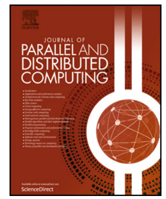
Journal of Parallel and Distributed Computing 187 (2024) 104832

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc



Starlight: A kernel optimizer for GPU processing

Alberto Zeni [a,*], Emanuele Del Sozzo [a,b], Eleonora D'Arnese [a], Davide Conficconi [a], Marco D. Santambrogio [a]

[a] Politecnico di Milano, Milan, Italy
[b] RIKEN Center for Computational Science, Kobe, Japan

ARTICLE INFO

Keywords:
Performance analysis
Performance optimization
High performance computing
GPU
Roofline model

ABSTRACT

Over the past few years, GPUs have found widespread adoption in many scientific domains, offering notable performance and energy efficiency advantages compared to CPUs. However, optimizing GPU high-performance kernels poses challenges given the complexities of GPU architectures and programming models. Moreover, current GPU development tools provide few high-level suggestions and overlook the underlying hardware. Here we present Starlight, an open-source, highly flexible tool for enhancing GPU kernel analysis and optimization. Starlight autonomously describes Roofline Models, examines performance metrics, and correlates these insights with GPU architectural bottlenecks. Additionally, Starlight predicts potential performance enhancements before altering the source code. We demonstrate its efficacy by applying it to literature genomics and physics applications, attaining speedups from 1.1× to 2.5× over state-of-the-art baselines. Furthermore, Starlight supports the development of new GPU kernels, which we exemplify through an image processing application, showing speedups of 12.7× and 140× when compared against state-of-the-art FPGA- and GPU-based solutions.

1. Introduction

The rapid growth of complexity and the amount of data that modern High-Performance Computing (HPC) applications have to analyze daily have exceeded the capabilities of general-purpose processors, creating a gap between the demand for computational power and achievable performance [1,2]. Consequently, as we reach the end of Moore's Law [3,4], we need new architectural solutions to satisfy continuously growing performance demand. In this context, hardware accelerators, e.g., Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs), incarnate an effective solution to offload compute-intensive tasks from the Central Processing Unit (CPU) [5–12]. In particular, Graphics Processing Units (GPUs) have proven over the years to be a much more efficient architecture compared to Central Processing Unit (CPU) in the HPC context in terms both of performance and energy efficiency [13,14]. However, the process of developing highly performing GPU kernels is significantly more complex than CPU software development and requires domain-specific knowledge and expertise to leverage the architecture effectively. State-of-the-art tools for GPU performance analysis [15–24] lack of clarity and detailed information. To exemplify, NVPROF [24] and NSIGHT [23] provide valuable information and suggestions to the end-user, but do not clearly identify specific code regions where the optimizations should be performed. Indeed, the information these tools provide results unclear for users who do not have high-level expertise in GPU programming.

In recent years, multiple research studies [25–30] relied on the Roofline Model [31] to provide an intuitive analysis of the performance of a given application running on CPU, GPU, or FPGA. However, these tools are often limited to a specific architecture and do not provide valuable suggestions to developers. For example, NSIGHT [32] provides a tool for the definition of the Roofline Model for the analyzed kernel; however, its described model lacks in detail, as it considers Global Memory (GMEM) bandwidth and Floating-Point (FP) operations only, and does not correlate the derived information with the suggestions it provides to the user. Conversely, other literature tools offer fine-grained profiling, allowing for computational bottlenecks to be associated with the corresponding parts of code [15,16,22]. Nevertheless, these tools are often capable of identifying a very limited number of bottlenecks, and do not correlate the kernel's attained performance with the underlying hardware capabilities. For instance, if the target application is memory-bound and performs poorly, such tools do not suggest any additional optimizations to the end-user.

* Corresponding author.
E-mail address: alberto.zeni@polimi.it (A. Zeni).

https://doi.org/10.1016/j.jpdc.2023.104832
Received 8 August 2023; Received in revised form 10 November 2023; Accepted 17 December 2023

Available online 22 December 2023
0743-7315/© 2023 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

In this complex scenario, we present Starlight, an open-source [1] tool that guides the user to develop highly optimized GPU kernels by combining Performance Counter (PC)-sampling and the Roofline Model to provide effective and accurate optimizations. We demonstrate its applicability on three different kernels executed on multiple systems, showing the flexibility of our tool when analyzing the performance of different GPU generations. The tool provides a performance analysis of the algorithm considering the underlying architecture and the GPU kernel performance. In particular, Starlight starts by creating a Roofline Model of the target architecture. Then, it proceeds to deeply analyze the kernel by finding the various stalls and assigning them to their respective line of source code. Next, Starlight combines the performance data from the Roofline Model with various details regarding the kernel stalls, generating a series of accurate suggestions for the end-user to follow to optimize the code. Finally, the tool provides an improvement estimation associated with each suggestion using the information on the stalls and their operations, enabling us to also highlight kernel hot spots. Starlight can perform this analysis on any GPU that supports CUDA from version 11.0. Moreover, to the best of our knowledge, it is the first tool in the literature to support the automatic generation of the Roofline Model on any CUDA-capable GPU plotting the target kernel performance and GPU device capabilities for every native data type (half-, single-, and double-precision FP, and integer) and multiple memory hierarchies definitions (L1 cache, L2 cache, and GMEM).

To summarize, the main contributions of this work are:

- Starlight, an open-source Roofline Model-based tool for the analysis and optimization of GPU kernels.
- Automatic Roofline Model generation for any CUDA-capable GPU with support for multiple memory hierarchies and integer applications, making Starlight the first tool in the literature to support the benchmarking and automatic generation of the Roofline for every native datatype.
- Analysis of kernel stalls with information about the stall itself and the stall-causing source code lines without the need for any manual modification/instrumentation of the source code of the GPU application.
- Kernel optimization suggestions associating information from the Roofline Model and kernel analysis decorated with in-depth kernel performance measurements.
- Kernel performance improvement prediction and execution's hot spots detection by precomputing the benefits of the suggested optimizations.

The rest of the paper is organized as follows: Section 2 provides an overview of Starlight and the motivation behind this work; Section 3 describes the Roofline Model in-depth; Section 4 details the characteristics of our tool and defines both how we generate the Roofline Model (Section 4.2) and how we collect and correlate performance data (Section 4.3) to suggest optimizations to the end-user (Section 4.4); Section 5 reports the experimental results obtained by tool-optimized applications and the description of the implemented algorithms; Section 6 overviews the related work regarding tools for GPU-kernel optimization and studies that expanded the Roofline Model for GPUs. Finally, Section 7 states the conclusions.

2. Starlight overview and motivation

Starlight automatically examines the target kernel and provides suggestions about possible optimizations of the analyzed kernel, highlighting under-performing code regions. First, we provide a high-level analytic model for an intuitive bounding analysis in the form of the Roofline Model (Section 3). Second, our optimization methodology also offers an in-depth performance analysis of the target GPU kernel, therefore, overcoming most of the limitations of the Roofline Model representation (Section 6). In this way, Starlight aids the user in identifying which part of the algorithm needs to be improved to achieve better performance and reduce the user expertise required to optimize GPU algorithms. Furthermore, we exploit the Roofline Model performance analysis as an alternative and effective evaluation tool for kernel efficiency to NVIDIA tools. Indeed, NVPROF and NSIGHT [24,32] propose Device Occupancy (DO) as the primary measure on which they base their performance metrics. Summarizing all the kernels' requirements to DO significantly reduces the efficiency of the proposed suggestions, as resource occupancy is a very error-prone metric and often inaccurately depicts the capabilities of a device [33]. Moreover, Starlight overcomes the limitations of state-of-the-art implementations by being the first tool able to depict the performance of any native datatype while correlating its performance with the target GPU architecture bottlenecks into its Roofline Model analysis (Section 6). We do not limit our analysis to the Roofline Model only, but rather we guide users to hot spots, exploiting SASS [34] and CUPTI [35] intermediate assembly representations to accurately characterize the analyzed kernel. Starlight also provides insights into possible performance gains, by correlating the information of the various kernel stalls to their respective code lines and stall nature while weighting the number of stalls removed if a suggested fix is applied to the code (Section 4.4).

We believe that Starlight offers both a more accessible and more effective way for kernel optimizations than other state-of-the-art tools, given its ability to overcome the limitations of the other Roofline-based implementation and provide the user with easy-to-understand and precise suggestions to optimize the source code.

3. The roofline model

The Roofline Model [31] represents a valuable resource for HPC developers, as it offers a visually intuitive method to portray and understand the performance (usually expressed in Floating-Point Operations Per Second (FLOPs/sec)) and bottlenecks of an application. Such a model depends on the target architecture's peak performance and Memory Bandwidth (MBW), obtainable through either the hardware specification or micro-benchmarks. It exploits the same analysis applied by Amdahl's Law [36], e.g., the *bound and bottleneck* analysis, to couple the attained performance of an application and its achieved MBW in a single graph. In particular, the Roofline Model showcases the various application limits associated with the characteristics of the underlying architecture, indicating if such an application is either memory- or compute-bound. In this way, users can understand performance issues at a glance, as developers can exploit this model to observe the different bottlenecks of algorithms and architectures to better comprehend how to improve the application performance. Originally, the model was conceived only to describe the performance of CPUs, while in recent years, the Roofline Model has also been adapted to better suit other architectures, e.g., GPUs and FPGAs [25–30], or extended to include additional features, such as multiple levels in the memory hierarchy, ranging from the off-chip memory to on-chip caches [37,27,38,39]. Fig. 1 shows an example of the vanilla Roofline Model on a log-log scale for a FP-based application. Here, the y-axis represents the reachable Floating-Point Performance (FPP) in Giga Floating-Point Operations Per Second (GFLOPs/sec). The x-axis denotes the Operational Intensity (OI), which indicates the number of operations performed per byte of GMEM traffic, depicting the relationship between the target architecture performance and the off-chip memory bandwidth. The horizontal blue line in Fig. 1 shows the peak FPP of the system. Thus, the actual FPP of an application cannot exceed that line, since it is a hardware limitation. The diagonal orange line exhibits the maximum FPP, in terms of MBW, that the memory system of the target architecture supports for a given OI. Given this setup, the following formula denotes the top attainable performance:

[1] https://github.com/albertozeni/starlight

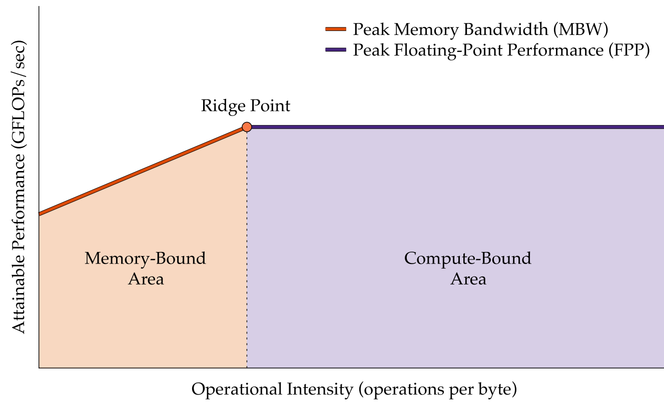


Fig. 1. An example of the Roofline Model chart.

$$GFLOPs/sec = \min(Peak\ FPP, Peak\ MBW \cdot OI) \quad (1)$$

Equation (1) considers the two previously mentioned lines: Peak FPP (blue) and Peak MBW (orange), which depends on the relative OI. The two lines intersect at the point of peak computational intensity and peak MBW. Such a point, called *ridge point*, provides the user with insights regarding the overall performance of the target system. Indeed, the OI of the ridge point divides the chart into two areas: a memory-bound one (on the left, highlighted in orange in Fig. 1) and a compute-bound one (on the right, highlighted in blue). Therefore, the OI of a given kernel plays a crucial role in determining its peak performance. For example, if the OI is lower than the ridge point's one, the kernel is in the memory-bound area, which means that the MBW of the system limits the attainable performance. Conversely, if the kernel is in the compute-bound area, the peak performance depends on the computing resources available in the system. In summary, developers aim to improve the OI to reach the compute-bound area, if feasible; then, they can enforce optimizations to increase the GFLOPs/sec until the kernel "touches the roof."

4. Proposed solution

This Section provides a detailed description of Starlight. First, we overview its structure (Section 4.1) and define our methodology for the Roofline Model generation (Section 4.2). Then, we illustrate how we perform fine-grained performance analysis (Section 4.3) and correlate this information with kernel performance predictions and optimization suggestions (Section 4.4).

4.1. Starlight structure overview

Starlight comprises three main modules, as depicted in Fig. 2. The first one is the *Roofline Generator* (Fig. 2 Ⓐ), which profiles the application and draws the Roofline Model, highlighting the actual performance of the application. Then, the *Performance Analyzer* (Fig. 2 Ⓑ) exploits NVIDIA CUPTI to correlate performance bottlenecks with the application's source code. Finally, the *Optimization Parser* (Fig. 2 Ⓒ) module correlates such information to produce optimization suggestions.

Starlight takes as input the binary of the target application. At first, the tool queries the target GPU and extracts its characteristics. From such information, the Roofline Generator selects the proper GPU Profiler (NVPROF [24], if the compute capability of the GPU is lower than 7.0, NCU [23] otherwise) and proceeds to build the GPU Roofline Model and to profile the application to map it onto the Roofline. Then, the Performance Analyzer collects the performance data by sampling the target application through CUPTI and associates performance information and bottlenecks with the application's source code lines. Next, the Optimization Parser processes the data from the two previous modules and combines them with optimization suggestions. Starlight output is the Roofline Model of the target kernel and the list of optimizations. After improving the application, the user can rerun Starlight to evaluate the current performance and check whether the tool suggests additional optimizations. Lastly, please note that Starlight currently targets single-GPU systems.

4.2. Roofline generator

The first module of Starlight is the Roofline Generator (Fig. 2 Ⓐ), which produces a Roofline Model tailored to a given application and the underlying GPU; please note that the Roofline Model generated by Starlight is inspired by the Roofline variants [37,27,38,39] that integrate the entire memory hierarchy to highlight the role and impact of each GPU memory level. To this end, it detects the target GPU and retrieves its characteristics (e.g., memory bus width, GPU frequency, and memory frequency) by querying the target GPU via the NVIDIA APIs and a custom kernel. In this way, Starlight builds a Roofline Model based on theoretical data; alternatively, the user can provide a JSON file containing empirical data about the GPU performance and memory to derive a more realistic Roofline Model. After this step, the tool starts the kernel analysis through the appropriate NVIDIA API's. More specifically, Starlight can benchmark every GPU supporting either NVPROF [24] or NCU [23], and according to the target GPU architecture compute capability, it automatically selects the appropriate APIs and the corresponding metrics to collect. Indeed, we collect specific performance information using NVIDIA's proprietary APIs, and deeply analyze it to accurately depict the target kernel's performance, and the GPU capabilities. We enrich the incomplete information provided by NVIDIA profilers with additional insights, exposing more useful and accurate information to the users through our Roofline Model. During this analysis phase, the tool collects data regarding the number and type of instructions and operations executed, the number of memory load/store transactions per each memory type (e.g., off-chip, L1 and L2 caches), the branch efficiency, the utilization of the Streaming Multiprocessors (SMs), the percentage of stalled threads, the achieved occupancy of the GPU, and, finally, the target kernel execution time. Once the data retrieval is over, Starlight correlates such results with the specifications of the target GPU. In particular, we use the number of available SMs, compute units per SM, Tensor Cores, if available, and the GPU's achievable frequency to compute the performance ceiling. Similarly, we employ memory-related data (e.g., memory clock and memory bus width) to calculate the GPU's memory bandwidth. Finally, after correlating such data, Starlight plots the Roofline Model for the given kernel.

Fig. 3 shows the typical outputs of the Roofline Generator module based on theoretical (Fig. 3a) and empirical data (Fig. 3b), respectively. The chart provides the user with a visually-intuitive method to understand the kernel performance using a *bound and bottleneck* analysis approach. In particular, we express the performance using different metrics depending on the type of performed operations. Indeed, Starlight supports the profiling of both integer and FP applications, the y-axis reports either Giga Integer Operations Per Second (GIOPs/sec) or GFLOPs/sec, whereas the x-axis exhibits the corresponding OI, here indicating the operations per byte of L1/L2/GMEM traffic. Within the chart, we use different colored lines to display the bandwidth of these three memories and various symbols for the performance of the supported arithmetic precision. We highlight different areas of the graph in different colors to better indicate if the analyzed kernel is either compute- or memory-bound. Finally, we calculate the performance ceiling and GPU's memory using the data previously mentioned. For instance, the NVIDIA RTX A5000 can reach a frequency of $1.695GHz$ and has 8192 CUDA Cores available. In particular, since each Core can schedule two 32-bit operations per clock cycle, the theoretical peak performance for both integer and single FP precision is $1.695GHz \times 8192 \times 2 = 27770.88$ GIOPs/sec or GFLOPs/sec (Fig. 3a). Similarly, we compute the theoretical memory bandwidth available on the target GPU. In this instance, the A5000 has a 384-bit memory bus

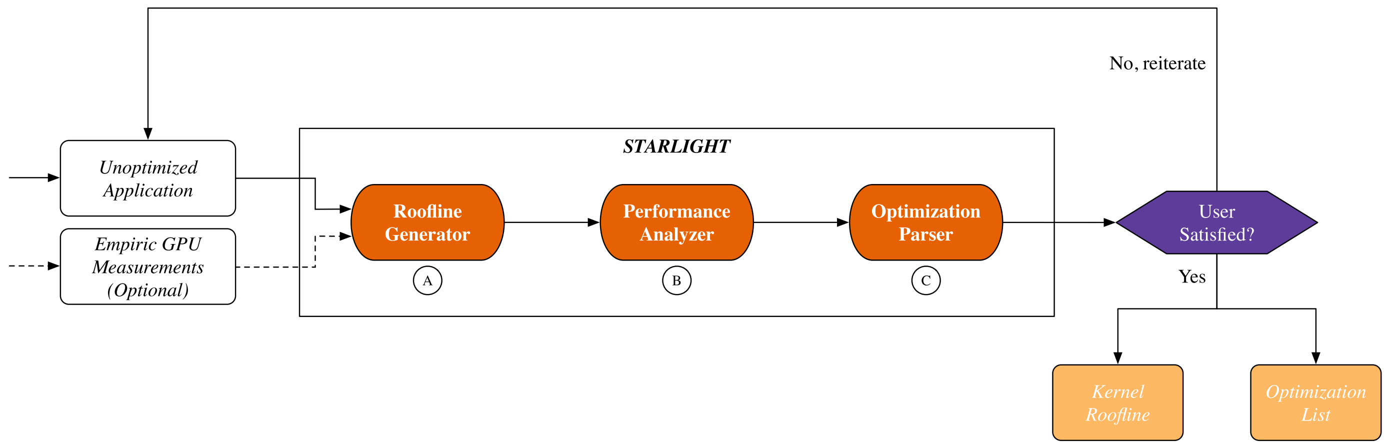
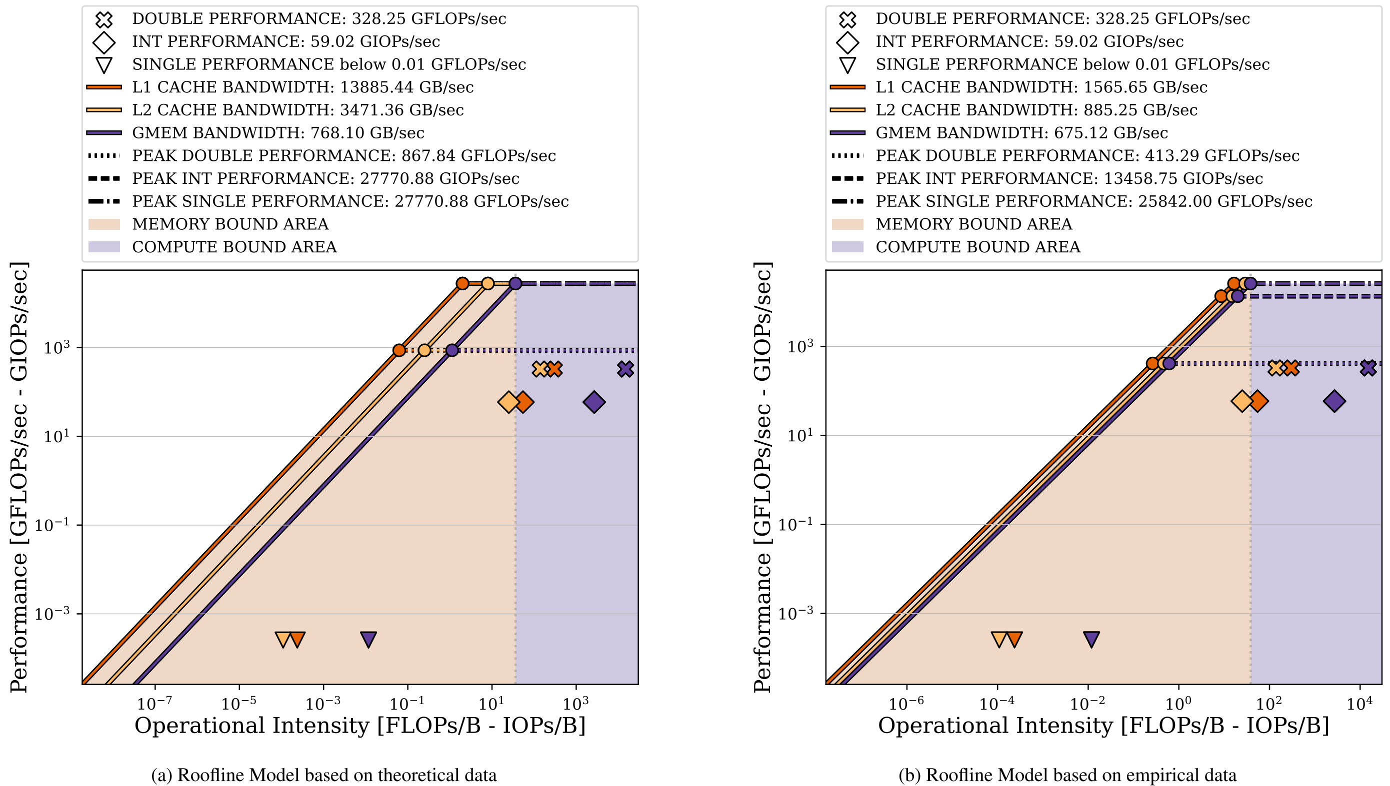


Fig. 2. From an unoptimized application, Starlight analyzes the application using three modules. First, it profiles the application and generates the Roofline Model reporting its performance and, optionally, a JSON file containing the benchmarked GPU performance (Ⓐ). Then, Starlight samples the application through CUPTI APIs and associates kernel stalls and performance measurements with the corresponding source code lines (Ⓑ). Finally, the tool correlates the information from the two previous modules and produces optimization suggestions (Ⓒ). The output of Starlight is the Roofline Model of the application and the optimization list. The user can also re-iterate Starlight analysis to further optimize and examine the code after each pass.



(a) Roofline Model based on theoretical data

(b) Roofline Model based on empirical data

Fig. 3. Example Roofline plot outputs of the Roofline Generator, using theoretical (a) or empirical (b) measurements. The three differently colored lines represent the bandwidth of the memories available on the GPU (an RTX A5000 in this example). The two differently colored zones represent compute- and memory-bound areas of the model. The three different shapes represent the performance attained by the different kernel computations related to the OI of the various levels of GPU memory (each shape corresponds to a different arithmetic precision).

width, a memory frequency of $2000MHz$, and can schedule 8 memory accesses per clock cycle. Thus, the memory bandwidth for the off-chip memory is $8 \times 2000MHz \times 384/(1000 \times 8) = 768\ GB/s$. Conversely, L1 and L2 caches have their frequency tied to the GPU rather than memory. Therefore, the L1 cache bandwidth depends on the SMs on the target GPU, differently from the L2 cache, which instead cannot always be accessed by all SMs, being its access configuration dependent on the GPU's architecture generation. In our use-case, the A5000 has a total of 64 SMs available, each accessing 128 bytes on the L1 cache, and the total number of SMs accessing the L2 cache reading 32 bytes per clock cycle at maximum. Because of this the peak L1 and L2 maximum bandwidth is $64 \times 128 \times 1.695GHz = 13885.44\ GB/s$ and $64 \times 32 \times 1.695GHz = 3471.36\ GB/s$ respectively. It is important to note that, while this information is unique for every GPU, even within the same architectural generation, every measure is completely automated by Starlight, which adapts the various measurements at runtime and ensures accuracy when targeting the different GPU characteristics.

Although this automated approach guarantees portability to different GPUs, the resulting theoretical Roofline Model may inflate the achievable performance and mislead users. For this reason, Starlight allows the user to supply a JSON file containing data about performance and memory ceilings derived from external experiments/benchmarks [40–43]; the result is an empirical Roofline Model that better fits the target GPU capabilities.

4.3. Performance analyzer

The Performance Analyzer module (Fig. 2 Ⓑ) collects various information regarding the kernel performance and bottlenecks and correlates them with the application source code without manually modifying or instrumenting the original CUDA application. To this end, this module leverages NVIDIA CUPTI APIs, which enable seamless kernel instruction sampling. Moreover, CUPTI instruments GPU binaries to gather data about executed instructions and memory accesses, something which instruction sampling alone cannot measure. Consequently, our module can collect all the various stalls of the application to offer more accurate performance optimization suggestions to the end-user. Finally, the Performance Analyzer also exploits the CUPTI Continuous Sampling APIs, which prevent the serialization of the various kernels within the application, guaranteeing that our performance modeling reflects the actual execution accurately. During this phase, the Performance Analyzer samples the target executable at the maximum frequency reachable by CUPTI Sampling APIs (every 32 clock cycles) to better model the various performance bottlenecks. Of course, CUPTI does introduce some overhead when profiling the target application, which significantly varies according to the target application and the density of CUDA activities within it. Nonetheless, the time spent by Starlight analyzing the target application is negligible with respect to the optimizations and future gains that the tool suggests, increasing the execution time of the application during the data collection phase only by a factor of 4.85× on average.

After retrieving the data from CUPTI APIs, the Performance Analyzer correlates PCs to SASS (the low-level assembly that compiles to binary GPU microcode [34]) and then SASS to the correspondent CUDA source line. At first, the Analyzer extracts CUDA binaries (*CUbins*) from the application executable. CUbins contain CUDA executable code sections, symbols, relocators, and debug information necessary to associate stalls with the correspondent lines of code at the SASS level. Then, the module correlates the PC sampling and SASS scheduled instructions. Once this phase ends, we can associate the different SASS assembly instructions producing stalls with their respective lines in the source code. Finally, the Performance Analyzer stores an intermediate raw representation of these results and passes them to the following module.

4.4. Optimization parser

The final module of Starlight (Fig. 2 Ⓒ), namely the Optimization Parser, is in charge of associating the various performance stalls and kernel performance with optimization suggestions for the end-user, according to the information collected in the two previous modules. Furthermore, it is in charge of detecting hot spots and providing information on the potential benefits of the applied optimizations. In particular, the module starts by taking into account measures collected by the Roofline Generator (Section 4.2), observing whether the target application is memory- or compute-bound. In the former case, Starlight suggests applying changes to the computation (e.g., decreasing the arithmetic precision) or compressing the input data. These optimizations aim to increase the Operational Intensity (OI) and, potentially, move the kernel to the compute-bound area. In the latter case, if the kernel is far from touching the compute roof, the module analyzes the GPU occupancy and SM efficiency, suggesting adapting the number of scheduled blocks and threads accordingly. Then, the module checks the kernel's branch efficiency, observing the efficiency of non-predicate warp instructions to advise the end-user on how to schedule the kernel threads better to avoid divergence.

Starlight proceeds to correlate this analysis with the corresponding code lines highlighted by the Performance Analyzer (Section 4.3) and their relative stalls. In particular, the module associates each stall with a specific cause and, accounting for the previously computed performance metrics, suggests the different optimizations to apply at the appropriate source code line. More specifically, Table 1 provides an overview of the different optimization recommendations according to the performance bottleneck observed during the construction of the kernel's Roofline Model and PC sampling.

As a final step, this module predicts the performance improvement (expressed in percentage and GFLOPs/sec or GIOPs/sec) that the kernel can obtain after resolving the various stalls, detecting code hot spots to facilitate the user in the research for problematic code sections. In particular, Starlight uses the peak performance of the target GPU, the kernel's achieved OI, the total kernel runtime, and the number of stalls, together with their previously detected cause (Table 1), to provide the end-user with accurate information regarding the various gains of every optimization. Indeed, the module parses information regarding the various code stalls together with the previously computed correlations, associating each suggestion, and corresponding code line, to an estimated improvement. First, we detect the kernel's hot spots correlating problematic source code lines (previously detected by the optimization parser module) with their corresponding operations. By doing so, we can precisely depict which parts of the kernel account for most of its execution time. We describe the runtime percentage of a hot spot code line as follows:

$$\textit{Line Runtime Percentage}_l = \frac{N_{op,l} + N_{stall,l}}{\sum_{i=1}^{M}(N_{op,i} + N_{stall,i})} \times 100 \quad (2)$$

Equation (2) describes the proportion of the runtime for every problematic code line in the kernel. We define M as the total number of code lines of our kernel, l as the analyzed source code line, while $N_{op,l}$ and $N_{stall,l}$ indicate the number of operations and stalls required by the execution of l, respectively. Then, to compute the possible attainable gain of each optimization, we analyze the nature of the various operations of every l line and associate every performance bottleneck to the corresponding line stalls and operations. Therefore, we can weigh the impact of every optimization for the target l line by characterizing each hot spot per bottleneck type, knowing that the same source line might have bottlenecks, and only some can be addressed with different optimizations.

We describe the impact of an optimization against a code line l in terms of speedup as:

Table 1
List of code optimizations suggestions provided by the tool and how they affect the kernel performance and OI on the Roofline Model.

Performance Bottleneck	Suggestion	Kernel improvement and movement on the Roofline
Latency/Dependency	Code Reorder Function Inline Loop Unroll	Increase performance ↑
Poor Memory Usage	Increase Shared Memory usage Coalesced Memory Access Register Reuse	Increase performance ↑ Increase OI →
Poor Resource Usage	Increase/Reduce Number of Threads Increase Number of Blocks	Increase performance ↑
Thread Synchronization/Branching	Reduce Number of Threads Code Reordering Remove Sync Split Computation	Increase performance ↑
Low OI	Reduce Operation Precision Compress Analyzed Data	Increase OI →

$$SpeedupAfterOptimization_l = \frac{\sum_{i=1}^{M}(N_{op,i} + N_{stall,i})}{\sum_{i=1}^{M}(N_{op,i} + N_{stall,i}) - N_{optype-stalls,l}} \quad (3)$$

where $N_{optype-stalls,l}$ is the number of stalls per specific operation type, e.g., *Latency*, we identified in the l line. Finally, since the runtime percentage of each code line is computed with respect to the total execution time of the kernel, by assuming the same number of computed operations by the kernel, we estimate the new performance of the optimized kernel, in either GFLOPs/sec or GIOPs/sec, as:

$$New\ Performance = Speedup\ After\ Optimization \times Original\ Performance \quad (4)$$

Experimental results (Section 5) show the effectiveness of our performance improvement prediction methodology, highlighting its accuracy in depicting performance stalls, as the difference between the predicted performance improvements against the obtained ones is below 2.5%.

5. Experimental results

This Section describes the experimental evaluation of Starlight. In particular, we evaluate our tool's analysis and optimization features on three different examples, each one exposing different compute patterns and arithmetic precision. We report the experimental settings of our experiments in terms of NVIDIA tools and target GPUs (Section 5.1). Then, we illustrate the analysis and optimization process we applied to the three examples.

First, we evaluated Starlight against two openly available HPC GPU applications. The former is an N-Body simulation application available in the NVIDIA examples[2] [45] (Section 5.2); the latter is LOGAN[3] [5], an HPC GPU algorithm for aligning very long genome sequences implementing the X-drop heuristics [46] (Section 5.3). Here, we focused on optimizing the computing kernels for these two algorithms without changing their overall structure to prove the effectiveness of Starlight in identifying the bottlenecks of already-optimized applications and suggesting further performance improvements. Moreover, we show the tool's capability in analyzing kernels employing single/double FP precision (N-Body simulation) or integer operations (LOGAN). Finally, the third example is an open-source solution for the computation of the Mutual Information (MI) in the Image Registration field [47–51] (Section 5.4). In this case, we demonstrate how Starlight can help guide the implementation and optimization process of an application initially not designed for GPU from the ground up.

Table 2
Different Machine Configurations used to test Starlight.

NVIDIA GPU	GPU RAM	Host CPU	Host RAM
RTX 3060 Mobile	6GB GDDR6	Intel i9 11900H	64GB
RTX A5000	24GB GDDR6	AMD Ryzen 7 5800X	32GB
A100	40GB HMB2	AMD Epyc 7542	2TB
V100	16GB HBM2	Intel Xeon Platinum 8167M	768GB
P100	16GB HBM2	Intel Xeon Platinum 8167M	256GB

5.1. Experimental settings

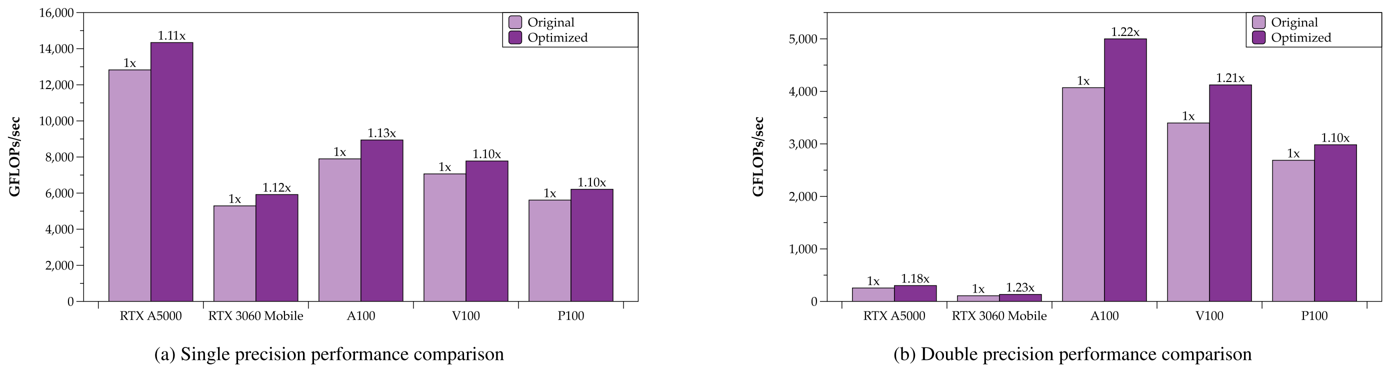
All the applications have been implemented using C++ and NVIDIA CUDA 11.8. The PC sampling utilities have been described using NVIDIA CUDA Toolkit 11.8 APIs and Perl. We collected the performance results on multiple systems covering multiple generations of NVIDIA GPUs (Table 2) to show the tool flexibility and its cross-architecture effectiveness. Finally, we report the Roofline Model of the NVIDIA RTX A5000 using empirical measurements [40–43], to show the consistency of the results with Starlight's generated Roofline Models.

5.2. N-body simulation analysis and optimizations

The *N-Body* simulation algorithm approximates the evolution of a system of bodies where they continuously interact under a force of attraction [44]. A simple example is a gravitational system where the bodies represent celestial entities (e.g., galaxies, stars, or planets). Generally, the N-Body simulation algorithm is a crucial part of many scientific applications, such as global illumination, fluid simulation, and protein folding. For this reason, the literature contains various versions of this algorithm. Among these, the *All-Pairs* is the most time-consuming yet accurate variant. At every simulation step, the algorithm computes the forces of attraction of each body by considering its interaction with all the others, resulting in the time complexity of $\mathcal{O}(n^2)$. Here, we analyze the All-Pairs implementation available in the CUDA examples,[4] which corresponds to the one work of Nyland et al. [45]. In particular, this implementation proposed a solution based on tiling, dividing the bodies into multiple tiles of the same dimension and updating the position of each body within a tile in parallel. Besides, the authors employ loop unrolling, assign a GPU block per tile of bodies, and update their position using multiple threads.

During the considered N-Body simulation algorithm analysis, Starlight highlighted multiple issues with latency and poor resource utilization on the GPU. Hence, we first proceeded with inlining the

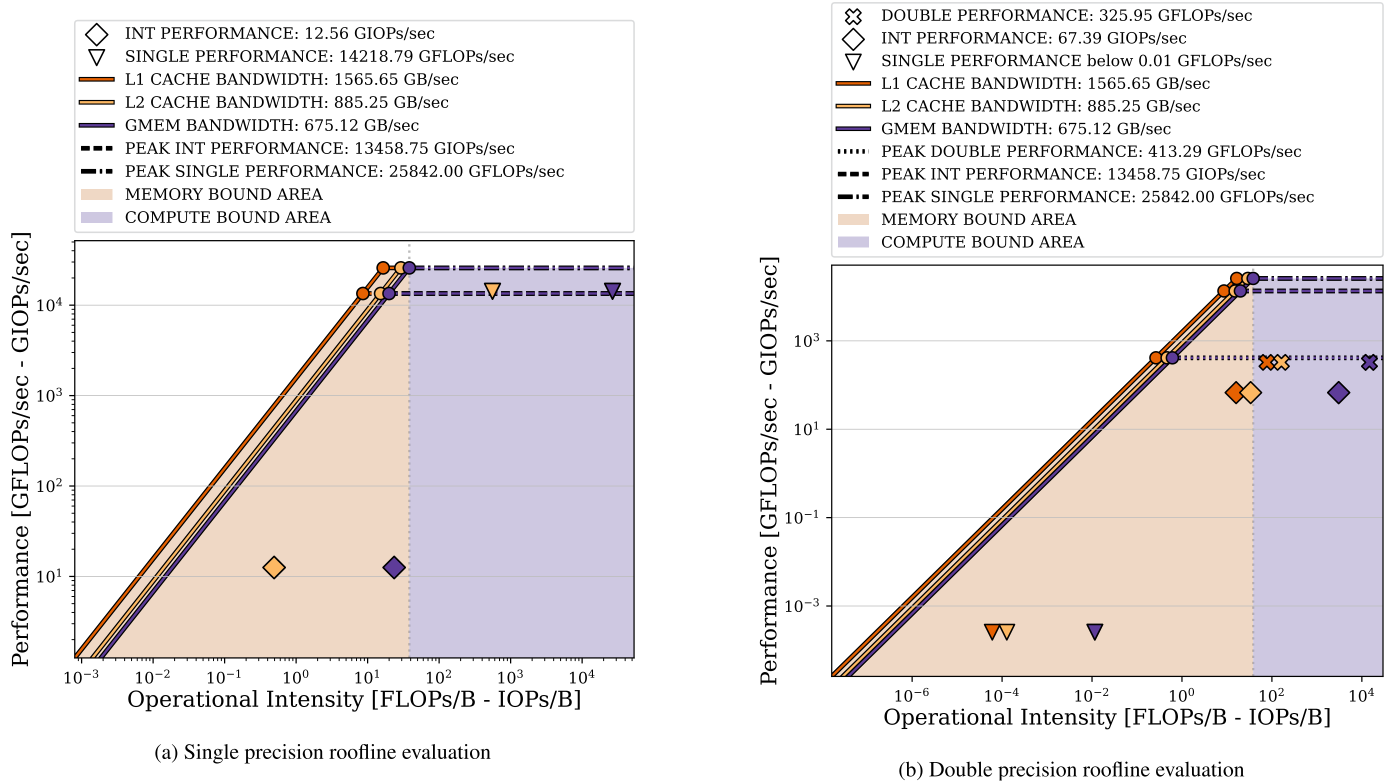
[2] https://github.com/NVIDIA/cuda-samples.
[3] https://github.com/albertozeni/LOGAN.git.
[4] https://github.com/NVIDIA/cuda-samples.



(a) Single precision performance comparison

(b) Double precision performance comparison

Fig. 4. Performance comparison of the Starlight -optimized N-Body single- and double-precision kernels and relative speedup on multiple GPU boards.



(a) Single precision roofline evaluation

(b) Double precision roofline evaluation

Fig. 5. Plot output of the Roofline generator for optimized N-Body single- and double-precision kernel on the NVIDIA RTX A5000.

functions flagged by the tool. Then, we adapted the already unrolled code section to use the same number of threads scheduled to compute the algorithm, improving the GPU resource utilization. Indeed, the original code version unrolled the execution of the inner loop of the N-Body simulation by a fixed factor of 128. This choice limited the parallelism up to 128 computations in parallel and significantly impacted the latency of the algorithm, since in the case of more than 128 scheduled threads, these would stall, limiting the capabilities of the GPU to overlap the execution warps properly. Finally, according to the target board, Starlight also suggested increasing the number of scheduled threads to compute more body interactions in parallel.

We evaluated our optimized code against the original version using both single and double FP precision. We considered a system of 65536 bodies for these experiments and simulate 100 time steps. Starlight's predicted performance speedup against the original implementation of the single precision workload was 1.15× on average. Fig. 4a compares the performance of the original algorithm against the one optimized with Starlight on different boards in terms of GFLOPs/sec. On the other hand, Fig. 5a shows the Roofline Model of our final single-precision implementation on the NVIDIA RTX A5000. In this scenario, our solution achieves up to 1.13× performance improvement compared to the original software, with different gains attained according to the target board. Moving to the double-precision version, Fig. 4b and Fig. 5b report the performance comparison and the Roofline, respectively. In this instance, given the additional complexity of double precision operations, Starlight highlighted additional stalls, thus resolving them accounting for additional performance gains with respect to the single-precision implementation and achieving improvements of up to 1.22× concerning the baseline double precision solution. Starlight's predicted improvements for the solved stalls indicated a performance increment of 1.24×. Moreover, we can observe that in the double precision use case, our optimized kernel is close to touching the Roofline, while in the single precision instance, the performance reached by the N-Body simulation algorithm is still not touching the Roofline ceiling, even after our optimizations, indicating that further improvements are possible. Indeed, Starlight still showed us all the remaining bottlenecks within the code, indicating dependency issues in the innermost part of the body's position update computation. Solving these issues would require a sig-

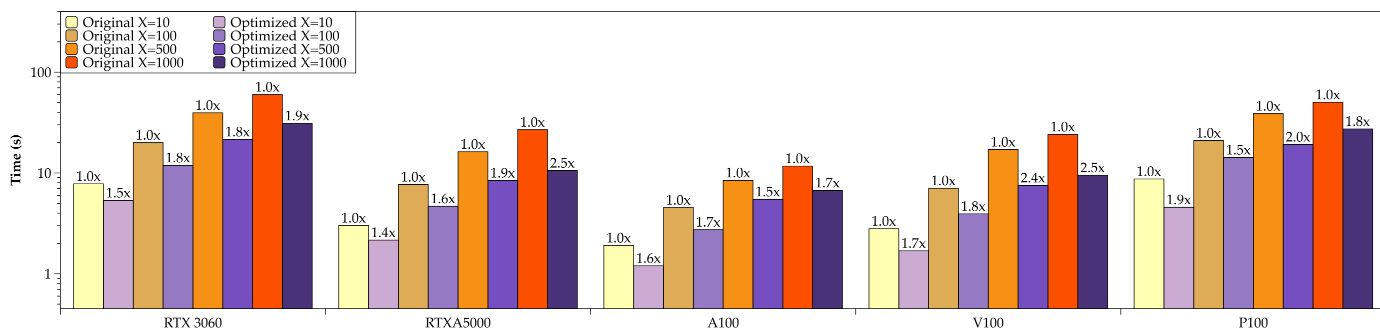


Fig. 6. Runtime comparison of the Starlight -optimized LOGAN kernel on multiple GPU boards with four X values and relative speedup (log-scale).

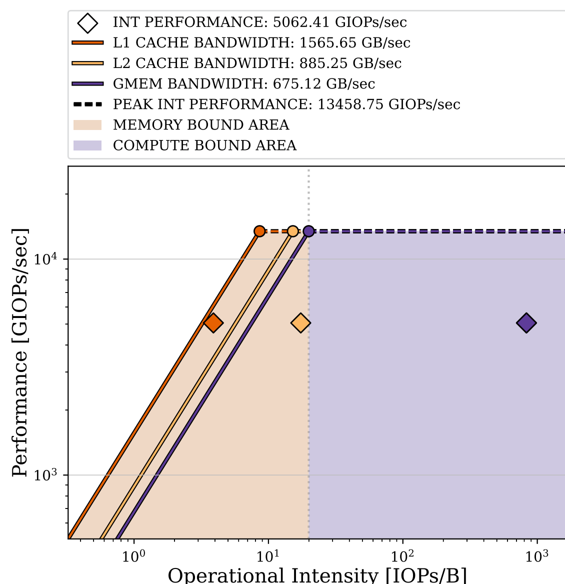


Fig. 7. Roofline plot output of the Roofline generator for optimized LOGAN on the NVIDIA RTX A5000 with $X = 100$.

nificant change in the code's structure, such as reordering most of the kernel's operations. Besides, although identifying these issues further confirms Starlight's efficiency, the required additional optimizations are beyond the scope of this example, whose objective, as previously mentioned, is to show how already available optimized GPU applications can still be improved and how our tool can aid this process.

5.3. LOGAN analysis and optimizations

LOGAN [5] implements a high-performance algorithm for the pairwise alignment of genome sequences [52–55]. X-drop [46] is a heuristic that avoids the entire quadratic cost of exact alignment algorithms such as Needleman-Wunsch [56] and Smith-Waterman [57] by searching only for high-quality alignments. In practice, X-drop eliminates searches between sequences that are clearly diverging. Indeed, instead of exploring the whole $m \times n$ space (where m and n are the lengths of the sequences to align), X-drop searches only for alignments that result in limited edits between the two sequences. Moreover, to reduce the search space, the algorithm keeps a maximum running score and does not explore cell neighborhoods whose score decreases below a user-specified parameter X.

Starlight analysis of the original version of LOGAN highlighted multiple performance issues. In particular, the tool identified multiple dependencies and latency issues and suggested inlining the highlighted functions accordingly and reordering some intensive operations. Besides, Starlight also recognized other stalls related to memory throttling and poor branching performance. Given this context, we solved the issues related to data dependency and latency by rescheduling some operations in the priors parts of the code, which led to a significant gain in performance. Then, Starlight suggested adapting the number of scheduled threads according to the input X. More specifically, changing X impacts the algorithm runtime: increasing X renders the heuristic less aggressive and causes the algorithm to run longer, whereas decreasing it makes the alignment stop sooner. For this reason, the tool advised reducing the number of threads when using a small value of X, leading to enhancements in performance and branch efficiency since we improved resource usage and diminished thread stalling. On the other hand, with a larger X, Starlight suggested raising the number of threads to speed up the alignment process, as the computation of the alignment matrix requires much more time in this scenario.

Fig. 6 compares the results of the original and optimized versions of LOGAN when run on various GPUs using multiple X values. To directly compare our design against the original implementation of LOGAN, we used the same dataset of $100K$ long reads employed by the authors of LOGAN for their experiments on the original version of the code. Our optimizations led to an average speedup of 1.73× when comparing the original LOGAN software to the Starlight -optimized one, while Starlight's predicted performance for the indicated stalls showed an average of 1.75× performance improvements. In contrast, Fig. 7 shows the final Roofline attained with the performance improvements of Starlight. We can observe that the kernel performance is touching the Roofline when looking at the line described by the L1 cache bandwidth. Due to the structure of LOGAN's heuristic algorithm and the multiple dependencies present in the kernel, improving the kernel's performance would require significant changes in the code structure, which, as previously stated, is beyond the scope of these examples.

5.4. Mutual information analysis and optimizations

Image Registration is the procedure of aligning a *floating* image F to a *reference* one R, widely employed in multiple and different fields, ranging from medicine to satellites [59]. Identifying the geometric transformation for such an alignment is a compute-intensive optimization process that requires calculating the likeness of the two images iteratively through a similarity metric. *MI* is one of the most employed metrics in this context [60], as well as in domains such as genomics [61], relevance networks [62], Hidden Markov Models training [63], and features selection [64]. In particular, the MI concept comes from Information Theory, and it measures the statistical dependency of two random variables, F and R (the two images in our case) [65]. MI computation requires first calculating the joint and single histograms of the two images, then, according to Shannon's equations [66], deriving the entropies from such histograms. Finally, the aggregation of entropy values produces the MI.

Our implementation builds upon the state-of-the-art open-source implementation design[5] [47], which offers an FPGA-accelerated kernel

[5] https://github.com/necst/iron.

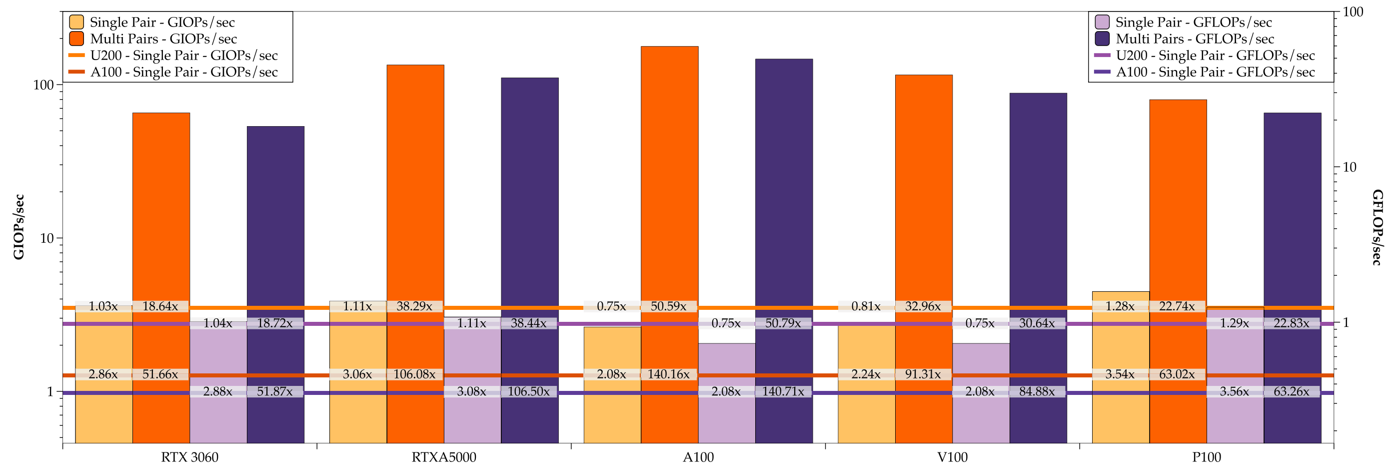


Fig. 8. Performance comparison of the Starlight -optimized Mutual Information kernel on multiple GPU boards against FPGA [47] and GPU [12,58] in terms of GFLOPs/sec and GIOPs/sec (log-scale). The speedup of each GPU version refers to the corresponding single-/multi-pair FPGA/GPU design on the corresponding line.

for the MI process. This implementation closely resembles the aforementioned algorithm but needs relevant changes to become suitable for GPU. Our initial solution employed multiple threads to compute the histograms directly on GMEM through atomic instructions. Starlight suggested exploiting the shared memory to reduce memory bottlenecks. In this way, we entirely privatized the single histogram computation and partially the joint histogram one. Indeed, since the entire joint histogram cannot fit into shared memory, we only store a portion of the input images on shared memory for fast and coalesced memory access during its computation. Since we were only using a single block for the MI computation of a single couple of images, Starlight also advised increasing the number of scheduled GPU blocks to boost the performance further. Generally, the number of MI calculations depends on both the convergence of the alignment process of an image couple and the number of active registration processes. Thus, once integrated within an Image Registration framework, our solution can support the parallel registration of multiple image couples.

To evaluate our design, we used a medical dataset of 227 Computed Tomography (CT) images with a dimension of 512×512 pixels, and a corresponding number of Positron Emission Tomography (PET) ones, resized from 128×128 pixels to a dimension of 512×512 pixels, each down-scaled to 8-bit data width[6] [67]. Furthermore, we compare our results against the state-of-the-art and open-source FPGA-accelerated implementation of the same algorithm proposed by Conficconi et al. [47] running on the accelerator card Alveo U200 and against a GPU solution exploiting PyTorch [12,58] running on an NVIDIA A100. Fig. 8 shows the performance comparison of our design against the FPGA and GPU state-of-the-art solutions, while Fig. 9 displays the Roofline Model of the final design of our MI kernel. When computing the MI of a single image couple at a time, we can observe that our implementation achieves similar performance to the FPGA design. Besides, our designs attain a significantly higher accuracy since the FPGA solution computes entropies using 23-bit fixed-point operations, whereas we maintain single-precision (32-bit) FP. The FPGA implementation supports up to 4 parallel kernels due to the number of available off-chip memory banks of the Alveo U200; thus, our approach outperforms the FPGA version up to 12.7× in terms of GFLOPs/sec when computing multiple MI in parallel. The state-of-the-art GPU solution is significantly under-performing with respect to our implementation, showing that our design is 2× faster even in its worst-performing instance. Furthermore, the PyTorch solution has no support for the parallel computation of multiple image couples at the same time; thus when computing multiple MI instances at the same time, our solution is capable of achieving performance improvements of over 140×. Concerning the performance predictions of Starlight, the first version of the code for MI computation showed an average performance of 52.74 GIOPs/sec and 8.66 GFLOPs/sec, and Starlight estimated a performance improvement for our applied optimizations of 2.59× and 4.42× for integer and FP performance respectively, while we attained a measured improvement of 2.55× and 4.32×, confirming the accuracy of Starlight's predictions. On the Roofline Model, we can observe that our solution is very close to touching the roof of the L1 cache. Starlight correctly highlighted that the dependency causing this bottleneck is related to the numerous atomic additions required to compute the MI exactly. If we removed this constraint our kernel would touch the roof, indicating that our implementation is indeed optimal.

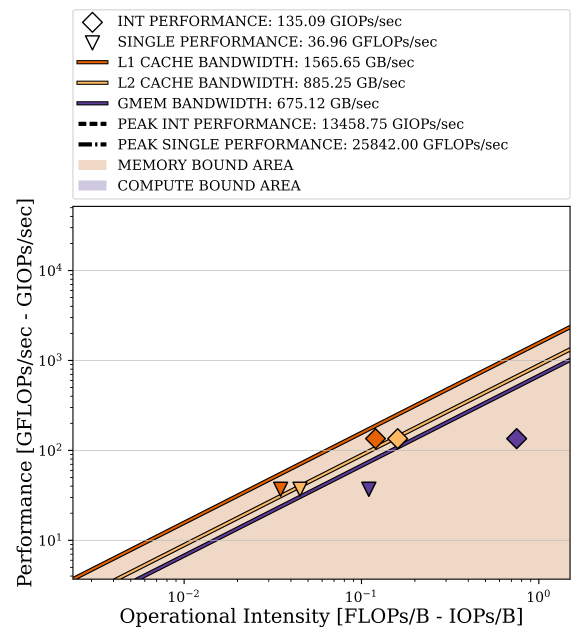


Fig. 9. Roofline plot output of the Roofline generator for optimized Mutual Information kernel on the NVIDIA RTX A5000.

6. Related work

In this Section, we first overview the related work regarding the various extensions of the Roofline Model for GPUs and tools for GPU

[6] Patient: C3N-00704, Study: Dec 10, 2000 NM PET 18 FDG SKULL T, CT: WB STND, PET: WB 3D AC.

kernel analysis and optimization. Then, we summarize how Starlight differs from other literature solutions through a qualitative comparison.

6.1. GPU roofline model

In recent years, several studies in the literature proposed extensions to the classical CPU-centric Roofline Model in terms of memory hierarchies/levels and target architectures. Indeed, the traditional model only investigates the bottlenecks related to GMEM and peak performance. For this reason, many researchers developed multiple extensions that consider different memory hierarchies to enable a deeper understanding of various architectures' performance [37,27,38,39]. For instance, Ilic et al. [37] and Koskela et al. [39] included multiple levels of caches (typically L1 and L2 caches) along with the more traditional off-chip memory in the Roofline Model for CPUs. Yang et al. [38] proposed ERT to determine the peak capabilities of the Volta generation of NVIDIA GPUs through micro-benchmarks automatically. Specifically, the tool can depict GMEM and GPU caches but does not benchmark a kernel's performance against the Roofline.

On the architectural side, multiple studies adapted the Roofline Model to GPUs, incorporating the support for multiple memory levels [25,29,30,40,41]. Besides, some of these approaches also introduced information regarding integer operation performance [29,30,42,43]. Despite the relevant and attractive features of such studies, they showcase some limitations regarding usability, portability, and considered GPU characteristics. For instance, some tools do not offer automatic GPU detection but instead rely on a preset of performance measurements to define peak performance and memory bandwidth [25,29,30]. Moreover, such tools require the user to instrument the target application and extract performance metrics manually. On the other hand, other approaches offer useful micro-benchmarks to empirically evaluate (part of) the computing or memory capabilities of a given GPU [40–43]. However, it is then up to the user to manually plot the Roofline Model and map the target application onto it. NVIDIA proposed an automatic definition of the Roofline Model inside the recently released suite for their GPUs (starting from the Volta generation), called NVIDIA NSIGHT [23], bringing this model to a much broader user base. Nonetheless, the tool has some limitations; for instance, it exclusively benchmarks FP computations and displays cache hierarchies only if the user provides data regarding the cache bandwidths.

Moving to GPUs by other vendors, Intel Advisor [68] is a tool that advises the user during the development of performant code for Intel's CPUs and GPUs. Among the various available features, this tool can automatically construct a Roofline Model for GPUs (and CPUs) that includes multiple performance and memory roofs (e.g., for integer/FP operations and cache levels). In addition, Intel Advisor offers interesting utilities such as optimization suggestions and performance estimations when porting (part of) an application from CPU (GPU) to (another) GPU. Lastly, Leinhauser et al. [26] proposed a Roofline Model definition for AMD GPUs. Initially, the authors planned to employ the AMD ROC Profiler (rocProf) [69] to build the Roofline Model and map a given application onto it; however, they then switched to micro-benchmarks [70,71] since rocProf did not acquire enough metrics (e.g., memory bandwidth/transactions).

Given the different approaches in the literature, we identified the most prominent features we believe a tool for building a Roofline Model for GPUs should implement (e.g., supported data types). Table 3 reports a qualitative overview of the discussed tools and our own according to such features.

6.2. GPU analysis and optimization tools

Fully exploiting the capabilities of GPUs requires a significant understanding of the underlying architecture and expertise on the developer's side. For this reason, researchers proposed multiple tools to aid non-experienced users in analyzing and optimizing GPU kernels. In particular, analysis tools profile a target application through binary instrumentation and hardware performance counters. In contrast, the optimization ones exploit analysis information to provide detailed suggestions to improve the GPU kernel performance. Given these premises, this Section discusses the most prominent GPU analysis and optimization tools. In particular, we concentrate on GPUs by NVIDIA since Starlight targets such vendor's devices.

Starting from the first category (i.e., analysis tools), NVBit [19] is a framework for dynamic binary instrumentation that offers various high-level APIs to analyze applications for NVIDIA GPUs. In particular, NVBit leverages dynamic recompilation at the SASS level to produce instrumented code compliant with the target architecture. Besides, it can also deal with pre-compiled binaries and libraries. However, despite these features, NVBit also exhibits some limitations. For instance, it does not support applications employing *shared* or *constant* GPU memory or accelerated libraries (e.g., cuBLAS, cuTENSOR). Additionally, this tool, as others based on code instrumentation, may generally incur significant overheads to profile the target application accurately. CUDA Flux [20] is an alternative to profiling based on hardware performance counters. Such a tool requires code instrumentation through LLVM to collect statistics about the control flow. Then, it calculates the resulting instruction count based on these statistics combined with an analysis at PTX level (the low-level parallel-thread execution virtual machine and instruction set) [34]; however, the exact characterization of all types of PTX instructions may result in large verbosity. Since CUDA Flux profiles only one thread to limit the execution overhead, this tool produces meaningful results for kernels with regular compute patterns. Besides, it does not support texture memory, concurrent kernels, and non-inlined functions. Finally, CUDAAdvisor [22] is a framework for fine-grain code instrumentation and performance analysis based on LLVM. It offers a feature set similar to NVIDIA SASSI [74] and overcomes limitations such as portability and expansibility. However, it only supports NVIDIA GPUs up to Maxwell architectures.

Moving to tools that combine analysis and optimization guidance, Zhou et al. [15,16] recently presented GPA. This performance advisor analyzes instruction samples to guide performance optimization on NVIDIA GPUs. In particular, GPA attributes stalls to their causes, matches patterns of inefficiency with optimization strategies, and estimates the potential speedup for each applicable optimization. Finally, GPA combines metrics extracted from instruction sampling and binary instrumentation to yield a comprehensive performance report. Despite these features and advantages, GPA exhibits some limitations. On the one hand, the binary instrumentation causes the expected overhead; on the other, GPA lacks support for thread divergence and warp synchronization detection and optimization and does not correlate kernel improvements with the underlying hardware capabilities. For instance, if a target kernel is memory-bound, GPA cannot suggest further enhancements to improve its performance. Zhou et al. also presented an extension to HPCToolkit [18] for GPU performance modeling [17]. This work attributes metrics to calling contexts spanning both CPU and GPU, it constructs approximations of call path profiles for GPU computations, and it employs a wait-free data structure to coordinate monitoring and attribution of GPU performance metrics. In particular, HPCToolkit retrieves such metrics from performance counter samples and attributes them to source lines and loops, enabling fine-grain analysis and tuning. After receiving the tool's detailed suggestions, the developer can then optimize the code. Generally, HPCToolkit showcases limitations similar to GPA. Hong et al. proposed SAAKE [21], a kernel emulator and bottleneck analyzer for GPU code optimizations based on latency and throughput as performance metrics. After identifying the bottlenecks and their causes, the user can exploit such information to optimize the code or rely on optimizers such as OpenTuner. Regarding the limitations, the overall analysis is performed solely on emulation for a single architecture and kernels with regular execution patterns; thus, SAAKE's characterization may partially represent the target GPU, and the internal profiling model might require modifications to support additional

Table 3
Comparison between tools implementing the Roofline Model for GPU.

Work	Data Types Integer	FP	Cache Levels	Automatic Roof. Gen.	Open Source	Built Upon	Supported Vendor
[25]	✗	✓	✓	✗	✗	CUPTI [35]	NVIDIA
[29,30]	✓	✓	✓	✗	✗	NVPROF [24]	NVIDIA
[40,41]	✗	✓	✓	✗	✓	Micro-benchmarks	NVIDIA
[42,43]	✓	✓	✗	✗	✓	Micro-benchmarks	NVIDIA
[23]	✗	✓	✗	✓	✗	NSIGHT [23]	NVIDIA
[68]	✓	✓	✓	✓	✗	Intel Advisor [68]	Intel
[26]	✗	✓	✗	✗	✗	Micro-benchmarks	AMD
Ours	✓	✓	✓	✓	✓	**CUPTI [35]**	**NVIDIA**

Table 4
Comparison between tools for performance analysis on NVIDIA GPUs.

Work	Open Source	Profiling Yes/No	Profiling Level	Profiling Built Upon	Profiling HW Performance Correlation	Profiling Other Info	Optimization Yes/No	Optimization Stall-Code Correlation	Optimization Improvement Estimation
[19]	✓	✓	SASS	CUDA [72]	✗	Binary Instrumentation, No Shared/Constant Memory	✓	✗	✗
[20]	✓	✓	PTX	LLVM [73]	✗	Regular Compute Patterns only, No Concurrent Kernels, No Inlined Functions	✓	✗	✗
[22]	✓	✓	PTX	LLVM [73]	✗	Support only for GPUs up to Maxwell	✓	✗	✗
[15,16]	✓	✓	SASS	CUPTI [35]	✗	Binary Instrumentation	✓	✓	✓
[17]	✓	✓	SASS	CUPTI [35] and HPCToolkit [18]	✗	Binary Instrumentation	✓	✓	✗
[21]	✗	✓	PTX	CUDA [72]	✗	Emulation Only, Model Tuning Required for every Kernel	✗	✗	✗
[24]	✗	✓	SASS	CUDA [72]	✗	Profiling Only	✓	✗	✗
[23]	✗	✓	SASS	CUDA [72]	✗	Profiling Only	✓	✗	✗
Ours	✓	✓	**SASS**	**CUPTI**[35]	✓ **(w/ Roofline)**	**Binary Instrumentation**	✓	✓	✓

GPUs or applications. NVIDIA proposed multiple profilers for GPU code over the years, namely NVPROF [24] and NSIGHT [23]. These provide multiple suggestions to the end-user, mostly relative to device utilization and kernel's stalls, but do not identify specific code regions where the optimizations should be applied. Besides, NSIGHT defines the Roofline Model for the target kernel, as we stated before.

As in Section 6.1, we collected the principal qualitative characteristics of the analyzed tools and Starlight in Table 4.

6.3. Discussion

Tables 3 and 4 provide a comprehensive qualitative comparison of the various studies in the literature on GPU kernel analysis and optimization, including our own. The investigation and comprehension of these prominent studies and their features represented the primary inspiration for Starlight, which aims to improve and address the current limitations of the literature on Roofline Model-based analysis and optimization tools for NVIDIA GPUs.

Concerning GPU Roofline Model definition, generally, all the solutions in the literature either lack flexibility or features concerning data types or memory hierarchy (Table 3). Only the solution proposed by Ding et al. [29,30] and Intel Advisor [68] can define a Model as feature-rich as the one described in this paper. However, the former supports two models of GPUs only, while Starlight supports all the critical aspects of the Roofline Model definition. On the other hand, the latter offers additional valuable features other than the Roofline Model generation, such as CPU-to-GPU and GPU-to-GPU migration, which may become intriguing future components of Starlight. Focusing on GPU vendors other than NVIDIA and Intel, AMD, as mentioned in Section 6.1, developed rocProf, which offers performance measurements to the end users, although being limited when compared to other profilers such as NSIGHT, NCU, and Intel Advisor. However, given the recent efforts in the open-source community by AMD, we believe that these limitations might be solved in the near future, enabling Starlight to support AMD GPUs.

Table 4 shows an overview of the tools focusing on NVIDIA GPU performance and bottleneck analysis, highlighting the various supported features of the tools and their limitations. We can observe that most state-of-the-art tools lack support for an accurate correlation of source code and hardware capabilities with the kernel's performance. Among such tools, GPA [15,16] is the most feature-rich tool available in the literature. However, it does not consider the capabilities of the underlying hardware (e.g., resource availability), making it less effective when analyzing poorly implemented stall-less kernels. Furthermore, these tools incur very expensive performance overheads (often requiring for more than 10× of the application's run-time) and only support a few generations of NVIDIA GPUs. Instead, Starlight can provide suggestions to the user by defining a Roofline Model of the target kernel that considers the capabilities of the target GPU hardware while also performing in-depth performance analysis account for an average overhead of only 4.85× in terms of the application's execution time.

As previously mentioned, Starlight's current implementation still leaves open research paths. Currently, Starlight only supports the benchmarking of applications on single GPU systems. Additionally, given the aforementioned complexities in implementing GPU code, we understand that non-experienced users might still have difficulties in applying Starlight's suggested optimizations. Indeed, an additional feature that one might consider implementing in Starlight is the automatic parsing and optimization of the original code. To perform this task, one would parse the optimization suggestions provided by Starlight and the correspondent file to which they refer and implement some of these suggestions without the user's intervention. On the one hand, some optimizations like function inlining and loop unrolling might be more straightforward to implement in this manner; on the other hand, optimizations like code reordering and restructuring may still require partial user intervention.

7. Conclusions

In this work, we presented Starlight, an open-source tool for GPU kernel optimization based on the Roofline Model. First, we provided a detailed description of the tool's structure, highlighting the mechanisms behind its various modules. We defined how Starlight exploits the Roofline Model to provide accurate and practical suggestions to the end-user for optimizing the target kernel. Then, we proved the tool's effectiveness by analyzing multiple state-of-the-art algorithms and optimizing them through Starlight. In particular, our final designs achieve performance improvements ranging from 1.10× to 2.5×. In addition, we showcased how Starlight can support the development of an application from the ground up targeting a MI computation. Our solution reaches a final performance improvement up to a factor of 12.7× compared to the literature implementation FPGA implementation and over 140× when compared against the state-of-the-art GPU solution. Finally, we provided an overview of the various state-of-the-art solutions for GPU kernel optimization and Roofline Model generation, qualitatively evaluating their feature set against the one provided by Starlight.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Alberto Zeni reports a relationship with Xilinx Inc., Advanced Micro Devices Inc., NVIDIA Corporation, Massachusetts Institute of Technology and with Dana-Farber Cancer Institute.

Emanuele Del Sozzo reports a relationship with RIKEN Center for Computational Science, University of Toronto, IBM and with Massachusetts Institute of Technology.

Eleonora D'Arnese reports a relationship with ETH Zurich and University of Illinois Chicago.

Davide Conficconi reports a relationship with Xilinx Inc. that includes, Advanced Micro Devices Inc., NVIDIA Corporation, IBM and with Huawei Technologies.

Marco Domenico Santambrogio reports a relationship with NVIDIA Corporation, Oracle Corporation, Xilinx Inc., Advanced Micro Devices Inc. and with Huawei Technologies Co Ltd.

Acknowledgments

Data used in this publication were generated by the National Cancer Institute Clinical Proteomic Tumor Analysis Consortium (CPTAC). The Authors would like to thank the NVIDIA University Program for the hardware donations and Oracle Research Program for the Oracle Cloud Credits.

References

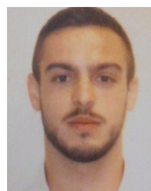
[1] J.M. Shalf, R. Leland, Computing beyond Moore's law, Computer 48 (2015) (SAND-2015-8039J).
[2] T. Sterling, Hpc in phase change: towards a new execution model, in: International Conference on High Performance Computing for Computational Science, Springer, 2010, p. 31.
[3] A.A. Chien, V. Karamcheti, Moore's law: the first ending and a new beginning, Computer 46 (12) (2013) 48–53.
[4] T.N. Theis, H.-S.P. Wong, The end of Moore's law: a new beginning for information technology, Comput. Sci. Eng. 19 (2) (2017) 41.
[5] A. Zeni, G. Guidi, M. Ellis, N. Ding, M.D. Santambrogio, S. Hofmeyr, A. Buluç, L. Oliker, K. Yelick Logan, High-performance gpu-based x-drop long-read alignment, in: 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2020, pp. 462–471.
[6] A. Zeni, G.W. Di Donato, L. Di Tucci, M. Rabozzi, M.D. Santambrogio, The importance of being x-drop: high performance genome alignment on reconfigurable hardware, in: 2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), IEEE, 2021, pp. 133–141.
[7] A. Zeni, K. O'Brien, M. Blott, M.D. Santambrogio, Optimized implementation of the hpcg benchmark on reconfigurable hardware, in: European Conference on Parallel Processing, Springer, 2021, pp. 616–630.
[8] X. Shi, Z. Zheng, Y. Zhou, H. Jin, L. He, B. Liu, Q.-S. Hua, Graph processing on gpus: a survey, ACM Comput. Surv. 50 (6) (2018) 1–35.
[9] E.D. Sozzo, D. Conficconi, A. Zeni, M. Salaris, D. Sciuto, M.D. Santambrogio, Pushing the level of abstraction of digital system design: a survey on how to program fpgas, ACM Comput. Surv. 55 (5) (dec 2022), https://doi.org/10.1145/3532989.
[10] D. Conficconi, E. Del Sozzo, F. Carloni, A. Comodi, A. Scolari, M.D. Santambrogio, An energy-efficient domain-specific architecture for regular expressions, IEEE Trans. Emerg. Top. Comput. (2022).
[11] A. Caulfield, P. Costa, M. Ghobadi, Beyond smartnics: towards a fully programmable cloud, in: 2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR), IEEE, 2018, pp. 1–6.
[12] E. D'Arnese, E. Del Sozzo, D. Conficconi, M.D. Santambrogio, Exploiting heterogeneous architectures for rigid image registration, in: 2021 IEEE Biomedical Circuits and Systems Conference (BioCAS), IEEE, 2021, pp. 1–5.
[13] S.A. Manavski, G. Valle, Cuda compatible gpu cards as efficient hardware accelerators for Smith-Waterman sequence alignment, BMC Bioinform. 9 (2) (2008) S10.
[14] M. Taher, Accelerating scientific applications using gpu's, in: 2009 4th International Design and Test Workshop (IDT), IEEE, 2009, pp. 1–6.
[15] K. Zhou, X. Meng, R. Sai, D. Grubisic, J. Mellor-Crummey, An automated tool for analysis and tuning of gpu-accelerated code in hpc applications, IEEE Trans. Parallel Distrib. Syst. 33 (4) (2021) 854–865.
[16] K. Zhou, X. Meng, R. Sai, J. Mellor-Crummey, Gpa: a gpu performance advisor based on instruction sampling, in: 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), IEEE, 2021, pp. 115–125.
[17] K. Zhou, M.W. Krentel, J. Mellor-Crummey, Tools for top-down performance analysis of gpu-accelerated applications, in: Proceedings of the 34th ACM International Conference on Supercomputing, 2020, pp. 1–12.
[18] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, N.R. Tallent, Hpctoolkit: tools for performance analysis of optimized parallel programs, Concurr. Comput., Pract. Exp. 22 (6) (2010) 685–701.
[19] O. Villa, M. Stephenson, D. Nellans, S.W. Keckler, Nvbit: a dynamic binary instrumentation framework for nvidia gpus, in: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, 2019, pp. 372–383.
[20] L. Braun, H. Fröning, Cuda flux: a lightweight instruction profiler for cuda applications, in: 2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), IEEE, 2019, pp. 73–81.
[21] C. Hong, A. Sukumaran-Rajam, J. Kim, P.S. Rawat, S. Krishnamoorthy, L.-N. Pouchet, F. Rastello, P. Sadayappan, Gpu code optimization using abstract kernel emulation and sensitivity analysis, in: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2018, pp. 736–751.
[22] D. Shen, S.L. Song, A. Li, X. Liu, Cudaadvisor: Llvm-based runtime profiling for modern gpus, in: Proceedings of the 2018 International Symposium on Code Generation and Optimization, 2018, pp. 214–227.
[23] N. Corporation, Nvidia® nsight Compute, NVIDIA Corporation, 2020, https://developer.nvidia.com/nsight-compute.
[24] N. Corporation, Nvprof: Nvidia® gpu profiler, NVIDIA corporation, https://docs.nvidia.com/cuda/profiler-users-guide/index.html, 2012.
[25] A. Lopes, F. Pratas, L. Sousa, A. Ilic, Exploring gpu performance, power and energy-efficiency bounds with cache-aware roofline modeling, in: 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), IEEE, 2017, pp. 259–268.
[26] M. Leinhauser, R. Widera, S. Bastrakov, A. Debus, M. Bussmann, S. Chandrasekaran, Metrics and design of an instruction roofline model for amd gpus, ACM Trans. Parall. Comput. 9 (1) (2022) 1–14.
[27] Y.J. Lo, S. Williams, B. Van Straalen, T.J. Ligocki, M.J. Cordery, N.J. Wright, M.W. Hall, L. Oliker, Roofline model toolkit: a practical tool for architectural and program analysis, in: International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, Springer, 2014, pp. 129–148.
[28] M. Siracusa, E. Delsozzo, M. Rabozzi, L. Di Tucci, S. Williams, D. Sciuto, M.D. Santambrogio, A comprehensive methodology to optimize fpga designs via the roofline model, IEEE Trans. Comput. (2021).
[29] N. Ding, S. Williams, An Instruction Roofline Model for Gpus, IEEE, 2019.
[30] N. Ding, M. Awan, S. Williams, Instruction roofline: an insightful visual performance model for gpus, Concurr. Comput., Pract. Exp. (2021) e6591.
[31] S. Williams, A. Waterman, D. Patterson, Roofline: an insightful visual performance model for multicore architectures, Commun. ACM 52 (4) (2009) 65–76.
[32] N. Corporation, Nvidia® nsight Systems, NVIDIA Corporation, 2020, https://developer.nvidia.com/nsight-systems.
[33] V. Volkov, Understanding Latency Hiding on GPUs, University of California, Berkeley, 2016.
[34] N. Corporation, Parallel thread execution isa, NVIDIA Corporation (2006), https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.
[35] N. Corporation, Cupti: cuda profiling tools interface, NVIDIA Corporation (2006), https://docs.nvidia.com/cuda/cupti/index.html.
[36] G.M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, in: Proceedings of the April 18-20, 1967, pp. 483–485, Spring joint computer conference, 1967.
[37] A. Ilic, F. Pratas, L. Sousa, Cache-aware roofline model: upgrading the loft, IEEE Comput. Archit. Lett. 13 (1) (2013) 21–24.
[38] L.B.N. Laboratory, Empirical roofline tool, https://crd.lbl.gov/divisions/amcr/computer-science-amcr/par/research/roofline/software/ert/, 2020.

[39] T. Koskela, Z. Matveev, C. Yang, A. Adedoyin, R. Belenov, P. Thierry, Z. Zhao, R. Gayatri, H. Shan, L. Oliker, J. Deslippe, R. Green, S. Williams, A novel multi-level integrated roofline model approach for performance characterization, in: R. Yokota, M. Weiland, D. Keyes, C. Trinitis (Eds.), High Performance Computing, Springer International Publishing, Cham, 2018, pp. 226–245.
[40] D. Ernst, M. Holzer, G. Hager, M. Knorr, G. Wellein, Analytical performance estimation during code generation on modern gpus, J. Parallel Distrib. Comput. 173 (2023) 152–167.
[41] D. Ernst, G. Hager, J. Thies, G. Wellein, Performance engineering for real and complex tall & skinny matrix multiplication kernels on gpus, Int. J. High Perform. Comput. Appl. 35 (1) (2021) 5–19.
[42] E. Konstantinidis, Y. Cotronis, A quantitative roofline model for gpu kernel performance estimation using micro-benchmarks and hardware metric profiling, J. Parallel Distrib. Comput. 107 (2017) 37–56.
[43] E. Konstantinidis, Y. Cotronis, A practical performance model for compute and memory bound gpu kernels, in: 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, IEEE, 2015, pp. 651–658.
[44] E. Del Sozzo, M. Rabozzi, L. Di Tucci, D. Sciuto, M. Santambrogio, A scalable FPGA design for cloud n-body simulation, in: 2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP), 2018, pp. 1–8.
[45] L. Nylons, M. Harris, J. Prins, Fast n-body simulation with cuda, GPU gems 3, 2007, pp. 62–66.
[46] Z. Zhang, S. Schwartz, L. Wagner, W. Miller, A greedy algorithm for aligning DNA sequences, J. Comput. Biol. 7 (1–2) (2000) 203–214.
[47] D. Conficconi, E. D'Arnese, E. Del Sozzo, D. Sciuto, M.D. Santambrogio, A framework for customizable fpga-based image registration accelerators, in: The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2021, pp. 251–261.
[48] G. Sorrentino, M. Venere, E. D'Arnese, D. Conficconi, I. Poles, M. Santambrogio, ATHENA: A GPU-based framework for biomedical 3D rigid image registration, in: IEEE Biomedical Circuits and Systems Conference (BioCAS), 2023, pp. 1–5.
[49] E. D'Arnese, D. Conficconi, E. Del Sozzo, L. Fusco, D. Sciuto, M. Santambrogio Faber, A hardware/software toolchain for image registration, IEEE Trans. Parallel Distrib. Syst. 34 (2022) 291–303.
[50] E. D'Arnese, D. Conficconi, M. Santambrogio, D. Sciuto, Reconfigurable architectures: The shift from general systems to domain specific solutions, in: Emerging Computing: From Devices to Systems: Looking Beyond Moore and Von Neumann, 2022, pp. 435–456.
[51] G. Sorrentino, M. Venere, D. Conficconi, E. D'Arnese, M. Santambrogio, Hephaestus: Codesigning and automating 3D image registration on reconfigurable architectures, ACM Trans. Embed. Comput. Syst. 22 (2023) 1–24.
[52] A. Zeni, G. Di Donato, A. Della Valle, F. Carloni, M. Santambrogio, On the genome sequence alignment FPGA acceleration via KSW2z, in: 2023 IEEE International Symposium on Circuits and Systems (ISCAS), 2023, pp. 1–5.
[53] G. Gerometta, A. Zeni, M. Santambrogio, TSUNAMI: A GPU implementation of the WFA algorithm, in: 2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT), 2023, pp. 150–161.
[54] B. Branchini, G. Gerometta, L. Cicolini, A. Zeni, E. Del Sozzo, M. Santambrogio, Surfing the wavefront of genome alignment, in: 2022 IEEE International Symposium on Circuits and Systems (ISCAS), 2022, pp. 1754–1758.
[55] A. Zeni, F. Peverelli, E. Cabri, L. Di Tucci, L. Cerina, M. Santambrogio, circFA: A FPGA-based circular RNA aligner, in: 2019 IEEE EMBS International Conference on Biomedical & Health Informatics (BHI), 2019, pp. 1–4.
[56] S.B. Needleman, C.D. Wunsch, A general method applicable to the search for similarities in the amino acid sequence of two proteins, J. Mol. Biol. 48 (3) (1970) 443–453.
[57] T.F. Smith, M.S. Waterman, et al., Identification of common molecular subsequences, J. Mol. Biol. 147 (1) (1981) 195–197.
[58] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, A. Lerer, Automatic differentiation in pytorch, in: NIPS 2017 Workshop on Autodiff, 2017, pp. 1–4, https://openreview.net/forum?id=BJJsrmfCZ.
[59] I. Stratakos, D. Gourounas, V. Tsoutsouras, T. Economopoulos, G. Matsopoulos, D. Soudris, Hardware acceleration of image registration algorithm on fpga-based systems on chip, in: Proceedings of the International Conference on Omni-Layer Intelligent Systems, 2019, pp. 92–97.
[60] F. Maes, A. Collignon, D. Vandermeulen, G. Marchal, P. Suetens, Multimodality image registration by maximization of mutual information, IEEE Trans. Med. Imaging 16 (2) (1997) 187–198.
[61] F. Lichtenstein, F. Antoneli, M.R. Briones Mia, Mutual information analyzer, a graphic user interface program that calculates entropy, vertical and horizontal mutual information of molecular sequence sets, BMC Bioinform. 16 (1) (2015) 1–19.
[62] A.J. Butte, I.S. Kohane, Mutual information relevance networks: functional genomic clustering using pairwise entropy measurements, in: Biocomputing 2000, World Scientific, 1999, pp. 418–429.
[63] L. Bahl, P. Brown, P. De Souza, R. Mercer, Maximum Mutual Information Estimation of Hidden Markov Model Parameters for Speech Recognition, ICASSP'86. IEEE International Conference on Acoustics, Speech, and Signal Processing, vol. 11, IEEE, 1986, pp. 49–52.
[64] P.A. Estévez, M. Tesmer, C.A. Perez, J.M. Zurada, Normalized mutual information feature selection, IEEE Trans. Neural Netw. 20 (2) (2009) 189–201.
[65] T.M. Cover, Elements of Information Theory, John Wiley & Sons, 1999.
[66] C.E. Shannon, A mathematical theory of communication, Bell Syst. Tech. J. 27 (3) (1948) 379–423.
[67] K. Clark, B. Vendt, K. Smith, J. Freymann, J. Kirby, P. Koppel, S. Moore, S. Phillips, D. Maffitt, M. Pringle, et al., The cancer imaging archive (tcia): maintaining and operating a public information repository, J. Digit. Imag. 26 (6) (2013) 1045–1057.
[68] Intel, Intel Advisor, https://www.intel.com/content/www/us/en/developer/tools/oneapi/advisor.html, 2021.
[69] A.M.D. (AMD) Roc-profiler ROCm-Developer-Tools, https://github.com/ROCm-Developer-Tools/rocprofiler, 2020.
[70] T. Deakin, J. Price, M. Martineau, S. McIntosh-Smith, Gpu-stream v2. 0: benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models, in: High Performance Computing: ISC High Performance 2016 International Workshops, ExaComm, e-MuCoCoS, HPC-IODC, IXPUG, IWOPH, P^ 3MA, VHPC, WOPSSS, Frankfurt, Germany, June 19–23, Springer, 2016, p. 2016, pp. 489–507, Revised Selected Papers 31.
[71] E. Konstantinidis, Y. Cotronis, A quantitative performance evaluation of fast on-chip memories of gpus, in: 2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), IEEE, 2016, pp. 448–455.
[72] N. Corporation, Nvidia® cuda, NVIDIA corporation, http://developer.nvidia.com/object/cuda.html, 2006.
[73] C. Lattner, V. Adve Llvm, A compilation framework for lifelong program analysis & transformation, in: International Symposium on Code Generation and Optimization, in: CGO 2004, IEEE, 2004, 2004, pp. 75–86.
[74] NVlabs, SASSI instrumentation tool for NVIDIA GPUs, https://github.com/NVlabs/SASSI, 2015.

Alberto Zeni is a Ph.D. Candidate in Information Technology at Politecnico di Milano. He received his B.Sc. and M.Sc. in Computer Engineering from Politecnico di Milano in 2017 and 2019 respectively. His research interesets revolves around heterogeneous architectures, especially GPUs and FPGAs, genomics, computer architectures and High Performance Computing.

Emanuele Del Sozzo got his Ph.D. in Information Technology from Politecnico di Milano in 2019. He received his B.Sc. and M.Sc. in Computer Engineering from Politecnico di Milano in 2012 and 2015 respectively. He also receives in 2015 M.Sc. degree in Computer Science from the University of Illinois at Chicago (UIC), and Alta Scuola Politecnica Diploma. His research focuses on reconfigurable architectures, code generation and optimization. He is currently a PostDoc at RIKEN Center for Computational Science.

Eleonora D'Arnese got her Ph.D. Information Technology from Politecnico di Milano in 2023. She received her B.Sc. and M.Sc. in Biomedical Engineering from Politecnico di Milano in 2016 and 2018 respectively. She also received in 2018 M.Sc. degree in Bioengineering from the University of Illinois at Chicago, Chicago, IL, USA. Her research focuses on pipeline generation for medical image processing and machine learning. She is currently a PostDoc at Politecnico di Milano.



Davide Conficconi got his Ph.D. Information Technology from Politecnico di Milano in 2022. He received his B.Sc. and M.Sc. in Computer Engineering from Politecnico di Milano in 2015 and 2018 respectively. His research interests revolves around reconfigurable architectures, especially FPGAs, design methodologies, computer architectures, design automation techniques, and abstraction layers. He is currently a PostDoc at Politecnico di Milano.

Marco Domenico Santambrogio received the Laurea (M.Sc. equivalent) degree in computer engineering from the Politecnico di Milano, Milan, Italy, in 2004, the M.Sc. degree in computer science from The University of Illinois at Chicago, Chicago, IL, USA, in 2005, and the Ph.D. degree in computer engineering from the Politecnico di Milano, in 2008. He was a Post-Doctoral Fellow with the Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, USA. He has been with the NECST Laboratory, Politecnico di Milano, where he founded the Dynamic Reconfigurability in Embedded System Design project in 2004 and the CHANGE (self-adaptive computing system) project in 2010. He is an Assistant Professor with the Politecnico di Milano. His current research interests include reconfigurable computing, self-aware and autonomic systems, hardware/software co-design, embedded systems, and high-performance processors and systems.