International Journal of Core Engineering & Management
Volume-6, Issue-10, 2020 ISSN No: 2348-9510

BRIDGING C++ AND CUDA FOR HIGH-PERFORMANCE FINANCIAL COMPUTING: A SURVEY OF AUTOMATIC OFFLOADING

Sheshank Kodam

Abstract

High-Performance Computing (HPC) has emerged as a critical component of modern scientific and industrial innovation, permitting the speedy resolution of more sophisticated, data-heavy problems. Within the financial sector, where accuracy and computation speed are critical, HPC techniques have enabled financial institutions to run large-scale simulations, risk analytics, and algorithmic trading. C++ continues to be the most widely-used language in financial software engineering because of its efficiency, object-oriented approach, and hardware controls. Graphics Processing Units (GPUs) using CUDA offer unmatched parallel processing power to speed up numerical workloads. The integration of C++ and CUDA automatically raises fundamental challenges associated with programming paradigms, memory management, and synchronization models. The aim of this paper is to review current implementations of C++ and CUDA integration in high-performance financial computing with an emphasis on automatic offloading mechanisms that allow programs to benefit from GPU acceleration without extensive manual offloading. Summarize compiler-assisted approaches, template-based programming that retains the flexibility of C++ with a performance increase, and framework-based approaches. Monte Carlo simulations, option pricing, and portfolio optimization are among the practical applications of automatic C++–CUDA integration, and the accompanying talks discuss the pros and cons of existing systems. Scalability, memory management, and domain knowledge challenges are addressed in the study, which also discusses autonomous GPU offloading in financial HPC systems.

Keywords: High-Performance Computing, CUDA, C++, automatic offloading, financial computing, hybrid programming, Monte Carlo simulation.

I. INTRODUCTION

High-Performance Computing (HPC) is an attractive and rapidly evolving field within computer science. By leveraging parallelism and distributed resources, HPC enables the solution of complex problems that are beyond the capabilities of conventional systems [1]. It has found applications across diverse domains such as molecular biology, genetic engineering, space exploration, cosmology, financial modeling, artificial intelligence, and cryptography. Broadly, HPC can be classified into three paradigms: Cluster Computing, Grid Computing, and Cloud Computing. Cluster computing refers to a system in which two or more autonomous nodes are interconnected to collaboratively solve computationally intensive problems beyond

the capabilities of traditional computing systems [2]. By leveraging parallelism and distributed resources, HPC accelerates computations across domains such as molecular biology, astrophysics, artificial intelligence, cryptography, and financial modeling. Among these, financial computing stands out as a domain where computational efficiency is not only a performance metric but also a direct determinant of economic success [3][4]. The ability to perform rapid, large-scale computations can influence investment decisions, risk assessments, and market predictions, tasks that require both speed and accuracy. The economic and social structure of contemporary civilization is significantly influenced by financial markets [5]. In recent decades, various statistical and soft computing mechanisms have been proposed to assist investors and analysts in predicting trends and optimizing decisions across different financial market segments [6]. However, the growing complexity and scale of financial datasets demand more powerful computing architectures beyond conventional CPUs [7][8].

C++ has long been recognized as the backbone of high-performance financial software development. Its combination of execution speed, low-level memory control, and strong type safety makes it particularly well-suited for computationally intensive financial applications [9]. In domains such as quantitative finance, risk analytics, and algorithmic trading, even millisecond-level delays can result in substantial financial losses. C++ provides developers with the flexibility to optimize performance at both the hardware and software levels, making it a preferred choice for latency-sensitive systems. Financial institutions and fintech firms rely heavily on C++ to implement pricing models, Monte Carlo simulations, and portfolio optimization algorithms. The language's support for object-oriented and template-based programming allows the development of scalable, reusable components that support complex mathematical frameworks [10][11]. To further accelerate such computational workloads, developers increasingly turn to Graphics Processing Units (GPUs), which excel at parallel data processing. NVIDIA first unveiled the Tesla microarchitecture in 2007 as part of their Compute Unified Device Architecture (CUDA) parallel computing platform and programming methodology. Since then, CUDA has evolved across multiple generations, including Fermi, Kepler, Maxwell, and Pascal architectures [12]. CUDA enables general-purpose GPU programming, allowing developers to exploit thousands of lightweight threads for numerical computation. Unlike OpenGL, which was primarily designed for graphics rendering, CUDA offers a dedicated API and programming environment focused on general-purpose computation on GPUs (GPGPU) [13].

While CUDA provides immense computational power, integrating it seamlessly with existing C++ financial applications remains a significant challenge. Traditional C++ programs are primarily designed for CPU-based sequential execution, whereas CUDA requires developers to explicitly define kernels, memory transfers, and device synchronization. This fundamental difference in programming models complicates the process of porting legacy C++ code to GPU architectures. Moreover, manual GPU programming demands specialized expertise in parallel computing and low-level memory management, which increases development complexity and reduces maintainability. Consequently, despite the clear performance benefits of CUDA, many

financial institutions hesitate to fully adopt GPU acceleration due to the high cost and time involved in rewriting large C++ codebases. Hence, an efficient bridge between C++ and CUDA is essential to exploit GPU performance while retaining the productivity and stability of C++ environments [14]. To overcome these integration challenges, researchers have developed automatic offloading techniques, which aim to identify compute-intensive sections of C++ code and transfer them automatically to GPUs for parallel execution. These methods eliminate the need for manual CUDA programming by relying on compiler analysis, runtime profiling, and code transformation tools [15]. Frameworks such as Open ACC, OpenMP 4.5, SYCL, and Kokkos provide abstraction layers that allow developers to maintain standard C++ syntax while benefiting from GPU acceleration [16]. In the context of financial computing, automatic offloading holds significant promise for accelerating simulations, pricing models, and data analytics without extensive code refactoring. However, the effectiveness of these systems varies in terms of performance gain, portability, and ease of adoption. Therefore, a comprehensive survey is necessary to examine existing techniques, evaluate their applicability to financial workloads, and identify open challenges in bridging C++ and CUDA for high-performance financial computing.

A. Structure of the Paper

The paper structure is as follows: Section II reviews automatic offloading frameworks. Section III covers C++–CUDA integration techniques. Section IV discusses GPU acceleration applications in finance. Section V outlines challenges and future directions. Section VI reviews related literature, and Section VII concludes with future work.

II. OVERVIEW OF AUTOMATIC OFFLOADING FRAMEWORKS

In advanced scientific research, the massively parallel architecture of GPU accelerators is being used to speed up computing workloads. The vast parallelism and energy efficiency of GPU-based clusters have increased the desire of academics and developers to migrate their applications to these.

A. Tacit Computing Outline

The technique known as "tacit computing" finds and arranges the appropriate network, cloud, and device layer resources for users at that time to deliver services tailored to each individual (Figure 1). The goal of Tacit Computing, despite its three-layer structure, is to handle changes in real time by processing as much as possible at the device layer, which is closest to the user site. Through device-layer processing, implicit computing can lower network traffic and stop critical data breaches. In order to find and use the right devices for users, live data discovery and device virtualization technologies are essential components of tacit computing.

International Journal of Core Engineering & Management
Volume-6, Issue-10, 2020 ISSN No: 2348-9510

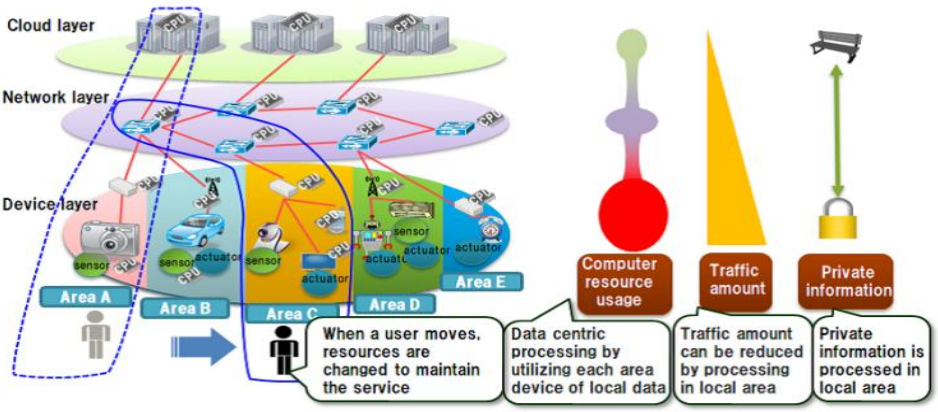

Fig. 1. Outline of Tacit Computing

Technology for live data discovery finds devices that provide consumers the info they need. Consider a scenario where a fixed-point camera is used to record a person and provides instruction or warning notifications. The individual in question is only on camera for a brief moment and is only interested in the portion of the video that they are featured in. Information that is constantly changing is referred to as live data in this context. Live data discovery technology assigns analytical jobs to lower levels so that users may locate the vital real-time information without waiting for it to reach the cloud tier.

Consider a scenario where a user wishes to view videos featuring their buddy who is competing in a marathon competition. Here, the friend's bib number is entered by the user as a search key in Tacit Computing.

B. Improvement by offloading specific processing for Tacit Computing applications

In this approach, Tacit Computing helps consumers find and use the right devices, which is a concept of Open IoT. Nevertheless, Tacit Computing disregards cost and performance when it employs devices dynamically. If the scenario in instead of tracking a marathon runner, the preceding part focused on police monitoring of terrorists or town cameras looking for elderly individuals who have disappeared. As a result, continuous and affordable image analysis services of camera footage are required [17].

International Journal of Core Engineering & Management
Volume-6, Issue-10, 2020 ISSN No: 2348-9510

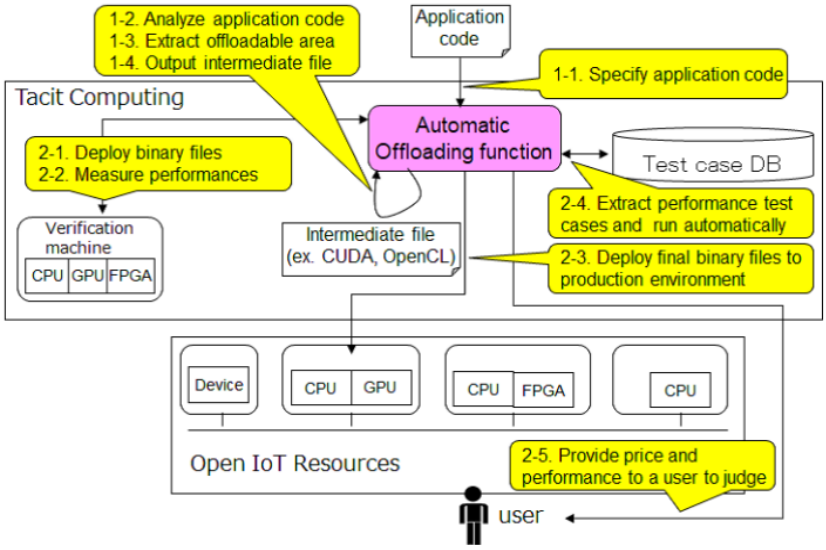

Fig. 2. Steps of automatic offloading

The offload region is extracted from the user application source codes by Tacit Computing, which then generates the intermediate language, deploys the binary file created from the intermediate language on the verification machine, runs it, and confirms the offload effect. The binary file is sent to the production environment and made available as a production service after Tacit Computing restarts the verification and identifies the proper offload zone. Here, Figure 2 is used to illustrate the unloading procedures.

Tacit Computing uses the automated offloading technique in 1-1 to allocate user applications, like as picture processing. To find processing structures like loop statements and library calls like FFT (Fast Fourier Transformation), the automated offloading feature in apps 1-2 examines the source code. In 1-3, the automated offloading feature finds and extracts the intermediate language for off loadable logics, such as loop statements and FFTs, on a GPU or FPGA. In 1-4, intermediate files are produced. The extraction of intermediate language happens more than once, as explained in the next subsection. To determine the ideal offloading location, recursive extraction and execution are utilized.

The intermediate language binary file is sent to a GPU-FPGA verification machine in 2-1 by the automated offloading mechanism. Executing deployed files and measuring offloading performance in 2-2. In order to better unload the region, the automated offloading function goes back to steps 1-3 and uses the results of performance measurements to derive another pattern. In phases two and three, it decides on the final offloading area layout and forwards the user's binary files to production. Performance test cases are extracted from the test case database (DB) and run by Jenkins or other tools in stages 2-4 to demonstrate user performance following binary file deployment. Users can choose whether to pay for the IoT service in steps two

International Journal of Core Engineering & Management
Volume-6, Issue-10, 2020 ISSN No: 2348-9510

through five after receiving pricing, performance, and other information depending on the outcomes of performance tests.

C. Proposal of automatic GPU offloading technology using Genetic Algorithm

It is commonly stated that while GPUs can increase throughput through parallel processing, they cannot ensure latency. Although there are many IoT applications, common examples include machine learning (ML) processing for large-scale sensor data analysis with various loop processes and image processing for camera movies. Consequently, try to increase application throughput by automatically shifting loop activities to the GPU [18].

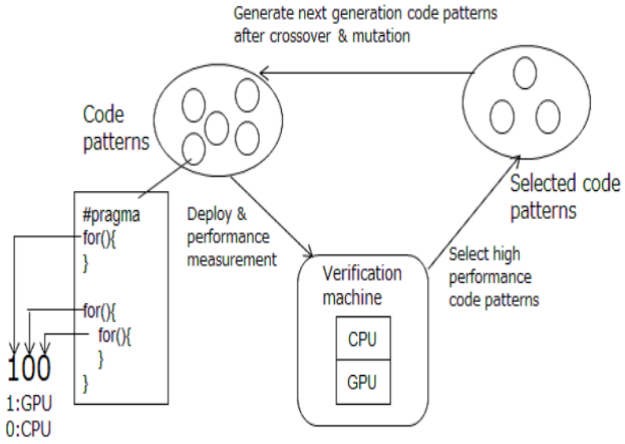

Fig. 3. Image of GPU offloading area search by GA

However, parallel processing in an appropriate location is necessary for optimum performance, as stated in Section 2. Unless there is a large quantity of data or loops, performance may not increase when memory data is moved between the CPU and GPU. Parallel processing may not always provide the best performance for certain loop statement combinations, as it depends on memory process status and data transfer timing. It might not be the fastest to use a three-parallelization arrangement when parallel processing can process loop statements #1, #5, and #10 more quickly than CPU processing in loop statements 10. To find a suitable parallel processing region, optimize loop statement parallelization or sequentialization through round robin trials.

A picture of GA activities is displayed in Figure 3. GA models biological evolution using combinatorial optimization. The GA flow phases include Initialization, Evaluation, Selection, Crossover, Mutation, and Complete Judgment. Use Simple GA in the recommended manner. Roulette selection is used to choose genes, which are then flipped from 1 to 0 or vice versa at a specific rate during mutation. At one point in time, genes move from one individual to another. This is known as simple GA, which is a simpler variant of GA [16].

International Journal of Core Engineering & Management
Volume-6, Issue-10, 2020 ISSN No: 2348-9510

III. HIGH-PERFORMANCE COMPUTING WITH C++ AND CUDA

High-performance computing in finance often requires both computational efficiency and programmer productivity. C++ is widely used in quantitative finance due to its efficiency, object-oriented features, and rich ecosystem for numerical computing. CUDA, NVIDIA's parallel computing platform, enables programmers to utilize GPUs' enormous computing capabilities. Bridging C++ and CUDA enables financial applications—such as Monte Carlo simulations, option pricing, and risk analysis—to execute computationally intensive tasks on GPUs, achieving significant speedups compared to CPU-only implementations [19].

Automatic or semi-automatic integration of C++ and CUDA reduces the manual effort required to manage device memory, write kernel functions, and handle host-device synchronization, which are traditionally complex and error-prone. Surveying existing techniques and frameworks is essential to understand how C++ constructs can be mapped effectively to GPU architectures for financial computing workloads.

A. C++–CUDA Integration Techniques

Integrating C++ with CUDA can be approached at several levels:

- **CUDA Kernels and Device Functions:** Traditional integration involves writing CUDA kernels directly in .cu files and invoking them from C++ host code. The developer is responsible for overseeing configurations for memory allocation, host-to-device data transmission, and kernel launch.
- **Wrapper Libraries:** Libraries like Thrust and cuBLAS provide high-level abstractions for vector and matrix operations, allowing C++ developers to leverage GPU acceleration without writing low-level CUDA code.
- **Template-Based Approaches**: Modern C++ features, such as template metaprogramming, enable generic programming patterns that can be compiled into efficient CUDA kernels. This allows automatic generation of GPU code from C++ abstractions.
- **Compiler Directives and Pragmas:** OpenMP 4.5+, Open ACC, and similar frameworks allow developers to annotate C++ loops and functions for GPU offloading, where the compiler handles the translation and scheduling of code on GPU devices.

By combining these techniques, developers can balance performance, portability, and maintainability in financial HPC applications.

B. Expression Templates and Meta-Programming

Expression templates are a powerful C++ technique that enables lazy evaluation of mathematical expressions. Instead of performing intermediate computations immediately, expression templates build a representation of the computation at compile time. This representation can then be translated into optimized CUDA kernels, minimizing temporary storage and maximizing parallelism.

- **Example in Financial Computing**: Consider a portfolio risk calculation involving multiple matrix and vector operations. Using expression templates, the entire computation tree can be captured as a single expression and offloaded to the GPU as a single kernel launch, reducing overhead and memory operations.
- **Benefits**:
  - Improves maintainability by keeping financial formulas expressed in readable C++ syntax.
  - Enables automatic generation of efficient CUDA kernels from high-level C++ code.
  - Reduces runtime overhead and temporary memory allocations.
- **Challenges:**
  - Debugging compile-time errors can be difficult due to heavy template usage.
  - Complexity in designing template hierarchies.

C. Code Transformation and Kernel Generation

Code transformation techniques automatically convert C++ constructs into GPU-executable CUDA code, reducing the need for manual kernel development. Key approaches include:

- **Static Analysis and Compiler-Assisted Transformation:** Compilers analyze loops, dependencies, and data structures in C++ code to determine parallelizable regions, generating corresponding CUDA kernels.
- **Genetic Algorithm or AI-Assisted Offloading:** Some advanced frameworks evaluate multiple code partitioning strategies to determine optimal offloading regions automatically, balancing GPU utilization and minimizing data transfer overhead.
- **Automatic Kernel Generation Libraries:** Libraries like TLoops and other template-based frameworks allow developers to write high-level tensor or matrix operations in C++, which are then transformed into optimized CUDA kernels at compile time.
- **Benefits in Financial HPC:**
  - Enhanced portability across different GPU architectures.
  - Improved kernel performance through automated optimization.
  - Reduced development time for GPU-accelerated financial algorithms.
- **Limitations:**
  - Performance depends heavily on compiler heuristics and GPU hardware characteristics.
  - Transformation frameworks may not capture all domain-specific optimizations.

Table I summarizes the fundamental differences between C++ and CUDA, highlighting their distinct purposes, execution models, memory architectures, and performance focuses. While C++ is a versatile, general-purpose programming language designed for a wide range of applications, CUDA is a GPU-focused parallel computing platform that extends C++ to leverage massive parallelism for high-performance workloads. This comparison underscores how C++ and CUDA complement each other in high-performance financial computing.

International Journal of Core Engineering & Management
Volume-6, Issue-10, 2020 ISSN No: 2348-9510

TABLE I. DIFFERENCE BETWEEN C++ AND CUDA

Aspect	C++	CUDA
Purpose	General-purpose programming language for building software across domains.	A programming paradigm and parallel computing platform created especially for NVIDIA GPUs.
Execution	Runs on CPU (serial or limited parallelism using threads).	Runs on GPU (massively parallel execution of thousands of threads).
Parallelism Model	Relies on CPU multi-threading and parallel libraries (e.g., OpenMP, TBB) for concurrency.	Uses SIMT (Single Instruction Multiple Thread) model to achieve massive parallelism.
Programming Model	Imperative, object-oriented, and generic programming with templates.	Extension of C++ with GPU-specific constructs such as kernels, threads, and memory management directives.
Memory Architecture	Uses CPU memory hierarchy (registers, caches, RAM).	Has distinct memory spaces — global, shared, constant, and registers — requiring explicit management.
Compilation	Compiled by a standard C++ compiler (e.g., g++, clang).	Requires CUDA compiler (nvcc) to compile device (.cu) code alongside C++ host code.
Code Structure	Includes functions, classes, templates, and STL constructs.	Includes kernels, device functions, thread hierarchies, and host–device synchronization.
Development Scope	Broad applications (desktop, embedded, scientific, finance, system software).	Specialized for GPU-accelerated computation in domains like HPC, AI, scientific computing, and finance.
Performance Focus	Optimized for single-thread performance and algorithmic efficiency.	Optimized for throughput and massive parallelism in compute-intensive workloads.
Ease of Use	Mature language with extensive libraries and toolchains.	More complex; requires explicit memory management and understanding of GPU architecture.

IV. APPLICATIONS OF GPU ACCELERATION IN FINANCIAL COMPUTING

Integration of C++ with CUDA is key to high-performance financial computing. By combining C++'s efficiency and flexibility with GPUs' huge parallel processing capabilities via CUDA, developers may expedite Monte Carlo simulations, option pricing, and portfolio optimization. Automatic offloading simplifies GPU-specific code writing, while bridging these two technologies produces high-performance, maintainable code. This section discusses processes, frameworks, and strategies for seamlessly connecting C++ applications to CUDA for optimized financial computations [20].

A. Monte Carlo simulations in statistical physics

Most traditional integration strategies fail for high-dimensional integrals, motivating Monte Carlo integration. Additionally, typical physical systems' phase spaces have a significant

International Journal of Core Engineering & Management
Volume-6, Issue-10, 2020 ISSN No: 2348-9510

number of dimensions. In three space dimensions, the phase space dimension for N classical particles is d = 6N since a particle can be described using just three coordinates and three momentum components. Figure 4 illustrates the Binder ratio's finite-size scaling and the system's corresponding phase transition behaviour.

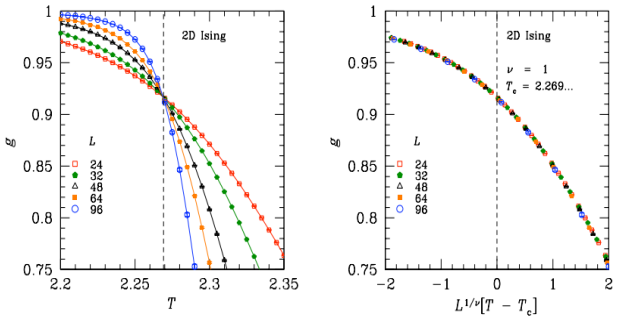

Fig. 4. Finite-Size Scaling of the Binder Ratio for the 2D Ising Model

Left panel: The two-dimensional Ising model with nearest-neighbor contacts and the relationship between temperature and its binder ratio. A transition is shown by data that almost crosses at a single point (the dashed line is the exact known Tc for the two-dimensional Ising model).

Right panel: Using the known Tc = 2.269. and v = 1, the data in the left panel is scaled to a finite size. The Binder ratio data is shown against the scaling variable L 1/v [T − Tc]. A universal curve that shows data for various system sizes indicates that the parameters being utilized are appropriate.

It is a perfect match for the Monte Carlo integration with significance sampling. States are now chosen based on the Boltzmann distribution, where P(s) represents the Boltzmann distribution. This is achieved when the factors cancel out. There is now a challenge in identifying an algorithm that permits a Boltzmann distribution sampling. It is referred to as the Metropolis algorithm [21].

B. Optimal trading strategy

The IT uses mid-price innovations to keep her informed before modifying her approach by determining how to use limit orders and the market to trade in and out of positions between now and T < T. Market orders guarantee speedy execution but are more expensive since the trader must pay the spread in addition to liquidity taking expenses. To fill her limit order, however, the trader must wait for an incoming market order. Limit orders, however, do not incur costs (they may be eligible for refunds in some markets), but execution is not guaranteed [22].

The IT only publishes passive orders at the touch (best bid or ask), where ` ± t = {' 1± t, . . . , `k± t} ∈ {0, 1} k indicates that she has decided to post a sell (+) or buy (−) limit order for one unit of asset at time t, with ` i± t = 0 indicating that there is currently no post. Additionally, suppose

that the IT's limit order is filled with probability p = {p1, . . . , pk} each time an incoming market order is received. Additionally, the number of market orders transmitted by the IT up to time t is counted by m± t = {m1± t..., mk± t} ∈ Z k +.

- A uniformed trader who believes that uniformed traders (UTs) lack the ability to generate a prior or learn from price innovations, hence the midprice dynamics are dictated by an arithmetic Brownian motion. In this case, the UT's strategy becomes that of a liquidity-supplying investor, akin to the models of market makers described in the corpus of current research.
- The other trader, known as the uninformed learner (UL), has a uniformed background but has the capacity to learn from market dynamics. In contrast to the UT's, his prior does not necessarily have an independent normal with mean zero and variance σ 2 i T, even though it is symmetric about the current midprice. Consequently, the UL can get insight from the midprice trends.

C. Impact of high-performance computing in finance

The advent of the above-mentioned mathematical models and the employment of cutting-edge workstations to solve them have led to significant advancements, particularly in risk pricing. The computing power of workstations would be depleted by some of the more intricate risk pricing models, forcing analysts to employ workstation clusters. Additionally, before a pricing model is implemented in practice on a group of workstations, it may occasionally be prototyped and tested using highly performant or massively parallel computers [23].

The majority of models for integrated risk management, especially those that use stochastic programming, depend on high-performance computers to solve their problems. It has even been extensively researched in the development of special-purpose parallel algorithms. Finally, the fact that computer-aided financial product creation entirely relies on high-performance parallel computations is only a hint that this field of study is still in its early experimental phase. Expect to see computer-aided design used on a broad range of platforms as it becomes increasingly common, encompassing workstation clusters, high-end workstations, and maybe systems with large parallel processing.

V. ROADMAP FOR OVERCOMING CHALLENGES IN C++ AND CUDA-BASED FINANCIAL HPC

This section presents a comprehensive roadmap for addressing the critical challenges in integrating C++ and CUDA for automatic offloading in high-performance financial computing. Table II summarizes each major challenge, its impact on financial HPC, current solutions and their limitations, and proposes future directions and novel trends to overcome these issues. The table also highlights the potential impact of these advancements, emphasizing how emerging techniques such as AI-assisted compilers, domain-specific frameworks, and adaptive runtime systems can enhance performance, portability, scalability, and efficiency in GPU-accelerated financial applications.

International Journal of Core Engineering & Management
Volume-6, Issue-10, 2020 ISSN No: 2348-9510

TABLE II. CHALLENGES, CURRENT SOLUTIONS, FUTURE DIRECTIONS, NOVEL TRENDS, AND IMPACTS IN C++ AND CUDA AUTOMATIC OFFLOADING

Challenge	Impact on Financial HPC	Current Solutions	Limitations of Current Solutions	Future Direction	Novel Trends	Potential Impact
Compiler and Portability Issues	Inconsistent builds, reduced portability across hardware and software environments.	CUDA updates, compiler version control, cross-platform frameworks.	Manual tuning still needed; poor automation; limited cross-platform support.	Develop AI-assisted compilers with adaptive code transformation.	Cross-platform offloading frameworks supporting multiple GPUs and heterogeneous hardware.	Increased portability, reduced development time, and wider deployment across systems.
Memory Management and Data Transfer	Latency and bandwidth bottlenecks, reduced GPU throughput.	Unified Memory, pinned memory, asynchronous data transfer APIs.	Still requires manual tuning; overhead in large datasets.	Improved Unified Memory and asynchronous data movement models.	Hardware–software co-design for dynamic memory management.	Reduced latency, improved throughput, and efficient memory utilization.
Domain-Specific Optimization	Suboptimal performance for financial workloads due to generic frameworks.	Manual kernel tuning, domain-specific libraries.	Time-consuming; lacks automation; non-scalable for diverse workloads.	Create financial-domain-specific offloading frameworks.	Incorporation of domain heuristics into GPU kernel generation.	Improved accuracy and performance for domain-specific algorithms such as option pricing and risk modeling.
Debugging and Profiling Complexity	Increased development time, difficulty diagnosing performance issues.	Nsight, CUDA-GDB, profiling tools.	High learning curve; limited automation; poor integration with offloading	Develop advanced profiling/debugging tools for hybrid programs.	AI-driven performance analysis and kernel optimization tools.	Faster development cycles, reduced debugging time, and improved reliability.

			frameworks .			
Scalability and Multi-GPU Coordination	Challenges in workload distribution and synchronization across GPUs.	MPI, NCCL, CUDA streams for GPU parallelism.	Limited automation; inefficient load balancing for heterogeneous tasks.	Design scalable multi-GPU scheduling algorithms.	Distributed GPU frameworks with adaptive load balancing.	Large-scale financial model acceleration with improved scalability.
Runtime Adaptability	Static offloading fails under dynamic workload conditions.	Manual runtime scheduling, workload profiling.	Not adaptive; lacks real-time optimization.	Implement adaptive runtime scheduling between CPU and GPU.	Integration of AI runtime systems for dynamic resource allocation.	Better performance consistency and adaptability to workload variations.
Integration with Emerging Architectures	Limited exploitation of novel hardware capabilities.	Vendor-specific libraries (cuBLAS, cuDNN, etc.)	Vendor lock-in; poor portability; steep learning curve.	Develop frameworks supporting FPGA, TPU, and quantum accelerators.	Hybrid computing platforms combining GPU, FPGA, and quantum acceleration .	Enhanced computing power, reduced runtime, and support for cutting-edge financial models.

VI. LITERATURE REVIEW

In this section, give a comprehensive overview of the research in Bridging C++ and CUDA for High-Performance Financial Computing, with some short summary information summarized in Table III.

Yamato et al. (2019) The automated graphics processing unit (GPU) offloading technology proposed in this research is a novel basic technique of Tacit Computing that automatically extracts suitable offloading regions from parallelizable loop statements using a genetic algorithm. IoT apps may perform better as a result. In a one-hour tuning period, evaluate the effectiveness of the proposed GPU offloading approach on five C/C++ image processing,

matrix manipulation, and other applications and find that it can process them more than 10 times quicker than using only central processing units [16].

Jin and Finkel (2019) The findings show that using vector data types in the kernels is not for speed and that more work items are better than large vectors per work item on the GPU. When GPU resources are constrained, streams should be handled carefully, However, CUDA streams and OpenCL may achieve almost identical GPU performance. When there is only one stream, the best performance on the FPGA may be obtained using kernel vectorization utilizing 16 vector lanes. In addition to reducing the kernel computation time for each stream, increasing the vector width per work-item and the number of streams will reduce the number of concurrent operations across the streams. The FPGA uses 3.4X less power than the GPU, despite the GPU's 3.1X more raw performance. The benefits of kernel offloading over a cutting-edge implementation on an Intel CPU server are becoming more and more apparent [24].

Shin et al. (2019) According to the workload characteristics, this paper proposes a workload-aware auto-parallelization framework (WAP) for DNN training, which automatically distributes the burden over several GPUs. They assess WAP's training throughput using TensorFlow against popular DNN benchmarks (AlexNet and VGG-16) and compare it to the most advanced frameworks. Additionally, they demonstrate how WAP automatically optimizes GPU assignment according to the computational requirements of the application, enhancing energy efficiency [25].

Mortatti, Yviquel and Araujo (2018) This article optimizes the design workflow, reduces the complexity of integrating cloud services and removes significant end-user interactions with an improvement of OpenMP 4.X. In a ray-tracing application, it uses this technique using a simplified version of the engines used in professional 3D modeling software (such as Blender). The rendering process is automatically moved from the user computer to a cluster of computers in the Microsoft Azure cloud after the calculation is complete. The completed pictures are then brought back and shown on the user computer's screen. This provides substantial speedups over local execution and a clear programming style [26].

Lewis and Pfeiffer (2018) offer the TLoops C++ library, which represents operations on tensorial quantities using a system of expression templates in single lines of C++ code that replicate analytic equations. It is possible to run these expressions in their original form or to generate comparable low-level C or CUDA code, which either speeds up the CPU's execution or enables a speedy translation to NVIDIA GPUs. The C++-class hierarchy and expression template that represent the expressions and enable automated code-generation are described in depth. Then, using several NVIDIA GPU generations, provide benchmarks for the expression-template code, which generated C code and CUDA code automatically [27].

Vulcan and Nicolae (2017) The method is comparable to GPU cloud computing, except the solution uses heterogeneous hardware and geographically dispersed computers. model is

International Journal of Core Engineering & Management
Volume-6, Issue-10, 2020 ISSN No: 2348-9510

available for usage online as a service. The number of available and running computers serves as a representation of the service's nodes. The programmer has the option to select between two load balancing methods offered by the service: automated and manual. In this manner, they wish to assess the viability of launching a business that offers processing capabilities by leasing processing resources from household customers [28].

TABLE III. SUMMARY OF AUTOMATIC OFFLOADING FOR HIGH-PERFORMANCE FINANCIAL COMPUTING

Reference	Focus On	Key Findings	Challenges	Limitations
Yamato et al. (2019)	Automatic GPU offloading using genetic algorithms in C/C++ applications	Proposed a GPU offloading method that extracts parallelizable loops automatically; achieved up to 10× speed improvement over CPU-only execution	Identifying optimal offloading regions automatically, balancing between automation and developer control	Limited evaluation scope (five applications); not specialized for financial workloads
Jin and Finkel (2019)	Comparative study of GPU and FPGA kernel offloading performance	Found that GPU offers 3.1× higher raw performance, while FPGA provides 3.4× lower power consumption; CUDA and OpenCL achieve similar GPU performance	Optimizing kernel vectorization and managing streams effectively	Performance highly dependent on workload type and vectorization strategy
Shin et al. (2019)	Workload-aware automatic parallelization (WAP) for DNN training	Automatically distributed workload to multiple GPUs; improved energy efficiency and throughput with TensorFlow benchmarks	Dynamic workload balancing and GPU scheduling	Limited to DNN training; not generalized to traditional C++ workloads
Mortatti, Yviquel & Araujo (2018)	Cloud-based GPU offloading via OpenMP extensions	Extended OpenMP 4.X for automatic offloading to Azure cloud clusters; achieved transparent integration and speed-ups over local runs	Efficient resource management in cloud environment	Focused on ray-tracing applications; latency from cloud transfer
Lewis & Pfeiffer (2018)	C++ template-based automatic CUDA code generation (TLoops library)	Developed TLoops library for automatic C/CUDA code generation from tensorial C++ expressions; improved portability and execution speed	Maintaining abstraction without sacrificing low-level optimization	Targeted mainly at tensor computations; requires CUDA-compatible hardware

Vulcan & Nicolae (2017)	Distributed GPU computing model as a cloud service	Proposed a heterogeneous, geographically distributed GPU service with manual/automatic load balancing	Coordinating distributed nodes efficiently; ensuring reliability over internet	Prototype-level implementation; scalability and real-world validation untested

VII. CONCLUSION AND FUTURE WORK

This survey illustrates the importance of bridging C++ and CUDA to enhance financial computing performance and highlights the value of using automatic offloading mechanisms. By evaluating compiler-assisted, template-based, and framework-based approaches, find that high-performance computing for financial applications, such as Monte Carlo simulations, option pricing, and portfolio optimization, can be achieved with a reduced need for manual effort to the benefit of performance. Even with all these approaches, however, there are still challenges in compiler compatibility, memory management, parallelization verification, and multi-GPU coordination. Although expression templates, code transformation, and AI-assisted offloading show promise for finagling solutions, the domain-specific optimizations are still limited in practice. Overall, using automatic offloading from C++ to CUDA for financial computing is a significant innovation that offers a good balance of developer productivity, code maintainability, and high-performance in execution, so financial institutions can productively utilize GPU acceleration.

Future research needs to investigate domain-aware automated offloading for financial workloads, parallelization accuracy, and memory transfer resource overheads. More research on multi-GPU and heterogeneous system coordination will lead to scalable methods. AI-derived optimization, adaptive runtime profiling, and predictive workload management can boost hardware efficiency. Standardizing compiler support across platforms and improving debugging tools helps boost adoption. Real-world benchmarks on complex financial models will support the claimed performance and efficiency benefits that will enable robust, high-performance financial computing with minimal human intervention.

REFERENCES

1. R. Rajak, "A Comparative Study: Taxonomy of High Performance Computing (HPC)," Int. J. Electr. Comput. Eng., vol. 8, no. 5, pp. 3386–3391, Oct. 2018, doi: 10.11591/ijece.v8i5.pp3386-3391.
2. J. Dongarra, T. Herault, and Y. Robert, "Fault Tolerance Techniques for High-Performance Computing," 2015, pp. 3–85. doi: 10.1007/978-3-319-20943-2_1.
3. B. R. Cherukuri, "Future of cloud computing: Innovations in multi-cloud and hybrid architectures," World J. Adv. Res. Rev., vol. 1, no. 1, pp. 068–081, Feb. 2019, doi: 10.30574/wjarr.2019.1.1.0002.

4. A. Balasubramanian, "Proactive Machine Learning Approach to Combat Money Laundering in Financial Sectors," Int. J. Innov. Res. Eng. Multidiscip. Phys. Sci., vol. 7, no. 2, pp. 1–15, 2019.
5. R. C. Cavalcante, R. C. Brasileiro, V. L. F. Souza, J. P. Nobrega, and A. L. I. Oliveira, "Computational Intelligence and Financial Markets: A Survey and Future Directions," Expert Syst. Appl., vol. 55, pp. 194–211, Aug. 2016, doi: 10.1016/j.eswa.2016.02.006.
6. Z. Q. Jiang, W. J. Xie, W. X. Zhou, and D. Sornette, "Multifractal analysis of financial markets: a review," Reports Prog. Phys., vol. 82, no. 12, Dec. 2019, doi: 10.1088/1361-6633/ab42fb.
7. V. M. L. G. Nerella, "Automated Cross-Platform Database Migration and High Availability Implementation," Turkish J. Comput. Math. Educ., vol. 9, no. 2, pp. 823–835, Jul. 2018, doi: 10.61841/turcomat.v9i2.15284.
8. S. S. S. Neeli, "Serverless Databases: A Cost-Effective and Scalable Solution," Int. J. Innov. Res. Eng. Multidiscip. Phys. Sci., vol. 7, no. 6, pp. 1–7, 2019.
9. R. S. Dehal, C. Munjal, A. A. Ansari, and A. S. Kushwaha, "GPU Computing Revolution: CUDA," in 2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN), IEEE, Oct. 2018, pp. 197–201. doi: 10.1109/ICACCCN.2018.8748495.
10. D. J. Duffy, "Beyond Object-Orientation: C++ Application Design for Computational Finance - A Defined Process from Problem to Parallel Code," Wilmott, vol. 2018, no. 93, pp. 60–69, Jan. 2018, doi: 10.1002/wilm.10647.
11. S. S. S. Neeli, "The Significance of NoSQL Databases: Strategic Business Approaches and Management Techniques," J. Adv. Dev. Res., vol. 10, no. 1, pp. 1–11, 2019.
12. V. M. L. G. Nerella, "MIGRATE: A Rollback-Enabled Framework for Automated Oracle XTTS-Based Cross-Platform Database Migrations," J. Electr. Syst., vol. 14, no. 4, pp. 85–95, 2018.
13. R. S. Dehal, C. Munjal, A. A. Ansari, and A. S. Kushwaha, "GPU Computing Revolution: CUDA," in 2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN), 2018, pp. 197–201. doi: 10.1109/ICACCCN.2018.8748495.
14. P. Diehl, M. Seshadri, T. Heller, and H. Kaiser, "Integration of CUDA Processing within the C++ Library for Parallelism and Concurrency (HPX)," in 2018 IEEE/ACM 4th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2), IEEE, Nov. 2018, pp. 19–28. doi: 10.1109/ESPM2.2018.00006.
15. C. Ceissler, R. Nepomuceno, M. Pereira, and G. Araujo, "Automatic Offloading of Cluster Accelerators," in 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), IEEE, Apr. 2018, pp. 224–224. doi: 10.1109/FCCM.2018.00058.
16. Y. Yamato, T. Demizu, H. Noguchi, and M. Kataoka, "Automatic GPU Offloading Technology for Open IoT Environment," IEEE Internet Things J., vol. 6, no. 2, pp. 2669–2678, Apr. 2019, doi: 10.1109/JIOT.2018.2872545.

17. E. S. Mtsweni and N. Maveterra, "Issues affecting application of tacit knowledge within software development project," Procedia Comput. Sci., vol. 138, pp. 843–850, 2018, doi: 10.1016/j.procs.2018.10.110.
18. Y. Yamato, T. Demizu, H. Noguchi, and M. Kataoka, "Proposal of Automatic GPU Offloading Technology on Open IoT Environment," in 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), 2018, pp. 634–639. doi: 10.1109/COMPSAC.2018.10309.
19. M. Springer and H. Masuhara, "Ikra-Cpp: A C++/CUDA DSL for Object-Oriented Programming with Structure-of-Arrays Layout," in Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing, New York, NY, USA: ACM, Feb. 2018, pp. 1–9. doi: 10.1145/3178433.3178439.
20. R. Orús, S. Mugel, and E. Lizaso, "Quantum computing for finance: Overview and prospects," Rev. Phys., vol. 4, p. 100028, Nov. 2019, doi: 10.1016/j.revip.2019.100028.
21. R. G. McClarren, "Introduction to Monte Carlo Methods," in Computational Nuclear Engineering and Radiological Science Using Python, Elsevier, 2018, pp. 381–406. doi: 10.1016/B978-0-12-812253-2.00024-8.
22. Á. Cartea, S. Jaimungal, and D. Kinzebulatov, "Algorithmic Trading with Learning," Int. J. Theor. Appl. Financ., vol. 19, no. 4, 2016, doi: 10.1142/S021902491650028X.
23. L. Hong, L. Zhong-hua, and C. Xue-bin, "The Applications and Trends of High Performance Computing in Finance," in 2010 Ninth International Symposium on Distributed Computing and Applications to Business, Engineering and Science, IEEE, Aug. 2010, pp. 193–197. doi: 10.1109/DCABES.2010.45.
24. Z. Jin and H. Finkel, "Base64 Encoding on Heterogeneous Computing Platforms," in 2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP), IEEE, Jul. 2019, pp. 247–254. doi: 10.1109/ASAP.2019.00014.
25. S. Shin, Y. Jo, J. Choi, S. Venkataramani, V. Srinivasan, and W. Sung, "Workload-aware Automatic Parallelization for Multi-GPU DNN Training," in ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), IEEE, May 2019, pp. 1453–1457. doi: 10.1109/ICASSP.2019.8683053.
26. M. Mortatti, H. Yviquel, and G. Araujo, "Automatic Ray-Tracer Cloud Offloading in OPENMP," in 2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), IEEE, Sep. 2018, pp. 428–435. doi: 10.1109/CAHPC.2018.8645871.
27. A. G. M. Lewis and H. P. Pfeiffer, "Automatic generation of CUDA code performing tensor manipulations using C++ expression templates," pp. 1–47, 2018.
28. A. M. Vulcan and M. Nicolae, "A smart grid model for high performance computing service," in 2017 10th International Symposium on Advanced Topics in Electrical Engineering (ATEE), IEEE, 2017, pp. 925–928. doi: 10.1109/ATEE.2017.7905124.