Optimizing Savitzky-Golay Filter on GPU and FPGA Accelerators for Financial Applications

Ioannis Oroutzoglou
Aristotle University of Thessaloniki
Thessaloniki, Greece
ioroutzo@physics.auth.gr

Argyris Kokkinis
Aristotle University of Thessaloniki
Thessaloniki, Greece
akokkino@physics.auth.gr

Aggelos Ferikoglou
Aristotle University of Thessaloniki
Thessaloniki, Greece
aferikog@physics.auth.gr

Dimitrios Danopoulos
Aristotle University of Thessaloniki
Thessaloniki, Greece
ddanopou@physics.auth.gr

Dimosthenis Masouros
Aristotle University of Thessaloniki
Thessaloniki, Greece
dmasoura@physics.auth.gr

Kostas Siozios
Aristotle University of Thessaloniki
Thessaloniki, Greece
ksiop@physics.auth.gr

***Abstract*—Over the last few years, computational power and intelligence are becoming more and more necessary in the sector of finance. More specifically, computational finance turns into a very popular topic for both academia and industry, where numerous published works from this field and especially investment and risk management, showcase the effects of these technological advancements. At the same time, the ever-increased computational demands have led to the deployment of various accelerators in order to meet both latency and power constraints for financial applications that vary from special purpose, made by economists, to general purpose Digital Signal Processing (DSP) applied in financial time-series. One of the most widely used applications, belonging to the 2nd category, is the Savitzky-Golay algorithm, a filter used for smoothing time-series data. In this work, we propose a mechanism that automatically creates different accelerated Savitzky-Golay filters for GPUs and FPGAs, based on a set of pre-accelerated templates. By evaluating the provided templates with a set of real use-case parameters, a speedup of x33.5 on the NVIDIA T4 GPU and x21.9 on the Alveo U50 FPGA is achieved compared with an Intel Xeon Gold 5218R CPU as a baseline, while achieving a decrease in power consumption of 89% and 70% respectively, disclosing a real latency-power trade-of between both accelerators.**

***Index Terms*—Savitzky-Golay Filter, GPU, FPGA, Computational Finance, FinTech**

I. Introduction

In recent years, the advancements in computing have dramatically changed the human understanding in a wide range of scientific fields. The ever-increased available processing power enabled the analysis of enormous volumes of data, providing meaningful insight in both academia and industry. From the High Performance Computing (HPC) point of view, among the technologies that allowed the management of the ever-increased computational demands, was the introduction of various specialized hardware acceleration platforms. Such devices (e.g., GPUs, FPGAs) can achieve higher performance than typical processing systems for the same power envelope [1]. By mapping the computationally intensive parts of an application to these highly specialized units, CPUs can be enhanced and their power consumption can be significantly decreased.

We thank Mellanox Technologies for their kind contribution of the NVIDIA T4 GPU device.

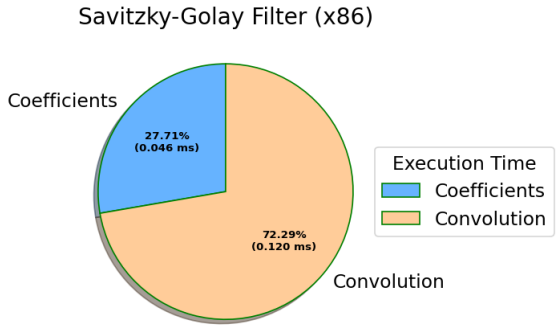

Fig. 1. Execution time of a Savitzky-Golay filter with a window size of 15 and polynomial order of 4 with input's signal size 20 KB.

The implications of hardware acceleration are immense in the finance sector and especially in investment/risk management. The main bottleneck in the calculations of this domain is the existence of no (semi)closed-form pricing formulas in the market models [2]. Evaluating by applying numerical approximations, for a complete portfolio, can be very time consuming, lasting from hours to days on state-of-the-art compute clusters with thousands of cores [3]. The situation becomes even worse as the complexity of the underlying market models and financial products increases, showcasing the necessity of using specialized hardware platforms.

In investment management, the stock closing price signal pre-processing constitutes an important phase for portfolio analysis. In particular, smoothing is a frequently used operation, as it is an essential preparation step for training models for market prediction. The Savitzky-Golay filter has been proven to be an excellent algorithm for this operation [4]. Nevertheless, its computationally intensive nature forms significant limitations.

The execution of every FIR filter consists of two phases. At first, the coefficients are calculated based on the filter's characteristics. Secondly, the filter's output is generated given an input signal and the calculated filter's coefficients. However, the process of filtering an input signal by performing discrete convolutions creates a bottleneck for achieving low-latency filtering, as it is depicted in Figure 1 .

Over the last years, researchers have leveraged hardware platforms for accelerating the computationally intensive part of the Savitzky-Golay filter. Agarwal et al [5] in their work implemented a Savitzky-Golay filter on a FPGA device using an HDL (Hardware Description Language) design flow for smoothing ECG and EEG signals. They showed that low-area designs on FPGA can lead to high-performance signal processing for biomedical applications. In another work, Belloch et al [6] accelerated multiple 1D convolutions from different audio channels on GPU devices using a concurrent execution method, exhibiting acceleration of massive convolution operations. Nevertheless, in the proposed bibliography, users are not able to easily modify the filter's parameters (i.e. polynomial order and window size) in the final accelerated designs.

In this paper, a mechanism for creating different accelerated Savitzky-Golay filters is proposed. By applying the filter's parameters to a pre-designed accelerated template, users are able to easily try different filters for different acceleration devices. The templates are evaluated using the filter parameters of a real use case for a Nvidia T4 GPU and an Alveo U50 FPGA.

The paper is organized as follows: Section II describes the filtering algorithm and the workflow for application development on FPGA and GPU devices. Section III presents the implemented methodology of creating the pre-accelerated filter templates. Section III shows the evaluation results. Section IV shows the conclusion of the paper.

II. Background

Savitzky-Golay Filtering Algorithm:

The Savitzky-Golay finite impulse response (FIR) smoothing filter is also known as polynomial smoothing or least-squares smoothing filter. It is the general form of the FIR average filter that is able to preserve the high frequency component of a signal, at the cost of removing less noise in comparison to the average filter [7]. The Savitzky-Golay filter is frequently used for smoothing time series data [8] as it is able to fit adjacent data points with a low-degree polynomial.

The filter is specified by two parameters: a) the window that defines the number of adjacent data points and b) the order of the desired polynomial. The filter's coefficients are solely dependent on the polynomial order and on the window size, meaning that for a given window and polynomial order the filter's coefficients are fixed. The smoothing process is performed by applying convolution between the signal's coefficients and the window adjacent data points. Once, the convolution for a specific window has been completed the window is shifted and the convolution is performed for the next set of data.

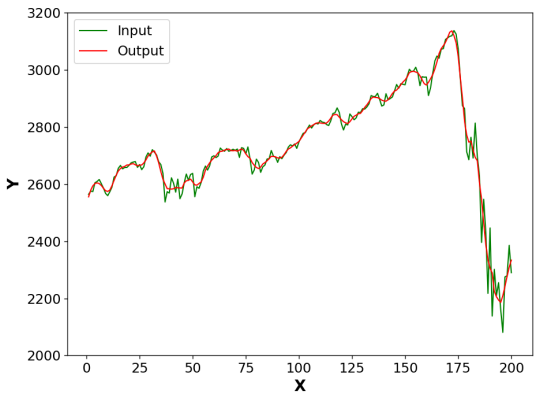

Fig. 2. Smoothing time-series using Savitzky-Golay filter

Figure 2 depicts a signal smoothed by applying the Savitzky-Golay filter with a window size of 10 and polynomial order of 3 on a time series composed of 200 data points.

FPGA Design Methodologies: Designing hardware for FPGAs can be performed at varying levels of abstraction with the commonly used being the register-transfer level (RTL) and the algorithmic level. These methodologies differ; RTL (i.e., VHDL, Verilog) is used for circuit-level programming while algorithmic level methodologies such as High-Level Synthesis (HLS) are used for describing designs in a higher level of abstraction. Compared to RTL, HLS provides a faster and more flexible development process, as designers instruct the HLS compiler on how to synthesise kernels by adding different directives on a C/C++ or OpenCL code.

Xilinx [9], one of the main FPGA vendors on the market, has put significant effort into providing a user-friendly toolset for employing High-Level Synthesis on FPGA design. It introduced Vitis [10], a framework able to provide a unified OpenCL interface for programming edge (e.g., MPSoC ZCU104) and cloud (e.g., Alveo U200) FPGAs. Simplifying the designing process, developers are able to focus on finding the optimal HLS directives with respect to the target device's architecture.

GPU Design Methodologies:

Accordingly, GPU's programmability is significantly improved over the years with high level languages such as CUDA [11] and OpenCL [12] to be the most popular approaches. CUDA programming model, released by NVIDIA, stands for Compute Unified Device Architecture and is a parallel programming paradigm supporting only NVIDIA GPUs, while OpenCL was launched by Apple and the Khronos group as a way to provide a portable language for various types of processors. Even though CUDA and OpenCL offer different interfaces for programming GPUs, they both handle their data in a very similar way, while providing an abstraction of GPU architecture, exposed in general-purpose programming languages such as C/C++. For the scope of this work, our description relies on the CUDA terminology, as this is the programming model that is used for accelerating the filters.

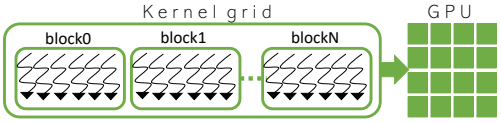

Fig. 3. CUDA grid

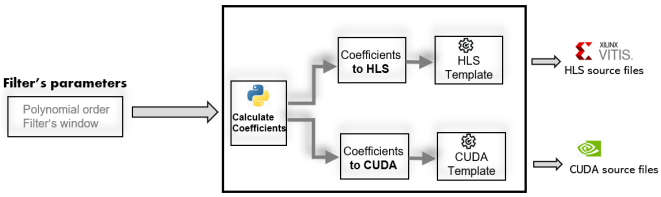

Fig. 4. Proposed methodology

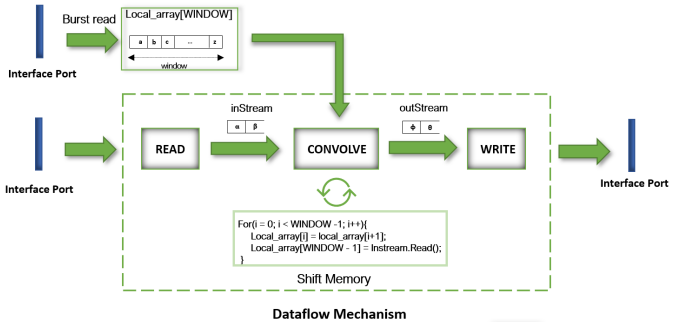

Fig. 5. Designed Dataflow Mechanism

There are 2 keywords widely used in CUDA model: `host` and `device`. `Host` constitutes the CPU available in the system while the system memory associated with it is called host memory. Additionally, the GPU is called `device` and its memory likewise is called device memory. For a CUDA program execution, the developers, at first, need to copy the input data from the host to device memory, also known as host-to-device transfer. Afterwards they need to load their CUDA kernel to be executed and, finally, to copy their results from the device to host memory, also called device-to-host transfer. Every CUDA kernel starts with a `__global__` declaration specifier, with programmers providing a unique global ID to each thread by using built-in variables. Total threads are grouped in CUDA blocks, where all blocks form a kernel grid. Figure 3 depicts an abstract representation of how CUDA threads are organized and mapped on the GPU.

III. Implemented Methodology

To speed-up the workflow of generating and implementing different versions of accelerated Savitzky-Golay filters, a designing flow that generates performance optimized HLS and CUDA kernels based on the pre-designed optimized filter templates was developed (depicted in Figure 4).

A. High-Level Synthesis Filter Template

Each FPGA kernel is designed to implement the dataflow execution flow that is depicted in Figure 5. Every filter's design consists of three distinct sub-units (**read**, **convolve**, **write**) that pipeline the memory read, memory write and the convolution computations.

- **read**: The read module performs memory read transactions through the interface port and store the input's signal values in a First-In-First-Out (FIFO) stream. The FIFO stream has a depth of two and buffers the input elements, allowing the **convolve** module to perform operations continuously without stalling its operation.
- **convolve**: This sub-component performs the convolution task and it starts it's operation when the first W input values have been stored in the FPGA's BRAMs. The convolution is performed in pipelined loops with each loop executing W multiplications between the input signal and the filter's coefficients in parallel. A shifting mechanism is utilized for accessing the next available input data. Specifically, when all of the W input values that are stored locally have been convolved, then the elements of this array are shifted, and the first element of the FIFO stream is moved in the last address of the local array. This process is repeated until all the input data elements have been moved in the local memory array and have been processed. The output of each convolution is stored in a FIFO stream that is connected to the write module.
- **write**: This module performs the memory write transactions by writing the output data elements that are stored in the FIFO stream, provided by the convolve unit, to the global memory.

W: filter's size of the window

B. CUDA Filter Template

The GPU implementation isolates only the compute intensive convolution filter in order to convert it to CUDA parallel kernel. At first, the input and output matrices are allocated on device memory through `cudaMalloc()` command, in order to be transferred through PCI Express with `cudaMemcpy()` command. After manipulating the required data, the kernel grid is organized into 32-size thread-blocks while the total size of threads that launch the kernel constitute the total input array size. In this way, each kernel's thread corresponds to each input's element in order to perform convolution with its surrounding elements. Finally, the resulted matrix is transferred from device to host memory through the `cudaMemcpy()` operation.

IV. Experimental Results

The proposed methodology was evaluated on a use case scenario from the finance sector that performs market and portfolio analysis. Specifically the scenario of accelerating on FPGA and GPU a Savitzky-Golay filter with a polynomial order of 3 and a window size of 11 for 10 different financial assets was tested. Each financial asset is represented by a time series composed of 20000 32-bit data points. Every data point corresponds to the closing price of the specific stock for one day of the year.

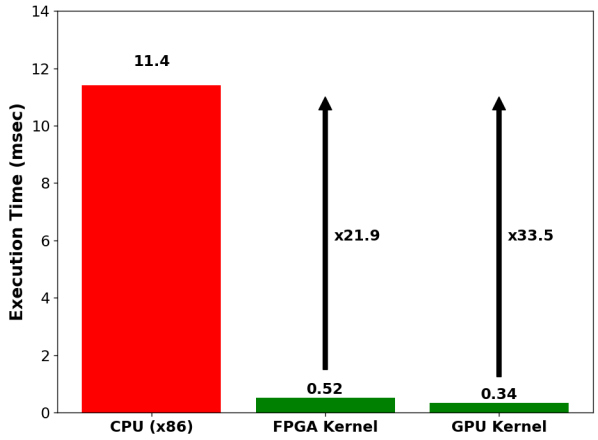

Fig. 6. Execution time speedup for Savitzky-Golay filter on Alveo U50 and Nvidia T4

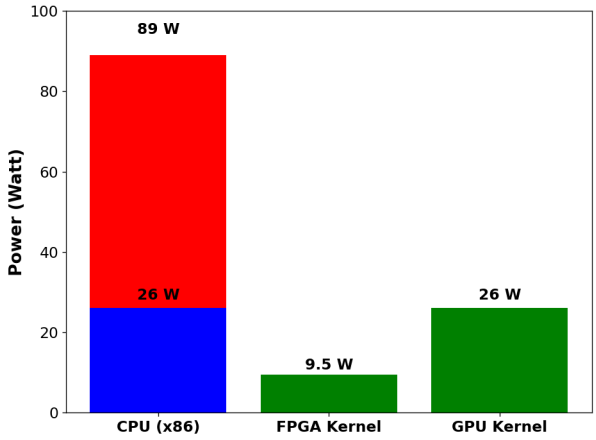

Fig. 7. Power consumption for Savitzky-Golay filter on Alveo U50 and Nvidia T4

Device Setup: For the device setup we tested our evaluation scenario on an Alveo U50 data center acceleration card and on a Nvidia T4 GPU. Our baseline reference architecture was an Intel Xeon Gold 5218R at 2.10 GHz.

Figure 6 depicts the execution time of the aforementioned use case on different computing architectures. The evaluated scenario has an execution latency of 11.4 ms on the Intel Xeon processor, while the execution time of the same computational kernel through the proposed designing flow, is decreased to 0.52 and 0.34 ms on the data center FPGA and GPU devices respectively. Therefore, a speedup of x21.9 and x33.5 is achieved when the pre-accelerated filter templates are used. Additionally, during the execution of the proposed implementation, the power consumption is monitored. For the power measurements of the CPU socket, the Intel PCM tool [13] is used, while the `nvidia-smi` and the `vitis_analyzer` tools are used for the case of GPU and FPGA respectively. The recorded power consumed for the baseline CPU, the FPGA and the GPU platform is presented in Figure 7. As shown, the total power consumption of the baseline CPU socket rises to 89 Watts (with idle consumption at 26 Watts) and FPGA and GPU implementations to consume 9.5 and 26 Watts accordingly. Therefore, it is obvious that a latency-power trade-off is disclosed between FPGA and GPU accelerated Savitzky-Golay filters, with GPU to provide the most latency efficient solution with x33.5 speedup and FPGA to provide the most power efficient solution with 89% reduction compared with the baseline CPU implementation.

V. Conclusion

In this work, a tool for creating different accelerated Savitzky-Golay filters for GPUs and FPGAs is proposed. The mechanism automatically applies the user-defined polynomial order and window size of the filter to a set of pre-accelerated templates. These templates are evaluated using the Savitzky-Golay parameters of a real use case, targeting a Nvidia T4 GPU and an Alveo U50 FPGA. It is shown that a speedup of **x33.5** and **x21.9** is achieved for the Nvidia T4 GPU and the Alveo U50 FPGA compared with the CPU baseline, with a power consumption of **26** and **9.5** Watts respectively. Finally, a latency-power trade-off is disclosed between the both accelerators with GPU to provide the most latency efficient solution with **x33.5** speedup and FPGA to provide the most power efficient solution with **89%** reduction compared with the baseline CPU implementation.

Acknowledgment

This work has been supported by the E.C. funded program SERRANO under H2020 Grant Agreement No: 101017168

References

[1] Danopoulos, D., Kachris, C., Soudris, D.: Fpga acceleration of approximate knn indexing on high- dimensional vectors. In: 2019 14th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC). pp. 59–65 (2019).
[2] De Schryver, C.: Design methodologies for hardware accelerated heterogeneous computing systems. PhD thesis, University of Kaiserslautern (2014)
[3] Weston, S., Spooner, J., Marin, J.-T., Pell, O., Mencer, O.: FPGAs speed the computation of complex credit derivatives. Xcell J. 74, 18–25 (2011)
[4] Górriz, J. M., Carlos García Puntonet, and Moisés Salmerón. "Preprocessing time series with ICA and savitzky-golay filtering." Neural Networks and Computational Intelligence. 2004
[5] Agarwal, Shivangi, et al. "Performance evaluation and implementation of FPGA based SGSF in smart diagnostic applications." Journal of medical systems 40.3 (2016): 1-15.
[6] Belloch, Jose A., et al. "Real-time massive convolution for audio applications on GPU." The Journal of Supercomputing 58.3 (2011): 449-457.
[7] Orfanidis, S. J., Introduction to signal processing. Prentice Hall, Englewood Cliffs, 1996
[8] Kennedy, Hugh L. "Improving the frequency response of Savitzky-Golay filters via colored-noise models." Digital Signal Processing 102 (2020): 102743.
[9] "Xilinx - Adaptable. Intelligent." Xilinx.com, 2019, www.xilinx.com/.
[10] "Vitis Software Platform." Xilinx, www.xilinx.com/products/design-tools/vitis/vitis-platform.html. Accessed 10 Feb. 2022.
[11] Cheng, John, Max Grossman, and Ty McKercher. Professional CUDA c programming. John Wiley & Sons, 2014.
[12] Munshi, Aaftab, et al. OpenCL programming guide. Pearson Education, 2011.
[13] https://www.intel.com/content/www/us/en/developer/articles/technical/performance-counter-monitor.html. Accessed 16 Aug. 2012.