A model of architecture for estimating GPU processing performance and power

Saman Payvar, Maxime Pelcat, Timo D. Hamalainen

Design Automation for Embedded Systems
https://doi.org/10.1007/s10617-020-09244-4

A model of architecture for estimating GPU processing performance and power

Saman Payvar[1] · **Maxime Pelcat**[2,3] · **Timo D. Hämäläinen**[1]

Received: 5 February 2020 / Accepted: 15 December 2020

Abstract
Efficient usage of heterogeneous computing architectures requires distribution of the workload on available processing elements. Traditionally, the mapping is based on information acquired from application profiling and utilized in architecture exploration. To reduce the amount of manual work required, statistical application modeling and architecture modeling can be combined with exploration heuristics. While the application modeling side of the problem has been studied extensively, architecture modeling has received less attention. Linear System Level Architecture (LSLA) is a Model of Architecture that aims at separating the architectural concerns from algorithmic ones when predicting performance. This work builds on the LSLA model and introduces non-linear semantics, specifically to support GPU performance and power modeling, by modeling also the degree of parallelism. The model is evaluated with three signal processing applications with various workload distributions on a desktop GPU and mobile GPU. The measured average fidelity of the new model is 93% for performance, and 84% for power, which can fit design space exploration purposes.

Keywords Modeling · Model of architecture · Design space exploration · Signal processing systems

1 Introduction

Heterogeneous platforms that contain GPUs and DSPs alongside general-purpose processors have become the mainstream for many signal processing applications, such as image, video and audio processing. One of the design decisions that should be made in the early stage is

✉ Saman Payvar
saman.payvar@tuni.fi

Maxime Pelcat
maxime.pelcat@insa-rennes.fr

Timo D. Hämäläinen
timo.hamalainen@tuni.fi

1 Tampereen Yliopisto, Tampere, Finland

2 IETR/INSA Rennes, Rennes, France

3 Institut Pascal, Clermont Ferrand, France

Published online: 16 January 2021

mapping of the application to the platform i.e. resource allocation for processing elements. Unfortunately, the exploration of mapping alternatives is still mostly performed case by case, which is a work-intensive and time consuming task. An approach that considerably reduces the effort is building models of the target platform and the application, and exploiting them with automatic tools. There are different approaches to the system modeling and workload mapping. For example, the Distributed Operation Layer (DOL) [21] is a framework for automatically optimizing parallel algorithm mapping on heterogeneous platforms. ArchC [19] is an architecture description language (ADL) for architecture design which provides early stage system verification. In contrast, rather than jointly designing optimization methods and architecture representations, this paper concentrates on learning a model from structure hypotheses and platform measurements, with the objective to obtain an abstract, repeatable and application decorrelated model usable in a wide set of optimization contexts.

In statistical system modeling, the application and the architecture are often considered together. Originally introduced for modeling of signal processing systems [14], the Linear System Level Architecture (LSLA) [15], Model of Architecture (MoA) separates the underlying architecture from the algorithmic aspects following a Y-chart approach [8]. LSLA specifically models the architecture and distinguishes the concepts of Model of Computation (MoC) from the MoA. The MoA and MoC separation reduces the modeling effort by formulating the system modeling as mapping of MoC activity to the MoA, so that the MoA and the MoC can be treated independently when needed. In LSLA it is possible to map different types of MoC to the LSLA, such as Synchronous Data Flow (SDF) [9] that is popular in signal processing.

In LSLA, an application described by a MoC is mapped to the architecture modeled by the LSLA MoA. Considering the activity of the application, a cost function is computed for each processing element in the platform. For estimating the performance of various mapping alternatives, the cost functions of the processing elements are summed up while varying the mapping parameters. For example, the energy consumption of the Odroid XU3 platform was modeled in [15]. In this particular case, eight processing elements interconnected by three communication nodes model the asymmetric eight-core CPU of the platform. The LSLA experiments model the energy consumption of a Stereo Matching application that computes a depth map from a pair of views of a single scene, while the GSLA experiments in this paper cover the execution time and the power consumption costs of matrix multiplication, digital predistortion and Gaussian filtering applications.

LSLA provides a model for linear Key Performance Indicators (KPIs). However, most contemporary platforms include a GPU, in which the performance with respect to the application activity is non-linear. This is the key motivation to extend the LSLA model. Our initial work was presented in [13] with a GPU performance model. Consequently, in this work we introduce power modeling and collect all the results for an LSLA model extended to GPU modeling.

The key contributions of this work are:

- An extension of the LSLA model with non-linear GPU processing modeling called GSLA (GPU-oriented System-Level Architecture). The model includes both performance and power.
- Prototype tooling implemented as shell scripts and Matlab code for both execution of the application for model creation and costs prediction in exploration use.
- Proof-of-Concept with three representative applications that are implemented in OpenCL and executed on two different GPU-equipped platforms for setting the model parameters and comparing the measured and predicted values for fidelity.

Experimental results show that the proposed GSLA model can help predicting with low complexity the performance and the power consumption of an application with varying parameters. On the other hand, modeling the key performance indicators of strongly differing applications or platform configurations is shown to require model parameter recalculation. Even in cases where parameters cannot be reused, experimental results show that the model lightweight structure can be kept, and retrained through a lightweight procedure and with good accuracy. Two different utilization examples of GSLA are discussed in Sect. 2 to clarify the intended usage of the MoA.

This paper is structured as follows: Sect. 2 introduces related work and provides a comparison to the proposed work. Sect. 3 introduces the MoA concept. Section 4 explains the proposed LSLA extension. Section 5 explains the parameters. Section 6 elaborates the performed experiments. Section 7 presents the novel power model for GPUs. Section 8 compares the fitness of the models. Finally, Sect. 9 explains the conclusions.

2 Model creation and potential usage

The design and usage of the proposed performance and power models in practice are represented in Fig. 1. The blue stages depict the steps of the model construction while the green steps show two examples of using the models. Here the dashed lines demonstrate the manual estimations while the continuous lines present the flow of the work.

Model development starts by selecting the test data and a target parallelizable algorithm i.e. matrix multiplication, Gaussian filtering and predistortion for execution on Mali and Baffin GPUs. Then, the algorithms are coded with OpenCL function calls and receive two command line argument inputs named S and γ which impact the size of the input data and the number of their parallel executions, offering variations in their structure. Later, two value sets for these parameters are defined where the S values set is inferred from the test data and the algorithm input data size and the γ values set is extracted from the platforms number of processing elements and memory sizes. In addition, the values for both S and γ sets were checked for feasibility e.g. execution of the OpenCL implementation of the parallel algorithm with a large S and a small γ values could take up to some days while multiple S and γ combinations are executed for profiling which makes it impractical. Shell scripts are used for passing the values of the S and γ parameters as command line arguments for each iteration of the execution and storing the execution time and power consumption values in files while the memory overflow errors are monitored. Linear regression is used to fit the GSLA model. The MoC and MoA for OpenCL application on GPU is then developed for showing the application mapping on the platform. The results of this paper show the suitable parameters of the models equations for Mali and Baffin platforms i.e. α,β or a, b, c. These values in conjunction with the model equations could be used to estimate the power and performance of similar applications on Mali and Baffin GPUs.

The produced models could be used with or without the presented parameters which are shown as two utilization cases in Fig. 1. In the first case, a similar algorithm to the selected three algorithms is executed on Mali or Baffin platforms while in the second case a different algorithm is executed on a platform X. The first case does not require any compilation or execution of a training code and the performance and the power estimations are done merely by simple equation calculations using the reported parameters values. The S and γ values of an algorithm implemented in OpenCL are computed by checking the code files for the global work size and number of kernel calls which is usually defined as constant values in

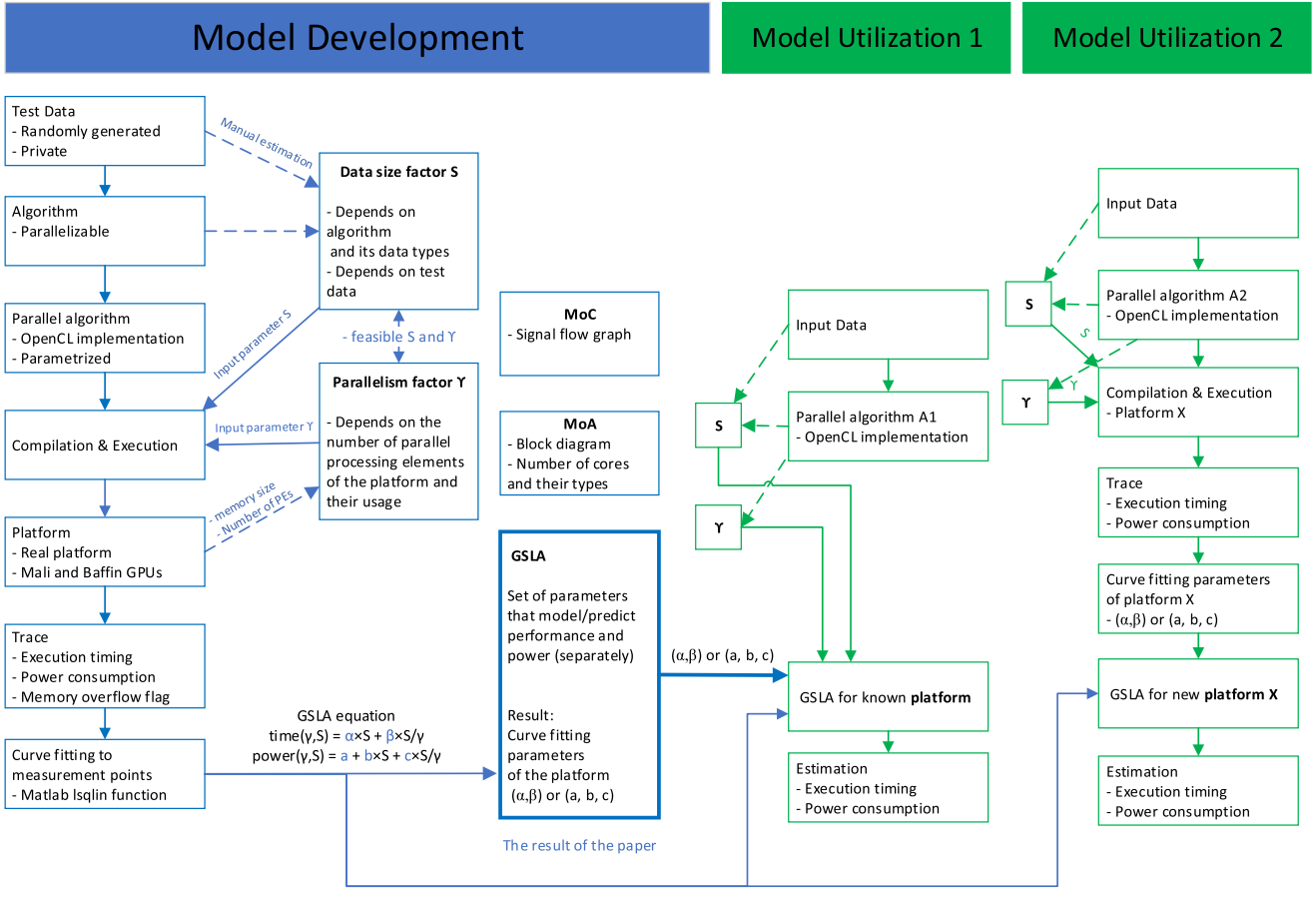

Fig. 1 Development and utilization work flow

a header file and considering the input data of the application. In the second case where the algorithm or the platform vary much from the presented training, values of parameters shall be determined by the execution of a representative algorithm on the platform and measuring the execution time or power consumption and using the presented models equations for the parameters calculations. Later, the calculated parameters could be reused for performance and power estimations of similar cases. These equations reduces the manual work of a researcher using the OpenCL for parallelizing where the appropriate values are typically determined by trial and error i.e. editing the code and benchmarking the application.

3 Related work

There are different methodologies in performance modeling studies. One of them, which provides a general model is the statistical analysis method. For example, Moren, et al. [11] present a statistical approach for work load scheduling on heterogeneous platforms consisting of CPU and GPU. Authors have modified the OpenCL API code for dynamic code feature collection which is used for performance prediction. In modeling methods, it is common to use a graph to present a software or a hardware system, or a system of systems. These methods are divided into two different categories: data flow graphs and non data flow graphs. In a data flow graph, a vertex is used to model a run-to-completion block of computation called an *actor*. *Edges* are used to model data token communications between actors, realized by FIFO queues (First In First Out). In addition, weights on FIFOs, called delays, are used to represent initial data present on edges. The execution of a data flow actor is called *firing* and is triggered when an actor has sufficient data on each input edge. Table 1 lists modeling approaches and the graph semantics used in related works.

SDF (Synchronous Data Flow) [9] is a well-known static MoC. In SDF, a system is modeled with a data flow graph where the firing rules specify the constant token consumption and production rates for all actors. These constant rates introduce limitations in terms of algorithmic behavior representation.

CFDF (Core Functional Data Flow) [17] is a form of EIDF (Enable Invoke Data Flow) [18] where a limited set of modes influence token consumptions and productions. CFDF limits mode transitions to only one alternative, making the model deterministic.

BSP (Bulk Synchronous Parallel) [22], unlike SDF or CFDF, is a system modeling method rather than an application modeling method, and it has its own graph representation. In BSP, there are processing units with local memories connected over a router. Processing elements access each other's memories by remote access messages.

DAL (Distributed Application Layer) [20] has a dynamic mapping methodology. It employs Kahn process networks to explore application mappings and a finite state machine to represent execution scenarios. Multiple scenarios are precomputed at design-time and the suitable one is selected at run-time.

Bezati et al. [4] present a data flow modeling method according to the CAL language [5]. Their method has six steps. First, two different models for application and architecture are designed. Second, simulation and profiling results are collected. Third, the application is mapped to the architecture. Fourth, C++ and HDL codes are generated from CAL. Fifth, the code is compiled and synthesized. Finally, compiled code is executed.

LSLA [15] is a MoA, modeling hardware architecture separately from the MoC. The LSLA MoA includes Processing Elements (PE) and Communication Nodes (CN). PEs and CNs of the LSLA MoA have cost functions including parameters that may be retrieved from

Table 1 Modeling approaches

Method	Target	Application graph
SDF	Application	Dataflow
CFDF	Application	Dataflow
BSP	System	Non Dataflow
DAL	System	Non Dataflow
[4]	System	Dataflow
LSLA	Architecture	Not restricted to dataflow
GSLA	Architecture	Non dataflow

representative platform benchmarking. In that case, the calculated cost functions are obtained from measured application executions and the cost function parameters can be used to predict system efficiency for a set of comparable applications.

Holmbacka et al. [7] studied the energy consumption of different phases of the applications on multi-core CPUs. For utilizing the Dynamic Voltage and Frequency Scaling (DVFS) and Dynamic Power Management (DPM) of parallel platforms, they ran the parallel phases with as low as possible clock frequency on multiple cores without missing any deadline and sequential phases with higher clock frequency on a single core. For controlling the hardware features, they introduced two parameters including the level of parallelism and the quality of service and call it QP-aware (QoS and Parallel) strategy. Running a program with lower clock frequency instead of a race-to-idle strategy provides an energy efficient solution by reducing the frequent frequency switching overhead. Authors used PREESM [16] for compiling the applications and extracting the level of parallelism and deployed a non linear programming solver for the QoS handling. In addition, they presented a platform specific power model as a function of DVFS and DPM usage.

The energy efficiency survey [10] classifies the utilized techniques for improving GPU energy efficiency and compares them with methods applied to other computing units such as FPGAs. Authors use five categories including dynamic voltage frequency scaling (DVFS), division-based CPU-GPU, architectural techniques, dynamic workload variation and application-specific programming-level approaches. A conclusion to this work is that the power consumption of the GPU should be considered at multiple design phases with several techniques to achieve desirable efficiency. The proposed GSLA model falls under this objective, as it aims at making power estimates available early during the design phase.

The proposed work on the GSLA model provides a system modeling approach as an extension to LSLA. It has the benefits of a reduced modeling effort due to its re-usability. Table 1 summarizes the modeling approaches in order to compare to this work. As can be seen, our work is focusing on the architecture and supports wide range of applications.

3.1 Polynomial modeling comparisons

The power model developed in [7] uses Dynamic Voltage and Frequency Scaling (DVFS) and Dynamic Power Management (DPM) platform variables. Authors use Levenberg-Marquardt's algorithm and aim at a high modeling precision which resulted in a third degree polynomial with seven terms. As an MoA, GSLA is not specialized to power modeling and has a lower complexity with three terms, keeping its complexity minimal. The usage of the fidelity metric for evaluation results in a model with lower complexity differentiates this work

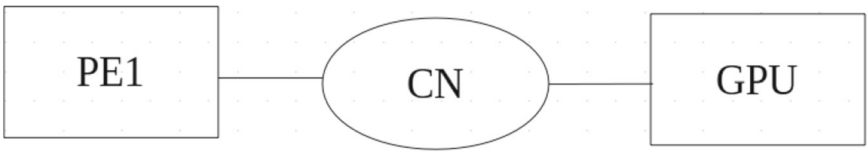

Fig. 2 Model of architecture

from studies using similar methodology, as the objective is not to have an accurate model but rather to take the right design decisions.

The energy model [6] presents the number of active cores and frequency as variables. Authors have considered three possibilities for the workload processing and depict three variations of their model. The variations of the terms number in their model is at least two and is impacted by the number of active cores. The proposed model is designed for static scheduling and requires timing data such deadlines and power consumption values of the active cores for predicting energy consumption. Compared to this tailored model, GSLA presents a simpler formula and we show that it still can capture several key performance indicators.

The energy per cycle model introduced in [12] uses normalized frequency variable. Authors use the Levenberg-Marquardt's algorithm for calculating their model equations which has three terms and it is in degree three. This model targets power and frequency data for the energy computation while the experiments are depicted for a limited set of measurements. On the other hand, GSLA with lower complexity is demonstrated as a two dimensional model fitting thirty six average measurement points.

4 Models of architecture

The MoA concept [15] is used to distinguish the processing architecture from the MoC, which should only address applications. Consequently, in this concept a system of an application running on a platform is presented with a MoC mapping on a MoA. A MoA is defined as a graph that in conjunction with the mapped MoC can be used for reproducible execution cost (time, energy, etc.) calculations. A MoA is designed for each specific processing architecture and it covers the processing elements and their interconnect. Figure 2 depicts an MoA which contains two processing elements and one interconnection.

Each element in the MoA graph has a cost function whose parameters can be estimated statistically according to measurement results of the mapped MoC element to the considered MoA element. The calculated parameter values depend on the application and the application configuration, but the search of the parameters is automated and requires only executable application(s) and test data from profiling.

4.1 Linear system level architecture

LSLA is a specific type of MoA that uses *linear* cost functions for each MoA graph element. The total cost of the modeled platform is calculated according to the Eq. 1, which depicts the total cost of application activity A on the LSLA graph P. In this equation, the total cost e.g. execution time cost is equal to the sum of the processing cost, and of the communication cost, λ being a scaling coefficient between processing and communication cost units. Tp

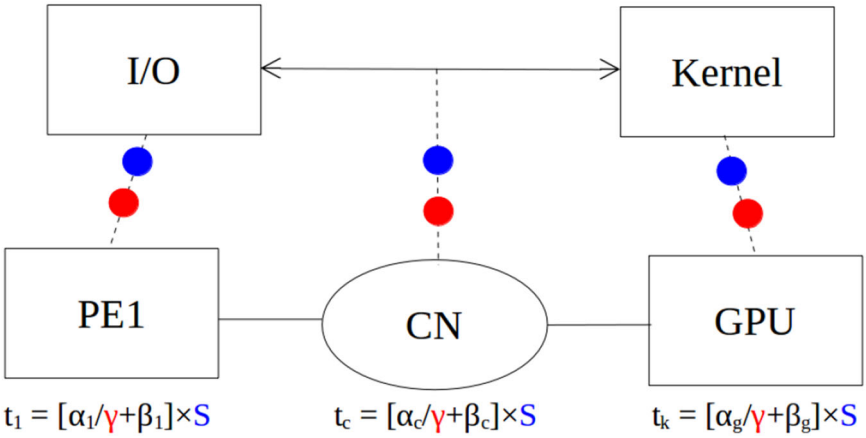

Fig. 3 Mapping of the application to the architecture model

depicts set of all mapped tokens to the processing elements and Tc shows set of all mapped tokens to the communication nodes. The activity of the mapped MoC is calculated as *tokens*, consisting of *quanta*, resulting in an affine cost model per communication and per processing. The quanta are an application-independent unit of execution cost.

$$cost(A, P) = \sum_{t \in Tp} cost(t, map(t)) + \lambda \sum_{t \in Tc} cost(t, map(t)) \tag{1}$$

In a system running an application with multiple dependent tasks on a platform with multiple processing elements, parallel application mapping and scheduling are required. While mapping refers to assigning tasks to processing elements, scheduling refers to ordering task execution on each processing element. On the modeling side, mapping an activity to a platform modeled with an MoA refers to the assignment of a unique processing element or communication node to each token in the application activity. The activity abstracts the *pressure* the application puts on hardware, resulting in physical ,properties such as time and energy consumption. In our experiments, GPU and CPU tasks are mapped manually, but the designed model can feed automated mapping processes.

In LSLA, the application and its activity (i.e. the pressure that it puts on hardware) are mapped as activity tokens to the LSLA model of the platform. Activity of the application includes *processing tokens* and *communication tokens*. These tokens are mapped to their associated elements in the platform model: processing tokens are mapped to processing elements and communication tokens are mapped to interconnection nodes that are used to transfer data between PEs.

5 GSLA: execution time modeling

This work adds GPUs that can be present in a modern heterogeneous platform. Figure 3 shows a simple GSLA graph that includes a CPU core *PE1*, the GPU, and the interconnection *CN* between the CPU core and the GPU. *PE1* is assumed to act as the host processor that communicates with the GPU. This simple GSLA model has three elements including two processing elements and one communication node.

Due to the very different characteristics of power and execution time modeling, we use different parameters and a slightly different model for both; the power model is presented in Sect. 8, whereas the execution time model is presented as follows: each element has its own cost function (presented beneath the nodes) that has two variables named γ and S, as well as two linear parameters α and β whose values are estimated for modeling purposes. One may note that the γ and S symbols refer to variables while bold characters represent the sets of values used during the profiling phase. The presented GSLA is used to model the execution time of the platform, thus time samples are used in parameter calculations. As presented in Equation 1, the total cost is a sum of all cost elements, i.e. the execution time of the GPU (t_k), the execution time of the host processor (t_1) and the execution time of the interconnect (t_c). The model of Eq. 2 is justified by the consideration that times (t_k), (t_1) and (t_c) do not overlap in time, i.e. the kernels of the GPU application are managed by the host device, then executed by the GPU during separate time intervals.

$$t_w = t_k + t_1 + t_c \tag{2}$$

$$t_k(\gamma, S) = (\alpha_g/\gamma + \beta_g) \times S \tag{3}$$

$$t_1(\gamma, S) = (\alpha_1/\gamma + \beta_1) \times S \tag{4}$$

$$t_c(\gamma, S) = (\alpha_c/\gamma + \beta_c) \times S \tag{5}$$

In these equations γ and S are variables, where γ is the parallelism factor, and S is the input data quantity. As it can be seen, increasing the parallelism factor γ decreases the total execution cost asymptotically. Each Eqs. 3, 4 and 5 follows an Amdahl's law [1] with β representing the incompressible time cost of a sequential section and α representing the compressible time cost of a perfectly parallel region. Conventionally, LSLA does not deal with internal processing element parallelism, which limits its usage to cores with limited concurrency. GSLA adds the parallelism factor γ that makes it possible to include parallel processing elements. For each GPU-related MoA graph entity (i.e., the GPU itself, the host PE and the interconnect) there are separate parameters α and β, where α can be regarded as the reciprocal of slope, and β as intercept. The Sect. 6 describes the proposed approach of estimating each α and β.

5.1 Usage of the methodology

Figure 4 depicts the tool flow steps for creating the cost prediction using the models either for performance or power. Applications are characterized with the **S** and $\boldsymbol{\gamma}$ sets, written in OpenCL source code and compiled for execution and measurements. The employment of the models is provided by a shell script and a Matlab script which is depicted in Fig. 4. First, the **S** and $\boldsymbol{\gamma}$ sets are selected and the shell script is edited accordingly. Then, the scripts perform the mapping, compilation and execution of the OpenCL application. The application receives S and γ as command line arguments. Pseudocode 1 presents an example of S and γ implementation in the matrix multiplication. Later, the Matlab script is used to extract the parameters α and β. Finally, the parameters are used to estimate the appropriate workload.

The parameters can be used for similar applications on equivalent platforms, or recalculated for other kind of platforms and applications.

Pseudocode 1 The S and γ Implementation in Matrix Multiplication

```
INPUT: S, γ
OUTPUT: execution time print
int main(int argc, char * argv[ ])  {
 ...
 size_t globalWorkSize[3];
 int parMtx = atoi(argv[1]);
 int globalWorkSize[0] = MTX_SIDE;
 int globalWorkSize[1] = MTX_SIDE;
 int globalWorkSize[2] = parMtx;
 ...
 int workSize = globalWorkSize[0] * globalWorkSize[1] * globalWorkSize[2];
 int totalLen = (atoi(argv[2]) * 256 * 4)/4;
 int iterations = totalLen/workSize;
 ...
 for(int i = 0; i < iterations; i + +){
 ...
 clEnqueueNDRangeKernel(commands, krnMatMul, 3, NULL, globalWorkSize, ...);
 ...
 }
 ...
}
```

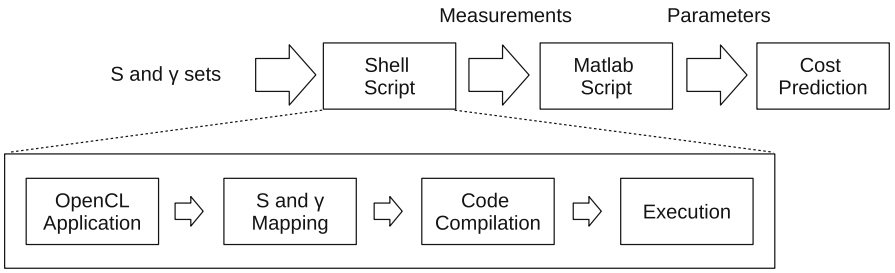

Fig. 4 Steps in the tool flow

6 Estimation of parameters

Acquiring an accurate execution time model for an application running on a GPU requires reliable profiling data. The proposed estimation approach assumes three accurate terms that can be profiled on the platform

- Application total *wall-clock time* t_w,
- Host code execution time t_1, and
- GPU kernel execution time t_k.

The remaining term t_c, in contrast, is derived using t_w, t_1 and t_k. The proposed procedure for acquiring accurate measurements for the terms are as follows: t_w is measured using the operating system clock, and t_k is read from the profiling data available from the GPU application programming interface. The measurement of t_1 is performed by modifying the application so that all GPU-related calls are disabled and the application only performs data I/O. Finally, t_c is derived from the other terms by subtracting t_1 and t_k from t_w.

The Pseudocode 2 presents the Matlab script used for modeling the performance of the applications according to the proposed Eqs. 2, 3, 4 and 5 . The $\mathbf{S}$ and $\boldsymbol{\gamma}$ sets have the same

number of elements and their values are considered according to the input data size of the applications and the memory of the platforms.

Pseudocode 2 Matlab Script of Performance Model

INPUT: $t_w = [...],\ t_k = [...],\ t_1 = [...]$
OUTPUT: $\alpha_g, \beta_g, \alpha_1, \beta_1, \alpha_c, \beta_c$

```
PerformanceModel {
 S = [...];
 γ = [...];
 tc = tw − t1 − tk;
 αg, βg = lsqlin(S, S/γ, tk);
 αc, βc = lsqlin(S, S/γ, tc);
 α1, β1 = lsqlin(S, S/γ, t1);
}
```

7 Experiments: execution time modeling

The experiments presented below serve to illustrate the suitability of the proposed model and Eqs. 2–5 for real-life GPU-equipped platforms. Typical signal processing applications were used as case studies: matrix multiplication, digital predistortion and Gaussian image filtering. The applications were written in OpenCL and were executed on two GPU-equipped platforms: the Odroid XU3 containing a Mali T628 GPU and a desktop workstation with the AMD RX 460 (Baffin) GPU. The test input data was randomly generated for matrix multiplication and predistortion, and private data for Gaussian filtering.

The α and β parameters of the cost functions were obtained with a Matlab script that invoked a least squares fitting algorithm (see Section 2 of [2]). In the Matlab script, the `lsqlin` function was used with a positive solution constraint as a standard method to perform linear regression.

Each application was profiled while varying two application variables, i.e., S and γ. For setting parameters values, the applications input data sizes, the memory of the platform, and the profiling duration have been considered. As the applications are simple, setting their data parallelism γ is straightforward and corresponds to the number of parallel fired kernels. Each obtained variable had six values where $S \in$ { 512, 1024, 2048, 4096, 8192, 16384 } and $\gamma \in$ { 8, 16, 32, 64, 128, 256}. The global work size of OpenCL applications was set application dependently. For matrix multiplication and predistortion, the work size set was calculated by 256*γ, while for gaussian filtering, it is 1024*γ. The reason for this variation is in the input data types i.e. gaussian filtering reads 1-byte data, while matrix multiplication and predistortion read 4-byte data items. For each (γ, S) combination the execution time was measured 10 times, giving a total of 360 samples per application/architecture combination.

7.1 Application-architecture mapping

In OpenCL, when computations are performed on a GPU, the CPU works as the host device that reads data from I/O, sends it to the GPU for processing, receives the computed result and stores it back to I/O. Based on the dataflow [9] MoC, a generic model for OpenCL applications was created. Data reading and writing of the CPU is mapped to an *I/O* node (see

Fig. 3). The *Kernel* node represents the computations performed on the GPU, whereas the communication between the *I/O* and *Kernel* nodes is presented with a bidirectional arrow in Fig. 3.

In each actor firing of the application graph, actors and the communication FIFO provide a token, which is mapped to their associated PE or CN node of the model. In other words, the tokens of the node *I/O* are mapped to the *PE1* architecture node, the tokens of *Kernel* are mapped to the *GPU* architecture node, and the communication FIFO tokens to the *CN* node. The cost functions shown below the architecture nodes have two variables, thus two tokens on the mapping lines in Fig. 3 are used to present the number of quanta for each variable.

7.2 Execution time results

This section shows how the proposed GPU execution time model fits with the measured execution time samples. In Figs. 5, 6 and 7 the bottom axes depict the variables S and γ, whereas the vertical axis depicts execution time. The dots represent the average of individual measured execution time samples.

The measured execution time samples are t_w (wall-clock time) values, and the mesh depicts the model-based sum of $t_k + t_1 + t_c$. For clarity, the measured time samples depict the average of the 10 measurements for each (γ, S) coordinate.

Table 2 depicts the calculated α and β parameter values for each application on Baffin and Mali GPUs. These parameters are used in the Eq. 2 for calculating t_w. The α value represents the cost of a token and equals to the slope of the mesh. The β value represents the constant time offset of the relevant GSLA element and is the t_w intercept of the mesh graph. Due to technical difficulties, values for the digital predistortion application were not acquired on the Mali platform. App 1 is matrix multiplication, App 2 is digital predistortion and App 3 is Gaussian filtering. M stands for Mali and B for Baffin platforms.

Table 3 demonstrates the fitting error between the model and measured samples for each application/platform combination as *fidelity* values. To highlight the improvement of the proposed GSLA model over conventional LSLA for GPU targets, Table 3 also shows the fidelity value for LSLA. Fidelity is computed similarly to [3] with the Kendall Tau Coefficient value, as calculated by the `corr` function of Matlab when configured for it. Fidelity assesses the capacity of the model to correctly order samples, 1 corresponding to a perfect order and -1 a perfectly reverse order. Indeed, a *good* model is a model that feeds good decisions more than a model with good absolute accuracy. A value of zero, as a worst case, would suggest independence in ranking between model and measurements. For computing the fidelity, the 360 execution time samples were randomly divided into a training set of 288 samples, and a test set of 72 samples.

In the Table 3 results, it can be seen that conventional LSLA yields considerably worse fidelity than the proposed GSLA for GPU architectures. The reason for this is evident: LSLA does not capture parallelism (γ), which is an integral part of GPU processing. An exception to this is the predistortion application on the Baffin GPU, where LSLA and GSLA yield almost identical fidelity. The reason for this is that on this platform, communication time dominates over parallelized kernel execution, making the whole application behave almost similar to a sequential application. Dominance of communication can be seen in Table 2 as the high value of coefficient α_c for application B2.

The measured fidelity values also show that the proposed non-linear GSLA model fits better the Mali platform than the Baffin platform. The difference is likely related to the

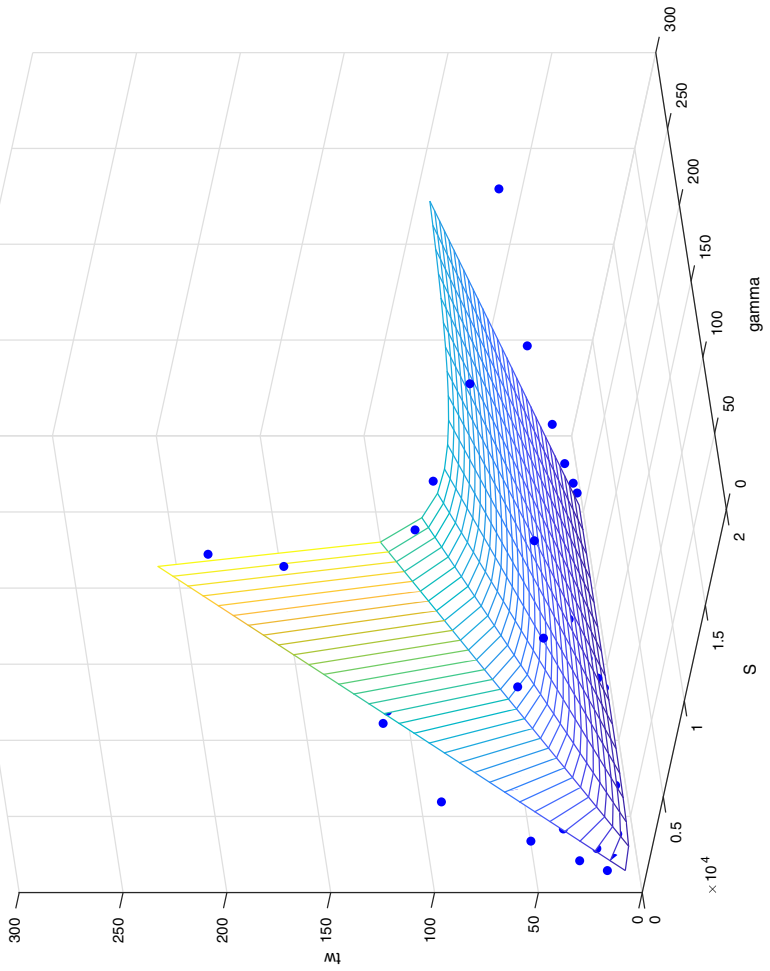

Fig. 5 Execution time of matrix multiplication on the Baffin GPU

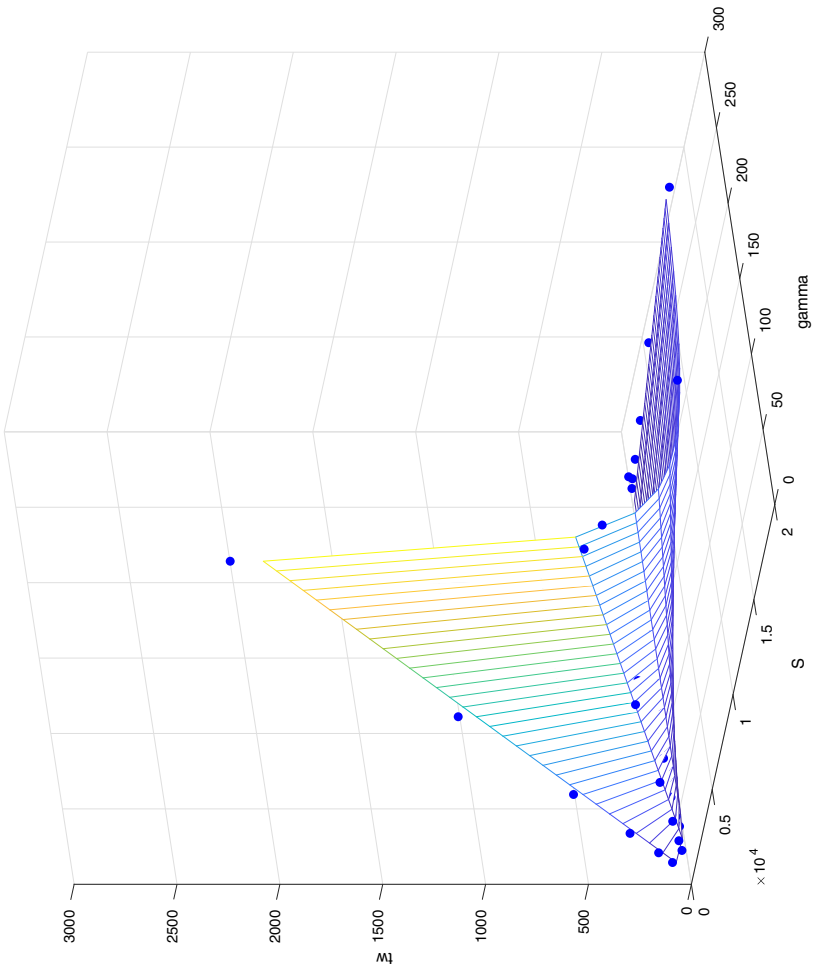

Fig. 6 Execution time of Gaussian filtering on the Baffin GPU

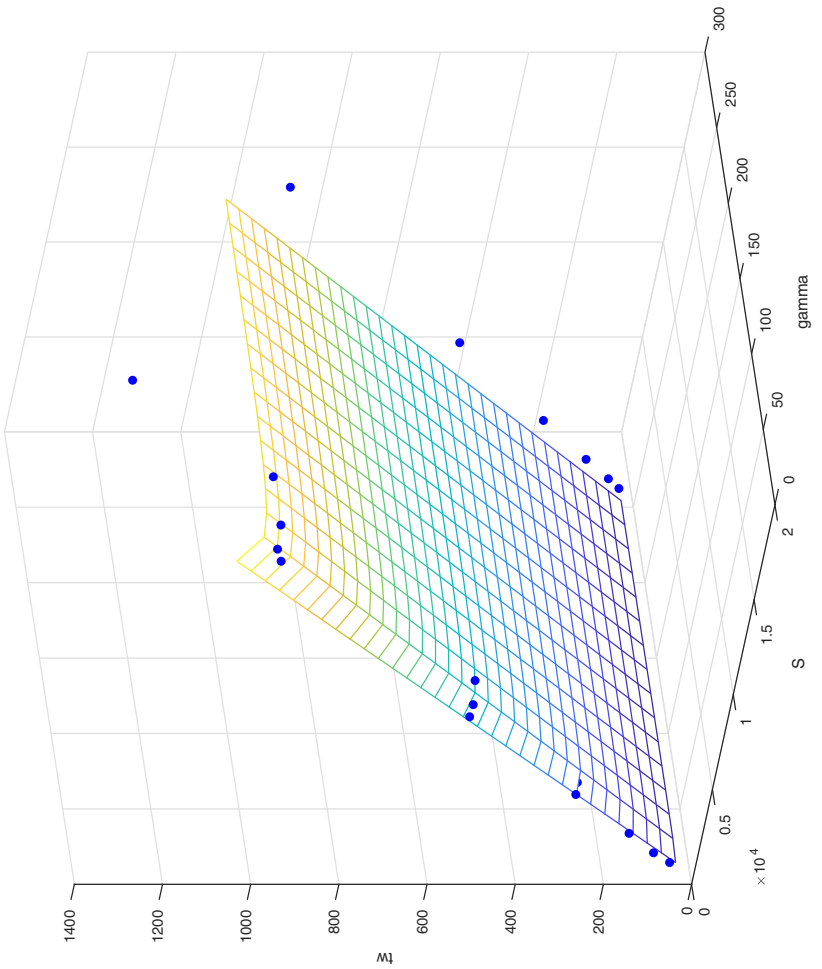

Fig. 7 Execution time of Predistortion on the Baffin GPU

Table 2 Calculated cost function parameters

App.	α_g	β_g	α_1	β_1	α_c	β_c
B1	0.001	0.008	0.000	0.004	0.005	0.068
M1	0.003	0.009	0.000	0.009	0.016	1.554
B2	0.005	0.049	0.002	0.003	0.060	0.000
B3	0.000	0.051	0.002	0.000	0.004	1.074
M3	0.003	0.023	0.022	0.000	0.082	0.460

Table 3 Fidelity of the test sets on execution platforms (Kendall tau coefficient between −1 and 1, 1 is the best)

Application	Platform	Fidelity GSLA (proposed)	Fidelity LSLA
1	Baffin	0.88	0.75
1	Mali	1.00	0.70
2	Baffin	0.90	0.92
3	Baffin	0.91	0.62
3	Mali	1.00	0.92

different memory architectures; Mali uses a shared memory between the CPU and the GPU, whereas the Baffin GPU is connected over PCI Express.

8 GPU power modeling

Besides GSLA for performance modeling, a power model is proposed for predicting the average power consumption of an OpenCL applications. The same Odroid XU3 platform is used for power profiling as it includes power sensors for CPU, GPU and memory. In our measurements we noticed almost constant CPU power which is expected from OpenCL applications running on GPU. Also, we ignored memory power consumption. Consequently, we only considered the GPU power dissipation as seen in Eq. 6. In this equation the p_t is the total power of the GPU.

In our experiments, we recognized that power consumption modeling with reasonable accuracy requires a third constant term in comparison to execution time modeling. In order to keep the complexity at reasonable levels, we tried to come up with the simplest possible model for capturing the GPU power. Consequently, the proposed power model has three parameters.

$$p_t(\gamma, S) = a_{GPU} + b_{GPU} S + c_{GPU} S/\gamma \tag{6}$$

As experiments, matrix multiplication and the Gaussian filtering are executed on the Mali platform and power values are read from the sensors. Profiling was similar to performance model i.e. with two application variables S and γ with the same values where $S \in \{512, 1024, 2048, 4096, 8192, 16384\}$ and $\gamma \in \{8, 16, 32, 64, 128, 256\}$. Figure 8 presents power modeling of matrix multiplication and Fig. 9 for Gaussian filtering. In both of these figures pt axis demonstrates only GPU power consumption. Table 4 shows the power model's parameters of these applications on the Mali GPUs. The values of the b_{GPU} and c_{GPU} are very small in comparison to other parameters. The a_{GPU} is almost constant at 0.12 representing a static

power consumption of 120 mW i.e. the intercept of equation and its larger values effect the total power consumption.

Table 5 depicts the fidelity of the proposed power model. These values could improve slightly with the cost of increasing the complexity of the model. For example, Eq. 7 was tested and rejected for the power model with R-squared value of 0.862 for matrix multiplication and 0.917 for Gaussian application, which does not justify the increased complexity.

$$p_t(\gamma, S) = a_{GPU} + b_{GPU} S + c_{GPU}/\gamma + d_{GPU} S/\gamma \tag{7}$$

The S and γ variables depicted in Eq. 6 show the input data quantity and parallelism factor. From the observations we noticed similar variable relations like the performance cost for the measured power samples. We observed that, logically, the input data quantity increases the power consumption while an increase of the parallelism reduces the power consumption. In addition, the Pseudocode 3 depicts the Matlab script of the power modeling according to the proposed Eq. 6. The values of the S and the γ sets are considered according to the memory capacity of the targeted hardware and the input data size of the applications.

Pseudocode 3 Matlab Script of Power Model

```
INPUT: p_t = [...]
OUTPUT: a_GPU, b_GPU, c_GPU
PowerModel {
 S = [...];
 γ = [...];
 a_GPU, b_GPU, c_GPU = lsqlin(p_t, [ones(S), S, S/]);
}
```

9 Fitness of the models

The proposed LSLA extension aims to model parallel execution on the GPU of the considered platform. With the following fitness study, we show that the created GSLA model is capable of fitting both GPU performance and GPU power consumption. The performance model shown in Eq. 8 and the power model presented in Eq. 6 have the same terms with exception of an extra constant term for Eq. 6. This constant logically models the static power of the platform while timing is null when no computation is requested. The fitness of the performance model and the presented power model are compared using the R-squared for the power measurements calculated with Matlab *regress* function. Table 6 depicts the fitness comparison of the models.

The R-squared values of 0.850 and 0.916 for Eq. 6 in comparison to values for the Eq. 8 justifies the selection of the Eq. 6 as the power model. The horizontal line has a better fit than Eq. 8 for power samples of the matrix multiplication application so the R-squared value. This suggests the requirement to add a constant value in Eq. 6.

The conclusion is that GSLA as defined by a sum of contributions obtained with Eq. 6, is capable of modelling both GPU power and performance with only 3 parameters.

$$t_w(\gamma, S) = \alpha\ S/\gamma + \beta\ S \tag{8}$$

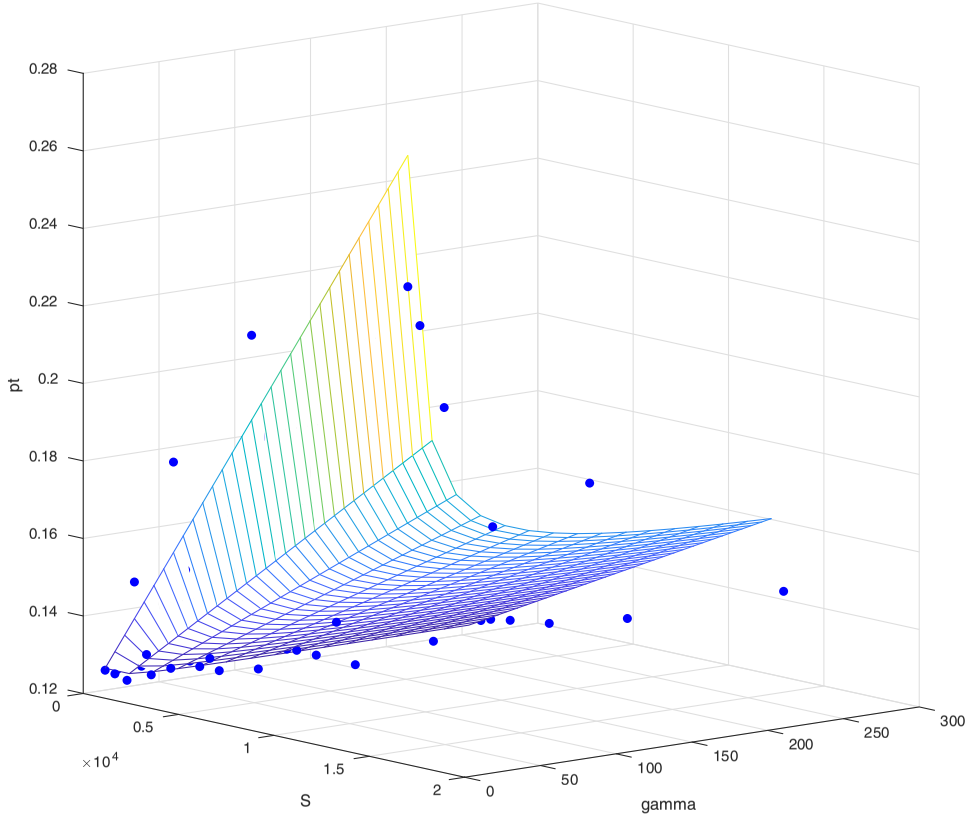

Fig. 8 Power modeling of Matrix multiplication on the Mali GPU (power in Watts)

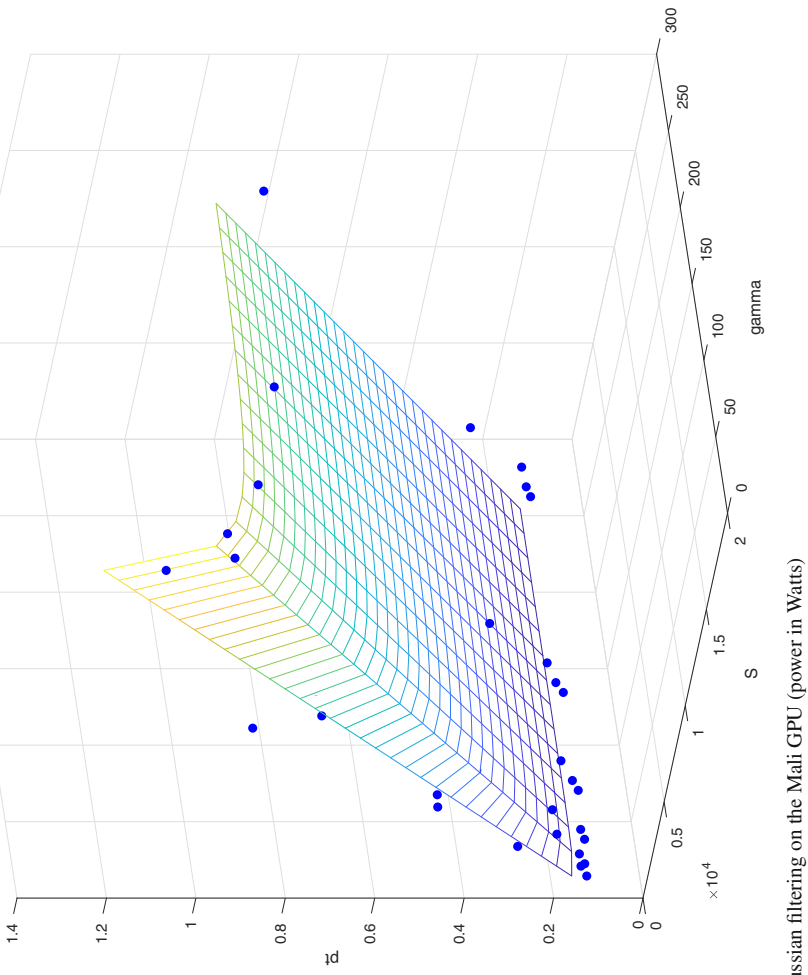

Fig. 9 Power modeling of Gaussian filtering on the Mali GPU (power in Watts)

Table 4 Cost function parameters of power modeling

Application	a_{GPU}	b_{GPU}	c_{GPU}
M1	0.121747	0.000003	0.000055
M3	0.121422	0.000052	0.000191

Table 5 Fidelity of the power model (Kendall tau coefficient between -1 and 1,1 is the best)

Application	Platform	Fidelity power model (proposed)	Fidelity LSLA
1	Mali	0.740	0.628
3	Mali	0.943	0.916

Table 6 Fitness of Models

Application	Equation	R-squared
1	8	-5.415
1	6	0.850
3	8	0.854
3	6	0.916

10 Conclusion

We presented a new Model of Architecture called GSLA (GPU-oriented System-Level Architecture). GLSA is tailored to GPU modeling but is capable of modeling both performance and average power of the targeted GPU. Contrary to the preexisting LSLA model, GSLA includes non-linear constructs, but reasonably fits the power consumption of a complex GPU with only 3 parameters. The validity of the proposed model is evaluated by profiling three OpenCL applications on two GPU-equipped platforms. The achieved model fidelity is 93% for execution latency and 84% for power. Such performances can be considered sufficient for design space exploration purposes.

In future, other lightweight machine learning techniques will be investigated for building models from platform measurements, expecially in more heterogeneous contexts combining e.g. CPU and GPU.

Acknowledgements This work was partially supported by the ITEA3 Project 16018 COMPACT and by the European Union Horizon 2020 research Grant 732105 CERBERO. We thank Jani Boutellier, Tapio Nummi, Antoine Morvan and Claudio Rubattu for their helpful guidance.

References

1. Amdahl GM (1967) Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18–20, 1967, spring joint computer conference, pp 483–485
2. Aster RC, Borchers B, Thurber CH (2018) Parameter estimation and inverse problems. Elsevier, Amsterdam
3. Bambha NK, Bhattacharyya SS (2000) A joint power/performance optimization algorithm for multiprocessor systems using a period graph construct. In: Proceedings of the 13th international symposium on System synthesis, pp 91–97. IEEE Computer Society
4. Bezati E, Thavot R, Roquier G, Mattavelli M (2014) High-level dataflow design of signal processing systems for reconfigurable and multicore heterogeneous platforms. J Real-Time Image Process 9(1):251–262
5. Eker J, Janneck J (2003) Cal language report. Technical Report, ERL Technical Memo UCB/ERL
6. Holmbacka S, Keller J, Eitschberger P, Lilius J (2015) Accurate energy modelling for many-core static schedules. In: 2015 23rd Euromicro international conference on parallel, distributed, and network-based processing, pp 525–532. IEEE
7. Holmbacka S, Nogues E, Pelcat M, Lafond S, Lilius J (2014) Energy efficiency and performance management of parallel dataflow applications. In: The 2014 conference on design & architectures for signal & image processing
8. Kienhuis B, Deprettere EF, Van der Wolf P, Vissers K (2001) A methodology to design programmable embedded systems. In: International workshop on embedded computer systems, pp 18–37. Springer
9. Lee EA, Messerschmitt DG (1987) Synchronous data flow. Proc IEEE 75(9):1235–1245
10. Mittal S, Vetter JS (2014) A survey of methods for analyzing and improving GPU energy efficiency. ACM Comput Surv (CSUR) 47(2):1–23
11. Moren K, Göhringer D (2018) Automatic mapping for opencl-programs on CPU/GPU heterogeneous platforms. In: International conference on computational science, pp 301–314. Springer
12. Nogues E, Pelcat M, Menard D, Mercat A (2016) Energy efficient scheduling of real time signal processing applications through combined DVFS and DPM. In: 2016 24th Euromicro international conference on parallel, distributed, and network-based processing (PDP), pp 622–626. IEEE
13. Payvar S, Boutellier J, Morvan A, Rubattu C, Pelcat M (2019) Extending architecture modeling for signal processing towards GPUS. In: 2019 27th European signal processing conference (EUSIPCO), pp 1–5. IEEE
14. Pelcat M, Desnos K, Maggiani L, Liu Y, Heulot J, Nezan JF, Bhattacharyya SS (2016) Models of architecture: reproducible efficiency evaluation for signal processing systems. In: IEEE international workshop on signal processing systems (SiPS), pp 121–126. IEEE
15. Pelcat M, Mercat A, Desnos K, Maggiani L, Liu Y, Heulot J, Nezan JF, Hamidouche W, Ménard D, Bhattacharyya SS (2017) Reproducible evaluation of system efficiency with a model of architecture: from theory to practice. In: IEEE transactions on computer-aided design of integrated circuits and systems
16. Pelcat M, Piat J, Wipliez M, Aridhi S, Nezan JF (2009) An open framework for rapid prototyping of signal processing applications. EURASIP J Embed Syst 2009:11
17. Plishker W, Sane N, Kiemb M, Anand K, Bhattacharyya SS (2008) Functional DIF for rapid prototyping. In: The 19th IEEE/IFIP international symposium on rapid system prototyping, 2008. RSP'08, pp 17–23. IEEE
18. Plishker W, Sane N, Kiemb M, Bhattacharyya SS (2008) Heterogeneous design in functional dif. In: International workshop on embedded computer systems, pp 157–166. Springer
19. Rigo S, Araujo G, Bartholomeu M, Azevedo R (2004) ArchC: A systemC-based architecture description language. In: 16th Symposium on computer architecture and high performance computing, pp 66–73. IEEE
20. Schor L, Bacivarov I, Rai D, Yang H, Kang SH, Thiele L (2012) Scenario-based design flow for mapping streaming applications onto on-chip many-core systems. In: Proceedings of the 2012 international conference on compilers, architectures and synthesis for embedded systems, pp 71–80. ACM
21. Thiele L, Bacivarov I, Haid W, Huang K (2007) Mapping applications to tiled multiprocessor embedded systems. In: Seventh international conference on application of concurrency to system design (ACSD 2007), pp 29–40. IEEE
22. Valiant LG (1990) A bridging model for parallel computation. Commun ACM 33(8):103–111