cache approach developed in the LPGPU project proposes to extend the data path of a GPU with an extra functional unit dedicated to perform value reuses. This value cache functional unit will be explicitly managed by special machine-level instructions (part of the GPU ISA). In addition, none of the above patents try to improve the power consumption of the target system (they try to offer only performance improvements). Furthermore, the said patents are too generic targeting to apply value memorization techniques in general purpose systems (e.g., server-like or PC- like applications or systems), while the proposed value cache approach is optimized for three-dimension graphics applications in which there are relatively small rendering code segments consisting of complex arithmetic operations (e.g., vector-like instructions).

6

Conclusions

As part of the 2nd year work we implement a frame-buffer traffic reduction scheme that significantly reduces read / write activity, which in term improves power efficiency. Briefly, we achieve power savings of:

1. 11.7% for the standalone selective update
2. 67.5% for the lossless compression scheme (additionally to [1]) or
3. 72.4% for the lossy compression scheme (additionally to [1]).

Additionally to that we propose value cache, a novel mechanism for GPUs that works as a lookup table that can bypass the step-by-step calculation required by specific instructions or group of instructions. This mechanism eliminates up to 70% of the total multiply instructions for the examined workloads with an average error threshold of 2% 4%. The impact of this is expected to be significant both in performance and power efficiency but time limitations did not allow us to include these results in the current report. In overall all types of redundancy have been studied and eliminated. CUDA redundancy has been studied by TU Berlin and is available in D.5.5.3. Finally reconsidering the results of our previous exploration on Slack we append the following Appendix to discuss new ideas that could potentially eliminate system Slack and further improve performance and power efficiency system-wide.

7

APPENDIX I.

7.1 Amendment to D5.5.2: New Developments on "Slack"

This section discusses new ideas to improve the overall system power consumption and performance by eliminating system-slack on a massively heterogeneous system. We have already observed that in a GPGPU system we do not have significant room to improve memory-slack due to the simultaneous multi-threading engine that allows the scheduling and execution of thousand outstanding threads at very low overhead. Although a reduced (device) memory bandwidth can create slack in the system this can be characterized more as a bad design. Our current work shows that even with the best configuration for the bandwidth and number of outstanding threads there can be a different type of slack in the system created by the non overlapping nature of the data transfers between main memory and device memory. This comes as a consequence of the programming models that try to coarsen the granularity of the transfers to reduce relative runtime overhead. These programming techniques use a single data transfer from host to device (HtoD) for all data in the beginning of the program, and either occasionally (during the execution) or once at the end of the program they gather the results with a device to host (DtoH) transfer. Using *Full/Empty (F/E)-Bits* with guided DVFS allow us to overlap data transfers with computation using a hardware controlled mechanism and adjust the frequency of each device. This can significantly reduce the power consumption of each peer (CPU/GPU) according to the workload and the capabilities of the other peer. F/E-Bits as a mechanism allow us to bypass a barrier and start the computation overlapping the independent data transfer with the executed kernel which is promising for a performance improvement of up to 2x, while adding DVFS to each device separately can further improve power efficiency by applying speed matching.

7.2 Description of Task

This task involves the following work:

- Modeling and estimation of slack. Create models based on the memory access behavior, exploit memory slack, find load imbalances in GPUs and eliminate the stalls they create. These models will also be used to predict performance, energy, and optimal voltage frequency settings (DVFS) to maximize power efficiency in WP5.
- Design and development of techniques to exploit load balancing slack: Develop dynamic (run-time) and static (profile-based) techniques necessary to apply DVFS or power down idle cores in situations involving an imbalanced load.
- Design and development of techniques to exploit memory access slack: Develop run-time mechanisms for GPU DVFS to determine optimal power-efficiency operational points according to memory behavior.

7.3 Understanding Memory Slack

Memory slack is defined as the time that the execution units are stalled due to off-chip misses. During this stall time the execution units keeps consuming power executing NOP instructions which is traditionally called bubbles 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) will increase execution. Because it overlaps with the stall time the performance is not damaged and energy benefits occur from the reduced frequency. This case is shown in Figure 11 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 trade-off in performance. This is because we cannot have instant DVFS and therefore we need to adjust a global frequency for a long instruction interval.

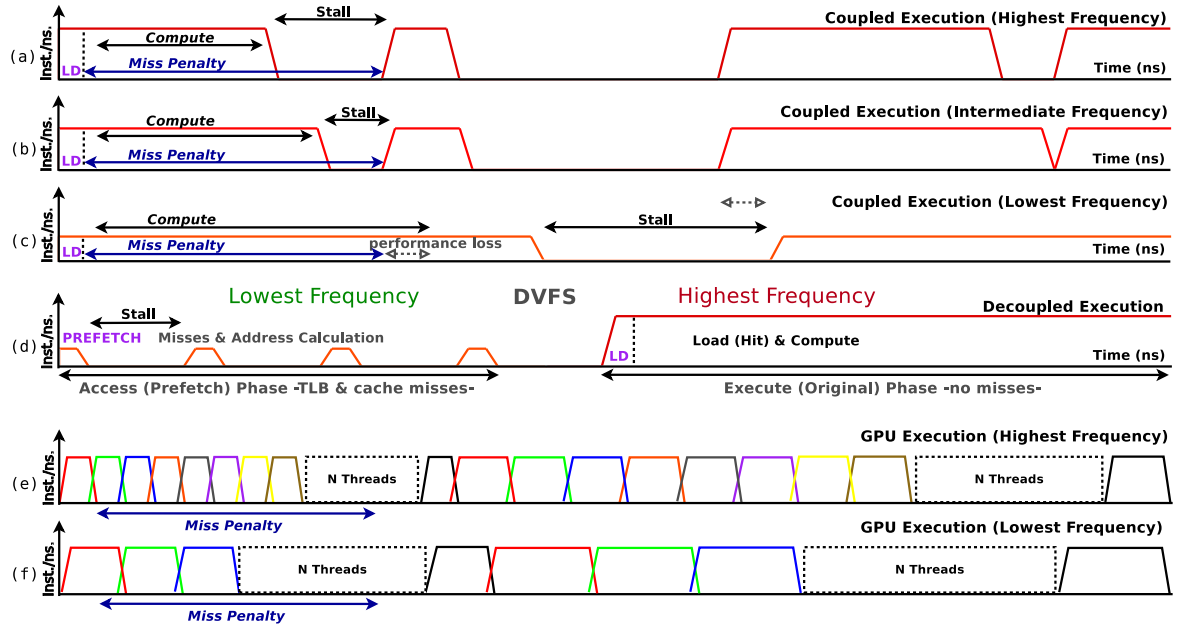


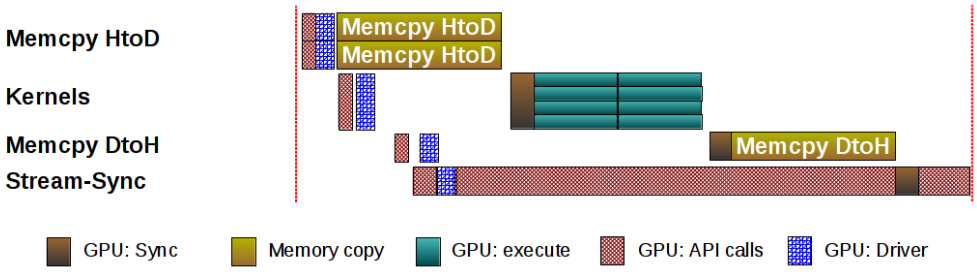
Figure 11: Example of coupled execution under different core frequencies (a,b and c), decoupled execution (d) and GPU executions (e and f). Figure from [9].

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 the program demands to DVFS granularity by splitting the program into 2 phases (access/execute) and DVFS each phase separately. In the access phase we prefetch the data to warm up the cache and therefore a low frequency is the best fit. For the execute phase which is scheduled to run immediately after the access phase we adjust the frequency to the highest expecting that very few cache misses will occur in it. This role distinction between totally memory bound and compute bound phases, allows us to maintain high performance which is highly correlated to the execute phase and at the same time, provide significant energy savings by running the access phase at the lowest frequency. Diagrams (e) and (f) show GPU executions under different frequencies. Because GPUs feature a hardware implementation to context switch between threads with 1 cycle overhead and support thousands of outstanding threads, they can hide all memory slack despite the long stalls per thread.

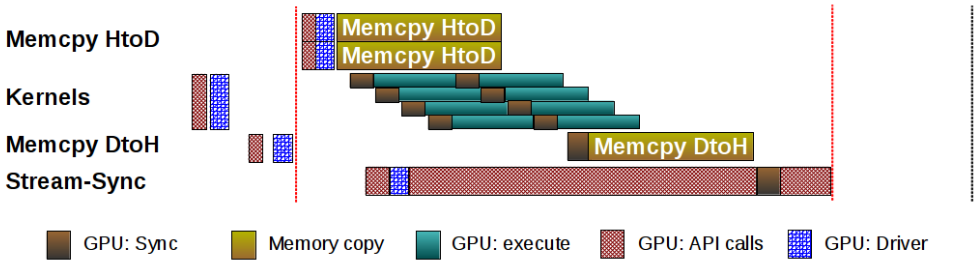
7.4 Understanding System Slack

System slack on the other hand refers to the slack that can be found in any component of the system including PCI-E links, I/O, memories, etc. Heterogeneous systems comprise of different components e.g., many-core CPUs, embedded GPUs, discrete GPUs, FPGAs, accelerators. Most of them coexist in a typical system demanding the transfer of the data from the main memory to the device memory. For the whole program lifetime many transfers from and to the device memory can occur whose performance is inelastic to the CPU frequency and the device frequency. The performance of these transfers is affected by the link speed and the memory capabilities of the host and device. Overlapping these transfers with computation gives us an opportunity for performance improvement in heavily streaming application with restricted data reuse. Applying DVFS to each of the devices allow us to maximize the energy savings without damaging performance.

Lustig et.al. [10] proposed Full/Empty bits as a method to exploit PCI-E link slack in discrete GPUs and improve performance by overlapping computation with data transfers. Their work proposes and evaluates hardware and software mechanisms to track data regions at very fine granularity and asynchronously start computation when the data required for some kernel has been transfered. This dependency tracking is fast and efficient and is able to mitigate most of the offloading overhead in state of the art GPUs. Figure 12 demonstrates the performance benefit of F/E bits for a simple streaming kernel. In Figure 12a execution has to wait for all data transfers to complete explicitly with a barrier even when the data for some kernel are available on the device. F/E bits as shown in Figure 12b can improve this situation by reserving a bit in the memory for every word (32bits) that will indicate if a kernel needs to block accessing this word (when the data transfer is not finished).



(a) Execution without full/empty bits support for a simple streaming kernel



(b) Execution with full/empty bits support for a simple streaming kernel

Figure 12: Full/Empty bits example to demonstrate the overlapping of data transfers to the device with computation. Figure similar to [10]

Figure 13 shows time distribution for a simple vector add. To further support and understand system slack we profile some of the CUDA kernels provided in NVIDIA COMPUTING SDK v.4.2. as shown in figure 14. This figure shows the theoretical system slack for different type of applications from totally memory-bound to totally compute-bound. F/E bits is promising to exploit an up to 2x performance improvement on applications that spent their time evenly on computation / data transfers if we can fully overlap these phases. In practice we expect to exploit only a part of the available system slack. For applications that are totally memory bound we hope to overlap all of the computation (small execution time) inside the data transfers while for compute bound applications we expect to overlap the (small) data transfers within the execution. F/E bits is not expected to be promising for performance to the extreme of any of these categories.

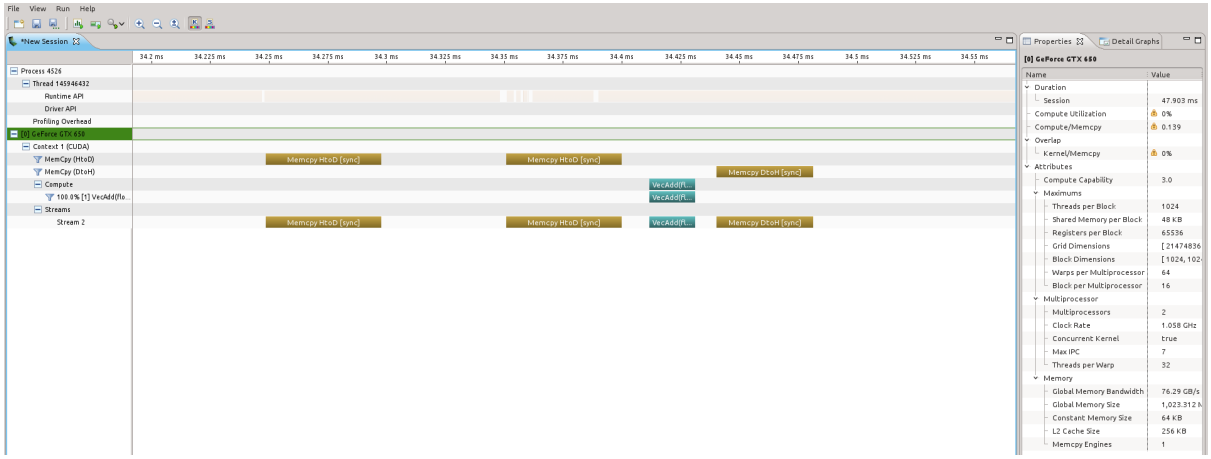


Figure 13: VectorAdd CUDA kernel from NVIDIA COMPUTING SDK v.4.2 executed on a GeForce GTX650

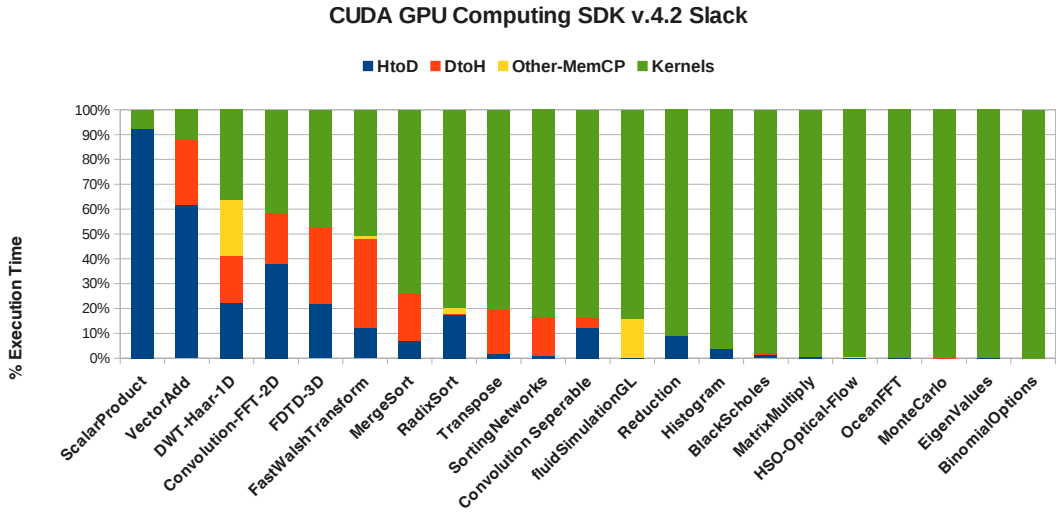


Figure 14: Various benchmarks from NVIDIA COMPUTING SDK v.4.2 executed on a GeForce GTX650

F/E bits is well known method to improve performance and therefore power efficiency of GPU

applications. The energy savings of this method is proportional to the speedup (Energy = Power * Time). As part of our work we are exploring DVFS in combination with F/E bits to optimize the overall *power* in a heterogeneous system: (1) in CPU (task producer), (2) in GPU (task consumer), and (3) in the interconnects.

7.5 Updated Task Proposal

Memory slack has been extensively explored in the original proposal of deliverable 5.5.2. Although our main conclusion was that there is not significant room for improvements in GPUs memory slack, we discovered system slack. System slack is a wider approach which also involves the slack that can be created in the communication of the main process with the device. This affects a series of architectural improvements that future GPUs will feature (e.g., unified address space). This idea comes further late in the project to be integrated in the hardware implementation. Despite that we consider this knowledge very valuable for future projects and we want to finish the theoretical study.

References

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

[2] 2013. http://www.arm.com/markets/mobile/computing.php.

[3] 2013. http://www.vivantecorp.com/p_mvr.html.

[4] 2013. http://www.semiconwest.org/sites/semiconwest.org/files/docs/Shekhar%20Borkar_Intel.pdf.

[5] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[6] S. Galal and M. Horowitz. Energy-efficient floating-point unit design. *Computers, IEEE Transactions on*, 60(7):913–922, 2011.

[7] 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/.

[8] Stefanos Kaxiras, Georgios Keramidas, Konstantinos Koukos, and Iakovos Stamoulis. D5.5.2. architectural techniques to exploit slack and accuracy trade-offs, 2013. http://www.lpgpu.org/wp/.

[9] 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.

[10] Daniel Lustig and Margaret Martonosi. Reducing GPU offload latency via fine-grained CPU-GPU synchronization. *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 354–365, February 2013.

[11] Kari Pulli, Tomi Aarnio, Kimmo Roimela, and Jani Vaarala. Designing graphics programming interfaces for mobile devices. *IEEE Comput. Graph. Appl.*, 25(6):66–75, November 2005.