A Machine Learning Approach for Efficient Parallel Simulation of Beam Dynamics on GPUs

Kamesh Arumugam*,†, Alexander Godunov ‡, Tunazzina Islam *,†, Desh Ranjan*,†, Balša Terzić†‡ and Mohammad Zubair*‡

*Department of Computer Science, Old Dominion University, Norfolk, Virginia 23529
†Center for Accelerator Science, Old Dominion University, Norfolk, Virginia 23529
‡Department of Physics, Old Dominion University, Norfolk, Virginia 23529

Abstract—Parallel computing architectures like GPUs have traditionally been used to accelerate applications with dense and highly-structured workloads; however, many important applications in science and engineering are irregular and dynamic in nature, making their effective parallel implementation a daunting task. Numerical simulation of charged particle beam dynamics is one such application where the distribution of work and data in the accurate computation of collective effects at each time step is irregular and exhibits control-flow and memory access patterns that are not readily amenable to GPU’s architecture. Algorithms with these properties tend to present both significant branch and memory divergence on GPUs which leads to severe performance bottlenecks.

We present a novel cache-aware algorithm that uses machine learning to address this problem. The algorithm presented here uses supervised learning to adaptively model and track irregular access patterns in the computation of collective effects at each time step of the simulation to anticipate the future control-flow and data access patterns. Access pattern forecast are then used to formulate runtime decisions that minimize branch and memory divergence on GPUs, thereby improving the performance of collective effects computation at a future time step based on the observations from earlier time steps. Experimental results on NVIDIA Tesla K40 GPU shows that our approach is effective in maximizing data reuse, ensuring workload balance among parallel threads, and in minimizing both branch and memory divergence. Further, the parallel implementation delivers up to 485 Gflops of double precision performance, which translates to a speedup of up to 2.5X compared to the fastest known GPU implementation.

I. INTRODUCTION

The data-parallel, SIMD nature of GPU architectures has traditionally been used to accelerate applications with dense and highly-structured workloads common in many problem domains ranging from graphics applications to molecular dynamics simulation and climate modeling. GPUs perform particularly well when the control-flow and memory access patterns are regular and predictable, as is the case in dense structured workloads like linear algebra computations. However, there has been considerable effort in recent years to develop GPU implementations to a wide variety of application from science and engineering that are unstructured and dynamic in nature. Many of these applications exhibit control-flow and memory access patterns that are not readily amenable to GPU’s architecture [1]–[5]. In particular, distribution of work and data in these applications cannot be characterized a priori because these quantities are input-dependent and evolve with the computation itself, and when implemented on GPUs, they demonstrate significant memory access irregularity [1], [6]. Applications with these properties are said to be irregular, and pose problems for high performance parallel implementations, where equal distribution of work over processor cores and locality of reference are required within each cache-sharing processor. Moreover, performance of applications on GPU architectures relies on high SIMD lane occupancy and efficient memory coalescing for inter-thread data locality, where the former requires minimal divergent branching for threads in a SIMD group, while the latter requires regular memory access patterns and data structure layouts [7], [8]. Unfortunately, irregular applications tend to present both significant branch and memory divergence on GPUs which leads to severe performance bottlenecks.

Numerical simulation of charged particle beam dynamics is one such irregular application that has gained increased interest in accelerator physics, especially in recent years, as these simulations are crucial in understanding and the design of: (i) high-brightness synchrotron light sources - powerful tools for cutting-edge research in physics, biology, medicine and other fields, and (ii) electron-ion particle colliders, which probe the nature of matter at unprecedented depths. In this irregular application, distribution of work and data in the accurate computation of collective effects at each time step of the simulation is highly unstructured and they cannot be characterized a priori, as these quantities are input-dependent and evolve with the computation itself. To obtain high-performance in such algorithms is extremely challenging, and to this end, researchers have developed two parallel algorithms to accurately calculate the collective effects at each time step of the simulation using GPUs, [9] and [10]. Algorithm presented in [9] targets equal distribution of work over processor to reduce control-flow irregularity, and [10] presents heuristics to maximize data reuse and to balance the workload among threads, thereby reducing both control-flow and memory access divergence. Both of these algorithms focus on optimizing the irregular, data-dependent memory accesses and control-flow during a single time step of the simulation independent of the other steps, with the assumption that these patterns are completely unpredictable.

Multiple analysis of beam dynamics simulation executing irregular workloads for a few hundreds or thousands of time
steps show that control-flow and data access patterns made during the computation of collective effects follow a loosely similar pattern between time steps. In such situation, one effective approach to reduce the irregularities is to analyze the control-flow and data access patterns at each time step of the simulation and then anticipate future data dependence and control-flow before it is needed. Given the complexity and diversity of control-flow and data access patterns in beam dynamics simulation, we believe anticipation strategies are best realized via intelligent application-specific prediction models that can adaptively model and track access patterns. Access pattern forecasts can then be used to formulate runtime decisions that optimize future computations of collective effects on GPUs, such as determining computations to thread mapping that maximize data reuse within a cache-sharing thread group and minimize thread divergence, data prefetching, computational workload balancing, linearizing the irregularities, etc.

This paper explores the use of predictive analytics and forecasting techniques to optimize the computation of collective effects on GPUs, thereby improving the overall performance of beam dynamics simulation. In particular, we present a cache-aware algorithm that use machine learning to forecast the control-flow and data access patterns required to calculate the collective effects at a future time step based on the computations and access patterns observed from earlier time steps. The parallel implementation of this novel algorithm on NVIDIA Tesla K40 GPU leads to 2.5X improvement in speedup compared to the fastest known parallel algorithm, thereby enabling unprecedented fidelity and precision in the study of all the relevant physics on synchrotron light sources and electron-ion particle colliders. The remainder of the paper is organized as follows. Section II outlines the algorithm for numerical simulation of electron beam dynamics and then provides a brief overview on the related work in its parallel implementations. In Section III, we present the machine learning approach to model irregular data access patterns in the computation of collective effects where the future values of control-flow and data access patterns are predicted based on the previously observed values. Next, the parallel algorithm to calculate collective effects using predictive analytics and its implementation on GPU architecture is illustrated in Section IV. Section V illustrates the performance of parallel algorithm on K40 GPU. Finally, in Section VI we summarize our findings and conclude.

II. ELECTRON BEAM DYNAMICS

A. Simulation Algorithm

Numerical simulation of electron beam dynamics on a 2D plane of the beam lattice consists of four consecutive steps that are computed at each time step of the simulation, and are repeated for a few hundreds or thousands of time steps (see Figure-1). Formally, for a simulation with step size \( \Delta t \), following four steps are executed during each time step \( k \), for some integer \( k \) in the range 0 to \( N_t \), where \( N_t \) is the number of time steps required for the simulation -

\[
\begin{align*}
1) & \text{ Particle Deposition:} \quad \text{Deposit the distribution function sampled by } N \text{ particles onto a 2D data grid of } N_X \times N_Y \text{ resolution using the particle-in-cell method [11]–[13], thereby yielding 2D grid of moments, one for each grid-point. The moments here is a multidimensional quantity representing the distribution’s deposited charge, current densities, etc. For simplicity, we use the notation } D_k \text{ to denote the 2D data grid of moments at time step } k. \\
2) & \text{ Compute Retarded Potentials:} \quad \text{The collective effects in the particle distribution due to beam’s self-interaction is modeled through retarded potentials, which are computed at each grid-point on the 2D data grid. Formally, suppose } V_k \text{ denotes the set of grid points on the 2D data grid at time step } k, \text{ such that } |V_k| = N_X N_Y \text{ and each grid-point } p \in V_k \text{ is a two-dimensional point } (x,y), \text{ denoting Cartesian coordinate of the grid-point on the 2D data grid (i.e., } (x,y) \text{ is the position of } p \text{ on } D_k). \text{ Then, the retarded potentials at } p \text{ is given by,} \\
3) & \text{ Compute Self-Forces:} \quad \text{Evaluate retarded potentials} \\
4) & \text{ Push Particles:} \quad \text{Advance particles by } \Delta t
\end{align*}
\]

\[
I(p) = \int_0^{R(p)} dr' \int_{\theta_{\min}}^{\theta_{\max}} (r', \theta', t') d\theta' \tag{1}
\]

where \( 0 < R(p) \leq \kappa c \Delta t \) for some positive integer \( \kappa \leq k \), retarded time \( t' = k \Delta t - r'/c, i \Delta t < t' \leq (i+1) \Delta t \) for some integer \( i \) in the range \( k - \kappa \leq i < k \), and the computation of integral limits is identical to that in [9]. For convenience, we shall refer to the integral in Equation-1 as \( \text{rp}-\text{integral} \).

The integrand \( f^{(p)}(r', \theta', t') \) in \( \text{rp}-\text{integral} \) denotes the moments deposited on the polar coordinate \( (r', \theta') \) (with center at \( (x,y) \)) by the particle distribution at simulation time \( t' \). However, \( f^{(p)} \) does not have an analytic form, and as a result, \( f^{(p)}(r', \theta', t') \) is approximated using 27 neighboring points from the data grids of moments, \( D_{i-1}, D_i, \text{ and } D_{i+1} \). In addition, these data grids from time steps \( i - 1, i, \text{ and } i + 1 \) are required for calculating integrand values for all \( r' \) sampled from the subregion \( [(k - i-1)c \Delta t, (k - i)c \Delta t] \) along outer dimension (i.e., for all \( t' \in [i \Delta t, (i+1) \Delta t] \)). Alternatively, let \( S_j \) denote the subregion \( [j c \Delta t, (j + 1)c \Delta t] \) along outer dimension, for all positive integer \( j \), then the data grids \( D_{i-1}, D_i, \text{ and } D_{i+1} \) are required to compute \( \text{rp}-\text{integral} \) along \( S_{k-i-1} \). Further, numerical approximation of \( \text{rp}-\text{integral} \) along the integration region \([0, R(p)]\) requires all the data grids
between time steps \( k - \kappa \) and \( k \), where the computation along \( S_j \) uses data from \( D_{k-j-1} \), \( D_{k-j-2} \), and \( D_{k-j-3} \). Adapting to this relation between the subregion and the set of data grids accessed during \( \text{rp}-\text{integral} \) evaluation along that subregion, the domain \([0, R(p)]\) can be divided into a set of subregions, \( S = \{S_0, S_1, \ldots, S_N\} \), and given such a set \( S \), Equation-1 is calculated as the sum of \( \text{rp}-\text{integrals} \) along all the subregions in \( S \).

Numerical approximation of \( \text{rp}-\text{integral} \) used repeated one-dimensional integration algorithm, as illustrated in [9], [14]. In particular, for a grid-point \( p \in V_k \) and a subregion \([a, b]\) along outer dimension, integration algorithm results in a partition, \( r^{(p)}_0, r^{(p)}_1, \ldots, r^{(p)}_n \), along the outer dimension, where \( r^{(p)}_0 = a < r^{(p)}_1 < \cdots < r^{(p)}_n = b \) and \( n > 0 \) is an integer, such that the partition has fine spacing where the integrand is varying rapidly and coarse spacing where the integrand is varying slowly. Given the partition, \( \text{rp}-\text{integral} \) is calculated as follows:

\[
I(p) = \sum_{i=0}^{n-1} Q(r^{(p)}_i, r^{(p)}_{i+1}) \tag{2}
\]

where \( Q(r^{(p)}_i, r^{(p)}_{i+1}) \) is a Quadrature rule estimate along the subregion \([r^{(p)}_i, r^{(p)}_{i+1}]\), and for every \( r' \) sampled along that subregion, inner integral is computed using Newton-Cotes formulae, as illustrated in [14].

Note that high-fidelity computation of collective effects by calculating retarded potentials at all grid points on the 2D data grid is the crucial and by far the most computationally intensive step of this simulation. As a result, to obtain high performance in the overall beam dynamics simulation, it is important to optimize this step.

3) Compute Self-Forces: The self-forces acting on each particle are computed by interpolation from the 2D data grid of moments and retarded potentials.

4) Push Particles: The particles are advanced to next time step by a small increment \( \Delta t \) in time by solving Lorentz equation [9], using leap-frog scheme [15].

B. Related Work

A number of prior studies have addressed acceleration of beam dynamics simulation using different methods [15]–[19]. Many of these methods employ approximations to simplify the computational complexity associated with high-fidelity calculation of collective effects [16]–[18], and this often trades computation time for the quality and accuracy of simulation. That is, approximation methods improve performance of beam dynamics simulation while sacrificing the accuracy. Recent research demonstrates that such approximations can be relaxed, and a high-fidelity simulation that models the collective effects much more accurately is feasible by leveraging the benefits of high-performance computing [9], [10].

The parallel algorithm in [9] is based on a globally adaptive parallel quadrature that exploits the power of GPU architecture to resolve the computational challenges associated with accurate evaluation of collective effects. However, the parallel implementation does not fully exploit the benefit of target architecture, and exhibits large number of irregular memory accesses that leads to inefficient use of the memory hierarchy, thereby significantly deteriorating the overall simulation performance on GPUs. The cache-aware algorithm presented in [10] address the memory inefficiencies in [9] by using heuristics. [10] propose two heuristics to reduce the control-flow and memory access irregularities in the parallel computation of collective effects on GPUs, where one heuristic maximize the data reuse by improving data locality between cache-sharing thread groups and the other heuristic maintains workload balance between parallel threads. The parallel algorithm and its implementation illustrated in [10] is currently the fastest known method for high-fidelity computation of collective effects in a charged particle beam dynamics simulations.

A more comprehensive review on the performance of GPU algorithms for parallel beam dynamics simulation is illustrated in [10], and a review on approximation based simulation algorithms is illustrated in [20].

III. MODELING ACCESS PATTERNS

An effective model for forecasting irregular data access patterns must predict when, what, and how many data blocks are required by an application before it is needed. To obtain these predictions, we have modeled data access patterns in the computation of retarded potentials (which induce collective effects) using application-specific supervised learning algorithms described in this section. First, we outline the representation of data access pattern in the numerical approximation of \( \text{rp}-\text{integrals} \), which are later used as input to the learning algorithm. Next, we present the online prediction model that use supervised learning on the observed data access patterns to train the model. Finally, we outline the algorithm to forecast the data access and control-flow patterns in \( \text{rp}-\text{integral} \) evaluations at a future time step using the prediction model learned at an earlier time step.

A. Representation of Data Access Pattern

The computation of \( \text{rp}-\text{integral} \) at a grid-point \( p \in V_k \) during time step \( k \) requires referencing data from the 2D data grids computed from one or more earlier time steps. In particular, calculating \( \text{rp}-\text{integral} \) along the subregion \( S_i \) requires referencing data from the 2D data grids computed during time steps \( k - i, k - i - 1, \) and \( k - i - 2 \) (i.e., data grids \( D_{k-i}, D_{k-i-1}, \) and \( D_{k-i-2} \)), for all positive integers \( i \) and \( k \) such that \( i < k \) and \( k \leq N_t \). Further, when \( S_i \) is subdivided into \( n_i \) partitions then \( \text{rp}-\text{integral} \) evaluation within that subregion will result in \( \alpha n_i \) memory references to each of the three data grids, where \( \alpha \) is the number of memory references made during the computation of inner integral, which is constant for a given Newton-Cotes formulae.

Adapting to this relation between the number of partitions and the memory references, we choose to represent the data access pattern in \( \text{rp}-\text{integral} \) evaluation using the partition generated along subregions, \( S_0, S_1, \ldots, S_{N_t-3} \). More formally, data
access pattern observed during rp-integral evaluation at a grid-point \( p \in V_k \) is represented by a list, \([n_0^{(p)}, n_1^{(p)}, \ldots, n_{N_p}^{(p)}]\), where \( n_j^{(p)} \) denotes the number of partitions along the subregion \( S_j \) required during rp-integral evaluation at \( p \), and given the access pattern, we can easily calculate the memory references to any data grid. As an example, number of reference to \( D_{k-i} \) is given by \( \alpha(n_i^{(p)} + n_{i-1}^{(p)} + n_{i-2}^{(p)}) \).

In our implementation, data access patterns are extracted during regular execution of the simulation with negligible overhead except for additional storage required to log the access patterns. These stored access patterns are later used by a supervised learning algorithm to train the online prediction model. Note that we have chosen to model the access patterns using coarser data grids instead of individual data elements. The rationale behind coarser modeling is that the irregular nature of beam dynamics simulation makes it challenging to track access to individual data elements. Moreover, even if tracking individual data elements was feasible, the overhead associated with storing the number of references to each data element will increase the memory requirement of beam dynamics simulation, which is already a memory intensive application.

**B. Online Prediction Model**

To capture and forecast the irregular data access patterns, we model the application access patterns using online prediction techniques where the future values are predicted based on the previously observed values. Formally, suppose the current time step of the simulation is \( k \) and we are given a set of access patterns observed during rp-integral computations at all grid points up to time step \( k \). Then, using all the data access patterns observed up to time step \( k \), the prediction model forecasts the access pattern \([n_0^{(q)}, n_1^{(q)}, \ldots, n_{N_q}^{(q)}]\) required to compute rp-integral at a grid-point \( q \in V_j \) for a future time step \( j \), where \( j > k \). This forecast is used to formulate intelligent runtime decisions that optimize the application execution during time step \( j \). Further, forecasts can be one-step ahead forecasting where \( j = k+1 \), or multiple step ahead forecasting where \( j >> k \). We use one-step ahead forecasting in this study.

1) **Training and Prediction:** For model training, given a set of training examples of the form \((x_1, y_1), \ldots, (x_n, y_n)\) where \( x_i \) is the grid-point of the \( i^{th} \) example and \( y_i \) is the access pattern observed during rp-integral evaluation at \( x_i \), a learning algorithm seeks a function \( g: X \rightarrow Y \), where \( X \) is the space of inputs (i.e., grid points) and \( Y \) is space of outputs (i.e., rp-integral data access patterns). In particular, at time step \( k \), the set of rp-integral computations at all grid points from one or more earlier time steps is used as training data by a supervised learning algorithm to seek a predictor function \( g_k: X \rightarrow Y \) at that particular time step. However, keeping track of rp-integral computations and access patterns from multiple time steps may increase the computational load and memory requirement of the application. In such situation, we can use supervised learning algorithms with online training where the function predictor at \( k^{th} \) time step, \( g_k \), is learned just from the access patterns observed during time step \( k \), and the previous best predictor \( g_{k-1} \).

The choice of learning algorithm depends on the data distribution, quality and nature of the data, required accuracy of prediction, and so on. Typically, each learning algorithm have different effect on a given problem, and as result, choosing the right algorithm often requires studying multiple algorithms and its effects on the problem before choosing the best performing one. In this study, we use \( k \)-nearest neighbor algorithm (kNN) in regression setting to train the prediction model, which, based on the heuristics in [10] is an intuitive choice. In addition to kNN, we also experimented with linear regression and noticed a negligible difference in the overall performance compared to kNN. However, hyperparameters tuning and other algorithms may have different effect on the beam dynamics problem. As the goal of this study was not to compare different learning algorithms but rather to explore the potential use of machine learning algorithms to improve the application performance, we limit our discussion to kNN and propose to explore the effects of different learning algorithm on applications performance in our future studies.

**C. Forecasting**

1) **Memory Access Patterns:** Given a predictor function \( g_{k-1} \) learned at time step \( k-1 \), the data access pattern for rp-integral evaluation at a grid-point \( p \in V_k \) for time step \( k \) is approximated as, \( g_{k-1}(p) \).

In the proposed algorithm for parallel beam dynamics simulation, illustrated in Section-IV, the predicted access patterns for all points \( p \in V_k \) are used to determine rp-integral computations to thread mapping that maximizes the data reuse within a cache-sharing thread group on the target architecture. This leads to an improved cache performance, even with the presence of memory access irregularity. Also, note that the predicted access patterns are just an approximation to the observed access patterns, and they are primarily used to reduce memory access irregularities in the simulation by optimizing computation to thread mapping, therefore, does not compromise the correctness of integral computation.

2) **Control-flow:** The flow of computation or control-flow for numerically approximating rp-integral at grid-point \( p \in V_k \) is typically determined by the algorithm used to compute the partition, \((r_0^{(p)}, r_1^{(p)}, \ldots, r_n^{(p)})\), along the outer dimension. Adaptive quadrature is traditionally used to compute such partitions, which, as illustrated in [21], [22], is characterized by control-flow and memory access irregularities that leads to severe performance bottlenecks on GPU architectures.

In the proposed algorithm we use predicted access patterns to approximate rp-integral partition, and given the partition, computation of rp-integral simply involves evaluating Equation-2. Such evaluation exhibit uniform and deterministic control-flow that can be mapped to GPUs with minimal thread divergences, thereby improving the overall performance. Formally, given a predicted access pattern \([n_0^{(p)}, n_1^{(p)}, \ldots, n_{N_p}^{(p)}]\) corresponding to a grid-point \( p \in V_k \), the forecasting algorithm computes a partition list, \((r_0^{(p)}, r_1^{(p)}, \ldots, r_m^{(p)})\), required to
calculate rp-integral at \( p \), which is an approximation to the partition required to calculate rp-integral within the required error tolerance. The following two methods are used to transform the data access pattern to integral partition:

1) Uniform partitioning - Each subregion \( S_i \) is divided into \( (n_i(p) - 1) \) finer subregions of equal size (i.e., \( n_i(p) \) partitions along \( S_i \)), for all integers \( i \) in range \( 0 < i < N_i \). This generates a global partition of size \( \sum_{i=0}^{N_i} n_i(p) \) on the entire integration region \( [0, R(p)] \).

2) Adaptive partitioning - Partition generated at an earlier time step is used alongside the access pattern forecast to approximate the partition at time step \( k \). The choice of partition from an earlier time step is identical to the approach used in [10], and this partition is updated using the access pattern forecast to generate a new partition which is used during time step \( k \). For example, let the partition from an earlier time step contain \( d_i \) partitions along \( S_i \), then each subregion in \( S_i \) from the earlier partition is divided into \( n_i(p)/d_i \) finer subregions to generate a new partition, which is used for rp-integral calculation at time step \( k \). The partition size generated using this approach is approximately \( \sum_{i=0}^{N_i} n_i(p) \) on the entire integration region \( [0, R(p)] \).

Note that the partition forecast computed from the access pattern is an approximation to the partition required to calculate rp-integral within the required error tolerance, and it is possible that the rp-integral estimate calculated using this partition forecast is not within the required error tolerance. The proposed algorithm illustrated in Section-IV handles this situation by considering the prediction as an initial condition for numerically approximating rp-integral and ensures that integral estimate always achieves the required error tolerance.

IV. PARALLEL SIMULATION ALGORITHM

The procedure COMPUTE-POTENTIALS in Algorithm-1 implements the second step of the four step beam dynamics simulation algorithm where it approximates the rp-integral at all grid points on a 2D data grid for a given time step. The procedure takes input \( k, V, \tau, g, \) and \( D \), where \( k \) is the current time step of simulation, \( V \) is a set of grid points on the 2D data grid at \( k^{th} \) time step such that \( |V| = N_X N_Y \), \( \tau \) is the required error tolerance for rp-integral evaluations, \( g \) denotes the predictor function learned using supervised learning algorithm at time step \( k-1 \), and \( D \) is the list of 2D data grids of moments from each time step stored linearly on the device memory. Each grid-point \( p \in V \) is a reference to 7-tuple object, \((x, y, t, I, \epsilon, access\_pattern, partition)\), where \((p.x, p.y)\) denote the Cartesian coordinate of the grid-point on the 2D grid at time step \( k \), \( p.I \) is the simulation time of the corresponding time step, \( p.I \) is the rp-integral estimate, \( p.\epsilon \) is the rp-integral error estimate, \( access\_pattern \) is a list containing the data access pattern for rp-integral computation, and \( partition \) holds a list containing the partition for rp-integral computation.

The procedure COMPUTE-POTENTIALS works as follows. Line 1-5 initializes different attributes of the grid-point object.

In particular, for each grid-point \( p \in V \), line-2 initializes integral and error estimates to 0, line-3 uses the best predictor function \( g \) learned at time step \( k-1 \) to forecast the access pattern required for rp-integral computation for the current time step \( k \), and line-4 calls a procedure that implements the algorithm described in Section III-C2 to convert access pattern forecast to rp-integral partition. Next, RP-CLUSTERING procedure at line-6 implements a clustering algorithm to partition the grid points based on their data access patterns such that access patterns for grid points in the same cluster are similar to one another. Formally, given a set of grid points \( V \) and an integer \( m \), RP-CLUSTERING procedure partitions the \(|V|\) grid points into \( m \) disjoint clusters, \( C = \{C_1, C_2, \ldots, C_m\} \), such that the sum of distance between the grid points access pattern in the cluster to its center is minimum,

\[
\arg\min_C \sum_{i=1}^{m} \sum_{p \in C_i} \|p.access_pattern - \mu_i\|^2
\]

where \( \mu_i \) is the center of cluster \( C_i \), and \( m = N_X \) or \( N_Y \). We use k-means clustering algorithm to implement RP-CLUSTERING procedure.

Furthermore, two grid points \( u, v \in V \), where \( u \neq v \), gets partitioned into same cluster when \( u.access\_pattern \) exhibits stronger similarity to that of \( v.access\_pattern \), which also implies that rp-integral computation on \( u \) \& \( v \) have maximum data reuse between them. This property of data reuse and access pattern similarity within a cluster is used to optimize rp-integral computations to thread mapping such that the overall memory performance on the target architecture is maximized. In other words, when a set of rp-integral

Algorithm 1 COMPUTE-POTENTIALS(k, V, \( \tau, g, D \))

1: for each grid-point \( p \in V \) parallel do
2: \( p.I \leftarrow 0, p.\epsilon \leftarrow 0 \)
3: \( p.access_pattern \leftarrow g(p.x, p.y, p.t) \)
4: \( p.partition \leftarrow COMPUTE-PARTITION(p.access_pattern) \)
5: end for
6: \( C \leftarrow RP-CLUSTERING(V) \)
7: \( L \leftarrow \emptyset \)
8: for each cluster \( c \in C \) parallel do
9: \( P \leftarrow \emptyset \)
10: for each grid-point \( p \in c \) do
11: \( P \leftarrow MERGE-LISTS(P, p.partition) \)
12: end for
13: for each grid-point \( p \in c \) parallel do
14: \( L' \leftarrow COMPUTE-RP-INTEGRAL(p, P, \tau, D) \)
15: \( L \leftarrow MERGE-LISTS(L, L') \)
16: end for
17: end for
18: for each \( ([a, b], p) \in L \) parallel do
19: \((I, \epsilon, P, A) \leftarrow RP-ADAPTIVEQUADRATURE(([a, b], p), \tau, D)\)
20: \( p.access_pattern \leftarrow MERGE-LISTS(p.access_pattern, A) \)
21: \( p.partition \leftarrow MERGE-LISTS(p.partition, P) \)
22: \( p.I \leftarrow p.I + I \)
23: \( p.\epsilon \leftarrow p.\epsilon + \epsilon \)
24: end for
25: \( g \leftarrow ONLINE-LEARNING(V, g) \)
computations that exhibit similar data access pattern between each other are mapped to parallel threads with one-to-one correspondence, they exhibit strong inter-thread locality. Such data locality among threads can be exploited by grouping them into one or more thread blocks in GPU architectures, where the memory performance is improved due to the benefit from data locality by using L1-cache or shared-memory. Note that, RP-CLUSTERING procedure in Algorithm-1 is similar to the one used in [10], however, [10] uses heuristics to measure the data reuse between two grid points, whereas in Algorithm-1, we use a more accurate measure of data reuse between two points by comparing their corresponding rp-integral computations access patterns. Even though we measure data reuse using predicted access patterns which are just an approximation to the observed data access patterns, experimental results in Section-V shows that this approach is effective in improving the memory performance.

The for loop in lines 8-17 evaluates rp-integral at all grid points using the partition approximated in line-4. First, for each cluster \( c \in C \), line-9 initializes a list \( P \), and for each grid-point \( p \in c \), the for loop in line 10 merges the list \( p.partition \) with \( P \) by calling an auxiliary procedure MERGE-LIST. The procedure MERGE-LIST\((P, P')\) returns the sorted list that is the merge of its two sorted input lists \( P \), and \( P' \) with duplicate values removed. In other words, lines 10-12 combine the predicted partition of all the points \( p \in c \) into a single unique partition \( P \), such that the combined partition is an approximation to individual partitions. The main objective behind combining the partitions is to have uniform control-flow in the for loop at line-13, which aids in minimizing the thread divergence when computations of this loop are mapped to GPU threads.

The procedure \( \text{COMPUTE-RP-INTEGRAL}(p, P, \tau, D) \) (see Listing-1) approximates rp-integral at a grid-point \( p \) using only the subregions from a partition list \( P \) where the rp-integral error estimate is less than \( \tau \), and the integral and error estimates along each subregion is approximated using Simpson’s quadrature rule. We use an auxiliary procedure \( \text{RP-QUADRULE}([a, b], p, D) \) to compute Simpson’s quadrature rule estimates, \((I, \varepsilon)\), along a subregion \([a, b]\) for rp-integral at a point \( p \), where \( I \) is the integral estimate and \( \varepsilon \) is the error estimate. We omit the pseudocode for RP-QUADRULE, as it is identical to the standard Simpson’s quadrature rule with the inner integral approximated using Newton-Cotes formulae, as illustrated in [9]. In \( i^{th} \) iteration of the for loop in \( \text{COMPUTE-RP-INTEGRAL} \) procedure, integral and error estimates along an integration region \([P[i], P[i+1]]\) is calculated by calling \( \text{RP-QUADRULE} \). Next, when the error estimate returned from \( \text{RP-QUADRULE} \) is less than \( \tau \), both integral and error estimates are accumulated to the output grid-point’s global estimates, \( p.I \) and \( p.\varepsilon \), respectively. Otherwise, the grid-point object and the corresponding subregion where the error estimate is larger than \( \tau \) is inserted to a list \( L \). Once the for loop terminates, partition list used to compute the integral is stored in \( p.partition \), data access pattern observed during the computation is stored in \( p.access_pattern \), and the list \( L \) is returned as the output from \( \text{COMPUTE-RP-INTEGRAL} \) procedure.

In the pseudocode for \( \text{COMPUTE-POTENTIALS} \), the method \( \text{COMPUTE-RP-INTEGRAL} \) is called on for each grid-point \( p \in c \) inside the for loop at line-13, and it returns a list \( L' \). Each element of this list is a pair \( ([a, b], p) \) where \([a, b]\) denotes a subregion such that the Simpson’s quadrature rule error estimate for rp-integral at a grid-point \( p \) along that subregion is larger than \( \tau \). Furthermore, individual list from each iteration of the for loop is merged to a global list \( L \) using the auxiliary procedure MERGE-LIST. The accumulated subregions and grid points from the global list are processed using traditional adaptive quadrature algorithm in lines 18-24. We use the procedure \( \text{RP-ADAPTIVEQUADRATURE} \) to implement Simpson’s adaptive quadrature algorithm, where, in addition to integral and error estimates, our implementation returns the integral partition along outer dimension and the data access pattern observed during the algorithm execution. In particular, \( \text{RP-ADAPTIVEQUADRATURE}([a, b], p, \tau, D) \) outputs a tuple, \((I, \varepsilon, P, A)\), where \( I \) and \( \varepsilon \) are the integral and error estimates, respectively. \( P \) is the partition along the outer dimension generated by adaptive quadrature’s control-flow, and \( A \) is the observed data access pattern. Next, access pattern and partition returned from \( \text{RP-ADAPTIVEQUADRATURE} \) is merged with the corresponding grid-point’s access pattern and partition, respectively, that is seen during \( \text{COMPUTE-RP-INTEGRAL} \) procedure. Furthermore, rp-integral estimates from adaptive quadrature are accumulated to the grid-point’s global estimates.
Next, in the procedure **ONLINE-LEARNING**, access patterns observed during rp-integral computations at all grid points \( p \in V \) is used by a supervised learning algorithm to train and update the predictor function \( g \). The updated prediction function \( g \) is used by **COMPUTE-POTENTIALS** procedure during the next time step.

A. Parallel Implementation

Initialization steps between lines 1-5 are implemented on the CPU, where the for loop is parallelized using OpenMP, and the grid-point attributes: \( \text{access\_pattern} \) and \( \text{partition} \), are stored only in CPU memory. The procedure **RP-CLUSTERING** implementing the \( k\text{-means} \) clustering algorithm is also implemented on CPU, using scikit-learn library [22]. Further, in the implementation of RP-CLUSTERING, we choose the number of clusters to be \( m = \max(N_X, N_Y) \), and since \( k\text{-means} \) algorithm prefers clusters of approximates similar size, each cluster size is approximately \( \min(N_X, N_Y) \).

Lines 8-17 is the heart of beam dynamics simulation that approximates the rp-integral at all grids points using the predicted partitions, and it is implemented on GPUs. In particular, computations of the for loop at line-8 is assigned to one or more thread blocks, where multiple thread blocks are used to take advantage of the Thread Level Parallelism (TLP). Within each thread-block, the computation of for loop at line-13 is assigned to GPU threads with one-to-one correspondence. In other words, each cluster \( c \in C \) is assigned to one or more thread blocks where the computation of each grid-point \( p \in c \) is assigned to GPU threads of the corresponding thread-block. The required number of threads for each block depends on the number of grid points in the cluster assigned to that particular block, and as a result, number of threads per block for GPU execution is chosen to be the maximum of all cluster sizes. Each thread implements the **COMPUTE-RP-INTEGRAL** procedure for the assigned grid-point and the data access patterns observed during the evaluation of this procedure is updated on CPU based on the partition used within the procedure. Further, TLP for the GPU execution is governed by assigning multiple thread blocks to each cluster’s rp-integral computation, such that the loop iteration in **COMPUTE-RP-INTEGRAL** procedure is shared between multiple thread blocks. It is important to note that the flow of computation in for loop at line-13 is uniform between different threads of a thread-block, and as a result, it eliminates the thread divergences between rp-integral computations at different grid-points assigned to the threads of the block.

Next, the computations in lines 18-24 is also implemented on GPU using a different kernel from the one explain before. In this kernel, the list elements are mapped to parallel GPU threads with one-to-one correspondence. Each parallel threads implements the **RP-ADAPTIVEQUADRATURE** procedure in parallel and independent of other threads. This implementation is identical to the globally adaptive algorithm illustrated in [9]. Finally, **ONLINE-LEARNING** procedure to update the prediction model using supervised learning is implemented on CPU using scikit-learn and OpenMP, where both the libraries use all the available CPU cores to speed up the training process.

V. Performance/Experimental Results

Simulation experiments for studying beam dynamics and performance analysis of the parallel implementation of compute retarded potentials stage of the simulation is carried out on NVIDIA Tesla K40 GPU with global memory accesses configured to be cached in both L1 and L2 (commonly called the Caching mode). Initial distribution for all the simulations are generated by Monte Carlo sampling of \( N \) particles with a total charge of beam bunch \( Q = 1\text{nC} \), and the rp-integral at all grid points are approximated to a error tolerance of \( \tau = 10^{-6} \). The performance metrics for all the GPU kernels illustrated in this section are measured using NVIDIA profiler, and the results are averaged over multiple runs. In this section, for convenience, we shall refer to the kernel implementing GPU specific code from **COMPUTE-POTENTIALS** procedure as **Predictive-RP Kernel**, and the GPU implementations from [9] and [10] as **Two-Phase-RP Kernel** and **Heuristic-RP Kernel**, respectively.

A. Validation

The correctness and accuracy of beam dynamics simulation depends on the fidelity of **COMPUTE-POTENTIALS** algorithm that use predictive analytics and forecasting techniques to calculate the retarded potentials in a multistep beam dynamics simulation. To ensure that such prediction based algorithm does not trade simulation’s correctness for performance, it is imperative to validate the parallel implementation and its effect on the simulation. We validate the parallel simulation by comparing the simulation output to the only special case for which the exact analytical results are available - that of a 1D monochromatic rigid bunch. Exact analytical solutions for the longitudinal and transverse force for a 1D rigid-line bunch study state model is given in [24], [25]. The parallel implementation presented here is benchmarked against the analytical results described in [24], [25] for the parameters of

![Fig. 2: Analytic versus computed effective longitudinal (left) and transverse (right) forces for the LCLS bend [15]; \( N = 1000000 \) particles on a \( 128 \times 128 \) grid, bend radius \( R_0 = 25.13 \text{ m} \), \( \theta_0 = 11.4^\circ \), longitudinal rms beam size \( \sigma_x = 50 \mu\text{m} \), emittance \( \epsilon = 1 \text{ nm} \), and total beam charge of \( Q = 1\text{nC} \).](image-url)
the LCLS bend [15]: bend radius \( R_0 = 25.13 \text{ m} \), \( \theta_b = 11.4^\circ \), longitudinal rms beam size \( \sigma_s = 50 \mu\text{m} \), emittance \( \epsilon = 1 \text{ nm} \), total beam charge \( Q = 1 \text{ nC} \). From Figure-2, it is evident that both longitudinal and transverse forces computed with our parallel algorithm agree perfectly with the exact analytical solution.

Further, a closer look into the nature of convergence of the computed forces to the analytic result is shown in Figure-3, which shows the mean-square error, defined as

\[
\epsilon = \frac{1}{N} \sum_{i=1}^{N} (F_i - F_i^{\text{exact}})^2,
\]

with \( N \) the number of particles, \( F_i \) the computed force and \( F_i^{\text{exact}} \) the analytic force on individual particles. As one should expect from Monte-Carlo type simulations, the accuracy of the computed forces, as measured by the mean-square error, scales as \( 1/N \) – inversely with the number of particles in the simulation [12].

### B. Performance Results

Table-I illustrates the double-precision floating-point performance of Predictive-RP kernel compared against the Heuristic-RP kernel for a beam dynamics simulation with 100000 particles and varying grid resolution. The results from Table-I are used to provide a quantitative analysis on the effects of using predictive analytics and forecasting techniques in improving the performance of beam dynamics simulations on GPUs.

1) Analysis Using the Roofline Model: Figure-4 shows the Roofline model for K40 GPU. The graph is on a log-log scale. The \( y \)-axis is attainable double-precision floating-point performance in units of GFlops/sec, and the \( x \)-axis is arithmetic intensity, varying from 0.125 Flops/DRAM byte-accessed to 32 Flops/DRAM byte-accessed. The system being modeled has a peak double precision floating-point performance of 1.4 Tflops/sec and peak memory bandwidth of \( BW_{\text{Theoretical-Peak}} = 288 \text{ GB/sec} \) from hardware specifications. However, peak memory bandwidth is often unachievable in practice. So, in order to analyze the performance more accurately, we measure the experimental memory bandwidth using the benchmarks from NVIDIA’s official SDK [26].

![Fig. 3: Mean-square error for the longitudinal force, as defined in the text, as a function of the number of particles per cell \( N_{ppc} = N/N_{\text{grid}} \), for a fixed grid of \( 128 \times 128 \) (or \( N_{\text{grid}} = 128^2 \).](image)

![Fig. 4: Roofline model analysis for Predictive-RP kernel (red line) compared against Two-Phase-RP kernel (grey line) and Heuristic-RP kernel (green line) on NVIDIA Tesla K40 GPU.](image)

<table>
<thead>
<tr>
<th>Grid Resolution ((N_X \times N_Y))</th>
<th>HEURISTIC-RP KERNEL</th>
<th>PREDICTIVE-RP KERNEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>(64 \times 64)</td>
<td>401</td>
<td>440</td>
</tr>
<tr>
<td>(128 \times 128)</td>
<td>445</td>
<td>475</td>
</tr>
<tr>
<td>(256 \times 256)</td>
<td>460</td>
<td>475</td>
</tr>
<tr>
<td>(256 \times 256)</td>
<td>485</td>
<td>485</td>
</tr>
<tr>
<td>Double precision performance ((\text{GFlops/sec}))</td>
<td>401</td>
<td>440</td>
</tr>
<tr>
<td>Experimental AI ((\text{Flops/DRAM byte-accessed}))</td>
<td>2.00</td>
<td>2.20</td>
</tr>
<tr>
<td>Warp Execution Efficiency</td>
<td>92%</td>
<td>96%</td>
</tr>
<tr>
<td>Global Load Efficiency</td>
<td>105%</td>
<td>115%</td>
</tr>
<tr>
<td>L1-cache Global Hit Rate</td>
<td>100%</td>
<td>99%</td>
</tr>
</tbody>
</table>

**TABLE I: Performance comparison of HEURISTIC-RP KERNEL and the new PREDICTIVE-RP KERNEL for computing the collective effects in a beam dynamics simulation with 100000 particles on NVIDIA Tesla K40 GPU.**
that particular GPU kernel. It is clear from Figure-4 that the parallel algorithm based on predictive analytics and forecasting technique has sufficiently high arithmetic-intensity when compared to the Two-Phase-RP and Heuristic-RP kernel. In particular, Predictive-RP kernel delivers up to 485 Gflops/sec of double precision performance on K40 GPU and achieves a experimental arithmetic intensity of up to 2.43 Flops/DRAM-byte accessed. The increase in arithmetic intensity indicates the effectiveness of the parallel implementation in utilizing the caches to filter the number of accesses that go to memory, thereby increasing the arithmetic intensity.

2) Branch divergence: The effect of branches on the execution efficiency is analyzed from the profiler metric, warp execution efficiency, which is the ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor. Values of less than 100% indicate the presence of threads with different control-flow paths which leads to performance bottlenecks on GPU architectures. In Predictive-RP kernel, warp execution efficiency is nearly 100%, illustrated in Table-I. This indicates that the GPU kernel has fewer divergent branches and has near uniform control-flow. In other words, use of anticipation strategies are effective in reducing the control-flow irregularity among parallel threads, which, as illustrated in [8], is one of the most important performance consideration in programming CUDA-capable GPU architectures.

3) Memory Performance: The following analysis about the memory performance of Predictive-RP kernel is inferred from Table-I -

1) Global load efficiency for the Predictive-RP kernel, which is the ratio of number of bytes requested by the kernel to number of bytes transferred, is greater than 100. Typically, efficiency of 100% indicates perfect coalescing, and values larger than 100% shows that, on average, the load requests of multiple threads in a warp are fetched from the same memory address and are also coalesced.

2) The L1-cache hit rate for global loads is nearly 100%, which indicates elevated data reuse between cache-sharing threads groups. Further, increased cache hit from the kernel reduces the DRAM bandwidth, which contributes to the increase in effective bandwidth and in subsequent increase of arithmetic intensity.

It is evident from the above observations that memory performance of Predictive-RP kernel is improved when compared to the Heuristic-RP kernel. In particular, computation to thread mapping based on the data access pattern forecast is effective in maximizing the data reuse within all cache-sharing thread groups. This leads to an improved cache performance and aids in reducing the impact of any unforeseen memory access irregularity.

4) Speedup: Table-II illustrates the performance of compute retarded potentials stage of the simulation using Predictive-RP kernel compared against the Heuristic-RP kernel for different simulation configurations on NVIDIA Tesla K40 GPU.

<table>
<thead>
<tr>
<th>Number of Particles (N)</th>
<th>Grid Resolution (N_x × N_y)</th>
<th>HEURISTIC-RP KERNEL</th>
<th>PREDICTIVE-RP KERNEL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GPU Time (sec.)</td>
<td>Overall Time (sec.)</td>
<td>Clustering Time (sec.)</td>
</tr>
<tr>
<td>100000</td>
<td>64 × 64</td>
<td>1.50</td>
<td>1.75</td>
</tr>
<tr>
<td></td>
<td>128 × 128</td>
<td>16.20</td>
<td>139.55</td>
</tr>
<tr>
<td></td>
<td>256 × 256</td>
<td>30.10</td>
<td>4.40</td>
</tr>
<tr>
<td>1000000</td>
<td>64 × 64</td>
<td>1.20</td>
<td>1.40</td>
</tr>
<tr>
<td></td>
<td>128 × 128</td>
<td>10.15</td>
<td>9.20</td>
</tr>
<tr>
<td></td>
<td>256 × 256</td>
<td>75.35</td>
<td>71.50</td>
</tr>
</tbody>
</table>

TABLE II: Performance of compute retarded potentials stage of the simulation using Predictive-RP kernel compared against the Heuristic-RP kernel for different simulation configurations on NVIDIA Tesla K40 GPU.

VI. CONCLUSION

We presented a novel machine learning approach to address the computational challenges in efficient parallel simulation of beam dynamics on GPUs. We primarily focused on optimizing the irregular algorithm to compute collective effects at each time step of the simulation which exhibit control-flow and memory access patterns that are not readily amenable to GPU’s architecture. The machine learning approach presented here relies on supervised learning algorithms to adaptively model and track irregular access patterns observed during the computation of collective effects at each time step of the simulation.
to anticipate the future control-flow and data access patterns. Further, we demonstrated that the access pattern forecast from anticipation strategies can be successfully used to formulate runtime decisions that minimize both branch and memory divergence on GPUs, thereby reducing the control-flow and memory access irregularities in the parallel implementation for computing the collective effects. This parallel implementation which use predictive analytics and forecasting techniques outperforms the fastest known GPU implementation and achieves a speedup gain of up to 2.5X and delivers up to 485 Gflops of double precision performance on NVIDIA Tesla K40 GPU.

We also quantified the impact of using machine learning approach in resolving the computational challenges in the parallel implementation of beam dynamics simulation on GPUs. Further, we presented a detailed performance comparison of this new algorithm against the only two published parallel algorithms for high-fidelity computation of collective effects on GPUs: Two-Phase-RP and Heuristic-RP algorithm. Two-Phase-RP algorithm is the first high-performance parallel algorithm for beam dynamics simulation that enabled high-fidelity simulation both feasible and computationally tractable. Heuristic-RP algorithm addressed the memory inefficiencies in Two-Phase-RP algorithm, which further provided a substantial boost in the performance. Now, with another factor of 2.5X improvement in speedup compared to the Heuristic-RP algorithm, this new algorithm enables unprecedented efficiency in numerical simulation of all the relevant physics of synchrotron light source and electron-ion particle collider. The newly improved efficiency, coupled with high-fidelity and precision of our earlier implementations, makes the previously inaccessible physics tractable.

ACKNOWLEDGMENT

We acknowledge the support of the National Science Foundation through grant 1535641.

We gratefully acknowledge the support of NVIDIA Corporation with the donation of Tesla K40 GPUs used for this research.


K. A. acknowledges the generous support of the Modeling and Simulation Graduate Research Fellowship Program provided by Old Dominion University during 2013-2016.

REFERENCES