A High-Precision, Hybrid GPU, CPU and RAM Power Model for Generic Multimedia Workloads

Kristoffer Robin Stokke, Håkon Kvale Stensland, Carsten Griwodz, Pål Halvorsen
Simula Research Laboratory & University of Oslo, Norway
{krisrst, haakonks, griff, paalh}@ifi.uio.no

ABSTRACT

Energy efficiency of multimedia processing is a hot topic in modern, mobile computing where the lifetime of battery-powered devices is low. Authors often use power models as tools to evaluate the energy-efficiency of multimedia workloads and processing schemes. A challenge with these models is that they are built without sufficiently deep hardware knowledge and as a result they have the potential to mispredict substantially depending on hardware configuration. Typical rate-based power models can for example mispredict up to 70 % on the Tegra K1 SoC. Inspired by multimedia workloads, we introduce a modelling methodology which can be used to build a generic, high-precision power model for the Tegra K1's GPU and memory. By considering hardware utilisation, rail voltages, leakage currents and clocks, the model achieves an average accuracy above 99 % over all operating frequencies, and has been rigorously tested on several multimedia workloads. Our method exposes detailed insight into hardware and how it consumes energy. This knowledge is not only useful for researchers to understand how power models should be built, but also helps to understand what developers can do to minimise power usage. For example, experiments show that for a DCT benchmark, 3 % power can be saved by utilising non-coherent caches and smaller datatypes.

CCS Concepts

•Computer systems organization → **Heterogeneous (hybrid) systems;** **•Human-centered computing** → *Mobile devices;* **•Applied computing** → *Electronics;*

Keywords

Multimedia, Tegra K1, CUDA, energy, performance, CPU-GPU frequency scaling

MMSys'16, May 10 - 13, 2016, Klagenfurt, Austria

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4297-1/16/05...$15.00

DOI: http://dx.doi.org/10.1145/2910017.2910591

1. INTRODUCTION

Energy consumption is an important topic in mobile computing. Today's battery-powered mobile devices such as smart phones, tablets, laptops or even drones have limited uptime due to a combination of small battery capacity and power-intensive hardware. As such, research in multimedia systems have in the past decade focused on trying to understand the power requirements of hardware to aid energy optimisation efforts. For example, Hosseini et al. [6] attempt to save energy in a game streaming scenario by selectively downloading and processing more important textures over less important ones, and Sharrab et al. [16] propose a power model for MPEG-4, MJPEG and H.264 and shows the effect in terms of power usage under various H.264 compression parameters.

Unfortunately, researchers' view of how modern mobile devices consume energy is limited. To evaluate energy usage of for example multimedia systems, they use power models which are too simple to justify the complex mechanisms that govern the power usage of hardware. Hosseini et al. [6] employ a smart phone power estimation tool called *PowerTutor* [21] to estimate the power of an HTC 3D evo phone under their game streaming scenario. However, inspecting the source code from PowerTutor we can see that the estimates are actually based on a power model from the HTC Dream. This leads to inaccurate estimates that give a false impression of how hardware consumes energy.

Powertutor [21] and other authors [6, 4, 20] implement *rate-based* power models, where for example power of a processor is assumed to grow linearly with the processor's utilisation level. Figure 1 shows prediction error for three types of power models running a motion estimation workload on the Tegra K1's GPU, where 1a is the commonly-used rate-based model. While the accuracy of this model type can be close to 100 %, the misprediction can also be substantial up to 70 % depending on the GPU and memory operating frequency. Thus the accuracy of such models is entirely dependent on the operating frequencies selected by the Dynamic Voltage and Frequency Scaling (DVFS) algorithms running on the platform at the time of the verification. The inaccuracy of rate-based models is not only limited to frequency settings. They also ignore many other mechanisms that have non-negligible effects on power usage such as core and rail power gating, frequency scaling and variations in rail voltage levels [15], variable cost of executing different instructions (see Figure 12), different software workloads and contention of hardware resources such as caches [1].

Some authors, such as Sharrab et al. [16] attempt to model more accurately the effects of power saving mechanisms such as frequency scaling. Their processor power model is based on the work by He et al. [22] and suggests that processor power is proportional to the cube of the number of cycles required to finish processing. However, this assumes that performance is inversely proportional to processor frequency. This has been shown to be an incorrect assumption for the Tegra K1, where performance scales sublinearly with frequency due to contention for resources [15]. The model will therefore mispredict on our platform.

In this paper, we propose a methodology to build high-precision, generic power models for the Tegra K1 SoC with special focus on its CUDA-capable GPU and main memory. We use several multimedia workloads intended for post-processing a live, raw video feed as a case study and demonstrate an average accuracy above 99 %. The model has been more rigorously tested than related work by running all experimental benchmarks over all possible GPU and memory frequencies. The accuracy is never below 96 %. Our method is based on estimating swithing capacitance of captureable hardware events, such as integer, floating point and conversion instructions, clock cycles and leakage currents by running a set of synthetic workloads stressing the various hardware components [1]. It also takes into account measured voltage levels on the rails supplying the SoC. Compared to related modelling efforts, the increased accuracy of our model does not compromise performance in power estimation. Our contributions are as follows:

- An accurate power model which gives good insight into power usage of a modern platform and can be used for evaluating energy efficiency of generic multimedia workloads.
- Our methodology shows other authors in the field how accurate power models should be built and validated for modern SoCs.
- By estimating the switching capacitance of various hardware events, our model shows how developers can be more energy-efficient by utilising local cache better or using smaller datatypes.
- We demonstrate a 3 % power saving for the DCT multimedia workload without compromising performance.

Our outline is as follows. Section 2 gives a short introduction to two common types of power models, and as a motivation compares the estimation accuracy of these and our model. Section 3 introduces our GPU multimedia workloads used to verify our model. In Section 4 and 5 we introduce the GPU, memory and CPU model predictor, and we derive the power model and the methodology to build it. The model is verified and discussed in Section 6, where we also discuss some preliminary energy-optimisation approaches. Finally, we conclude our work in Section 7.

2. BACKGROUND AND RELATED WORK

There is a plethora of work that attempts to model the power usage of various types of computing devices, such as smart phones, embedded development systems, laptops and stationary computers. These generally describe power or energy as linear systems where the cost of executing various types of instructions, accessing different cache hierarchies, disks or network interfaces is found using different methodologies. While some authors have used neural networks to estimate these costs [7, 10] the vast majority use multivariable, linear regression. The typical way of describing for example the power usage of GPUs [5, 9, 18] and CPUs [14, 19] is of the form:

$$P_{tot} = \beta_0 + \sum_{i=1}^{N_\rho} \rho_i \beta_i \qquad (1)$$

In Equation 1, β_0 is the power of idle components with constant power draw, ρ_i is a predictor with units of accesses per second, β_i is the cost in Watt-seconds per access and N_ρ is the number of predictors. Note that some works [5, 9] have slightly different interpretations of Equation 1. For example, β_i can be replaced with the maximum power of a hardware component $P_{i,max}$, and ρ_i can be replaced with the utilisation of that component ($\rho_i \in [0, 1]$). The general form of the expression remains the same and viable to solution with regression. During the past five years several researchers have extended these modelling efforts out of the laboratory, where such models are built on-line on smart phones using various types of power measurement sensors [4, 20, 21].

A liability with modelling power using Equation 1 is that it does not consider the physical rules that govern energy consumption. Modern SoCs such as the Tegra K1 (see Figure 6) are complex platforms featuring several *rails* powering the various components within the SoC. Figure 6 shows four of these: the GPU, memory and CPU cluster rails. Kim et al. [8] describes the rate of energy consumption for logic CMOS circuits as the sum of dynamic and static power. These equations can be readily applied to individual power rails [3, 15]. Static power is the product of the circuit's leakage current $I_{R,leak}$ and rail voltage V_R:

$$P_{R,stat} = I_{R,leak} V_R \qquad (2)$$

Dynamic power is caused by switching activity and is governed by both hardware and software. For example, as a processor executes instructions, loads data from cache, or the memory chip spends cycles serving memory read and write requests, dynamic power is being used:

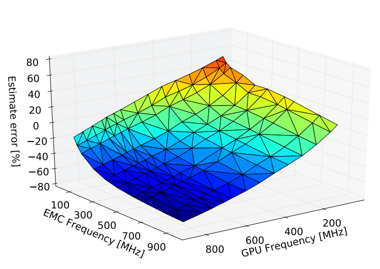
$$P_{R,dyn} = \alpha_R C_R V_R^2 f_R \qquad (3)$$

In Equation 3, C_R is a potential maximum switching capacitance per cycle (with a unit of coloumbs per volt per cycle) and $\alpha_R \in [0, 1]$ is a workload-specific factor which decides how much switching capacitance C_R is being consumed per cycle. f_R is the operating frequency in cycles per second.

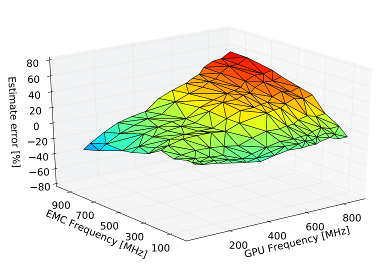
The take-away point from Equations 2 and 3 is that the cost of executing instructions (dynamic power) and using hardware (static power) varies with voltage. The voltage on a rail V_R further depends on the operating frequency on that rail f_R [8], and is regulated by DVFS algorithms (see Figure 2). When building rate-based power and energy models using Equation 1, it is therefore reasonable to expect that, depending on the frequency operating point where the model was built, the estimated costs β_i will vary.

We now conduct an experiment to find out how inaccurate power estimations β_i can be if rail voltage is not taken

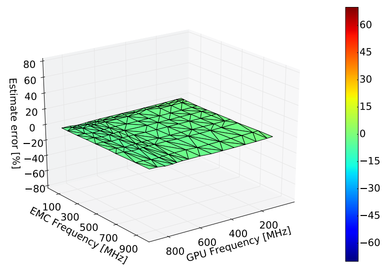
[1]Source code and trace files are available at http://folk.uio.no/krisrst/mmsys16/



(a) Rate-based model.



(b) Modelling switching capacitance αC.



(c) Our hybrid model.

Figure 1: Prediction error for a motion estimation kernel.

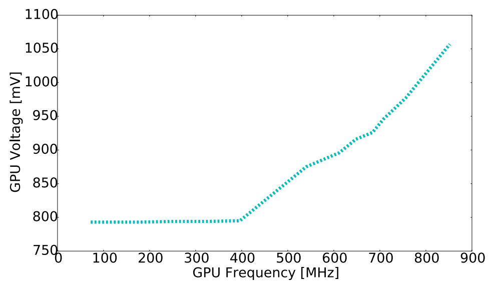


Figure 2: GPU voltage versus frequency.

into account. The details around this experiment is similar to the method presented in Section 5.2. We use Equation 1 and multivariable, linear regression to build a simple power model based on our synthetic benchmarks listed in Table 2 and our predictors in Table 1 (excluding memory, clock and leakage predictors which are not typically used but rather a part of our contributions). We verify the model with a motion estimation filter for videos running on the GPU (see Section 3.2). The result can be seen in Figure 1a, where model accuracy is plotted versus memory and GPU frequencies. As expected, we see that accuracy is very variadic depending on operating frequency because Equation 1 does not consider rail voltages. The resulting planes show a gradual increase in accuracy which for some frequency combinations is very close to 100 % (green and blue area). Our hypothesis is that some of these areas show better accuracy because they reflect better the frequency levels that were set in the model training phase by the memory and GPU DVFS algorithms. At high frequency levels the model underestimates up to 60 %. In the opposite case (low frequencies), the model overestimates by up to about 60 %. This is a direct effect of the fact that rail voltage is not considered.

There are few works which attempt to incorporate rail voltages into power models. Hong and Kim [5] model static power considering both rail voltage and temperature, but have a rate-based dynamic power model. Castagnetti et al. [3] model the power of an Intel XScale CPU. Pathania et al. [13] model power of a 4+4 ARM CPU in big. Little configuration as well as a PowerVR GPU for gaming workloads. Both take into account rail voltages as shown in Equations 2 and 3. However, they are based on finding the dynamic power coefficient $\alpha_R C_R$ either directly or through the processor hardware utilisation level, similarly

Figure 3: Video stream rotation.

to the work by Stokke et al. [15]. The error of such a model built for the Tegra K1 can be seen in Figure 1b. While the error rate of such a model is better than a pure rate-based one it can still be very high up to 50 %. The methodology also has two disadvantages. First, $\alpha_R C_R$ changes depending on the workload characteristics and must be subsequently re-estimated for any workload combination. Secondly, increasing frequencies in various domains (GPU or CPU) inevitably increases hardware utilisation α_R in other parts of the platform. In this aspect, we provide an entirely generic model which takes as inputs the various utilisation levels of different components in the Tegra K1 SoC as well as rail voltages, and achieve a much better accuracy close to 100 %, as shown in Figure 9c.

3. WORKLOADS

Our workload represents video processing operations intended for post-processing raw video streams stored in the YUV format. These have been implemented in CUDA for the Tegra K1 GPU. In our benchmarks, these workloads process 80 full-HD video frames.

3.1 Image Rotation

In the image rotation tests, each frame of a video stream is being rotated by a continuously increasing angle θ (see Figure 3). The algorithm treats each frame as a cartesian coordinate space centered in the middle of the frame. Reference pixel positions (u, v) are calculated by multiplying each original pixel coordinate (x, y) by the *rotation matrix* as follows:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} cos - \theta & -sin - \theta \\ sin - \theta & cos - \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \qquad (4)$$

Subsequently, each reference pixel at position (u, v) is put at its corresponding frame location (x, y).

3.2 Motion Vector Search

In the second test, we apply motion vector search (MVS) on the raw video stream. MVS is a common technique in

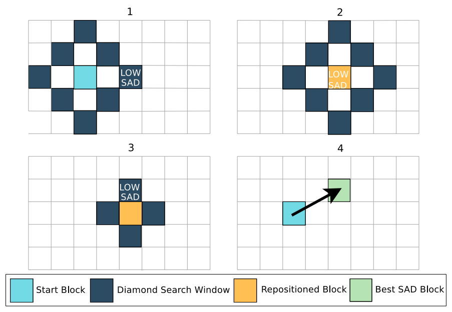


Figure 4: An illustration of the operation of the diamond search algorithm.

video encoding to reduce the amount of information that has to be stored with each frame. In our case, it works by dividing each frame into a set of macroblocks of 8x8 pixels, and then attempting to estimate each block's displacement (the vector) relative to the previous frame.

We have implemented the diamond search algorithm [23]. Diamond search estimates the displacement of each macroblock by computing the sum of absolute difference (SAD) of the current macroblock and the eight surrounding macroblocks in the previous frame (as shown in Figure 4). At every step, as long as the macroblock with the lowest SAD is not in the center of the "search window", the window will be re-centered at the macroblock with the lowest SAD. After three iterations, the pattern changes to a smaller diamond with only four surrounding macroblocks, where the one block with the lowest SAD is estimated to be correct.

3.3 Compression

The third and final test is MJPEG video compression. In this test, the video is compressed by removing high-frequency components from each frame. The image compression algorithm transforms each macroblock of 8x8 pixels into the frequency domain using the discrete cosine transform (DCT):

$$M_{u,v} = \gamma(u)\gamma(v)\sum_{x=0}^{7}\sum_{y=0}^{7} m_{x,y} cos[\frac{\pi}{8}(x+\frac{1}{2})u]cos[\frac{\pi}{8}(y+\frac{1}{2})v] \quad (5)$$

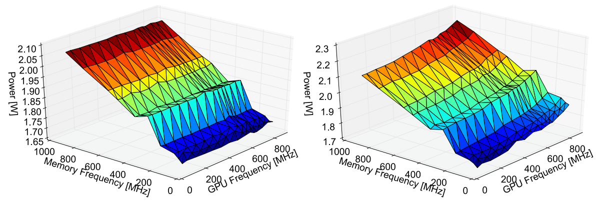
In Equation 5, $u, v \in [0, 7]$ are the DCT output coordinates, $M_{u,v}$ are the frequency components, $m_{x,y}$ are the original pixel values in the macroblock, and $\gamma(w)$ is a normalising function.

3.4 Debarreling

Barrel distortion is an effect that occurs with different lenses [17]. Our "debarreling" workload computes a constant debarreling map for one type of lens. This map only needs to be calculated once and is subsequently applied to each frame. The debarreling filter is the least compute-intensive filter we consider.

4. ENERGY CONSUMING ENTITIES ON THE TEGRA K1 SOC

To build a power model for the Tegra K1, it is important to have solid understanding of the relevant parts of the architecture (see Figure 6) which consumes energy under our workloads. In this section, we describe these components, the Hardware Performance Counters (HPCs) they expose to our power model as predictors, and why we use them (see Table 1 for an overview of the predictors).



(a) GPU is off (b) GPU is on and idle

Figure 5: Power usage depending on whether the GPU is off or not.

4.1 Memory

The Jetson-TK1 is equipped with off-chip DDR3 Random Access Memory (RAM) and two Embedded Memory Controllers (EMC) [11]. The EMCs arbitrate memory accesses from the CPU complexes (EMC_{cpu}, 32-bit accesses) as well as the GPU (EMC_{gpu}, 64 bit accesses). The Tegra K1 is thus capable of executing both CPU and GPU memory-intensive programs at the same time. Carroll and Heiser [2] show in their analysis of the OpenMoko Freerunner smartphone that RAM can account for a substantial amount of total power depending on the type of workload, even without an active GPU. Despite this, RAM power is usually not directly accounted for in literature.

4.1.1 Dynamic Power for RAM

The possibility to monitor memory activity is not trivial. Not accounting for two CPU HPCs that can count L2 cache misses and writebacks there are few predictors which can be used to trace memory utilisation. The CPU HPC implementation only allows for collection of a maximum of six counters simultaneously, so these should be avoided. The GPU is also equipped with set of HPCs which can be read during program execution, but these do not provide access to the total number of bytes read or written to memory by the GPU.

The Tegra K1 instead implements a hardware activity monitor (ACTMON) which is intended to guide the memory DVFS algorithm [11]. The ACTMON is capable of counting the following:

- The total number of memory cycles spent serving CPU memory requests.
- The total number of memory cycles spent serving any memory requests (including GPU and other sources).

We wrote a modified kernel driver for the ACTMON, enabling us to count these variables in any time interval above one ms. While this solution is not able to distinguish between memory reads and writes, it has several advantages. For example, it provides a fine-grained measure of hardware activity created by the GPU and the CPU. The method is also able to count *all* accesses to memory, for example caused by GPU driver overhead and memory accesses from other sources, and avoid occupying HPC space.

4.1.2 Clock Power

The memory bank is continuously spending energy to maintain its consistency. In Figure 5a), the total power of the

Jetson-TK1 is plotted versus memory and GPU frequency. In this example, the GPU rail is off. Therefore, there is no change in power as GPU frequency increases. However, the total power usage increases linearly with memory frequency. We assume that this is due to increased self-refresh frequency. The power model must take this factor into account.

At memory frequencies 204 and 300 MHz, we see that there is an inconsistency in our assumption. Despite the linear trend at other frequencies, power drops slightly at 204 MHz and increases at 300 MHz. We do not know the reason for this, but some PLL clocks are activated in the core domain at these frequencies. We take the drop and increase of power usage (relative to the "linear" increase in power at the other frequencies) into consideration in our model as simple offsets.

4.2 GPU

The Kepler-based GK20A GPU on the Tegra K1 is a more parallel architecture than the CPU, capable of running 128 threads in parallel. The GK20A contains a single SMX. An SMX implements four warp schedulers, each of which is capable of concurrently running groups of 32 threads called *warps*. Two independent instructions per warp can be executed at the same time [12]. The threads utilise 192 CUDA cores which provide arithmetic and floating point functionality and various other units such as code, data and texture caches (see Figure 6). The Special Function Units (SFUs) implement special functions such as sin and cosine. They are out of scope for this paper.

Due to the high number of concurrently active threads, memory pressure (and power usage) is substantial. The SMX features a complex memory hierarchy to improve memory access latency and bandwidth. Read accesses from memory are performed through the Tegra K1's 64 bit EMC2 interface. Accesses are cached in a L2 cache with size 128 kB, which is global to all threads running on the SMX. Memory read accesses can also be stored in a warp-local 64 kB L1 cache, either implicitly on RAM loads or directly through the use of shared memory (in which case it can also be written).

When considering the power usage of the Tegra K1's GPU it is important to have the best control over hardware utilisation of the different parts of the Kepler architecture. In this section, we outline the HPCs used to collect this information using NVIDIA's profiling tool, `nvprof`.

4.2.1 *Memory Hierarchy*

Any memory access can result in hardware utilisation on three different components of the Tegra K1. In the worst case, the memory is fetched or stored off-chip in RAM. Data can also be cached in L2 and L1.

An easy misconception is that the global memory (RAM) throughput HPCs, `gst_throughput` and `gld_throughput` reflect actual amount of data stored and loaded from RAM. However, we found in our experiments that these counters do not separate between actual RAM accesses and L2/L1 cached accesses. Instead, we use the ACTMON activity monitor to track the GPU's utilisation of RAM.

The `l2_subp0_total_read_sector_queries` HPC counts the number of 32 B read accesses to the L2 cache. In our attempts to train the model, we also used the `l2_subp0_total_write_sector_queries`, but we were unable to estimate a meaningful coefficient to it. We suspect this is because L2 writes are directly written to RAM, and that the power usage related to such events is instead captured by our RAM activity counter.

The L1 cache utilisation is more difficult to trace accurately because there is no single counter (as for the L2 cache) which enables us to trace the raw number of read and write transactions. L1 is used for caching thread-local memory accesses, global memory accesses, and finally, it can be configured for shared memory access between threads. To accurately trace L1 cache utilisation, we need to trace all these types of accesses.

Thread-local data consists of for example function parameters. This memory is allocated in RAM and automatically cached in L1. Accesses to local memory via the L1 cache is traced by the `l1_local_{store/load}_hit` HPCs, which increment by one for each 128-bit transaction. Memory reads from RAM may also be stored in L1, and are traced with the `l1_global_load_hit` HPC. Memory stores to RAM are not cached in L1 because L1 caches are not coherent, but are instead directly written to L2 cache or RAM.

L1 utilisation that occurs as a result of shared memory usage is harder to trace. These transactions are counted by the HPCs `l1_shared_{load/store}_transactions`. However, shared memory accesses are often broadcasted to all or several of the threads running in a warp, because all threads access the same memory location. In this case, the number of transactions is still counted for each thread-initiated share memory load or store. Therefore, the number of transactions reported by these HPCs is much higher than what is *actually* read or written in hardware. To estimate the actual utilisation, we found that it is possible to use the `shared_efficiency` HPC. This counter gives the ratio of requested (actual) shared memory throughput to the required (thread-initiated total) throughput. The actual L1 shared access utilisation can be estimated as follows:

$$L1_{shr\_act\_\{l/s\}} = L1_{\_shr\_\{l/s\}} \times shared\_efficiency \qquad (6)$$

Note that the estimate can be inaccurate because `shared_efficiency` does not separate between reads and writes.

4.2.2 *Functional Units*

Running GPU threads utilise various hardware units to perform additions, control operations, register loads and stores etc. NVIDIA provides fine-grained accounting over which types of instructions have been executed over time, which are specified in several groups. The grouping allows for more accurate tracking of hardware utilisation. For example, it is to be expected that floating point operations cost more power compared to pure integer operations. The groupings of HPCs are as follows:

- `inst_fp_32 / inst_fp_64` counts floating point operations on different datatypes (32-bit or 64-bit operations).
- `inst_integer` counts integer operations.
- `inst_bit_convert` counts bit conversion instructions.
- `inst_control` counts control flow instructions, such as branching and jumps.
- `inst_misc` counts miscellaneous instructions, such as register moves and NOP instructions.

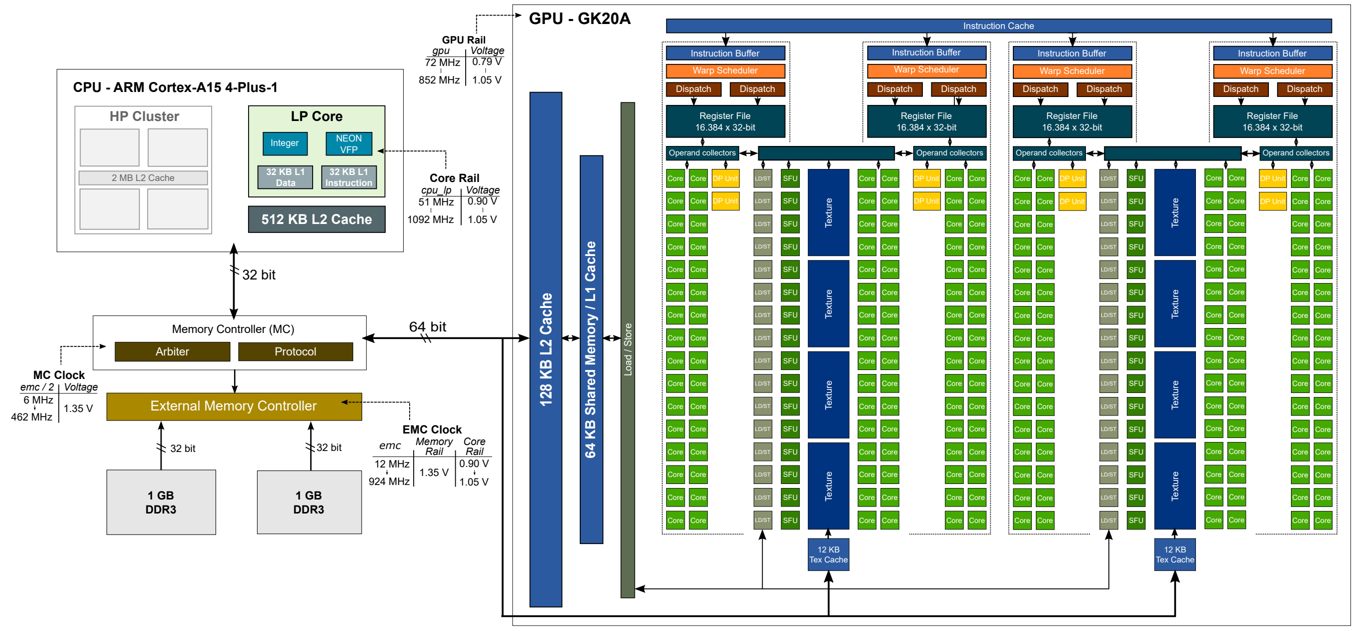


Figure 6: Overview of the Tegra K1 and the Kepler SMX architecture.

4.2.3 *Clock Power*

In addition to dynamic power caused by application utilisation in the GK20A's memory hierarchies and functional units, there is some power usage which is related to the GPU's own clock (clock power). This is visible when the GPU is on (GPU rail is powered), but idle. Consider for example Figure 5b. Here, we have set the GPU's inactivity timer (the time before the GPU rail is powered off after for example a kernel launch) to a few seconds, and logged the average power usage of the Jetson-TK1 in this period over all GPU and memory frequencies. Comparing with Figure 5a, where the GPU is off, we can clearly see that there is non-negligible additional switching activity when the GPU is powered and idle. Furthermore, we can see that the additional overhead scales linearly with frequency until GPU rail voltage starts to increase (at 400 MHz, see Figure 2). From these point, power scales linearly with $f_{gpu}V_{gpu}^2$. This confirms that the additional overhead is an effect of dynamic power on the GPU rail.

4.2.4 *Static Power*

Static power usage is caused by leakage currents in the GPU's circuitry (for example transistors). For the GK20A, we have found that the estimated leakage current on the GPU rail has higher standard deviation than the other predictors. This is likely due to power gating inside the circuitry.

4.3 LP CPU Core

The Tegra K1's CPU is not the main target of this study, but its power usage is non-negligible for several reasons. It is for example running the operating system and all its system services and drivers, as well as launching CUDA kernels through our benchmark programs. It also utilises RAM through its own EMC, in particular to fetch and store processed frames. For these reasons, one cannot assume that the power usage caused by CPU activity is negligible.

While the Tegra K1 has a complex, dual-cluster CPU with a High-Performance (HP) quad-core cluster in addition to the LP companion core, we restrict processing to the LP core because the main target of this study is the GPU. Ideally, it should only be necessary to model dynamic power of the LP core by disabling the HP cluster and fixing processor frequency to 1092 MHz. This causes static power to be estimated as part of the base power, because the core rail voltage (and static power contribution) will not vary. However, in our experiments, we observed that the estimated dynamic CPU power coefficients give more precise values if the static power usage is also modelled.

The Tegra K1's CPU has an HPC implementation which can be queried with the **perf** Linux framework. Compared to the GPU HPC implementation, the CPU has less fine-grained instruction counting, with only one global (total) instruction counter. It does, however, have better accounting for L1 and L2 cache accesses, which are able to separate between hits and misses for both data and instruction cache reads and writes. Only seven counters can be monitored simultaneously, and one of these is always occupied by the *active cycle counter*. Because we do not expect the dynamic CPU power to be substantial, we ignore the CPU cache hierarchy and variation in costs of different types of instructions. We define a new metric, $\rho_{cpu,ipc}$, which is defined as the ratio between the instruction count over the active cycle count which accounts for CPU dynamic power caused indirectly by software execution:

$$\rho_{cpu,ipc} = \frac{N_{cpu,inst}}{N_{cpu,cycles}} \tag{7}$$

Intuitively, $\rho_{cpu,ipc}$ will decrease estimated CPU dynamic power when the CPU is stalling more. We let this counter represent the software workload.

GPU and RAM have a clock-dependent dynamic power cost which is basically an estimated cost per clock cycle. This is also the case for the CPU, but the number of cycles per second is *not* the raw frequency point for the CPU (1092 MHz), as for RAM and the GPU. This is evident because attempting to use the raw frequency point to estimate the dynamic clock power completely breaks the regression results, severely reducing the accuracy of the model and causing several negative coefficient estimates. We believe that the CPU is power gating the clock aggressively when it is idle, and therefore, the actual cycles per second is the active cycle counter itself.

5. A HIGH-PRECISION POWER MODEL

In this section, we derive the power model used to estimate power usage of our GPU multimedia workloads. The main idea behind our methodology is that dynamic power is modelled in terms of measurable hardware activity, and that we compensate for variations in rail voltages. We describe in detail the methodology used to build the model, where we have implemented specialised benchmarks to stress the relevant parts of the Tegra K1 in such a way that the regression results become accurate.

5.1 Derivation

The total power usage of the Jetson-TK1 is the sum of power usage on each rail P_R and base power P_{base}. The Tegra K1 has 21 power rails, but only three are being used in our scenario: the *core*, *memory* and *GPU* rails (see Figure 6). All other rails are assumed to be idle with constant power usage as a part of P_{base}. The total power usage becomes:

$$P_{total} = P_{core} + P_{mem} + P_{gpu} + P_{base} \tag{8}$$

The total power usage on a rail is the sum of static and dynamic power (see Equation 2 and 3). However, the dynamic power equation only gives a crude average of the switching activity (instructions executed, memory requests, caching operations etc). As mentioned in Section 2, the dynamic power coefficient $\alpha_R C_R$ can be estimated using regression [15], but the process is prone to error because α_R is not always constant over all frequency combinations. For example, when increasing memory frequency, switching activity (α_R) in other architectural units such as the CPU or GPU can also increase, leading to misprediction. The core improvement of our model is that we express dynamic power in terms of measurable hardware activity while also accounting for changes in rail voltage. By doing this, our hypothesis is that power usage can be more accurately estimated for any workload on any operating frequency point. We model dynamic power on a rail R as:

$$P_{R,dyn} = \sum_{i=1}^{N_R} C_{R,i}\rho_{R,i}V_R^2 \tag{9}$$

In the equation above, for rail R, $\rho_{R,i}$ is a hardware activity predictor in occurrences per second, N_R is the number of predictors, and $C_{R,i}$ is the switching capacitance (coloumbs per volt) per occurence of event $\rho_{R,i}$. The total power usage of the Jetson-TK1 becomes the sum of power usage on each of the three active rails $R \in \mathbb{R}$ and base power P_{base}:

$$P_{jetson} = \sum^{R\in\mathbb{R}} (P_{R,dyn} + P_{R,stat}) + P_{base} \tag{10}$$

Note that static power of the memory rail is not possible to model because the voltage on that rail does not vary, as per our discussion in the next section.

5.2 Methodology

By extension of Equation 10, the unknown variables are the switching capacitances $C_{R,i}$, leakage currents $I_{R,leak}$ and base power P_{base}. Our methodology to find these terms is based on multivariable linear regression and is summarised as follows. We create twelve spesialised benchmarks designed to stress different hardware blocks of the Tegra K1 (see Table 2). Each of these are run over all possible combinations of processor and memory frequencies (see Table 3) for a total of 1830 samples. In each run, we log the predictors $\rho_{R,i}$, voltages V_R and operating frequencies. The predictors must now be processed to account for the rail voltage:

- All dynamic power predictors $\rho_{R,i}$ must be multiplied by V_R^2 (see Equation 9).
- The static power predictor is just the rail voltage V_R (see Equation 2).

The resulting "final" predictors are then passed to the regression solver and produces the coefficients seen in Table 1. Note that while related works often use multivariable regression with non-negative coefficients [19] we simply use normal regression without ending up with negative coefficients[2].

Running each benchmark over all possible frequency combinations has several advantages.

1. It creates natural variation between the model predictors (access rates). When memory frequency is low and GPU frequency is high, the memory access rate is lower while the GPU hardware utilisation is higher, and vice-versa. This is also helps test the full range of model predictors (rates), which is extremely important for regression to estimate accurately.
2. Rail voltages vary depending on operating frequency. This also increases diversity in the training dataset's predictors.
3. Increasing frequency like this also helps estimate clock power and leakage currents (which would be impossible otherwise) because higher rail voltage is higher at high GPU frequencies.
4. Increased dataset size.

Note that our method consists of a high number of runs (1830) to complete the model training phase because each of the benchmarks listen in Table 2 is run over all possible GPU and memory frequency levels. The number of runs can be reduced by including fewer frequency levels and should not affect the accuracy of the model, but care must be taken to force variation in rail voltages. For example, most of the GPU frequencies below 400 MHz are not needed because rail voltage does not vary (see Figure 2). We leave it to future work to investigate the impact in terms of model accuracy resulting from a reduced number of training frequencies.

In our initial attempts to train our model, we found that trivially using example code (for example CUDA examples)

[2]The only exception is our memory offset ”tweak” at 204 Mhz, which is expected.

Rail	Number	Predictor	Description	Coefficient	Value
GPU	0	V_{gpu}	GPU voltage	$I_{gpu,leak}$	$0.27A$
	1	$\rho_{gpu,clock}$	Total clock cycles per second	$C_{gpu,clock}$	$2.10\frac{nC}{V}$
	2	$\rho_{gpu,L2R}$	L2 cache 32B reads per second	$C_{gpu,L2R}$	$10.79\frac{nC}{V}$
	3	$\rho_{gpu,L1R}$	L1 cache 4B reads per second	$C_{gpu,L1R}$	$8.90\frac{nC}{V}$
	4	$\rho_{gpu,L1W}$	L1 cache 4B writes per second	$C_{gpu,L1W}$	$8.43\frac{nC}{V}$
	5	$\rho_{gpu,INT}$	Integer instructions per second	$C_{gpu,INT}$	$41.11\frac{pC}{V}$
	6	$\rho_{gpu,F32}$	Float (32-bit) instructions per second	$C_{gpu,F32}$	$38.15\frac{pC}{V}$
	7	$\rho_{gpu,F64}$	Float (64-bit) instructions per second	$C_{gpu,F64}$	$115.33\frac{pC}{V}$
	8	$\rho_{gpu,CNV}$	Conversion instructions per second	$C_{gpu,CNV}$	$72.42\frac{pC}{V}$
	9	$\rho_{gpu,MSC}$	Miscellaneous instructions per second	$C_{gpu,MSC}$	$28.36\frac{pC}{V}$
Memory	0	$\rho_{mem,clock}$	Total clock cycles per second	$C_{mem,clock}$	$258.66\frac{pC}{V}$
	1	$\beta_{mem,204}$	Power offset at 204 MHz	$P_{mem,204}$	$-0.03W$
	2	$\beta_{mem,300}$	Power offset at 300 MHz	$P_{mem,300}$	$0.05W$
	3	$\rho_{mem,CPU}$	CPU busy memory cycles per second	$C_{mem,cpu}$	$2.25\frac{nC}{V}$
	4	$\rho_{mem,OTH}$	Other (GPU) busy memory cycles per second	$C_{mem,oth}$	$2.17\frac{nC}{V}$
Core	0	V_{cpu}	CPU voltage	$I_{cpu,leak}$	$0.79A$
	1	$\rho_{cpu,cpi}$	CPU instructions per cycle	$C_{cpu,cpi}$	$3.72\frac{mC}{Vs}$
	2	$\rho_{cpu,acl}$	CPU active cycles per second	$C_{cpu,acl}$	$166.62\frac{pC}{V}$
Other	P_{base}		Base power	-	$0.78W$

Table 1: Overview of energy model predictors and coefficients.

Benchmark	Description	Components / instructions under explicit stress										
		CPU	RAM (CPU)	GPU	RAM (GPU)	L2	L1	INT	F32	F64	Conv.	Misc.
Idle CPU	GPU off, CPU in idle state.	✓										
CPU-workload	GPU off, CPU processing.	✓	✓									
Idle GPU	GPU on and idle, CPU in idle state.	✓		✓								
L2 Read	Stresses L2 cache reads only.	✓		✓		✓						
L1 Read	Stresses L1 cache reads.	✓	✓	✓			✓					
L1 Write	Stresses L1 cache writes.	✓	✓	✓			✓					
RAM	Stresses RAM activity (GPU EMC).	✓		✓	✓							
Integer	Stresses integer arithmetic unit.	✓		✓		✓		✓				
Float32	Stresses floating point unit.	✓		✓		✓		✓	✓			
Float64	Stresses floating point unit.	✓		✓		✓		✓		✓		
Control	Stresses conversion instructions.	✓		✓		✓		✓			✓	
Misc	Stresses miscellaneous instructions.	✓		✓		✓		✓				✓

Table 2: Overview of benchmarks and components under stress.

Clock	Rail	Description	Frequency		Voltage
			Steps	Range $[MHz]$	
cpu_lp	Core	LP core	9	$[51, 1092]$	$[0.80, 1.05]$
emc	Core	Memory	10	$[40, 924]$	$[0.80, 1.01]$
gpu	GPU	LP core	15	$[72, 852]$	$[0.79, 1.05]$

Table 3: The Tegra K1 clocks, voltage and frequency ranges.

has two major drawbacks. First, the measured model predictors are not diverse enough for the regression to estimate meaningful coefficients. For example, L1 and L2 read throughput will remain virtually the same independently of operating frequencies and benchmarks, as data passes both stages anyway. Secondly, only using benchmarks which are actively processing result in poor estimations. This is because some coefficients, such as leakage currents, clock and base power are independent of the workload. Much better estimations can be achieved by also profiling the power for an idle system.

Ideally, each benchmark should stress just one architectural unit of the system (memory writes, L2 reads, etc.). This is hard, and in some cases impossible, to achieve in practice. For example, attempting to *only* read 100 MB of data from memory to L2 cache results in no data being read at all, because the CUDA driver is very good at optimising code and detects that the data is not actually used. This optimisation happens at a driver level and can not be disabled. Furthermore, as mentioned above, at some points it is impossible to stress just one architectural unit. An example is stressing the L1 cache. Stressing the L1 cache inevitably results in stressing L2 cache *and* memory, because of cache eviction policies and the internal workings of the GPU.

We have written twelve spesialised different benchmarks, which can be seen in Table 2. Due to the limitations described above, these are written in a ”pyramid” fashion where we stress the (possible) small groups of hardware units first, before adding units on top. For example, it is possible to stress L2 cache reads only, without stressing memory or any other hardware units, by reading the same data from memory over and over without L1 caching, and then conditionally writing it back in an if-condition which never evaluates to true. Then, the process can be done again, but this time forcing the GPU to cache the global reads in L1 as well. The regression will now have diversity in both predictors to accurately estimate their coefficients.

The results of the regression can be seen in Table 1. Note that there is no coefficient for L2 writes. The reason for this is that the estimated coefficient is very small compared to the others, and has very high standard deviation. We believe this is because L2 writes are not actually cached there, but immediately written back to memory. The power usage related to this event is thus instead part of the active memory cycles.

6. EXPERIMENTS AND DISCUSSION

In this section, we perform extensive verication of our power model using generic multimedia workloads. We show that our model is able to predict power with high accuracy, discuss the model coefficients and finally look at possible ways to save power by exploiting local caches and shorter datatypes.

6.1 Setup

To verify our model, we have developed four different GPU benchmarks for video processing (debarreling, DCT, motion estimation and rotation). We let these process a full-HD video stream for 80 frames, over all possible GPU and memory frequency combinations. Figure 7 shows a simplified flow diagram for a single test run. Before power can be estimated, the GPU HPCs must be collected. This is challenging for a number of reasons:

- HPC collection for GPU kernels takes a substantial amount of time, slowing down the kernels. We must therefore log HPCs on a per-kernel basis to be re-used.
- HPC values vary not only based on the kernel executed, but also on launch configuration and (possibly) function parameters.
- Depending on the set of HPCs to collect, several runs over one kernel is needed.

To solve these issues, we implement callback functions to track kernel launches and kernel exits. On entry to these functions, a hash is computed based on the kernel's symbol name and execution configuration. We ignore variance in HPCs that result as changes in function parameters, which is only relevant for the rotation workload. On kernel launches, we initiate HPC collection as needed (in total three runs per hashed kernel). On kernel exits, if HPC collection is complete, we estimate power of the GPU, CPU and memory using the counters. GPU execution time is measured by reading the total elapsed GPU cycles for a hashed kernel (the `elapsed_cycles_sm` HPC, e_c) and dividing it with the frequency of the GPU:

$$T_{kern} = \frac{e_c}{f_{GPU}} \tag{11}$$

The experimental setup is based on a Keithley K2280S power source and monitoring unit using an external machine to avoid overhead on the Tegra K1 [15]. The power estimation phase is low-overhead which involves straightforward calculation of power using model coefficients and predictors collected with PERF and CUPTI APIs and equations from Section 5.

6.2 Accuracy

Figure 8 shows an example the measured and estimated power components in a single DCT frame encoding interval.

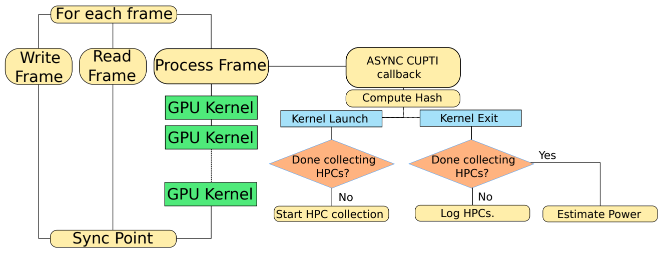


Figure 7: Per-Frame Benchmark flow.

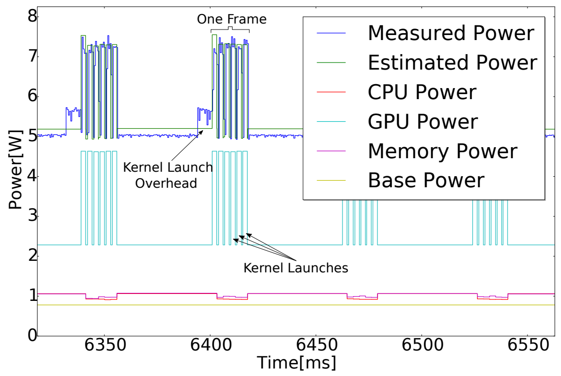
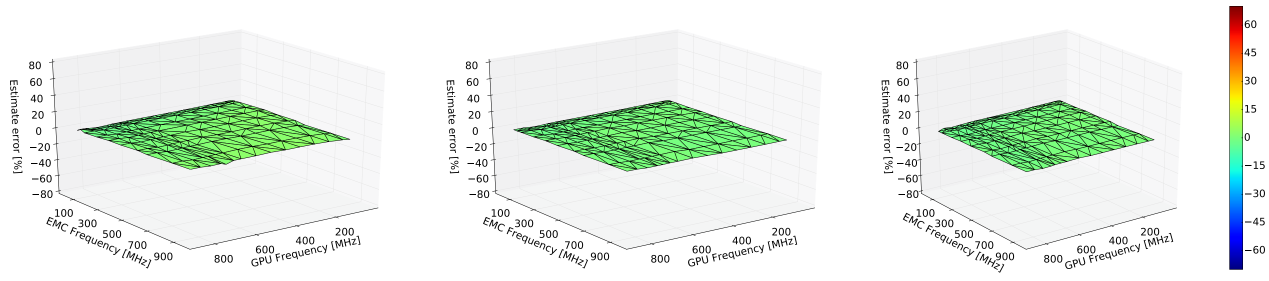


Figure 8: Power plot over time for the DCT kernel.

The peaks in the plot represent kernel launches. There are six kernel launches per DCT frame because the Y frame is divided into four subregions with the same size as a UV frame. We see that the estimation is very accurate over time, close to 100 %. This represents a substantial improvement from state-of-the-art methods that do not consider rail voltages (see Section 2). The total power estimate appears less precise between frames. This is because power estimation only occurs after each kernel launch, and there is a non-negligible kernel launch overhead before the first kernel launch at the beginning of each frame. This overhead is represented as an average since the exit of the last kernel of the previous frame.

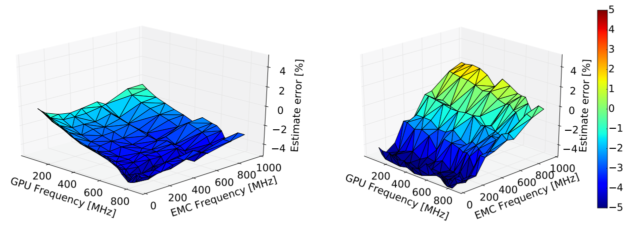
Figure 9 shows the model error in % for a total of 70 processed frames over all GPU and memory frequency combinations, for the debarreling, DCT and rotation workloads. The estimation is very accurate. Not considering the motion estimation filter, the accuracy of our model is over 99 % on average and always above 96 %. This demonstrates that our model, which is built using entirely synthetic benchmarks, is able to consisely capture the power usage of kernels comprised of a more complex mix of instructions and branches.

We now study Figure 10a, which shows a closed-up plot of prediction error for the motion estimation filter. We see that, in general, estimation is good (between 99 to 100 %) at low GPU frequencies. Moving towards the lowest memory and GPU frequency, estimation accuracy tends to degrade towards its lowest point. This is true for all the filters, but hard to see in Figure 9. Furthermore, we see that at 756 MHz GPU frequency, estimation accuracy shows a significant drop across all benchmarks. It is here important



(a) Debarreling. (b) DCT. (c) Rotation.

Figure 9: Prediction error for test benchmarks over all GPU and memory frequency combinations.



(a) Close-up view. (b) No GPU execution.

Figure 10: Motion estimation error over all frequencies.

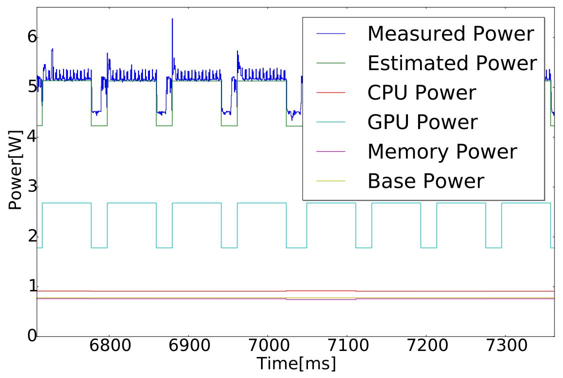


Figure 11: Power over time for a single motion estimation Y-frame processing at 756 MHz (GPU) and 924 MHz (memory).

to note that there is a substantial amount of time where the GPU is *not* actively processing, but the CPU is busy reading the next frame or writing results in memory (see Figure 8). When error starts increasing, we see that the power estimation during this period tends to be larger than when the GPU is active. Consider for example Figure 11, where a single peak (Y-frame motion estimation kernel) is shown. For this test, the error was 4 %. It is clear that the estimation during the kernel launch is very accurate. However, in the period between frames, the estimation is worse. This indicates that either the CPU or memory estimate is inaccurate.

To test the accuracy of the CPU and memory model, we run a differential test on the motion estimation benchmark. No GPU kernels are executed, but frames are read and results written as normal. To avoid compiler optimisations, we conditionally execute the motion estimation kernel in an if-statement which never evaluates to true. The result can be seen in Figure 10b. We see that the error over CPU-only execution can vary about the same amount as when the GPU kernels are also being run. We believe that a better power model for the LP core is needed to further increase the accuracy of the model.

The motion estimation filter has a lower accuracy than the rest of the filters. On average, it is 97 %, and the estimation accuracy decreases more with GPU frequency than for the other filters (at the lowest 95 % accurate). The motion estimation kernels stresses shared memory more, and each kernel runs for 5 to 26 times longer than for the other filters. We found that the CUPTI counters for shared memory transactions do not appear to be reliable, which can cause our model to mispredict more than the other filters. For the U and V frames, CUPTI is for example not able to count shared memory transactions at all. Only shared memory transactions for the Y frames are actually counted. Furthermore, the implementation used to run the motion estimation kernels is slightly different than for the filters. It is therefore also possible that the CPU model is causing the misprediction.

6.3 Model Coefficients and Power Breakdown

Table 1 shows all estimated model coefficients. Looking at the leakage currents and base power, we see that the CPU (LP core) leakage is estimated to be 0.79 A, and the base power 0.82 W. This is very similar to our previous work [15] where a different methodology is used to find dynamic, static and base power. GPU leakage is estimated to 0.27 A, which is much smaller than the CPU. Possible reasons for this is that the GPU is implemented using a smaller number of and/or another type of transistors. Related work has shown that the leakage of the HP cluster varies between 0.38 to 0.80 A [15] depending on the number of online HP cores. Given a choice between running a workload on any of these processors, the GPU therefore seems like a better choice, given that its static power dissipation will be much lower. However, dynamic power must also be taken into account to find the most energy-efficient alternative.

Figure 12 shows most of the dynamic power coefficients from Table 1 in coloumbs per volt. $C_{cpu,cpi}$ is not shown because it has units of coloumbs per volt per second. Considering clock power ($C_{*,clock}$), we can see that the memory

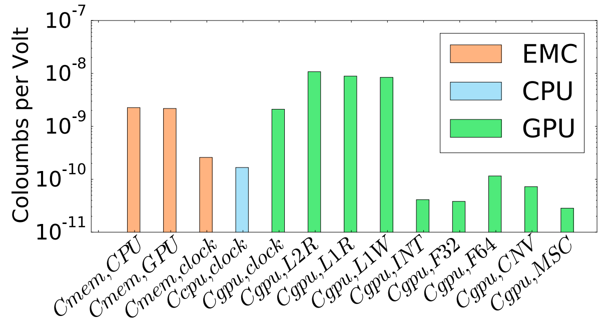


Figure 12: Model coefficients.

Source	Energy Cost			
	per Transaction		per Byte	
	LV	HV	LV	HV
Memory	$3.96nWs$		$0.49nWs$	
L2 Read	$6.73nWs$	$11.89nWs$	$0.42nWs$	$0.74nWs$
L1 Read	$5.55nWs$	$9.81nWs$	$1.38nWs$	$2.45nWs$

Table 4: Energy cost for reading from various cache and memory hierarchies (excluding static and clock energy).

and LP core clock coefficients are similar, i.e., the capacitance load per active clock cycle are the same. For example, at the highest operating frequencies (see Table 3), the LP core and memory clock power is estimated to be 0.20 and 0.43 W, respectively. The GPU clock capacitance is an order of magnitude higher than for the CPU and memory. At the highest frequency, the GPU clock power is estimated to be 1.97 W. From Section 4.2.3, we argue that clock power is independent of the workload. As a processor, the GPU therefore incurs a substantial overhead in terms of clock power compared to the LP core.

The capacitance load per active memory cycle, $C_{mem,*}$, is the same independently of the source. For example, if the CPU initiates a single memory active cycle, it will cost the same as one initiated by the GPU. Since the GPU EMC has a higher bandwidth than the CPU EMC (64-bit vs. 32-bit), it may be possible to save energy in this way. We wrote some simple programs to read and write memory from the GPU, and found that regardless of operation (read or write) the CPU EMC is capable of delivering 4 B per active memory cycle, and the GPU EMC is capable of delivering 8 B per active cycle. This means that reading and writing through the GPU EMC is twice as energy-efficient than the CPU EMC. Other factors must of course also be taken into account, such as the processing done on the data and cache hierarchies.

Studying the workload-relevant GPU coefficients in Figure 12, we see that the capacitance load per transaction on the cache hierarchies ($C_{gpu,L*}$) is larger than the cost for active memory cycles ($C_{mem,GPU}$). However, the actual energy cost depends also on the GPUs rail voltage (see Equation 9), as well as the number of bytes read or written per transaction (see Table 4). Interestingly, memory transactions appear to be more energy-efficient than reading from L1 and L2 GPU cache in most cases (except for the lowest GPU operating voltage). However, leakage and clock power also play important factors. If memory is only read through memory, memory fetches will take longer time, and so the contribution from leakage and clock power will be larger.

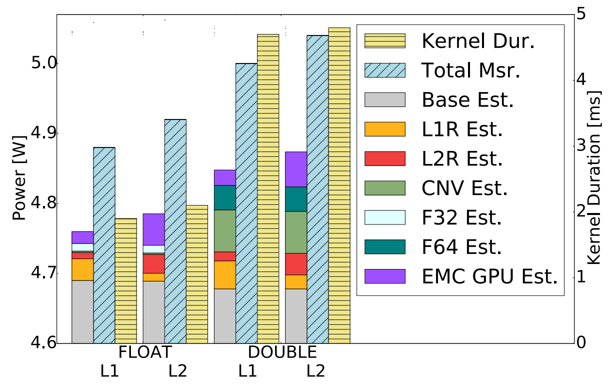


Figure 13: Difference in power usage when using different floating-point datatypes and cache strategies for the DCT.

6.4 Energy-Efficient Programming

In Section 6.3, we have seen that the cost of various instructions differ. For example, the model coefficients hint that loading memory through the L1 cache is cheaper than loading it through the L2 cache. 32-bit floating point instructions are also estimated to have a smaller switching capacitance (see Figure 12). We now attempt to energy-optimise some of our workloads using this information as a preliminary experiment.

We run the DCT benchmark at the highest GPU and memory operating frequencies where we study the impact in terms of performance and power when caching in L1 over L2 and using 32- and 64-bit floating point datatypes (see Figure 13). In these tests, we target a rate of five FPS. This is not because five FPS is very good, but only to equalise the contribution to power usage from the EMCs, CPU and other components. The variation in power usage will reflect only a change in the GPU power usage. At this framerate, the processing finishes within a window of 200 ms, and sleeps for the remaining time before starting processing of the next frame.

Studying Figure 13, we see through our model that using 64-bit floating point operations consumes above three times as much power as using 32-bit operations. Furthermore, a substantial amount of power is saved in bit conversion instructions. With L2 caching, changing the datatype to 32-bit saves 2.4 % power on average. Additionally caching in L1, we see that we are able to save 3.2 %. Our model indicates two main reasons behind this. Although the combined L1 and L2 read energy remains approximately the same, caching in L1 can reduce energy usage because it reduces GPU EMC hardware activity. We believe that this is because the L2 cache is coherent, and maintaining this coherency involves some activity in memory. Note that this is a system-wide measure with much idling between frames, and that the actual saving will vary depending on workload and framerate.

7. CONCLUSION

Power models commonly used for evaluation of multimedia systems in literature are very basic and have the potential for serious misprediction. This is because they do not take into account the physical phenomena which govern energy usage on modern platforms. In this paper, we have shown that the evaluation of such systems mandates more hardware insight than what is typical. Our method to model power usage achieves an average accuracy above 99 % for several GPU multimedia workloads on the Tegra K1. Our method achieves better accuracy than traditional power models by estimating the power costs of hardware utilisation on the Tegra K1's GPU, memory and CPU (for example instructions per second or active memory cycles) as well as clock and leakage power using measured rail voltages. The estimates are based on entirely synthetic benchmarks designed to stress specific hardware blocks. The model has also been more rigorously tested than normal, over all possible operating frequencies and with different, complex multimedia workloads that excersise more hardware components at the same time. Our model shows not only that it is possible to estimate power of workloads in a generic way (for any GPU multimedia workload) using available HPCs on the Tegra K1, but also helps us to understand better how software can optimise energy usage of multimedia workloads from a hardware point of view. Preliminary experiments show that 3 % power can be saved by using shorter datatypes (for example 32-bit floating point over 64-bit) and by exploiting local, non-coherent caches which do not consume power to maintain consistency with other memory hierarchies.

Acknowledgments

This work has been performed in the context of the BIA project *Unified PCIe IO* (#235530), with contributions from the FRINATEK project *EONS* (#231687) both funded by the Research Council of Norway (RCN).

8. REFERENCES

[1] R. Basmadjian and H. de Meer. Evaluating and Modeling Power Consumption of Multi-Core Processors. In *Proc of e-Energy*, pages 1–10, 2012.

[2] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In *USENIX Technical Conference*, volume 14, 2010.

[3] A. Castagnetti, C. Belleudy, S. Bilavarn, and M. Auguin. Power Consumption Modeling for DVFS Exploitation. In *Proc of DSD*, pages 579–586, 2010.

[4] M. Dong and L. Zhong. Self-Constructive High-Rate System Energy Modeling for Battery-Powered Mobile Systems. In *Proc of MobiSys*, pages 335–348, 2011.

[5] S. Hong and H. Kim. An Integrated GPU Power and Performance Model. In *Proc of ISCA*, pages 280–289, 2010.

[6] M. Hosseini, J. Peters, and S. Shirmohammadi. Energy-Budget-Compliant Adaptive 3D Texture Streaming in Mobile Games. In *Proc of MMSys*, pages 1–11, 2013.

[7] Q. Jiao, M. Lu, H. P. Huynh, and T. Mitra. Improving GPGPU Energy-Efficiency Through Concurrent Kernel Execution and DVFS. In *Proc of CGO*, pages 1–11, 2015.

[8] N. S. Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan. Leakage Current: Moore's Law Meets Static Power. *IEEE Computer*, pages 68–75, 2003.

[9] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi. GPUWattch: Enabling energy optimizations in GPGPUs. *SIGARCH Computer Architecture News*, 41(3):487–498, 2013.

[10] N. Limin, T. Xiaobin, and Y. Baoqun. Estimation of System Power Consumption on Mobile Computing Devices. In *Proc of CIS*, pages 1058–1061, 2007.

[11] NVIDIA. Tegra K1 Technical Reference Manual. Technical report.

[12] Nvidia. Kepler GK110. *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*, 2012.

[13] A. Pathania, A. E. Irimiea, A. Prakash, and T. Mitra. Power-Performance Modelling of Mobile Gaming Workloads on Heterogeneous MPSoCs. In *Proc of DAC*, 2015.

[14] M. Pricopi, T. S. Muthukaruppan, V. Venkataramani, T. Mitra, and S. Vishin. Power-Performance Modeling on Asymmetric Multi-Cores. In *Proc of CASES*, 2013.

[15] K. R. Stokke, H. K. Stensland, P. Halvorsen, C. Griwodz. Why Race-to-Finish is Energy-Inefficient for Continuous Multimedia Workloads. In *Proc of MCSoC*, pages 57–64, 2015.

[16] Y. O. Sharrab and N. J. Sarhan. Aggregate Power Consumption Modeling of Live Video Streaming Systems. In *Proc of MMSys*, pages 60–71, 2013.

[17] G. Vass and T. Perlaki. Applying and Removing Lens Distortion in Post Production. In *Proc. of Hungarian Conference on Computer Graphics and Geometry*, pages 9–16, 2003.

[18] J. M. Vatjus-Anttila, T. Koskela, and S. Hickey. Power Consumption Model of a Mobile GPU Based on Rendering Complexity. In *Proc of NGMAST*, pages 210–215, 2013.

[19] Y. Xiao, R. Bhaumik, Z. Yang, M. Siekkinen, P. Savolainen, and A. Yla-Jaaski. A System-Level Model for Runtime Power Estimation on Mobile Devices. In *Proc of GreenCom & CPSCom*, pages 27–34, 2010.

[20] F. Xu, Y. Liu, Q. Li, and Y. Zhang. V-edge: Fast Self-Constructive Power Modeling of Smartphones Based on Battery Voltage Dynamics. In *Proc. of USENIX NSDI*, pages 43–55, 2013.

[21] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang. Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones. In *Proc of CODES/ISSS*, pages 105–114, 2010.

[22] Zhihai He, Yongfang Liang, Lulin Chen, I. Ahmad, and Dapeng Wu. Power-Rate-Distortion Analysis for Wireless Video Communication Under Energy Constraints. *IEEE Transactions on Circuits and Systems for Video Technology*, 15(5):645–658, 2005.

[23] S. Zhu and K.-K. Ma. A new diamond search algorithm for fast block-matching motion estimation. *IEEE Transactions on Image Processing*, 9(2):287–290, 2000.