GPU Auto-tuning Framework for Optimal Performance and Power Consumption

Sunbal Cheema and Gul N. Khan

Department of Electrical, Computer and Biomedical Engineering, Toronto Metropolitan University, Toronto ON CANADA, sunbal.cheema@torontomu.ca, gnkhan@torontomu.ca

ABSTRACT

An auto-tuning framework for GPU devices is presented for tuning application kernels of OpenCL. The GPU tuner employs multi-objective optimization methodology to improve the performance and power consumption of applications. It efficiently explores a user defined solution space comprising of possible tunable algorithmic and hardware counter variations through code transformations. The methodology targets GPU code tuning situations where performance and energy consumption are critical. The proposed framework is evaluated for 2D convolution kernels. It utilizes a non-dominated sorting Genetic Algorithm with hardware power sensor data for application code transformation through code rewrite and validation. Various algorithmic variations such as loop unrolling, caching, workgroup size and memory utilization are applied. The final pareto optimal configurations code utilized around 30% less power and 4% faster execution time. The analysis shows the convergence of optimization, and 45% improvement in standard deviation.

KEYWORDS

Auto-tuning, Code transformation, Multi-objective optimization, GPU code regeneration, Performance power optimization

1 INTRODUCTION

According to the world economic forum, the technology sector will be consuming 20% of the produced electricity by the year 2025 [1]. Portable, and sustainable (power consumption efficient) computing is crucial for GPU-based technology and research. Power consumption has also become a bottleneck for many embedded and mobile platforms. Automatic code transformation that provides better performance and power efficiency of computing is of prime importance. As the CPU-GPU devices have improved over the years, built-in sensors can be utilized to steer the optimization process to a global optimal solution. Our research focuses on tuning and validation based on the hardware counters and with feedback tuning by using power measurement sensor data. Salient features of our proposed framework are listed below.

- Multi-objective auto-tuning (Pareto Optimal Points)
- Integrated power and time calculation
- Online and offline auto-tuning
- User defined parameters and constraints

Various researchers have adopted different code tuning methodologies such as iterative methods [2], differential evolutions [3], machine learning [4, 5], empirical approaches [6, 7], and open-source exploration tools [8].

1.1 GPU Code Tuners

Lin et al. present an auto-tuner involving execution time and energy consumption estimation model that employs simulated annealing or Genetic Algorithm (GA) to determine an optimal configuration for either power or performance [9]. OpenTuner is a generic tuner with a user defined cost function, which generates the search space based on the user defined parameters [10]. It provides several search space optimization techniques such as particle swarm optimization, random search, etc. OpenTuner employs multi-objective optimization, however multi-objective related details are not reported.

CLTune has been presented as a generic OpenCL and CUDA kernel tuner [11]. It uses design search space exploration strategies like particle swarm optimization, simulated annealing, random and full search. ATF has been reported as a generic auto-tuner, which is claimed to provide tuning of OpenCL kernels for optimal performance and/or power consumption [12]. Since there are no pareto optimal points reported, it is difficult for the user to make decisions regarding kernel configurations. Kernel Tuning Toolkit (KTT) is a kernel tuner that has been developed on the concept of CLTune [13]. The Periscope Tuning Framework provides tuning plugins for OpenCL kernels related to CPU, GPU, and Xeon Phi coprocessors [14].

1.2 Power Estimation

Optimizing the power utilization of heterogenous systems is of critical importance. One can find several studies regarding the GPU profiling of power utilization using internal or external sensors [15, 16]. CPUs related power utilization models show some degree of convergence, but the GPU power utilization models are not matured and recognized yet. CPU power is mostly measured using a linear model however, similar practice is not valid for GPUs. Some researchers argue that statistical linear regression models are more appropriate for GPUs [17]. Song et al. claim that these models are inadequate to monitor the intricacies of GPU [18]. They improved linear regression model by utilizing tools such as NVIDIA NVML and CUPTI [19, 20]. Adhinarayanan et. al. applied instantaneous power utilization and averaging to get the final value [21]. In the past, power measurement models were calculated with external sensors. However, recent GPUs have on board power sensors that provide instantaneous power measurement [22].

2 MOKAT FRAMEWORK

Our tuning framework for OpenCL kernels provides multi-objective optimization for performance and power utilization. A search space comprising of user defined tunable variables is created. An elitist Non-dominated Sorting Genetic Algorithm (NSGA-II [23]) in conjunction with the integrated power calculation is employed to compute a pareto optimal solution set. The proposed auto tuning framework, MOKAT (Multi-Objective Kernel Auto-Tuner) is depicted in Figure 1.

Kernel optimization is a combinatorial optimization problem, which is a process of searching the maximum or minimum value from a function. Software tuning and code regeneration is a multi-objective combinatorial optimization problem that strives to find the optimal solution in all possible configurations [24]. The search space is discrete and the resulting pareto optimal solution will not be necessarily a continuous function. In this paper, the quality of the resulting solution set is compared by applying multiple GPU specific objective functions (time, power, energy, etc.) to be optimized. Each objective function will be either minimized or maximized. MOKAT minimizes both objective functions of power utilization and execution time.

2.1 Search Space Creation

Search space is created by iterating through all possible values of the given variables. Constraints are applied to prune the search space and remove invalid configurations. For 'n' number of variables, each variable V_k has V_k^x number of possible values and the number of possible configurations would be: $V = \{V_1, V_2 ... V_n\}$

$$Possible\,Configurations = \left(\prod_{1}^{n} V_k^x \right) \quad (1)$$

For 'm' number of constraints on variables, the number of constraint values are C_l^x: $C = \{C_1, C_2, ... C_M\}$

$$Search\,Space(X) = \left(\prod_{1}^{n} V_k^x \right) - \left(\prod_{1}^{m} C_l^x \right) \quad (2)$$

2.2 OpenCL Kernel

OpenCL offers functional portability in terms of its execution on different devices; however, it does not offer performance portability. When a tuned code achieves high performance on one device then executing the same code on another device may often perform poorly. Therefore, the code must be re-tuned for each new device before its execution. The problem of performance portability is not just related to OpenCL. This issue arises when we try to transfer a code from one device to another of a different generation or vendor. However, this problem is aggravated with OpenCL as it is designed for a greater range of heterogenous devices with different architectures.

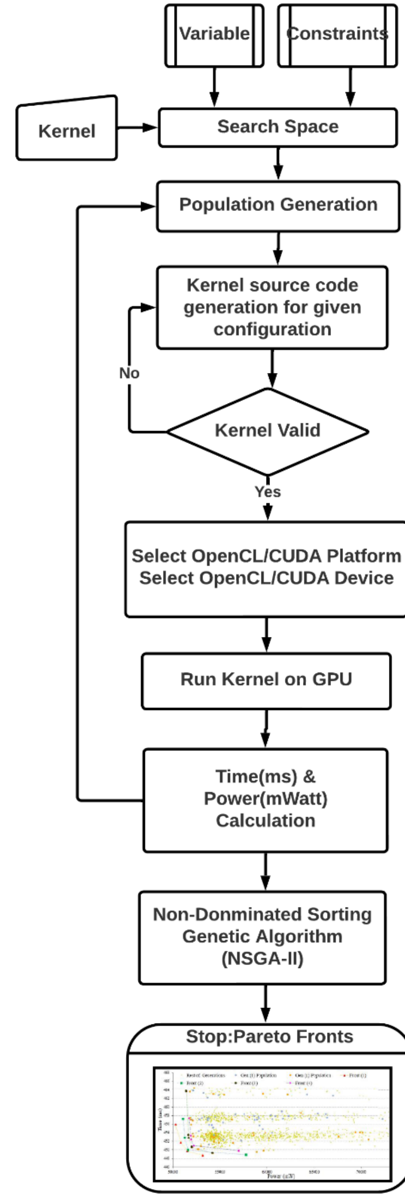

Figure 1: MOKAT Framework

2.3 Power Calculation

MOKAT utilizes the NVIDIA NVML [19] to acquire the internal sensor values of power consumption for GPU utilization above zero. It obtains readings at 10msec intervals and uses the average of these readings during the execution of kernel. Power utilization for high performance computer (HPC) systems is determined either through external or internal techniques. Many researchers have employed internal sensors for power calculations. Ferro and others have used GPU sensors [25], and Schöne et al. have calculated energy with the power measured during a given time interval [26].

2.4 Non-dominated Sorting Genetic Algorithm

Non-dominated Sorting Genetic Algorithm (NSGA-II) is an efficient algorithm featuring diversity preservation mechanism [23]. Initial parent population $popP_0$ of size N is created from the X search space created by the OpenCL kernel configuration. Child population $popQ_0$ of size N is created by crossover and mutation of the parent population. The two populations are joined into a 2N size of $popR_0$ population.

$$popR_k = popP_k \cup popQ_k \quad where: k = 1,2,..,gen \qquad (3)$$

Non-dominated sorting is applied onto the $popR_0$ population to get the non-dominated Fronts i.e., $F_1, F_2, ..F_n$.

$$popP_k = F_1 \cup F_2 \cup, ..F_n \quad where\ size(F_1 \cup F_2 \cup, ..) = N \qquad (4)$$

After the non-dominated sorting operation, the Fronts are joined to form the $popP_k$, where $k = 1, 2, .., gen$. We applied crowding sort and selected the configuration using crowding distance technique given in Figure 2. After creating the new $popP$, mutation and crossover processes are applied to get the child population of size N. The Genetic Algorithm is implemented by employing real variables. Genetic operators affect the performance of evolutionary process. We utilized single point crossover and random gene mutation process. Power utilization and execution time of the kernel is used to evaluate the final dominance.

Crowding Distance Assignment:

Step 1: l is the number of solutions in F. Assign $d_l = 0$ for all solutions.

Setp 2: Sort the set in the worst order for each objective function f_m $m = 1,2,3,..M$

Step 3: For $m = 1,2,3,..M$, assign a large distance to the boundary solutions, or $d_{I_1^m} = d_{I_l^m} = \infty$ and all other solutions like j=2 to j=$l-1$ assign:

$$d_{I_j^m} = d_{I_j^m} + \frac{f_m^{I_{j+1}^m} - f_m^{I_{j-1}^m}}{f_m^{max} - f_m^{min}}$$

Figure 2: Crowding Distance Technique

3 CASE STUDY: CONVOLUTION KERNEL

Since the advent of GPUs, their usage has increased continuously, and many data-intensive applications are also being executed by GPUs. GPUs are not limited to supercomputing and researchers in every field are using them for operational intensive applications such as neural networks. For that reason, research has been done on how deep learning applications run efficiently on HPC systems. Convolution is the most common operation in neural networks. We have employed the highly tunable convolution in Werkhoven's work for convolution kernel tuning [27].

Description of all the tunable parameters of 2D convolution kernel are given in Table 1. The total amount of work is divided into workgroups of size TBX and TBY, where TBX and TBY are the workgroup X-axis and Y-axis dimensions. WPTX and WPTY are defined as the work per thread. The architecture of the GPU dictates what is the best possible combination of work per thread in both dimensions. A small value of work per thread can cause a higher communication overhead. Vector width, V is either less or equal to work per thread value. It is applied only when the LOCAL in terms of cache is at level 2 (see Table 1). UF (unrolling factor) value depends on the filter size, which is either enabled or disabled for loops. PD defines the local memory padding. The parameter LOCAL is for three caching stages: no local memory, every thread caches the value at its coordinates (x, y), and helper threads are launched. We analyzed the 2D convolution kernel for 3424 possible number of configurations. We used the filter size of 7x7 for an image size 8192x4096. The results show that the process varies by analyzing the MOKAT results in different perspectives and explained in the next section.

Table 1: Tunable Variables of 2D Convolution Kernel

Variable Name		Detail	Possible Values
TBX, TBY		Workgroup size	8, 16, 32, 64
WPTX,WPTY		Work per Thread	1, 2, 4, 8
LOCAL		Caching	0, 1, 2
V		Vector Width	1, 2, 4
UF		Loop Unrolling Factor	1, Filter Size
PD		Padding	0, 1

3.1 Effect of NSGA-II Parameters

We evaluated MOKAT for power and performance by selecting different values of GA parameters for 2D convolution as shown in Table 2. For ease of usage, the GA parameters can be entered as runtime variables. User can input a list of various GA parameters such as number of generations (Gen), population size (Pop), mutation probability (MP) and crossover probability (CP) [28]. MOKAT is evaluated in detail for eight variations of the NSGA-II parameters. We have summarized Eight cases in Table 2. It can be concluded that higher mutation probability adds diversity to the solution. Therefore the algorithm needs a large number of generations for converging to optimal configuration as for Case 5.

It can also be observed in Case no 4 that by having lower mutations and crossover probabilities will result in fewer number of pareto optimal points, as it does not have diversity. For optimal results one should employ the following steps in the selection of optimal GA parameters for the auto-tuner.

- Select a population size and number of generations relative to each other and the search space size.
- For the search space X, MOKAT should iterate at least 30% of X i.e., X_n ≥ 30% of X, where X_n is the total number of configurations executed on the GPU.

For NSGA-II, if the generation size is 30 it will iterate through 60 configurations for the 1st generation. The number of configurations that will be iterated (X_n) can be calculated by equation (5).

$$X_n = (Pp * 2) + (Pop * (Gen - 1)) \quad (5)$$

Table 2: NSGA-II Optimization Parameters and Pareto-Optimal Points with Kernel Execution time, Power & Energy.

Case	Gen[a]	Pop[b]	CP[c]	MP[d]	[Power] [Time] [Energy]	2D Convolution First Pareto Optimal Front's Points[1]					
						1	2	3	4	5	6
1	30	30	0.5	0.5	[mW]	51073	52539	52682	52745	53425	55352
					[s]	453.2	452.68	451.4	450.62	449.23	448.96
					[kJ]	23.15	23.78	23.78	23.77	24.00	24.85
2	30	30	0.5	0.8	[mW]	50817	51574	51789	52789	55245	
					[s]	457.02	451	450.96	449.09	448.97	
					[kJ]	23.22	23.26	23.35	23.71	24.80	
3	30	30	0.8	0.5	[mW]	51882	52096	54327			
					[s]	450.84	449.88	448.8			
					[kJ]	23.39	23.44	24.38			
4	30	30	0.2	0.2	[mW]	51487	51820				
					[s]	451.63	449.13				
					[kJ	23.25	23.27				
5	30	30	0.8	0.8	[mW]	51487	51970	52141			
					[s]	450.61	449.74	448.73			
					[kJ]	23.20	23.37	23.40			
6	30	50	0.5	0.5	[mW]	51591	51757	53901	54543		
					[s]	456.63	450.15	449.77	448.7		
					[kJ]	23.56	23.30	24.24	24.47		
7	50	30	0.5	0.5	[mW]	50293	50816	51522	53161		
					[s]	455.91	451.74	449.77	448.68		
					[kJ]	22.93	22.96	23.17	23.85		
8	50	50	0.5	0.5	[mW]	51102	51447	51744	52031		
					[s]	454.71	453.52	450.91	449.63		
					[kJ]	23.24	23.33	23.33	23.39		

Whereas: a) Gen: Generations, b) Pop: Populations, c) CP: Crossover Probability, d) MP: Mutation Probability. Note 1: The points sorted in order of increasing power.

Convolution search space is not exceptionally large, and we have taken equation (5) into consideration. Therefore, the population size becomes a don't care condition if it is not significantly lower than the requirement. The number of generations will affect the quality of pareto points providing the most optimal solution.

3.2 Quality of Optimal Configurations

The best pareto front is for Case 7 as shown in Figure 3. It has pareto-points with the lowest energy and execution time. The main reason is the higher number of generations, which provide enough time for the algorithm to converge to the near global optimal configurations. Although the number of generations (Gen) for Case 7 and 8 are equal, the quality of Pareto-optimal points for Case 7 are significantly better than Case 8. Case 7 has lower population size that reduces the chances of redundant exploration, and the algorithm converges to an optimal solution. A detailed search space graph for Case 7 is depicted in Figure 4.

3.3 Case 7: The Best Reported Solution Set

The details of the best solution i.e. Case 7 is provided in Figure 4. All the best possible fronts of the final generation are drawn to understand the algorithm convergence towards the best solution. All the points in the objective space, which are considered by the auto-tuner during the tuning process can be observed in Figure 4. All the configurations in the objective space with respect to power and time is depicted in Figure 4 providing a clear insight. It shows how the first population of Gen 1 represented by purple dot (.) is spread in the objective space from top right (the worst) to bottom left (the most optimal area for minimization). Last Generation is also shown by the orange dot (.) that converges to the bottom-left area after 50 generations. The best and globally most optimal Pareto-front is represented by an orange triangle shaped legend, indicating the best possible configurations recorded from all the seven cases. Table 3 provides the details of all the variables for the most optimal configurations given by equation (1). After considering the kernel configurations from 1st generation of Case 7 and few Pareto optimal configurations, we noticed a 30% reduction in power and 4% in execution time.

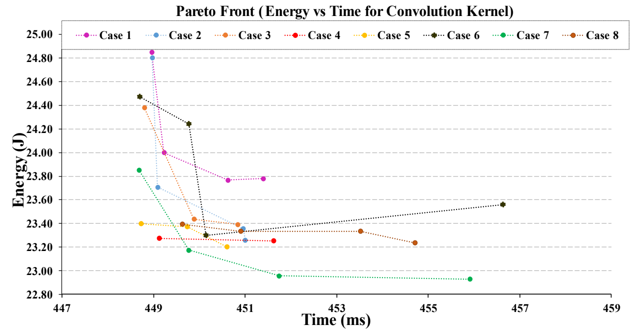

Figure 3: Time *vs* Power for the Pareto Optimal Fronts (for Table 2 Data)

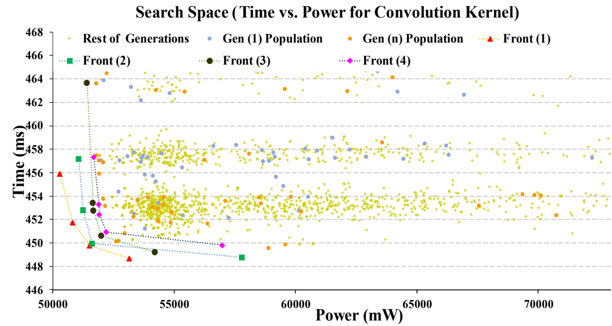

Figure 4: Generation 50, Population 30, Crossover Probability 0.5 & Mutation Probability 0.5 for Tesla K20c

3.4 GPU Tuning Variable Values

Table 3 lists the optimization variable and tunable parameter values regarding the Pareto-optimal solutions for all the eight cases. Case 7 (see Table 2 and Table 3) is selected for detailed discussion on the impact of tunable variables. It can be observed from Table 2 and Table 3 that the Pareto-point 4 of case 7 has minimum execution time, and it has the highest power/energy as compared to other Pareto-points. Pareto-point 4 has a workgroup size X:Y equal to 32:8 and all the threads use cache to store their coordinate values. The workgroup per thread values (X:Y) are (2:4) and a vector width of 2, where loop unrolling factor is 7. Considering all the pareto-fronts' values, it can be concluded that the higher values of the workgroup size do not always lead to the most reliable solution. Increasing the number of works per thread and workgroup size will add latency to access the global memory. In other words, GPU occupancy is limited by the registers available, local memory bandwidth and the hardware scheduler. The variation in the parameter values for Case 7 suggests a workgroup size of 8x32, 32x8, or 16x16 (a total of 256) as seen in Table 3. It can also be observed that for smaller workgroup size, the work per thread value is higher for keeping the GPU occupancy at a fair level. This is to compensate for the work loss. Points 3 and 4 have low execution times and both have utilized the

loop unrolling of 7 (filter size). The energy difference between the best and the worst point is 0.92kJ. It can be concluded that the four Pareto-points are the best choices.

Table 3: Best Pareto-Optimal Variable/Parameter Values 2D Convolution Kernel.

Case No.	Pareto Point No.	Tuning Variables Values									
		TBX[1]	TBY[2]	LOCAL[3]	WPTX[4]	WPTY[5]	V[6]	UF[7]	PD[8]	TBX_XL[9]	TBY_XL[10]
1	1	8	32	2	2	4	2	7	0	11	34
	2	16	16	0	4	2	1	1	0	16	16
	3	16	32	2	4	1	2	7	0	18	38
	4	8	8	1	8	2	2	1	1	8	8
	5	32	32	1	1	8	1	7	1	32	32
	6	64	16	0	1	8	1	1	0	64	16
2	1	8	8	2	2	4	2	7	0	11	10
	2	8	32	1	8	2	4	7	0	8	32
	3	64	8	2	2	4	1	7	1	67	10
	4	32	32	0	2	4	1	1	0	32	32
	5	32	8	0	8	1	1	1	0	32	8
3	1	8	32	1	8	2	4	1	0	8	32
	2	64	16	1	8	1	1	7	0	64	16
	3	8	16	2	2	2	1	7	1	11	19
4	1	32	32	1	1	8	1	7	0	32	32
	2	16	32	0	4	4	1	7	0	16	32
5	1	64	8	0	2	4	2	7	0	64	8
	2	8	16	1	2	8	1	7	0	8	16
	3	8	16	2	2	4	2	7	1	11	18
6	1	32	32	1	4	2	2	7	1	32	32
	2	8	8	0	4	2	2	7	0	8	8
	3	16	8	2	2	1	2	7	0	19	14
	4	8	8	1	4	1	2	7	0	8	8
7	1	8	32	2	8	1	2	1	0	9	38
	2	8	8	0	2	8	1	1	0	8	8
	3	16	16	1	8	4	1	7	1	16	16
	4	32	8	1	2	4	2	7	0	32	8
8	1	8	8	0	4	2	4	7	0	8	8
	2	8	8	0	4	4	4	7	0	8	8
	3	32	32	0	4	8	4	1	0	32	32
	4	8	8	1	8	1	1	7	0	8	8

Where as: 1) Work Group Size X 2) Work Group Size Y 3) Caching Strategy 4) Work Per Thread X 5) Work Per Thread Y 6) Vector Width 7) Loop Unrolling Factor 8) Padding 9) Work Group Size X- XL 10) Work Group Size Y-XL.

We can infer from the results shown in Table 3 that the best workgroup size is multiple of 256, and caching value '1' is better if the workgroup size is not less than 256. Loop unrolling factor at its best is 7 as it helps to improve the performance without significantly affecting the power. The higher values of TBX_XL and TBY_XL needs a caching value of 2, which has always increased the power values of the kernel.

3.5 Probability Distributions

A detailed analysis of Case 1 to Case 8 is depicted in Figure 5 and Figure 6 to show the probability distribution with respect to Power (Left Green) and Time (Right Blue). Figure 5 shows the probability distribution of the first generation for all the cases and Figure 6 provides the probability distribution for the final generation of the eight cases. It can be observed that the execution time has relatively less converging trend, but the probability distribution for power indicates a positive trend towards low power utilization in the final generation. We have discussed the best case (Case 7) earlier where Figure 4 indicates the power reduction by 30% and execution time reduction by 4%. The power and time measurments are summed up and standard deviation is calculated for the overall 1st and last population. After calculating the best case scenerio (i.e. Case 7), 45% reduction in standard deviation is observed that means the algorithm converges to the most optimal solution.

4 CONCLUSIONS

The proposed GPU code tuner is evaluated for 2D convolution kernel (test/benchmark problem) on a Tesla GPU. A detailed account of optimal configuration sets for the benchmark kernel is provided and discussed including their execution time and average power consumption. It is concluded that the unroll factor speeds up the process without affecting the power values. The energy consumption of a kernel and its power utilization shows a direct relation to each other as indicated in Table 2. Our framework provides optimal diverse solution for appropriately selected GA parameters. The most promising contribution of this research is that it provides a powerful, flexible, and generic GPU code tuning tool for designing energy efficient high performance computing applications without compromising computational performance. The HPC design engineer can systematically select the optimal configuration depending on the application's objective priority.

ACKNOWLEDGEMENT

Authors acknowledge the financial support from FEAS, Toronto Metropolitan Univ., and GPU-based HPC Systems from CMC.

REFERENCES

[1] N. Jones. 2018. How to stop data centres from gobbling up the world's electricity. *Nature News Feature*, 12 Sept. 2018. [Online]. Available: https://www.nature.com/articles/d41586-018-06610-y [Accessed: Feb. 2023].

[2] J. F. Fabeiro, D. Andrade, and B. B. Fraguela. 2013. OCLoptimizer: An Iterative Optimization Tool for OpenCL. *Procedia Computer Science*, vol. 18, pp. 1322–1331.

[3] H. Jordan et al. 2012. A multi-objective auto-tuning framework for parallel codes. *Int. Conf. on High Performance Computing, Networking, Storage and Analysis*, pp. 1-12.

[4] A. Magni, C. Dubach, and M. O'Boyle. 2014. Automatic optimization of thread-coarsening for graphics processors. *23rd Int. Conf. on Parallel Architectures and Compilation*, pp. 455-466.

[5] T. L. Falch and A. C. Elster. 2015. Machine Learning based Auto-Tuning for Enhanced OpenCL Performance Portability. *IEEE International Parallel and Distributed Processing Symposium Workshop*, pp. 1231-1240.

[6] J. Fang, H. Sips, P. Jaaskelainen, and A. L. Varbanescu. 2014. Grover: Looking for Performance Improvement by Disabling Local Memory Usage in OpenCL Kernels. *Int. Conf. on Parallel Processing*, 2014, pp. 162-171.

[7] S. Hirasawa, H. Takizawa, and H. Kobayashi. 2015. A Verification Framework for Streamlining Empirical Auto-Tuning. *3rd Int. Symp. on Computing and Networking*, pp. 508-514.

[8] E. Paone et al. 2015. Customization of OpenCL Applications for Efficient Task Mapping under Heterogeneous Platform Constraints. *Design, Automation and Test in Europe Conference & Exhibition* (DATE), pp. 736-741.

[9] C. Lin, S. Teng, and P. Hsiung. 2016. Auto-tuning for GPGPU applications using performance and energy model. *J. of Systems Architecture*, vol. 62, pp. 40-53.

[10] J. Ansel et al. 2014. OpenTuner: An extensible framework for program autotuning. *International Conference on Parallel Architecture and Compilation Techniques (PACT),* pp. 303-315.

[11] C. Nugteren and V. Codreanu. 2015. CLTune: A Generic Auto-Tuner for OpenCL Kernels. *IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, Turin Italy, pp. 195-202.

[12] A. Rasch, M. Haidl, and S. Gorlatch. 2017. ATF: A Generic Auto-Tuning Framework. *IEEE 19th International Conference on High Performance Computing and Communication*, pp. 64-71.[13] F. Petrovič et al. 2020. A benchmark set of highly-efficient CUDA and OpenCL kernels and its dynamic autotuning with Kernel Tuning Toolkit. *Future Generation Computer Systems*, vol. 108, pp. 161–177.

[14] R. Mijaković, M. Firbach, and M. Gerndt. 2016. An architecture for flexible auto-tuning: The Periscope Tuning Framework 2.0. *2nd International Conference on Green High Performance Computing* (ICGHPC), pp. 1-9.

[15] R. A. Bridges, N. Imam, and T. M. Mintz. 2016. Understanding GPU Power: A Survey of Profiling, Modeling, and Simulation Methods. *ACM Computing Surveys*, vol. 49, no. 3, pp. 41:1–41:27.

[16] Terpstra, H. Jagode, H. You and J. Dongarra. 2009. Collecting performance data with PAPI-C. In *Tools for High Performance Computing*, M. Müller, M. Resch, A. Schulz and W. Nagel, Eds., Springer, Berlin Heidelberg, pp. 157–173.

[17] H. Nagasaka et al. 2010. Statistical Power Modeling of GPU Kernels Using Performance Counters. *IEEE Int. Conf. on Green Computing*, pp. 115–122.

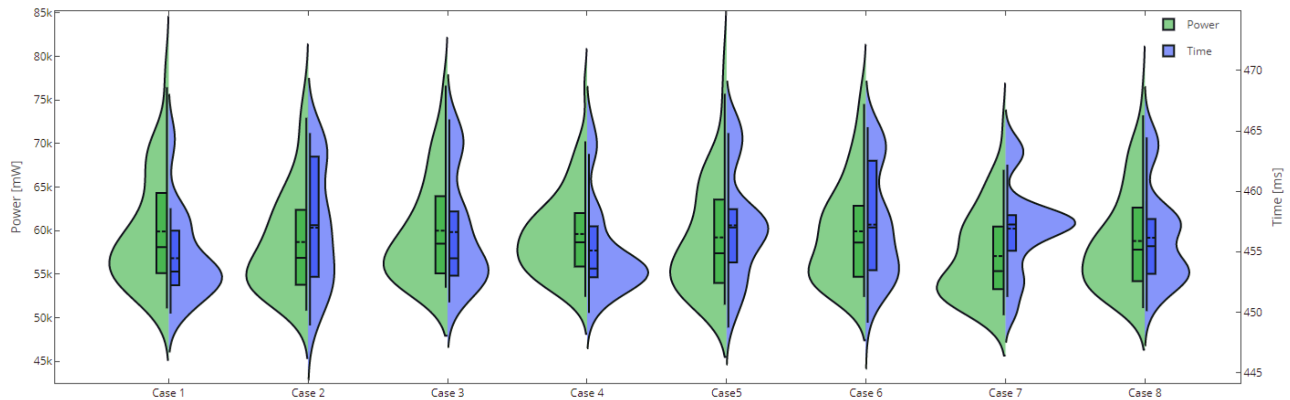

Figure 5: First Generation Probability Distribution of 2D Convolution Kernels - Power and Execution Time

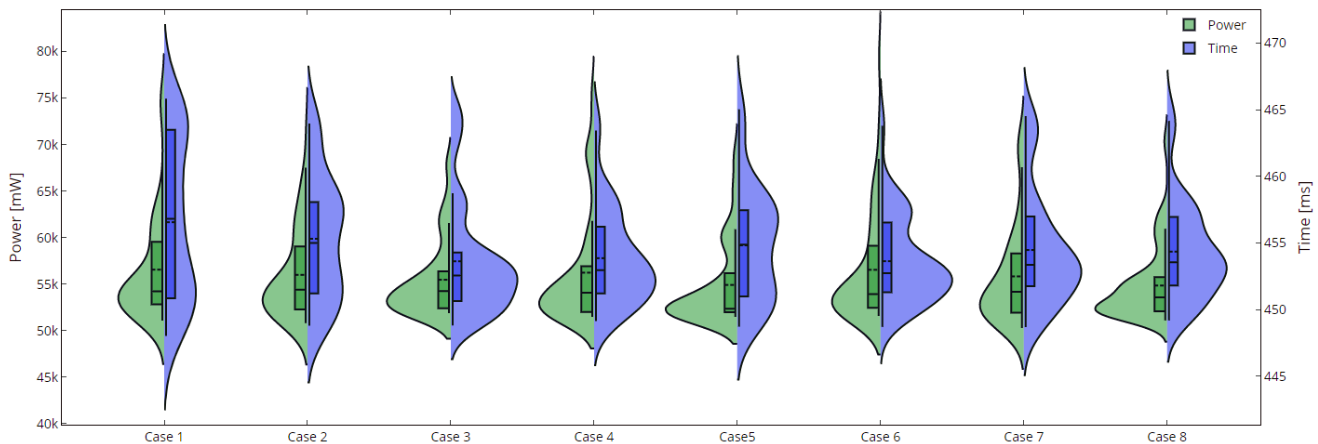

Figure 6: Last Generation Probability Distribution of 2D Convolution Kernels - Power and Execution Time

[18] S. Song, C. Su, B. Rountree, and K. W. Cameron. 2013. A Simplified and Accurate Model of Power-Performance Efficiency on Emergent GPU Architectures. *Int. Symp. on Parallel and Distributed Processing*, pp. 673–686.
[19] NVIDIA. NVML API Reference Manual. Available online: https://developer.nvidia.com/nvidia-management-library-nvml
[20] CUPTI: User's Guide, NVIDIA Corporation, July 2013. Available online: https://docs.nvidia.com/cuda/cupti/index.html
[21] V. Adhinarayanan, B. Subramaniam, and W. chun Feng. 2016. Online Power Estimation of Graphics Processing Units. *16th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing*, Cartagena, Columbia, pp. 245–254.
[22] M. Burtscher, I. Zecena, and Z. Zong. 2014. Measuring GPU power with the K20 built-in sensor. *ACM Workshop on General Purpose Processing Using GPUs*, pp. 28-36.
[23] K. Deb, A. Pratap, S. Agarwal and T. Meyarivan. 2002. A fast and elitist multiobjective Genetic Algorithm: NSGA-II. *IEEE Trans. on Evolutionary Computation*, vol. 6, no. 2, pp. 182-197.
[24] A. Blot, M. Kessaci, and L. Jourdan. 2018. Survey and unification of local search techniques in metaheuristics for multi-objective combinatorial optimisation. *Journal of Heuristics*, vol. 24, no. 6, pp. 853-877.
[25] M. Ferro, A. Yokoyama, V. Klôh, G. Silva, R. Gandra, R. Bragança , A. Bulcão, B. Schulze. 2017. Analysis of GPU Power Consumption Using Internal Sensors. *16th Workshop em Desempenho de Sistemas Computacionais e de Comunicação*, pp. 1698-1711.
[26] J. Schöne et al. 2014. Tools and Methods for Measuring and Tuning the Energy Efficiency of HPC Systems. *Scientific Programming*, vol. 22, pp. 273–283.
[27] B. Van Werkhoven, J. Maassen, H. E. Bal, and F. J. Seinstra. 2014. Optimizing Convolution Operations on GPUs Using Adaptive Tiling. *Future Generation Computer Systems*, vol. 30, pp. 14–26.
[28] C. A. Coello. 2000. An updated Survey of GA-based multi-objective Optimizing Techniques. *ACM Computing Surveys*, vol. 32, no. 2 pp. 109–143.