FLEXDP: Flexible Frequency Scaling for Energy-Delay Product Optimization of GPU Applications

Kaijie Fan
Technische Universität Berlin
Berlin, Germany
kaijie.fan@campus.tu-berlin.de

Biagio Cosenza
University of Salerno
Salerno, Italy
bcosenza@unisa.it

Ben Juurlink
Technische Universität Berlin
Berlin, Germany
b.juurlink@tu-berlin.de

ABSTRACT

Dynamic frequency scaling is broadly available among different modern computer architectures, making it possible to improve the performance and energy efficiency of an application by carefully setting the core frequency. However, while an exhaustive tuning is feasible on simple single-kernel applications, in real-world applications comprised of multiple tasks, the set of possible frequency setting combinations is too large to be exhaustively evaluated.

This work deals with the problem of optimizing a multi-task GPU application with frequency scaling. We focus on different scalarizations of the problem by optimizing for performance, energy consumption, as well as energy-delay product (EDP) and energy-delay-two product (ED^2P). We propose FLEXDP, a new flexible framework that finds the optimal core-frequency configuration over multiple kernels, allowing multiple frequency changes between kernel executions, and taking change overheads into account.

The proposed approaches are evaluated on an NVIDIA Titan X. Experimental results on five applications demonstrate that FLEXDP outperforms the default and autoboost configurations with respect to performance, energy efficiency, EDP, and ED^2P.

CCS CONCEPTS

• **Computer systems organization** → **Parallel architectures**; • **Hardware** → *Power and energy*.

KEYWORDS

Frequency scaling, Multi-task, Optimization, Energy-delay product, GPUs

ACM Reference Format:
Kaijie Fan, Biagio Cosenza, and Ben Juurlink. 2022. FLEXDP: Flexible Frequency Scaling for Energy-Delay Product Optimization of GPU Applications. In *19th ACM International Conference on Computing Frontiers (CF'22), May 17–19, 2022, Torino, Italy*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3528416.3530241

1 INTRODUCTION

Energy efficiency is one of the most critical aspects of modern computing systems. To cope with it, hardware vendors have implemented a variety of techniques that help the users to reduce the energy consumption of an application while trying to minimize its impact on performance.

CF'22, May 17–19, 2022, Torino, Italy

Dynamic voltage and frequency scaling (DVFS) is a technique that aims at decreasing the power consumption by dynamically adjusting voltage and frequency. On recent NVIDIA GPUs, the *NVIDIA Management Library* (NVML) [6] provides an API for monitoring and managing power state as well as changing the core frequency. These features pave the way to optimization that focuses not only on performance, but also on minimizing the energy consumption of a program. Many research works have recently addressed the energy and the performance optimization problem of a GPU application through frequency scaling, e.g., by using machine learning with static analysis [3] or performance counters [1, 5].

However, the optimization of multi-task GPU applications comes with additional challenges. First, the optimal frequency setting is kernel-dependent, therefore a frequency that is optimal for a kernel may not be optimal for others. Secondly, changing the frequency incurs in an overhead. Finally, it is also not realistic to perform an exhaustive search of all frequency combinations on multi-task applications. To solve these challenges we propose FLEXDP, an optimization framework for multi-task GPU applications. FLEXDP requires kernel sampling for kernel characterization with respect to energy, performance, and two scalarized metrics: the *energy-delay product* (EDP) and the *energy-delay2 product* (ED^2P), which are largely-used in the context of energy efficiency evaluation [7]. Overall, our work makes the following contributions:

- We frame the problem of selecting a fixed frequency on GPU programs made of multiple kernels as a constrained optimization problem: we optimize it for energy consumption, performance, and two multi-objective scalarizations that trade off energy against performance: EDP and ED^2P.
- We introduce FLEXDP, which extends the previous optimization problem to handle frequency change between kernels, i.e., dynamic frequency optimization. This framework takes frequency change overheads into account and finds efficient configurations for all the four objectives among the set of all possible allowed frequency changes.
- We implement our approach using NVML and experimentally evaluate its quality on five applications including both synthetic benchmarks and applications.

2 FREQUENCY OPTIMIZATION FRAMEWORK

Given an input application with m kernels $k_1, \ldots, k_m$, our optimization workflow is composed of the following three steps:

(1) For each kernel k_i, we profile both the performance and energy consumption so that all values of $f_p(x_i)$ and $f_e(x_i)$ are known for the per-kernel core frequency $x_i \in [c_{min}, c_{max}]$. To avoid sampling the whole frequency space, we limit profiling to few frequencies and fit, respectively, a linear model for performance

(i.e., $f_p(x_i) = a_i x_i + b_i$), and a quadratic model for energy consumption (i.e., $f_e(x_i) = a_i x_i^2 + b_i x_i + c_i$).

(2) For each kernel k_i and $x_i \in [c_{min}, c_{max}]$, we calculate the EDP and ED^{2}P values using:

$$f_{EDP}(x_i) = f_p(x_i) f_e(x_i) \quad (1)$$

$$f_{ED^2P}(x_i) = f_p(x_i)^2 f_e(x_i) \quad (2)$$

(3) Finally, we apply our optimization algorithms on all four objective functions (f_p, f_{EDP}, f_{ED^2P} and f_e) over the m kernels to find the best core frequency setting for the input application.

In this paper, we present three different strategies to solve the problem in step (3): *fixed-*, *always-* and *flexible-change*.

Fixed Frequency Optimization. We describe the optimization problem formulated to find the best *single* frequency of a multi-task application, i.e. without allowing frequency change between tasks.

In terms of performance, we assume to have a program that executes m kernels. For each kernel k_i, its performance value f_p is described by a linear function with coefficients a_i and b_i and $x \in [c_{min}, c_{max}]$. Thus, we formulate the performance problem as a constrained optimization problem in which the core frequency x is the solution for minimizing the application performance function $\sum_{i=1}^{m} f_p$. As all performance functions have positive coefficients a, the solution to this problem is trivial: $x = c_{max}$. The energy optimization problem is formulated in the same method with performance, while each per-kernel energy values f_e are described by a quadratic function with a unique minimum, which depends on the specific kernel.

In the case of the two energy-delay metrics EDP and ED^{2}P, which are calculated using Eq.1 and 2, we return the optimal frequency value for the specific objective function, i.e., the frequency value that minimizes the objective function.

For a given predefined objective metric, solved by the previously discussed optimization framework, we define the function CostFixed$(k_1, \ldots, k_m)$, which returns the cost for the optimal frequency x that minimizes the objective value, and the function FreqFixed$(k_1, \ldots, k_m)$, which returns the actual frequency x.

Always-Change Frequency Optimization. An alternative solution is to (locally) find the best configuration for each task/kernel and allow to set the frequency before each kernel invocation. We call this heuristic *always-change*, meaning that, before each kernel execution, we always set the core frequency to the one that minimizes the objective function of that specific kernel. This solution is optimal if the frequency changing overhead is proportionally very small with respect to the kernel's objective value (e.g., runtime or energy consumption). On the other hand, this strategy always incurs in $m - 1$ frequency changes, where m is the number of kernels in the application, and does not implement any attempt to reduce the number of frequency change for instance if the optimal frequency of consecutive task is very close to each other.

Flexible Frequency Optimization. The optimization strategies discussed so far represent two opposite approaches. They are beneficial, respectively, in case where the overhead is very large (*fixed-change*) or very small (*always-change*). Now we introduce a smarter *flexible-change* formulation, which is capable to find a solution where frequency changes are accurately performed only when they incur in a real benefit in terms of the objective function.

Given a sequence of m kernels $< k_1, \ldots, k_m >$ and the overhead cost ϵ, we introduce a new strategy that handles the case where flexible frequency changes are allowed. The approach comprises three algorithms and assumes to have access to two functions CostFixed and FreqFixed, which return respectively the cost and frequency of the optimal frequency configurations for m kernels.

Algorithm 1 One Frequency Change at Minimum Cost

```
OneChange(< k_1, ..., k_m >, ε)
1:  min ← +∞
2:  best ← 0
3:  for j ← 1, m − 1 do                    ▷ Cost of freq. change after kernel j
4:      left ← CostFixed(< k_1, ..., k_j >)
5:      right ← CostFixed(< k_{j+1}, ..., k_m >)
6:      cost ← left + right + ε
7:      if cost < min then
8:          min ← cost
9:          best ← j
10: return best
```

The first algorithm, called OneChange, returns the position of the optimal frequency change if we allow only one change after the first kernel execution (i.e., two frequencies are used during the program execution). In detail, this procedure evaluates the objective function of all $m - 1$ possible splits of the kernel, so that the first j kernels use a frequency different than the next $m - k$. We use CostFixed to calculate the optimal frequency for each of the two partitions. This function returns a kernel index i, which indicates that the optimal strategy that minimizes the cost must change the frequency after kernel k_i and before kernel k_{i+1}. Note that OneChange returns 0 if we have 0 or 1 kernel; it returns 1 if we have two kernels, as this is the only possible frequency change we can perform. It is simply possible to get the frequency value for the two lists of kernels by calling FreqFixed$(k_1, \ldots, k_{best})$ and FreqFixed$(k_{best+1}, \ldots, k_m)$.

Algorithm 2 Cost of Multiple Frequency Changes

```
CostFC(< k_1, ..., k_m >, S, ε)
1:  cost ← 0
2:  l ← 1
3:  for i ← 0, S.length - 1 do
4:      r ← S[i]
5:      cost ← cost + CostFixed(< k_l, ..., k_r >) + ε
6:      l ← r + 1
7:  cost ← cost + CostFixed(< k_l, ..., k_m >)
8:  return cost
```

As the flexible strategy allows for multiple frequency changes, we need a way to calculate the cost of multiple frequency changes, including the overheads. We define S as a list of indices referring to when the frequency is changed. E.g., for $S =< 2, 3, 7 >$ we have three frequency changes after kernel k_2, k_3, and k_7. Elements in S are ≤ 1 and $< m$. Given S the list of kernel indices where we enable the frequency change, we define a new algorithm, CostFC, which calculates the overall cost for m kernels.

Finally, the FlexibleChange algorithm uses a recursive divide-and-conquer strategy to determine which frequency changes are beneficial, returning a list of split indices. This recursive procedure starts with the baseline program comprised of only one kernel, for which an empty list is returned. Then, for a program with at least two kernels, we evaluate the cost of cases with and without

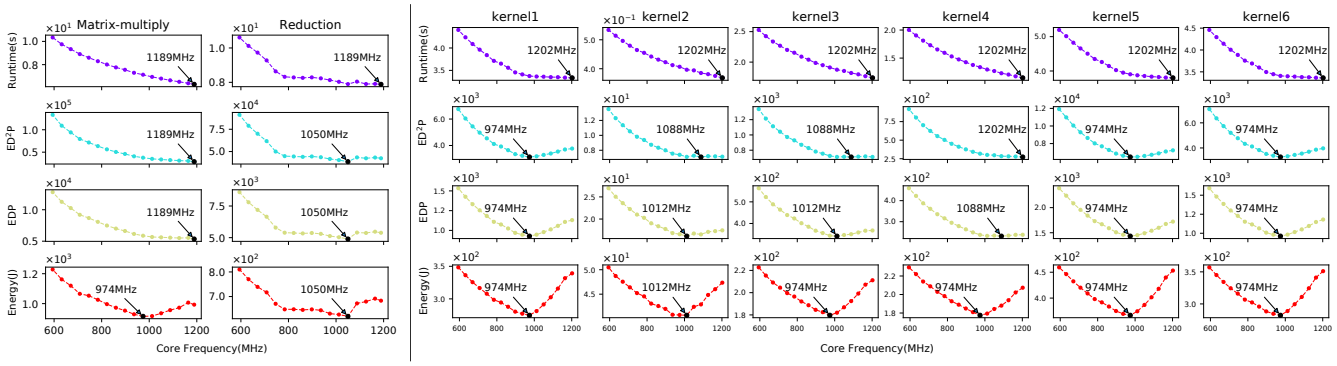


Figure 1: Per-kernel evaluation for Matmul-Reduction Pattern (left) and SRAD (right).

Algorithm 3 Flexible Change Algorithm

FlexibleChange($< k_1, \ldots, k_m >, \epsilon$)
▷ Base case: only one kernel, no change is possible
1: **if** $m = 1$ **then**
2: **return** ListEmpty() ▷ Return an empty list
▷ Case 1: No frequency change
3: $costNoChange \leftarrow$ CostFixed($< k_1, \ldots, k_m >$)
▷ Case 2: At least one frequency change
4: $j \leftarrow$ OneChange($< k_1, \ldots, k_m >, \epsilon$)
5: $L \leftarrow$ FlexibleChange($< k_1, \ldots, k_j >, \epsilon$)
6: $R \leftarrow$ FlexibleChange($< k_{j+1}, \ldots, k_m >, \epsilon$)
7: $costChange \leftarrow$ CostFC($< k_1, \ldots, k_j >, L, \epsilon$) + CostFC($< k_{j+1}, \ldots, k_m >, R, \epsilon$) + ϵ
▷ Cost comparison
8: **if** $costChange < costNoChange$ **then**
9: $L \leftarrow$ ListInsert(L, j)
10: **return** ListMerge(L, R) ▷ Return L, j and R
11: **else**
12: **return** ListEmpty() ▷ No change, empty list

frequency changes. In case of no change, the cost is returned by the CostFixed function. In case of changes, we first calculate the index of the best split j by using the OneChange algorithm. Then, we recursively apply FlexibleChange on the two lists of kernels obtained by splitting at j. The recursive step returns the two lists of indices L and R, for which the overall cost is calculated thanks to the function CostFC. After cost comparison, an empty list is returned if no change gives a smaller cost; otherwise, it returns a list including the splits on the left and right sublists, plus the new split on j. At the end, FlexibleChange finds a sequence of frequency changes that minimize the objective function for the m input kernels.

3 EXPERIMENTAL EVALUATION

We evaluate the proposed optimization framework on an NVIDIA GTX Titan X and compare these strategies against two basic approaches based on *default* frequency (1001MHz) and the *autoboost* configuration. The GTX Titan X supports four memory frequencies and 85 core frequencies, while this work focuses on varying core frequency and fixing memory frequency at 3505 MHz.

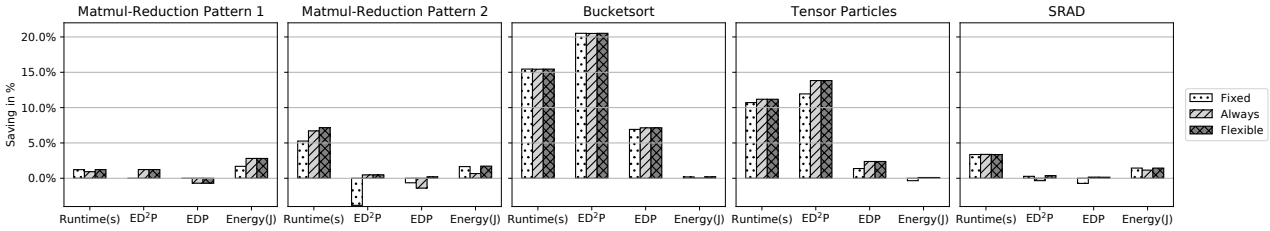
We evaluate five OpenCL applications with a different number of kernels and characterizations and take the frequency changing overheads into account. Those applications are two synthetic applications made of a different sequence of matrix-multiply (M) and reduction kernels (R), bucketsort (three kernels) and Speckle Reducing Anisotropic Diffusion (SRAD, six kernels) belong to the *Rodinia* benchmark [2], and a tensor particles computation [4] (four kernels). For each application, we consider four scenarios of interest: performance, ED^{2}P, EDP, and energy consumption. The detailed results of each kernel for MRMRMR pattern and SRAD are presented in the next paragraphs, while the general discussion of all five applications is available in Sec. 4.

Matmul-Reduction Pattern: MRMRMR. Matrix-multiplication and reduction kernels are the base of synthetic application benchmarks. Fig. 1 (left) shows the four objectives with increasing core frequency. The runtime benefits greatly from core frequency scaling for both kernels. Two trade-offs, ED^{2}P and EDP, are optimized at different frequencies: ED^{2}P is minimized at 1189 MHz and 1050 MHz, EDP is minimized at 1189 MHz and 1050 MHz, corresponded to matmul and reduction kernels. Optimal frequency to minimize the energy is also different with each kernel: matmul has the best energy efficiency at 974 MHz, while reduction at 1050 MHz.

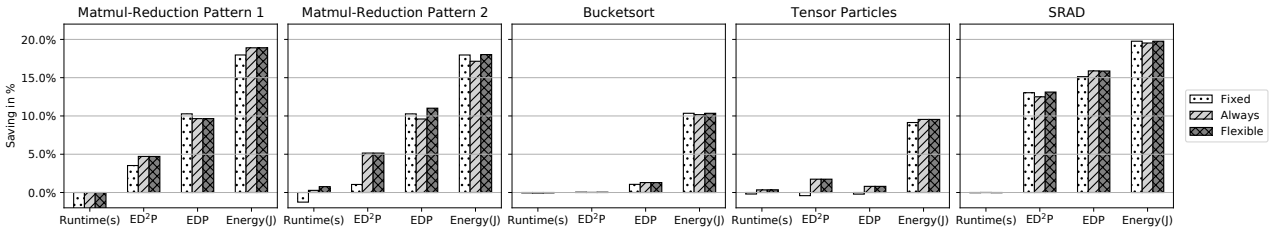
Fig. 2a and Fig. 2b indicate the savings in the four objectives with the three strategies, comparing against the default frequency configuration and autoboost, separately. In terms of runtime, *flexible* returns the same configuration as *fixed* and performs better than the other three, in particular, improves 2% against *default*. For energy efficiency, *flexible* returns the same configuration as *always* and reduces the energy consumption as much as 18.8% comparing against *autoboost*, and also consumes less than *default* and *fixed*. In fact, while for performance the difference between the two per-kernel optimal frequency leads to a large gain, favoring the change always strategy, for energy consumption the gain does not compensate for the overhead of changing. ED^{2}P and EDP correspond to two trade-offs in which *flexible* returns the same optimal configuration returned by *always*. In this application, the *flexible* algorithm returns a configuration that is the same as either *fixed* or *change-always* strategy.

SRAD. Fig. 1 (right) shows the four scenarios of each kernel (k_1, ..., k_6). In terms of performance, all six kernels benefit greatly from core frequency scaling with the optimal frequency at 1202 MHz. ED^{2}P is optimized at 974 MHz for k_1, k_5 and k_6, at 1088 MHz for k_2 and k_3, at 1202 MHz for k_4. EDP is optimized at 974 MHz for k_1, k_5 and k_6, at 1012 MHz for k_2 and k_3, and at 1088 MHz for k_4. The best energy efficiency occurs at 974 MHz for all kernels except k_2.

Fig. 2a and Fig. 2b give the evaluation of the *SRAD* application. For performance, all proposed configurations (*fixed*, *always*, and *flexible*) give similar results. With respect to the ED^{2}P and the sole



(a) Four objectives savings versus default config.



(b) Four objectives savings versus autoboost config.

Figure 2: Evaluation result.

energy consumption, *fixed* slightly outperforms *always*. Here, the *flexible* strategy finds the same frequency configuration as *fixed*.

4 DISCUSSION AND CONCLUSION

Our experimental evaluation provides interesting insight on the algorithms used for frequency scaling as well as the technical issues related to the experimental evaluation on an NVIDIA GPU.

The GPU default frequency is a value that is selected to provide good performance on a broad set of applications. In fact, it is relatively easy to find a configuration that performs better for a specific application. Moreover, it only works for performance: the proposed *always*, *fixed*, and *flexible* strategies are better than *default* in minimizing energy consumption.

The *autoboost* strategy improves over the fixed for performance but, similarly to *default*, does not provide configurations that minimize the energy consumption.

Our three proposed strategies are interesting in different aspects. The *fixed* strategy is always at least better than the *default*. They both use only one frequency for all the kernels, but our fixed strategy picks a configuration that is optimal over all kernels. The *flexible* strategy always finds the best. In most cases, it is as good as the best (minimum) of the *always* and *fixed* strategy. In a few cases such as the matmul-reduction pattern 2, it is capable to discover new frequency configurations that are better than any other configurations.

From an application perspective, kernels variety — i.e. very different per-kernel characterization in terms of performance, energy, EDP, or ED^2P — favors a strategy that is capable to adapt to the different per-kernel tuning, e.g., *change-always*. On the other hand, an application with a very similar kernel will favor fixed frequency configurations. The interesting aspect of the *flexible* strategy is that it adapts to the structure of the application: sometimes it behaves like the two opposite approaches (e.g., SRAD); sometimes is even able to find new configurations for mixed-type applications (e.g., matmul-reduction pattern 2).

Overall, on a broad range of applications, our proposed optimization framework, FLEXDP, improves the performance up to 15.4% with respect to the default fixed frequency, and 19.7% for energy efficiency with respect to the autoboost configuration. For EDP and ED^2P, we have an improvement compared against the default frequency (EDP: 7.1%, ED^2P: 20.5%) and autoboost (EDP: 15.8%, ED^2P: 13.1%).

ACKNOWLEDGMENTS

This research has been partially funded by the European High-Performance Computing Joint Undertaking (JU) under grant agreement No. 956137 (project LIGATE), by MIUR PON Ricerca e Innovazione 2014-2020 (grant No. AIM1872991-1), and by the China Scholarship Council (grant No. 201709110138).

REFERENCES

[1] Yuki Abe, Hiroshi Sasaki, Shinpei Kato, Koji Inoue, Masato Edahiro, and Martin Peres. 2014. Power and Performance Characterization and Modeling of GPU-Accelerated Systems. In *IEEE 28th International Parallel and Distributed Processing Symposium*. 113–122.

[2] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[3] Kaijie Fan, Biagio Cosenza, and Ben H. H. Juurlink. 2019. Predictable GPUs Frequency Scaling for Energy and Performance. In *Proceedings of the 48th International Conference on Parallel Processing*. 52:1–52:10.

[4] Ivan Grasso, Marcel Ritter, Biagio Cosenza, Werner Benger, Günter Hofstetter, and Thomas Fahringer. 2015. Point Distribution Tensor Computation on Heterogeneous Systems. In *Proceedings of the International Conference on Computational Science, ICCS (Procedia Computer Science, Vol. 51)*. Elsevier, 160–169.

[5] Joao Guerreiro, Aleksandar Ilic, Nuno Roma, and Pedro Tomas. 2018. GPGPU Power Modelling for Multi-Domain Voltage-Frequency Scaling. In *24th IEEE International Symposium on High-Performance Computing Architecture, HPCA*.

[6] NVIDIA. 2021. NVML Reference Manual. https://docs.nvidia.com/pdf/NVML_API_Reference_Guide.pdf

[7] V. Spiliopoulos, S. Kaxiras, and G. Keramidas. 2011. Green governors: A framework for Continuously Adaptive DVFS. In *2011 International Green Computing Conference and Workshops*. 1–8.