Commun. Comput. Phys.
doi: 10.4208/cicp.OA-2024-0233

Vol. **37**, No. 5, pp. 1358-1382
May 2025

Application of Computational Modelling to Particle Physics

Marco Barbone[1,*], Alexander Howard[1], Mihaly Novak[2],
Wayne Luk[1], Georgi Gaydadjiev[3] and Alex Tapper[1,*]

[1] *Imperial College London, London, United Kingdom.*
[2] *European Laboratory for Particle Physics (CERN), Geneva, Switzerland.*
[3] *Delft University of Technology, Delft, Netherlands.*

Received 27 September 2024; Accepted (in revised version) 11 February 2025

Abstract. This study introduces a methodology for forecasting accelerator performance in Particle Physics algorithms. Accelerating applications can require significant engineering effort, prototyping and measuring the speedup that might finally result in disappointing accelerator performance. The proposed methodology involves performance modelling and forecasting, enabling the prediction of potential speedup, identification of promising acceleration candidates, prior to any significant programming investment. By predicting worst-case scenarios, the methodology assists developers in deciding whether an application can benefit from acceleration, thus optimising effort. A Monte Carlo simulation example demonstrates the effectiveness of the proposed methodology. The result shows that the methodology provides a reasonable estimate for GPUs and, in the context of FPGAs, the predictions are extremely accurate, within 2% of the realised execution time.

AMS subject classifications: 68U01

Key words: High performance computing, Monte Carlo, FPGA acceleration, GPU Acceleration, performance modelling.

1 Introduction

Programming accelerators such as Graphics Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs), etc., presents many different challenges that developers need to overcome to successfully accelerate application. To avoid investing

*Corresponding author. *Email addresses:* `m.barbone19@imperial.ac.uk` (M. Barbone), `a.tapper@imperial.ac.uk` (A. Tapper), `alexander.howard@cern.ch` (A. Howard), `mihaly.novak@cern.ch` (M. Novak), `w.luk@imperial.ac.uk` (W. Luk), `g.n.gaydadjiev@tudelft.nl` (G. Gaydadjiev)

 ©2025 Global-Science Press

needless development time and resources into accelerating applications that may not benefit from acceleration, it is important to model and forecast the performance. This is most evident in the case of FPGAs where compilation time can take up to several days. Hence, it is crucial to accelerate only applications that can achieve a significant speedup whilst keeping the number of compilation iterations to a minimum.

This paper proposes methodology that aims to accelerate only the relevant hotspots while minimising changes to the original code-base. Minimising the number of changes has an additional benefit of reducing the likelihood of introducing bugs, which can be extremely difficult to fix due to the asynchronous execution and extreme threading of code on accelerators compared to the host CPU. Performance modelling provides an estimate of the potential speedup that can be achieved by off-loading portions of an application to an accelerator [3]. In addition, performance modelling helps to identify the most promising acceleration candidates and provides insight into the design parameters that need to be tuned to achieve optimal performance [19]. Performance forecasting involves predicting the performance of an application on an accelerator before any hardware implementation is made. This helps in the design space exploration and enables developers to identify the most promising hardware configurations and application partitioning schemes [15]. Our methodology also predicts the worst-case scenario, enabling developers to make informed decisions before investing significant engineering efforts in accelerating applications.

Notable works include PPT-GPU [2], a scalable and accurate simulation framework for predicting GPU performance; Choi et al. [7], who propose a methodology for distributed GPU performance modelling; Da Silva et al. [8], which extends the roofline model for FPGA optimisation in HLS tools; Goswami et al. [10], which introduces a machine learning-based estimator for FPGA-based CNN accelerator design; GCoM [11], which models GPU core-side stalls and predicts performance; and Voss et al. [19], who develop a methodology for reconfigurable hardware design and performance estimation.

In the context of particle physics, the two largest general-purpose experiments at the LHC, namely ATLAS and CMS, are responsible for generating $\sim$10 billion events per year with Monte Carlo (MC) detector simulation and event reconstruction. The execution of extensive event generation campaigns incurs a substantial computational cost [17]. ATLAS forecasts that by the year 2028, approximately 75% of its computational resources will be allocated for simulating various aspects of particle generation in collisions. This includes event generation, modelling interactions with detectors, converting signals into digital data, and ultimately reconstructing this data for analysis. Similar projections have been made by other experiments operating at the LHC. This significant increase in computational demand motivates the adoption of GPUs and FPGAs for MC simulations [6].

GPUs show promising results, achieving a speedup of over 7x for Leading Order calculations compared to CPUs using the MadFlow framework [5]. FPGAs also show promising results. Voss et al. in [18] implemented a simplified electromagnetic shower (EM) simulation on an FPGA that compared to CPU achieves a speedup of over 4x. Additionally, Barbone et al. in [4] demonstrated a massive speedup of 270x for Coulomb

scattering using FPGAs compared to CPU, which equates to a cost-equivalent speedup of over 10x based on the comparative market prices at the time of writing. These examples show the potential of parallel and heterogeneous computing architectures in accelerating MC simulations in various domains, including medical simulations [14,21].

2 Methodology

For our candidate accelerator applications we followed the approach proposed by Voss et al. [19] for FPGA and extended it to include both GPUs and FPGAs.

The methodology is applied before writing code and aims to:

- determine whether the application is suitable for acceleration;
- the achievable speedup;
- which parts of the application needs to be accelerated;
- estimate the engineering effort.

The workflow is iterative in nature and it requires incremental refinements until the requirements are met or further improvements are either impossible or not worth the effort. It should be noted that the approach provided in this section is intended for manual application by the developer and is not automated. However, certain tools can be employed to decrease development time, and with the availability of more advanced tools, automation of certain aspects may become possible.

Algorithm 1 Methodology for acceleration

```
 1: perform initial analysis of the original application
 2: determine the target accelerator
 3: create initial partitioning in accelerator and software
 4: create first initial performance model and architecture
 5: create first initial software model                          ▷ Applies only to FPGA
 6: while bottleneck exists do                                  ▷ Requirements are not met
 7:     identify bottlenecks with new architecture or accelerator/CPU partitioning
 8:     resolve bottlenecks with new architecture or accelerator/CPU partitioning
 9:     verify new architecture in updated software model       ▷ Applies only to FPGA
10:     perform further analysis as necessary
11:     refine performance model based on updated analysis and architecture
12: end while
13: start implementation
```

Algorithm 1 outlines the complete scenario, which begins with the analysis of the original application to identify its compute-intensive, data-intensive parts and workload

characteristics (line 1). The results of this analysis aid in determining the most suitable accelerator for the workload (line 2), and an initial partitioning between the CPU and accelerator is established (line 3). Subsequently, the accelerator architecture is outlined, including the portion of the application executed by the accelerator, data allocation, and transfer (line 4). A preliminary, platform-specific performance model is then constructed. The details for each platform will be presented later in Sections 2.4 and 2.5.

The fifth step applies solely to FPGA acceleration and involves the development of a software model that accurately simulates the hardware architecture (line 5). This serves two purposes: first, it clarifies the functionality of each component of the architecture, and secondly, it simulates the precise behaviour of the hardware, which is beneficial for debugging the FPGA implementation.

Subsequently, the iterative process starts. While a bottleneck exists, meaning that the performance requirements are not met, (line 6) the developer has to first identify the bottleneck (line 7). If the bottleneck is computational, it is possible to use better algorithms, better data structures, or even parallelise over multiple devices. In case the bottleneck is I/O bound then it is possible to use compression, caching or accelerating data-generating portions of the workload (line 8). Once the application is optimised it is necessary to update the performance model to account for the changes (line 10-11). If the requirements are met then it is possible to start the implementation (line 13), otherwise it is necessary to proceed to identify and resolve new bottlenecks (line 7-11).

2.1 Application analysis

The initial step involves measuring various metrics to determine the most appropriate accelerator for the application: **cache miss rate**, **branch miss rate** and whether the application is **embarrassingly parallel**.

Based on these three factors, it is possible to determine which platform is the most suitable candidate. Fig. 1 summarises the decision process. The starting point is the level of parallelism in the algorithm. If the algorithm is not embarrassingly parallel, GPUs are unlikely to offer a significant performance advantage. Then branch mis-prediction rate and cache miss rate needs to be evaluated. The former refers to the frequency with which the CPU branch predictor incorrectly predicts the outcome of a branch, while the latter refers to the frequency with which the data is not found in the cache and must be retrieved from main memory. In the case that workload exhibits both high-cache miss rates and high branch misprediction rates, FPGAs are almost certainly the accelerator that will perform best. Otherwise, if both these rates are low and the application is embarrassingly parallel GPU will offer the best performance. There is one last case to consider, when the application is embarrassing parallel but it has only high branch misprediction rates or only high cache miss rates. In this case, there is no clear winner between FPGAs and GPUs and further analysis is needed – the following sections will explore this further.

Lastly, FPGAs are the only viable choice in cases of strict ultra-low latency requirements or when jitter is not acceptable. Ultra-low latency requirements are usually of

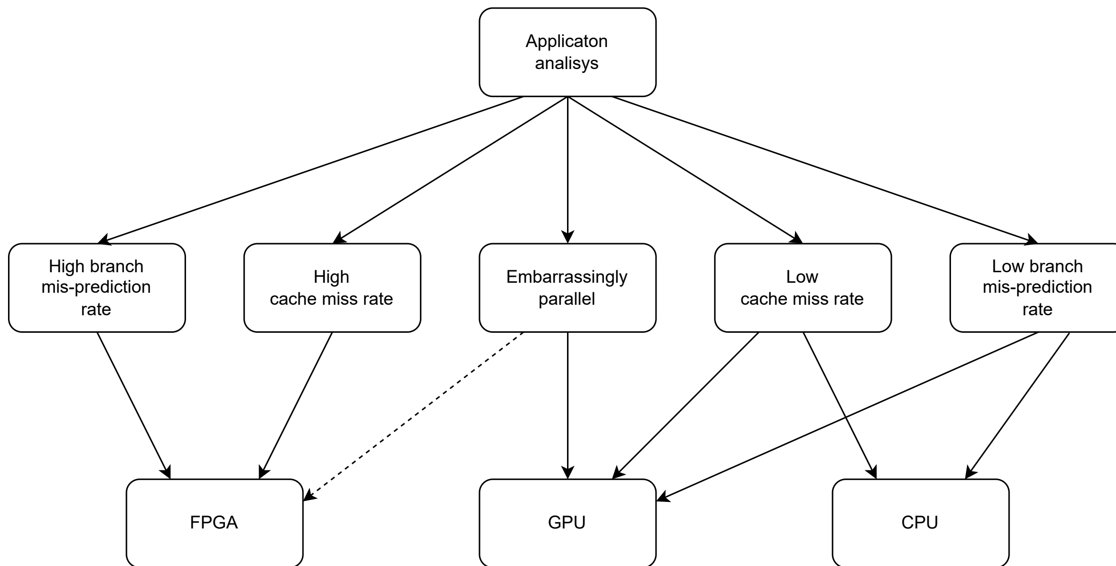

Figure 1: Application analysis deciding process. Depending on the characteristics of the application it is possible to determine which accelerator (if any) is suitable for acceleration.

the order of nanoseconds to microseconds, and CPUs cannot meet these performance requirements as the kernel scheduling overhead is milliseconds. Multi-threading on CPUs only increases the overhead, as creating a thread requires tens of microseconds. Moreover, the system scheduler cannot be controlled, and even threads with the highest priority can be de-scheduled, causing jitter as sleeping threads need to be scheduled before performing the computation.

2.2 Software model (FPGA)

FPGA programming presents multiple challenges in terms of design, implementation and execution. In particular, the long compilation time, which can span multiple days, makes design iterations in hardware extremely resource expensive. Hence, reducing the number of compilation steps is beneficial for successfully accelerating any application. The software model is a simplified implementation of the parts of an application that are to be ported onto the FPGA. It should mirror the intended FPGA implementation and use the same Application Programming Interfaces (APIs). It serves three purposes:

- insight into the selected code;
- testbench for verifying numerical and algorithmic changes;
- debugging reference for FPGA implementation.

To accomplish the initial purpose of the software model, the algorithm can be implemented as basic, unoptimised code, which aids in analysing the application and refining

the performance model and architecture. Runtime profiling is not typically done in the software, but rather metrics like memory access patterns and loop iterations executed are examined. Additionally, the software model is used to assess various algorithmic and numerical options, which inform the architecture and performance prediction. Consequently, the software model is usually developed in conjunction with both.

It is important to reintegrate the software model into the original application to ensure that the planned between the original application and the accelerated modules is adequate to recreate the original functionality. It should be noted that due to the non-associativity of arithmetic using floating-point numbers, minor differences in the result are expected at this stage. Using the same APIs for the software model as for the FPGA implementation offers an easy-to-use debugging tool for the FPGA implementation. It is essential to verify against both the original application and the software model because the software model undergoes the same numerical and algorithmic changes as the FPGA implementation. This makes it easier to determine if an unexpected result for a given input dataset is due to an error in the FPGA design, side-effects, or fundamental issues with the selected algorithm or its numerical properties. Resolving numerical and algorithmic problems in the software model is significantly easier than in an FPGA implementation. Additionally, this provides an FPGA mock-up implementation for earlier integration into the host application.

Although creating the software model and continuously improving it during the design process incurs additional design effort, it avoids the need to make these design iterations in hardware, reducing time and cost whilst allowing software engineers to optimise the application without specific hardware expertise.

2.3 Loopflow graphs

Loopflow graphs are a visual representation of the most important results of the application analysis. They are an extension of the control dataflow graph [9] and are useful in understanding the interactions between loops in a program. The loops are represented by rectangles, with the number of nested loops annotated by a factor, and the operation count can be annotated inside each rectangle. Directed arrows show the dataflow between loops, with arrow widths indicating transferred data amounts. An example can be found in Fig. 2. Loopflow graphs provide an easy way to visualise both computational and data requirements, which can help in identifying the portions of the code that should be moved to the accelerator. In this study, the loopflow graphs were extended from their original proposal by Voss et al. [19] to include the GPU context. The counting of operations differs between the two platforms as explained below, while iterations and the modelling of data transfers remain the same.

FPGA operations count. In the case of FPGA, the number of operations corresponds to the number of arithmetic operations $(+,-,*,/)$ and elementary functions (sin,cos,log,exp) present in the loop. These operations can be manually counted by

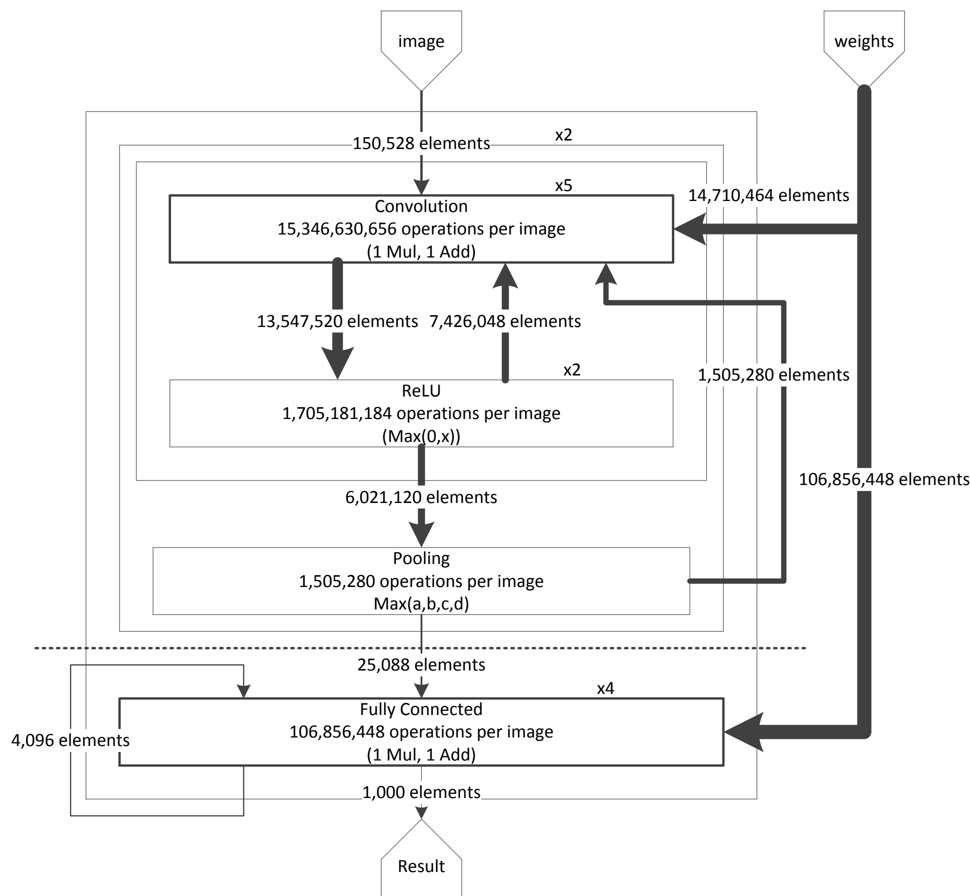

Figure 2: Loopflow graph from [19]. It shows the number of operations and the data transfers of each loop. It allows to identify which loops benefits from acceleration.

analysing each line of code of the software model which is a realistic representation of the hardware design. While some tools may exist to aid in the process, a manual count is typically necessary. In the case of while loops, the average number of iterations can be determined using statistics from the code executed on one or more representative inputs.

GPU operations count. When it comes to estimating the number of operations in the case of GPUs, simply counting the number of operations like in the FPGA case is not sufficient. On GPUs, the application is a sequence of instructions, and all of the instructions need to be executed. Therefore, to estimate the number of operations, one must compute the number of assembly language (ASM) instructions. While some operations require fewer clock cycles to execute than others, assigning a fixed constant cost to all operations does not significantly compromise the accuracy of the model.

To obtain the number of ASM instructions, the best approach would be to implement

the bottleneck on the GPU. However, this is not always practical since the aim of the methodology is to minimise development time. As an alternative, x86 CPU instructions can be used as a substitute. While the number obtained this way may be less accurate than that obtained from the GPU Instruction Set Architecture, a range analysis from best to worst case can still be performed to obtain reasonable estimates. The number of ASM instructions can be obtained by compiling the CPU code without vectorization and unrolling, and with forced *inlining*. If there are any function call instructions, the developer must manually insert the number of instructions for those that are not *inlined*. It should be noted that by forcing *inlining*, there should be very few such cases. Once the ASM is obtained, the developers analyses the bottleneck counting the number of instructions and the number of iterations of each loop reporting them in the graph. When the number of iterations is dynamic (e.g., the while-loops) profiling statistics can be used to model a reasonable average case.

2.4 Performance model

By using the code analysis results, the loopflow graph and the initial software model, a representative performance model can be created to capture the planned implementation on the selected platform. This involves predicting the hardware and bandwidth usage, as well as the execution time for a problem of a known size, and obtaining a preliminary speed-up estimation. The performance model not only enables design space exploration, but also highlights potential problems and bottlenecks of the architecture before the implementation begins. As the architecture aims to alleviate these bottlenecks, an iterative improvement of the performance model and architecture is often required. This allows for early design space exploration in the acceleration process, facilitating cost-benefit analysis and decision making.

2.4.1 Predicting I/O bandwidth usage

Data transfer T_{comm} and memory access T_{mem} or more in general any off chip data transfer ($T_{tranfer}$) can be calculated as the sum of the data ($\mathcal{S}_i$) divided by the relative bandwidth (BW) multiplied by the efficiency factor between maximum (bus) data rate and the actual one achieved. It is worth noting that efficiency ($\eta(\mathcal{S}_i)$) is a function that depends on the size of the data being transferred. A big sequential data transfer has higher efficiency than many small stochastic data transfers

$$\mathcal{T}_{tranfer} = \sum_i \frac{\mathcal{S}_i}{BW \times \eta(\mathcal{S}_i)}. \tag{2.1}$$

The bandwidth available for communication between the host and the accelerator relies on the physical interconnect employed. On most acceleration platforms, PCIe is used as the interconnect. PCIe is constructed using bidirectional links, implying that the full bandwidth can be utilised for both reading and writing concurrently. It is worth noting

that in case the application performs data transfers different magnitudes over PCIe, the smallest one can be ignored without affecting the accuracy of the result.

Memory transfers have a significant impact on performance, and therefore they must be carefully considered in the model. Both GPUs and FPGAs lack pre-fetchers, which means that memory accesses need to be handled with care. There is no automated mechanism that can mitigate the cost of memory accesses.

DRAM bandwidths are unidirectional, e.g., bandwidths in both directions (read and write) are shared. In most cases, lower magnitude data transfers could be omitted. However, the user needs to be careful when doing so. Small, non-sequential data transfers that happen multiple times during the computation, impact performance heavily due to their large latency as shown in Fig. 7, and cannot be omitted.

2.4.2 FPGA performance model

FPGAs consist of predictable building blocks, making it possible to estimate the performance of a given architecture accurately using a few simple equations that will be presented in this section. The total time it takes to process a workload on an FPGA is the sum of the time needed for initialisation and the time for actual execution. If the workload is sufficiently large, the execution time is expected to dominate over initialisation time which can be neglected. However, the exact initialisation time depends on the used platform and framework and can vary between 1 and 100 ms. In a streaming dataflow design, the execution time is determined by the longest latency, which is the maximum of the time needed for computations, data transfer between host and FPGA, and data transfer between FPGA and board memory. If utilisation is not constant, the execution time of these tasks is treated as an additive term to the overall runtime.

Predicting area usage. To determine the computational time requirements for a given device, the achievable degree of parallelism must first be identified. The main limiting factor is the availability of FPGA hardware resources. Typically, FPGA resources are used for three purposes: implementing arithmetic operations, scheduling operations, and supporting other IP modules such as PCIe and memory controllers.

To estimate the area usage for arithmetic operations, one must first determine the amount of hardware resources required to implement a simple operation on the target FPGA device. This can be done using automated tools, vendor documentation, or by creating micro-applications and running them through vendor tools. The area usage can then be estimated by multiplying the area cost of a single operation by the number of operations required on the fabric. Scheduling operations rely on FIFOs, which are typically implemented in on-chip memory or using registers.

For IP modules, the resource requirements can typically be predicted using micro benchmarks. To predict on-chip memory usage, it is important to understand the three main purposes for on-chip memory: (1) in operations and IP-blocks, (2) for buffering or reordering of data, and (3) for balancing the pipeline of computational structures. The

area usage for the first two cases can be estimated using either micro-benchmarks or Eq. (2.2), which calculates the number of memory blocks required (n_{mem}). This calculation is based on the required width (w_{req}), depth (d_{req}), and number of read ports (p_{req}), compared to the available width ($w_{hardware}$), depth ($d_{hardware}$), and read ports ($p_{hardware}$) of the memory type present on the chip, as specified in the documentation.

Predicting the area usage for operation scheduling is difficult since it requires knowledge of the scheduling algorithm used by the toolchain. However, it is possible to identify the largest FIFOs needed to access data from previous cycles and treat them as normal buffers. The amount of memory resources required by smaller FIFOs is highly dependent on the application, so only a rough estimate is possible. Overall, it is recommended to assume slightly higher memory and logic usage to ensure sufficient safety margins, especially for scheduling and additional control logic

$$n_{mem} = \frac{w_{req}}{w_{hardware}} \times \frac{d_{req}}{d_{hardware}} \times \frac{p_{req}}{p_{hardware}}. \tag{2.2}$$

Predicting the compute performance. The next step is to estimate the achievable parallelism and the number of data items that can be processed per clock cycle, without taking into account bandwidth limitations. The time required ($\mathcal{T}_{comp}$) to process a set of data items can be calculated using Eq. (2.3), where the total number of items to be processed n_{total} is divided by the product of the target frequency (f) and the number of items processed per cycle (n_{cycle}). In case the same item needs to be processed multiple times during the computation the total number of items to be processed (n_{total}) needs to be multiplied by the number of iterations ($n_{iterations}$)

$$\mathcal{T}_{comp} = \frac{n_{total}}{n_{cycle} \times f} = \frac{n_{total} \times n_{iterations}}{n_{cycle} \times f}. \tag{2.3}$$

Frequency prediction is not always precise. However, if chip resources are used less than 80% and sufficiently deep pipelining is implemented, the frequency on the same FPGA can be estimated based on previous design experience or by conducting experiments using artificial designs to fill up the chip. For instance, any modern FPGA can typically achieve a frequency of 200 MHz if the chip is filled up to 80% using any toolchain to program it.

2.5 GPU performance model

The performance of a GPU is influenced by various factors such as branches, data transfer, and parallelism, which were discussed in Section 2.1. To develop a complete performance model, these parameters, along with others, need to be taken into account.

The loopflow graphs described in Section 2.3 allow for the estimation of the number of assembly instructions ($\mathcal{N}_{ASM}$). The approach involves using half of the measured number of instructions as a lower bound and double that amount as a high range. The

lower bound provides an estimate of the best-case scenario, while the high bound gives an estimate of the worst-case scenario. Tests have shown that there is a nearly one-to-one correspondence between NVIDIA PTX ASM instructions, but it is still important to consider both the best and worst-case scenarios to gain insights into the likelihood of successfully accelerating the application and the best achievable performance.

Another important parameter to consider in GPU acceleration is efficiency (η). Fig. 6 illustrates the performance penalty of branching on GPUs. When branching occurs, the GPU de-schedules the diverging branches, and as a result, the execution time becomes the sum of the time to execute the "then" and the "else" branch. However, the number of ASM instructions already accounts for this, as the instructions of both sides of the branch are already considered. Nevertheless, the model does not take into account the idle CUDA cores that are not executing any useful work due to their thread being de-scheduled. The "efficiency" parameter accounts for this behaviour, and its initial value is set to 1, but it is halved each time a branch is encountered.

Input size ($\mathcal{S}$) is an important parameter that is required to forecast the total execution time of an application. The input size is used in combination with the number of ASM instructions ($\mathcal{N}_{ASM}$) to estimate the total amount of instructions that the GPU will execute. The method used to estimate the number of ASM instructions, as previously illustrated in Section 2.3, produces the number of instructions needed to process only one element of the input. To estimate the total number of instructions, the number of ASM instructions needs to be multiplied by the input size. In more complex workloads, the input size might be a vector instead of a scalar. In such cases, it is essential to be careful when obtaining the forecasts using data from the loopflow graph. The input size is kept as a separate parameter instead of incorporating it in the loopflow graph to allow for further analysis on the input size. For example, GPU acceleration might not be worth it for small-medium sized inputs. However, large inputs might change the outcome, and having the input size as a separate parameter enables exploring this aspect.

The remaining parameters are hardware dependent and they are the GPU **clock frequency** (f) which is used to estimate the processing time and **cores** ($\mathcal{N}_{CORES}$). $\mathcal{N}_{CORES}$ models the number of CUDA cores from NVIDIA or Stream processors in the AMD case. They estimate the parallelism achievable on the GPU. Each core is assumed to process one element.

Once the above parameters are computed the following equations help estimate the performance:

$$\mathcal{T}_{compute} = \frac{\mathcal{S} \times \eta \times \mathcal{N}_{ASM}}{f \times \mathcal{N}_{\mathcal{CORES}}}. \tag{2.4}$$

2.5.1 Total time

To accurately estimate the total execution time, data transfers must be taken into account, as this is a complex and extensive topic. Section 2.4.1 provides the necessary details for modelling data transfers. Using Eq. (2.1) the total compute time become as shown in Eq. (2.5) or (2.6) depending on the presence of overlap or not.

- Assuming no overlap:

$$\mathcal{T}_{total} = \mathcal{T}_{compute} + \mathcal{T}_{mem} + \mathcal{T}_{comm}; \tag{2.5}$$

- In case of overlap:

$$\mathcal{T}_{total} = \max\{\mathcal{T}_{compute}, \mathcal{T}_{mem}, \mathcal{T}_{comm}\}. \tag{2.6}$$

However, in case data transfers can be overlapped with the computation, that is to perform the transfers while executing some of the computation then the total time is given my the maximum between the three components, as shown by Eq. (2.6). In reality, the total time is the result a combination of data transfers that can be overlapped and some that cannot hence, in practical terms a Eqs. (2.5) and (2.6) should be used together.

2.6 Methodology derivation

This section describes the rationale behind the modelling methodology, and highlights the differences and extensions compared to state-of-the-art approaches from the literature.

2.6.1 Application analysis

Fig. 3 shows the evolution of NVIDIA CUDA cores over the past years, revealing a remarkable increase by two orders of magnitude within a decade.

Furthermore, recent years have seen an acceleration in this trend, indicating that this growth is likely to continue in the future. To achieve high performance on present-day GPUs, it is imperative that the application is embarrassingly parallel and can exploit

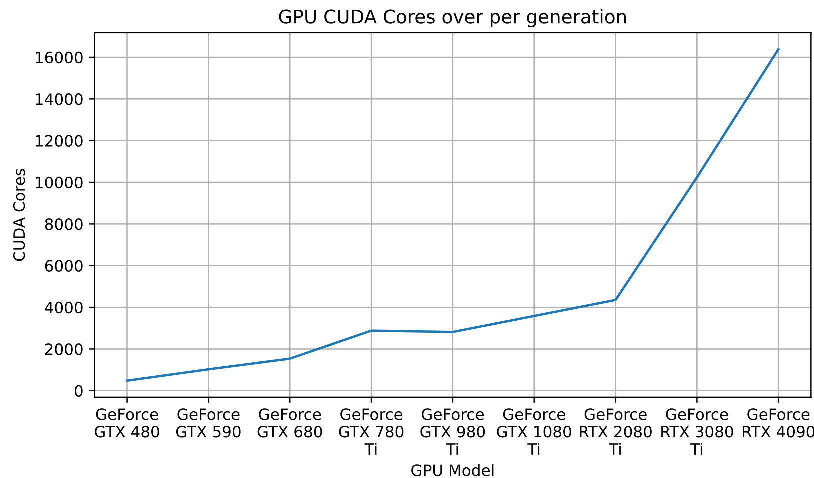

Figure 3: CUDA cores per GPU generation.

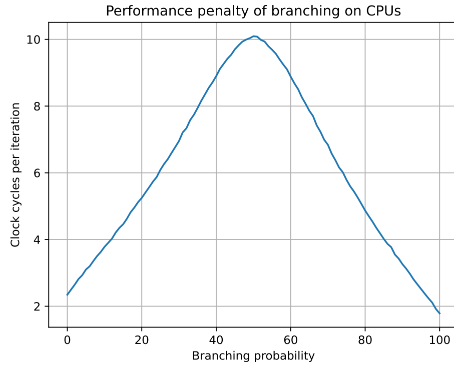

Figure 4: Clock cycles required to execute: $if(\mathcal{R}<P)\ sum \mathrel{+}= \mathcal{R}$; where $\mathcal{R}$ is a random number not known at compile time.

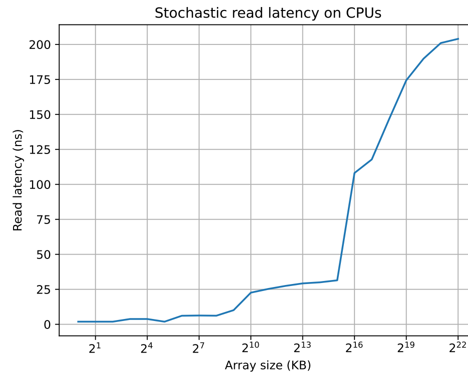

Figure 5: Read latency of the various memories in the system.

the increased number of CUDA cores. A few hundred parallel threads were sufficient for achieving significant speedup a decade ago, but to fully utilise top-of-the-line GPUs today, the level of parallelism required is in the order of tens of thousands. To ensure future-proofing, it is advisable to aim for at least one order of magnitude higher parallelism than the current number of CUDA cores available on state-of-the-art GPUs.

The following two metrics are branch miss rate and cache miss rate. It is important to note that rate refers to the amount of branches or caches missed relative to the total computation. This aims to filter branches that have almost zero probability to be executed, as error checks from the analysis or computation that have relative few expensive branches and memory accesses which do not impact the performance. Even on GPUs, branches that are either always or never executed do not impact performance.

If the workload exhibits a high branch miss rate or high cache miss rate, the performance of CPUs may be poor. Fig. 4 shows the number of clock cycles required to execute an addition at varying branching probabilities. When the probabilities are close to zero or 100%, the CPU requires approximately two clock cycles to execute $if(\mathcal{R}<P)\ sum \mathrel{+}= \mathcal{R}$, where $\mathcal{R}$ is a random number not known at compile-time. However, in the worst case, it requires approximately ten clock cycles, resulting in over five times slower performance. Memory reads are even more expensive. Fig. 5 illustrates the memory latency for stochastic reads. It reveals that compared to L1 cache, L2 cache, L3 cache, and DRAM are over 5, 15, and 108 times slower, respectively. Consequently, applications that execute a large number of stochastic memory reads can be over two orders of magnitude slower on CPUs.

In these scenarios, GPUs are unlikely to provide significant performance improvements, as branches are even more expensive on GPUs, caches are smaller, and memory read latency is higher. Fig. 6 illustrates the performance penalty of branching on GPUs. When threads diverge, GPUs de-schedule them, so even if there is a low branching prob-

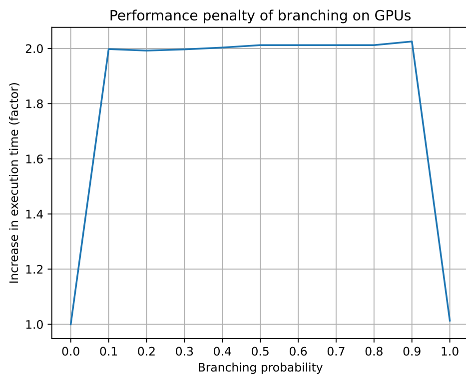

Figure 6: Increase in execution time due to branching on GPU.

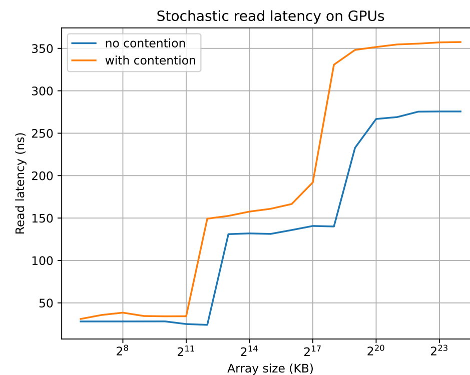

Figure 7: Read latency of the various GPU memories.

ability, the few diverging threads will be executed later, resulting in increased execution time. This implies that the cost of executing the code is the sum of the time needed to execute both branches of the *if* statement.

Fig. 7 shows that the memory access penalty is even higher than on CPUs. In cases of random access to the cache or on-board memory, the code slows down by 56 to 511 times, or even up to 660 times in cases of memory contention. For reference, shared memory is slightly faster, and accessing data from there slows down the computation by 32 times.

Summary

- **Parallelism and Memory Bandwidth:** GPUs are highly effective for embarrassingly parallel workloads and memory-bound applications reliant on sequential reads. With GDDR6 memory bandwidth exceeding 800 GB/s (Fig. 9), GPUs outperform CPUs with DDR5 memory, whose bandwidth is an order of magnitude lower, even in multi-channel configurations.
- **Branching and Cache Misses:** High branch mis-prediction rates combined with frequent cache misses degrade GPU performance. However, GPUs can still deliver significant speedups by exploiting their immense parallelism. A complete performance model (see Sections 2.4 and 2.5) is necessary in such cases to evaluate execution time and determine the best platform.

FPGAs offer unique strengths in scenarios where:

- **Branching:** FPGAs handle branches efficiently as both the "then" and "else" paths are executed in parallel, discarding the unneeded result without a performance penalty.
- **Cache Misses:** Custom pre-fetching and caching strategies can be implemented on FPGAs to overlap computation and communication, ensuring minimal idle time.

Despite these advantages, FPGAs require approximately an order of magnitude more engineering effort compared to CPUs. Therefore:

- FPGAs are considered viable only if they can deliver at least an order of magnitude higher performance than CPUs.
- If GPUs and FPGAs are predicted to achieve similar performance, GPUs are preferred due to their lower development complexity.

GPUs excel in embarrassingly parallel workloads and memory-bound applications, while FPGAs are more effective for tasks involving frequent branching or cache misses. The decision between these platforms should be guided by detailed performance modelling to balance computational efficiency with the engineering effort required, ensuring the optimal platform is chosen for the specific workload.

2.6.2 I/O bandwidth efficiency

Efficiency is a function that captures several factors, including the PCIe encoding scheme, which is the strategy used to convert data into packets for transfer over the PCIe bus. For example, the 8b/10b encoding scheme used in PCIe 1.0-2.1 introduces a 20% overhead, whereas PCIe 3.0-4.0 uses a 128b/130b encoding scheme resulting in a much lower 1.5% overhead. The different DMA implementations can also reduce efficiency, and DMA characteristics and benchmarks are well-documented in the FPGA case. In the case of GPUs, one can test PCIe bandwidth using CUDA samples or by implementing any custom code that transfers data to the GPU and measures the time it takes.

Fig. 8 shows the measured bandwidth for an NVIDIA 3090 Ti on an AMD system. Bandwidth is affected by the transfer size, with smaller transfers being more expensive due to the possibility of not fully filling PCIe packets. The measured bandwidth is 18 GB/s, lower than the theoretical maximum of 31.5 GB/s, which can be attributed to several factors, such as DMA not being designed to handle high bandwidth, CPU bottleneck or driver issues. It is important to consider these limitations in the model to allow applications to benefit from GPU acceleration even if the available bandwidth is far lower than the theoretical maximum.

2.6.3 Memory transfers efficiency

Voss et al. [19] measured the efficiency of DDR4 memory in FPGAs and found that for a single word, the efficiency is below 20%. However, it increases to 80% for ten accessed words and reaches 90% for 100 accessed words. This indicates that random memory accesses could result in poor FPGA performance. Hence, it is essential to utilise on-chip memory effectively and read and buffer more data to maximise FPGA performance.

GPU performance is also significantly affected by memory transfers, and it is important to consider these factors in the model. Fig. 9 shows the memory bandwidth over data size for GPUs, and it demonstrates that GDDR efficiency is initially worse than that of FPGAs until at least 128KB are read in one go. After this point, performance increases exponentially with the data increase and reaches a peak of over 800 GB/s.

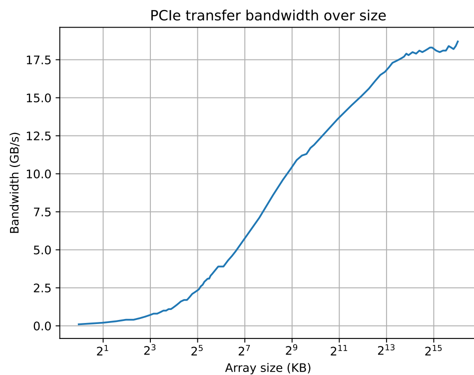

Figure 8: PCIe 4.0x16 bandwidth on a NVIDIA 3090 Ti.

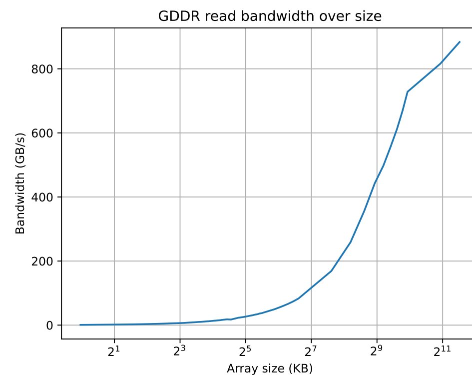

Figure 9: GDDR Bandwidth over data size.

3 Evaluation

To evaluate the proposed methodology, a MC simulation was considered: single Coulomb scattering. The decision to evaluate the methodology using a MC simulation stems from the computational complexity of MC simulations and, as previously stated in the introduction, ATLAS forecasts that the majority of its computational resources will be allocated to MC simulations by the year 2027 [6]. The simulation was modelled using the proposed decision process, which was then used to determine whether acceleration using either FPGA or GPU would be suitable. This evaluation serves as a validation of the decision process and the ability of the model to accurately determine the most suitable platform for acceleration.

3.1 Single Coulomb scattering

Simulation of electron trajectories, inside an infinite material and having only Coulomb scattering as possible interaction, was considered first as a simple but physically meaningful MC test case. The screened Rutherford Differential Cross-Section (DCS) [20] was selected for describing elastic scattering of electrons in the potential of the target atom. The main advantage of this DCS is its simplicity: the DCS and all derived quantities, required during the simulation algorithm, can be expressed in closed analytical forms, including even generation of the corresponding polar angle of scattering.

3.1.1 Application analysis

The MC simulation is embarrassingly parallel as each particle can be simulated independently. However, it should be noted that this requires a parallel random number generator (RNG) as well as a coordination mechanism to avoid race condition when updating the radiation deposition in the geometry.

According to the Linux *perf* command, the MC simulation analysed in this study suffers from a high branch misprediction rate relative to the total number of instructions, thus indicating that this MC simulation falls in the category of workloads that is suitable for FPGA acceleration. However, this information alone is not enough to conclude that FPGA would be faster as further efficiency analysis are needed. Moreover, even if the workload is suitable, there might not be enough resources on an FPGA to achieve a meaningful increment in performance.

The cache misprediction rate is low, the MC simulation supports only electrons who do not travel far in space hence, by using a proper geometry partitioning is possible to always access memory locations in cache. This applies to GPUs as well as geometry partition size is arbitrary.

Given the characteristics of this workload, FPGA seems to be a better candidate compared to GPUs, however as Section 2.1 explained, this is one of the border case applications where a complete performance model is necessary to determine the best accelerator to use.

The complete model that Sections 3.1.2 and 3.1.3 will explain show that there is no clear winner between FPGAs and GPUs as both should achieve similar performance. In this cases, the model suggests is to choose GPUs as they require less engineering effort.

In this case, the loopflow graph is omitted as the simulation is composed only by one main simulation loop with no nested loops. The number of operation required is reported directly in the model.

3.1.2 FPGA performance model

By applying Algorithm 1, a first architecture and software model was drafted. The initial architecture proposes that FPGA simulates one iteration of the main simulation loop per clock cycle. For this architecture to be at least one order of magnitude faster than CPUs, the resulting FPGA design must either reach a clock frequency of 250 MHz or achieve parallelism greater than one. HLS toolchains might not guarantee reaching 250 MHz hence the second option needs to be explored. According to the model's predictions, when targeting a Xilinx VU9P FPGA, simulating one iteration of one particle simulation main loop per clock cycle requires 127 DSPs and 10 Mb of on-chip memory. These values are obtained by manually analysing the code line by line and counting the required DSPs and memory requirement. For example in the following code:

```
1 double theLongiDistr[longiDistNumBin] = {0.0};
2 double theTransDistr[transDistNumBin] = {0.0};
3
4 ...
5
6 aTrack.fPosition[0] += aTrack.fDirection[0] * stepLength;
7 aTrack.fPosition[1] += aTrack.fDirection[1] * stepLength;
```

The memory requirement for line 1-2 is:

```
1 sizeof(double)*longiDistNumBin + sizeof(double)*transDistNumBin
```

The DSP requirement instead is according to Xilinx documentation: two single-precision floating-point additions and two single-precision floating-point multiplications. Hence, lines 6-7 require 2 DSPs for addition + 3 DSPs for multiplication each = 10 DSPs. The counting operation illustrated above needs to be repeated for the entire software model. In this case, modelling LUTs and FFs has been omitted as the DSP utilisation is one order of magnitude higher than the rest as most of the simulation is composed by floating-point arithmetic. From Xilinx documentation it is possible to notice that floating point arithmetic mostly requires DSPs instead of LUTs and FFs. While other operations (multiplexers, de-multiplexers) require LUTs and FFs, these operations are one order of magnitude less common in this MC simulation and can be neglected.

The VU9P FPGA is equipped with 6,840 DSPs and 345 Mb of on-chip memory, thus the simulation of one particle requires 1.86% and 3.35% of the total available DSPs and on-chip memory respectively. Trivially, since $3.35 > 1.86$, the limiting factor is the on-chip memory. Moreover, the compiler requires part of the on-chip memory to schedule the design, in our worst-case analysis we consider only 50% of the total memory available to store data.

Eq. (2.3) shows that $\mathcal{T}_{compute}$ is 1,150 ms where:

$$\begin{aligned} n_{total} &= 100 \times 10^6, \\ n_{iterations} &= 34.5, \\ f &= 200 \times 10^6 s^{-1}, \\ n_{cycle} &= 15. \end{aligned}$$

The data transfers account for less than 1 ms on a PICe 3.0x16 bus hence are neglected from the computation. As Section 3.1.1 showed memory accesses can also be omitted as the vast majority of the memory accesses happens in cache due to the geometry partitioning scheme adopted.

For the sake of simplicity aggregation of the raw data to the final histograms was not implemented on the FPGA. According to profiling results the CPU requires 198 ms to perform the final aggregation hence the $\mathcal{T}_{compute}$ is foretasted 1,348 ms.

3.1.3 GPU performance model

This section uses NVIDIA terminology as the GPU available for this study is an NVIDIA V100. However, the concepts illustrated here apply to AMD GPUs as well. Similarly to the FPGA performance model, the first architecture was drafted by applying Algorithm 1. The initial architecture exploits the embarrassingly parallel characteristics of the Coulomb scattering simulation by parallelising over particles – each CUDA core simulates one particle. This is possible as modern GPUs support atomics for updating the shared geometry. The efficiency is 0.0625 as each particle encounters an average of 16 diverging branches during the simulation. The number of ASM instructions counted using GCC and the previously mentioned methodology is rounded to approximately 5000. Rounding the value by a small margin does not impact the accuracy of the model. The

Table 1: Design space exploration.

Particles	Clock (GHz)	CUDA cores	Efficiency (0,1]	ASM instructions	Time (ms)
100×10^6	1.455	5,120	0.0625	2,500	**537**
100×10^6	1.455	5,120	0.0625	5,000	**1,074**
100×10^6	1.455	5,120	0.0625	10,000	**2,148**

parallelism depends on the physical hardware capabilities as the simulation is embarrassingly parallel and exploits all available resources. In this case, an NVIDIA V100 is used for this study, which offers 5,120 CUDA cores running at 1.455 GHz. Finally, the input size is 100×10^6 particles.

The geometry is assumed to be stored in a format that does not cause costly cache misses. This is possible thanks to electron position locality as they do not travel far at each iteration step. Hence, memory access overhead is negligible in this case. Note that any architecture that is designed not to be memory-bound can neglect data transfers from the model. PCIe data transfers are also negligible as the geometry can be generated directly on the GPU, and the results comprise only longitudinal and transverse distributions instead of the full geometry.

Table 1 shows the design space exploration and the parameters used. These values are substituted in Eq. (2.5) while zeroing data transfers as they are negligible in this case. The computation happens mostly in L1 cache, and results are written to GDDR asynchronously. Data transfers over the PCIe account for less than one millisecond. The expected computation time ranges between 0.5 and 2 seconds.

3.1.4 FPGA implementation

The model's predictions show that to meet the speedup requirement, the resulting architecture should be parallel. Fig. 10 shows the resulting architecture. In total, there are 15 instances, as predicted by the model: each instance is composed of a particle generator, a RNG capable of generating 96 bits per clock cycle, and a particle simulator. The particle generator models a pencil beam; hence, all particles generated share the same initial energy and direction. The RNG is implemented using three separate instances of the Mersenne Twister 32-bits (MT32) RNG [16]. This generator is not suited for use on FPGA, as it requires large memory resources compared to FPGA-optimised generators. However, it is the most popular RNG and is most likely to be available in any programming language; hence, for the sake of generality, it was used in this study.

The particle simulator is the core of the simulation as it implements the MC logic. It is composed of a while loop that iterates until the particle runs out of energy. The moment it runs out of energy, a sample point is taken and sent back to the CPU, which aggregates all of them at the end of the computation. It is worth mentioning that hardware implementations of while-loops cause backward edges in the dataflow graph. As shown in Fig. 11, this backward edge feeds the result back and is processed multiple times until

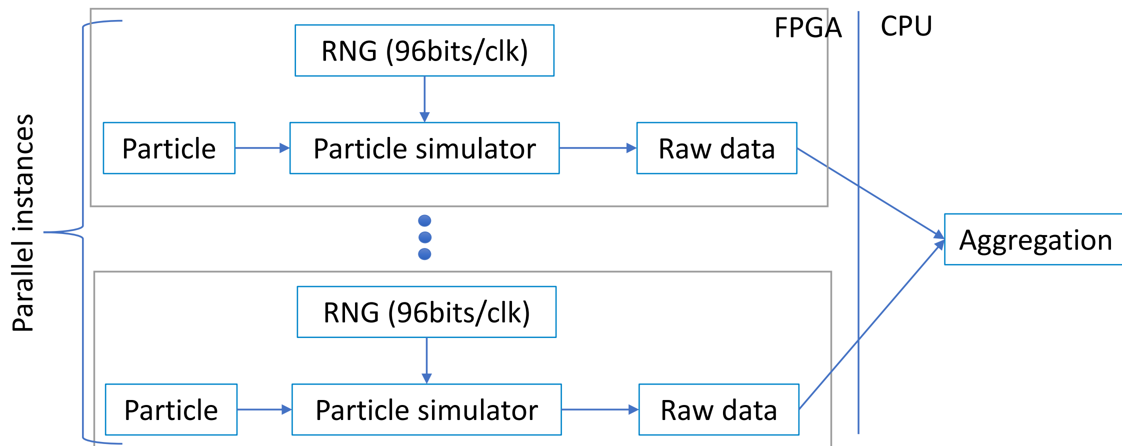

Figure 10: FPGA architecture schematic displaying the parallel FPGA instances and the CPU based aggregation. The grey rectangles represent instances of the simulation, composed by beam, RNG, simulator that generates raw data. These instances are replicated until all the FPGA resources are utilised. The raw data they generate is then sent back to the CPU for the final aggregation.

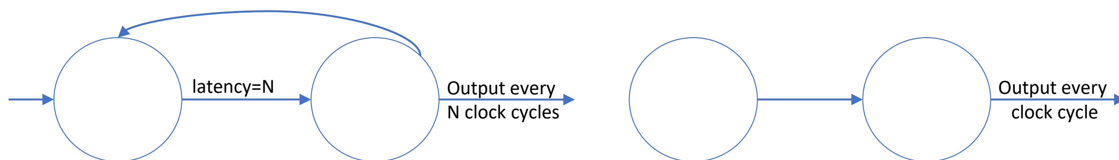

Figure 11: Illustration of the while loop optimisation implemented.

the guard is false; while-loops can produce a result every n cycles where n is the depth of the pipeline. However, one of the characteristics of MC is independence: multiple particles can be fed to the while-loop since the order of the result does not matter. If the guard is true, a particle is sent back; if it is false, a new particle is fed in and a result is produced. Exploiting this characteristic can increase MC performance by a huge margin. For example, if n corresponds to 68, without independence, the performance would be over 60 times slower.

This optimised particle simulator computes the final position of the particle, the moment it runs out of energy, and sends the result back to the CPU. For the sake of simplicity, aggregation of the results is performed by the CPU. According to the model, moving the aggregation onto the FPGA would improve the performance by 15%.

3.1.5 GPU implementation

The GPU implementation exploits the embarrassingly parallel characteristics of the simulation by processing particles in parallel. As the number of particles is on the order of

10^8, this is likely to exceed the number of CUDA cores/Stream processors for many generations. The MC simulation does not support any secondary generation, so each core integrates a pencil beam and a particle simulator. Replicating the pencil beam allows us to avoid using memory to store stacks of pre-generated particles, which reduces overhead and simplifies load-balancing. Differently from FPGA, GPUs offer native support to atomics hence, to reduce data transfers raw data aggregation to histogram is performed on the GPU itself.

The RNG used is Philox_4x32_10 [13], which is a fast and parallel RNG with reasonable quality. The implementation available in cuRAND, the free public NVIDIA RNG library for GPUs, allows each core to generate numbers independently in parallel. Other generators like Mersenne Twister for Graphics Processors [12] were tested, as they offer higher quality numbers, but they severely limit GPU performance.

Explicit use of shared memory is also investigated. Shared memory could be used to store the transverse and longitudinal distributions, RNG state, and a portion of the geometry. However, tests showed that manually managing shared memory results in lower performance.

Modern GPU micro-architectures have a monolithic on-chip memory, which is partitioned between shared memory and cache. The developer can explicitly use a portion of this memory to allocate frequently allocated data or to pre-fetch data from memory. However, doing so reduces the amount of memory available to the GPU to use as cache. Since the Coulomb scattering algorithm causes electrons to not travel long distances, leaving the cache management to the GPUs results in higher performance as this is done by specialised hardware. Any user-developed software, on the other hand, is executed in software by the CUDA cores, resulting in higher overhead, less parallelism, and lower general performance.

3.1.6 Validation

The final results of this MC simulation consist of a longitudinal and a transverse distribution. To evaluate the accuracy of the results, we executed a Kolmogorov–Smirnov test against reference distributions, which concluded that the distributions are equivalent using $\alpha = 5\times\sigma$. Achieving equivalence with smaller α requires a further increase in the number of histories.

Figs. 12 and 13 show the longitudinal and transverse distributions obtained by simulating 100×10^6 particles in gold using a pencil beam of 128 keV. It can be noted that the lines of the reference implementation, the GPU, and the FPGA-accelerated versions are overlapping perfectly.

3.1.7 FPGA results

To evaluate the performance we computed 100M histories of a 128 KeV electron interacting in water. The tests are executed Xilinx Alveo U200 that incorporates a VU9P FPGA device. The FPGA implementation was limited at 200 MHz, using single-precision

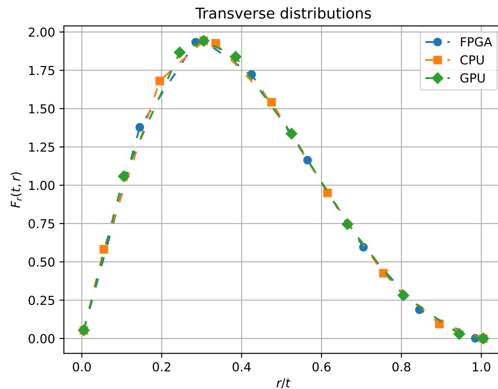

Figure 12: Transverse distribution.

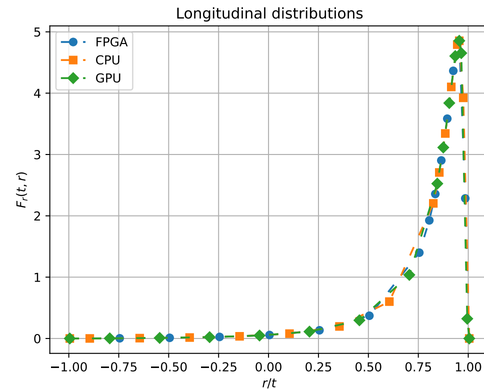

Figure 13: Longitudinal distribution.

floating-point, leaving out additional performance benefits given by the use of fixed-point data representations. The CPU implementation was parallelised using OpenMP while the FPGA implementation was developed in MaxJ and compiled using MaxCompiler 2021.1 and Vivado 2019.2. It is worth mentioning that we did not perform any hardware-specific optimisation or rely on any compiler-specific optimisation; these results are therefore representative of any HLS toolchain. Relying on MaxCompiler specific optimisations increases the achievable clock frequency up to 300 MHz obtained with no changes to the codebase. However, since these results are not representative of the performance obtainable with other HLS toolchains, they are mentioned but not included in our final evaluation. The final execution time is 1321 ms which is within margin of error compared to the 1,348 ms predicted by the model. This confirms the accuracy of the modelling methodology.

3.1.8 GPU results

To evaluate the performance we computed 100M histories of a 128 KeV electron interacting in water. The benchmarks were executed on an NVIDIA Tesla V100 PCIe 32GB equipped with 32GB GDDR5X and 5,120 CUDA cores running at 1,455 MHz. The code was compiled using GCC 9.3.1, NVCC 11.6 and OpenMP for parallelism. The time was obtained by running the tests five times and averaging the results. The GPU implementation achieves a performance of 873 ms which fall in the range predicted from the model and close to the best achievable performance. Explicit use of shared memory is also tested, however it fails to improve performance in any meaningful way hence the results are omitted.

3.1.9 Comparison between platforms

We compared the performance of 2x AMD EPYC 7551 32-Core Processor running at 2.0 GHz, boosting up to 3.0 GHz. The code was compiled using GCC 9.3.1 and OpenMP for

Table 2: Comparison between multiple platforms. Metrics are Time, speedup and cost normalised speedup.

	CPU	FPGA	GPU
Time (ms)	8,494	1,321	853
Speedup	1	6.42	9.95
Speedup/Price	1	1.60	3.52

parallelism. Table 2 shows the comparison between the platforms. In this case, FPGAs are 6.42 times faster and 1.60 times more cost-effective than a 64-core CPU. GPUs, on the other hand, are almost 10 times faster and 3.5 times more cost-effective according to market prices available at the time of writing. If the amount of engineering time is included in the equation, then FPGAs might not be as cost-effective compared to the other platforms. While the proposed model cannot capture these details, users are encouraged to consider them and make informed decisions.

4 Conclusions

This study introduces a methodology for forecasting accelerator performance. It allows developers to quickly analyse their applications prior to any investment in development. This facilitates informed decisions about the tradeoff between speedup and engineering effort early in the development process. Moreover, the methodology guides the entire process from analysis to implementation aiming to minimise development time by highlighting potential bottlenecks and pitfalls early on, reducing the need for major architectural changes to the application under development. The results show that the performance forecasts allow the developer to quantify the benefits of GPU and FPGA acceleration. In particular, running time estimates are within an acceptable range for GPU acceleration and accurate to within 2% for the more predictable FPGAs. These results align with findings from [10, 11, 19] while offering a more general approach that targets both GPUs and FPGAs. When a given application cannot benefit from acceleration, the proposed methodology identifies this early during the analysis phase. In such cases, performance modelling can prevent hundreds, if not thousands, of hours of unnecessary engineering effort. While the approach presented here is intended to be "manual," tools such as OneAPI [1] can be employed to automate metrics like data transfer analysis and operations count, further streamlining the process.

Acknowledgments

This research project was supported by the CRUK Convergence Science Centre at The Institute of Cancer Research, London, and Imperial College London (A26234). We acknowledge funding from the Cancer Research UK programme grant C33589/A19727.

The Institute of Cancer Research and The Royal Marsden NHS Foundation Trust are members of the Elekta MR-Linac Research Consortium. We are also grateful to Nils Voss for providing the LoopFlow graph example.

References

[1] oneAPI: A New Era of Heterogeneous Computing. URL https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html#gs.ow9meo.

[2] Yehia Arafa, Abdel Hameed A. Badawy, Gopinath Chennupati, Nandakishore Santhi, and Stephan Eidenbenz. PPT-GPU: Scalable GPU Performance Modeling. *IEEE Computer Architecture Letters*, 18(1):55–58, 1 2019. ISSN 15566064. doi: 10.1109/LCA.2019.2904497.

[3] Amin Bakhoda, Jason Kim, and Kathryn S McKinley. Analyzing CUDA workloads using a detailed simulator. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Parallelism (HotPar)*, pages 163–174, 2009.

[4] M Barbone, A Howard, A Tapper, D Chen, M Novak, and W Luk. Demonstration of FPGA Acceleration of Monte Carlo Simulation. *Journal of Physics: Conference Series*, 2438(1):1–6, 2 2023. ISSN 1742-6596. doi: 10.1088/1742-6596/2438/1/012023.

[5] Stefano Carrazza, Juan Cruz-Martinez, Marco Rossi, and Marco Zaro. MadFlow: towards the automation of Monte Carlo simulation on GPU for particle physics processes. *EPJ Web of Conferences*, 251:1–03022, 2021. ISSN 2100-014X. doi: 10.1051/EPJCONF/202125103022.

[6] James Catmore. Computing and software for the High Luminosity LHC. *Proceedings of Science*, 390:009, 4 2021. ISSN 18248039. doi: 10.22323/1.390.0009. URL https://pos.sissa.it/.

[7] Jaemin Choi, David F. Richards, Laxmikant V. Kale, and Abhinav Bhatele. End-to-end performance modeling of distributed GPU applications. *Proceedings of the International Conference on Supercomputing*, pages 1–30, 6 2020. doi: 10.1145/3392717.3392737. URL https://dl.acm.org/doi/10.1145/3392717.3392737.

[8] Bruno Da Silva, An Braeken, Erik H. D'Hollander, and Abdellah Touhafi. Performance modeling for FPGAs: Extending the roofline model with high-level synthesis tools. *International Journal of Reconfigurable Computing*, 2013:1–7, 2013. ISSN 16877209. doi: 10.1155/2013/428078.

[9] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits (1st ed.)*. McGraw–Hill Higher Education, 1994.

[10] Pingakshya Goswami, Masoud Shahshahani, and Dinesh Bhatia. Robust Estimation of FPGA Resources and Performance from CNN Models. *Proceedings - 2022 35th International Conference on VLSI Design, VLSID 2022 - held concurrently with 2022 21st International Conference on Embedded Systems, ES 2022*, pages 144–149, 2022. doi: 10.1109/VLSID2022.2022.00038.

[11] Jounghoo Lee, Yeonan Ha, Suhyun Lee, Jinyoung Woo, Jinho Lee, Hanhwi Jang, and Youngsok Kim. GCoM: A Detailed GPU Core Model for Accurate Analytical Modeling of Modern GPUs. *Proceedings - International Symposium on Computer Architecture*, 22:424–436, 6 2022. ISSN 10636897. doi: 10.1145/3470496.3527384. URL https://dl.acm.org/doi/10.1145/3470496.3527384.

[12] Mutsuo Saito and Makoto Matsumoto. Variants of Mersenne Twister Suitable for Graphic Processors. *ACM Transactions on Mathematical Software (TOMS)*, 39(2):1–12, 2 2013. ISSN 00983500. doi: 10.1145/2427023.2427029. URL https://dl.acm.org/doi/10.1145/2427023.2427029.

[13] John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. Parallel random numbers: As easy as 1, 2, 3. *Proceedings of 2011 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16, 2011. doi: 10.1145/2063384.2063405. URL https://dl.acm.org/doi/10.1145/2063384.2063405.

[14] Shobhit Sharma, Anuj J. Kapadia, Wanyi Fu, William P. Segars, Ehsan Samei, and Ehsan Abadi. A rapid GPU-based Monte Carlo simulation tool for individualized dose estimations in CT. *https://doi.org/10.1117/12.2294965*, 10573:982–990, 3 2018. ISSN 16057422. doi: 10.1117/12.2294965.

[15] Mark Silberstein. GPUs: High-performance Accelerators for Parallel Applications. *Ubiquity*, 2014(August):1–13, 8 2014. doi: 10.1145/2618401.

[16] Matsumoto M T and Nishimura. Dynamic creation of pseudorandom number generators. *Monte Carlo and Quasi-Monte Carlo Methods*, 2000:56, 1998.

[17] Andrea Valassi, Efe Yazgan, Josh McFayden, Simone Amoroso, Joshua Bendavid, Andy Buckley, Matteo Cacciari, Taylor Childers, Vitaliano Ciulli, Rikkert Frederix, Stefano Frixione, Francesco Giuli, Alexander Grohsjean, Christian Gütschow, Stefan Höche, Walter Hopkins, Philip Ilten, Dmitri Konstantinov, Frank Krauss, Qiang Li, Leif Lönnblad, Fabio Maltoni, Michelangelo Mangano, Zach Marshall, Olivier Mattelaer, Javier Fernandez Menendez, Stephen Mrenna, Servesh Muralidharan, Tobias Neumann, Simon Plätzer, Stefan Prestel, Stefan Roiser, Marek Schönherr, Holger Schulz, Markus Schulz, Elizabeth Sexton-Kennedy, Frank Siegert, Andrzej Siódmok, Graeme A Stewart, and The H S F Physics Event Generator W G. Challenges in Monte Carlo Event Generator Software for High-Luminosity LHC. *Computing and Software for Big Science*, 5(1):12, 2021. ISSN 2510-2044. doi: 10.1007/s41781-021-00055-1. URL https://doi.org/10.1007/s41781-021-00055-1.

[18] Nils Voss, Peter Ziegenhein, Lukas Vermond, Joost Hoozemans, Oskar Mencer, Uwe Oelfke, Wayne Luk, and Georgi Gaydadjiev. Towards Real Time Radiotherapy Simulation. *Journal of Signal Processing Systems*, 92(9):949–963, 9 2020. ISSN 19398115. doi: 10.1007/S11265-020-01548-9/FIGURES/5. URL https://link.springer.com/article/10.1007/s11265-020-01548-9.

[19] Nils Voss, Bastiaan Kwaadgras, Oskar Mencer, Wayne Luk, and Georgi Gaydadjiev. On Predictable Reconfigurable System Design. *ACM Transactions on Architecture and Code Optimization (TACO)*, 18(2):1–17, 2 2021. ISSN 15443973. doi: 10.1145/3436995.

[20] G. Wentzel. Zwei Bemerkungen über die Zerstreuung korpuskularer Strahlen als Beugungserscheinung. *Zeitschrift für Physik*, 40(8):590–593, 8 1926. ISSN 14346001. doi: 10.1007/BF01390457/METRICS. URL https://link.springer.com/article/10.1007/BF01390457.

[21] Peter Ziegenhein, Sven Pirner, Cornelis Ph Kamerling, and Uwe Oelfke. Fast CPU-based Monte Carlo simulation for radiotherapy dose calculation. *Physics in Medicine and Biology*, 60(15):6097–6111, 8 2015. ISSN 13616560. doi: 10.1088/0031-9155/60/15/6097.