Fall 2017

Efficient Machine Learning Approach for Optimizing Scientific Computing Applications on Emerging HPC Architectures

Kamesh Arumugam Karunanithi
Old Dominion University, a.k.kamesh001@gmail.com

EFFICIENT MACHINE LEARNING APPROACH FOR OPTIMIZING SCIENTIFIC COMPUTING APPLICATIONS ON EMERGING HPC ARCHITECTURES

by

Kamesh Arumugam Karunanithi
B.E. July 2010, Visvesvaraya Technological University, India

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
December 2017

Approved by:

Mohammad Zubair (Co-Director)

Desh Ranjan (Co-Director)

Balša Terzić (Co-Director)

Andrey Chernikov (Member)

ABSTRACT

EFFICIENT MACHINE LEARNING APPROACH FOR OPTIMIZING SCIENTIFIC COMPUTING APPLICATIONS ON EMERGING HPC ARCHITECTURES

Kamesh Arumugam Karunanithi
Old Dominion University, 2017
Co-Directors: Dr. Mohammad Zubair
Dr. Desh Ranjan
Dr. Balša Terzić

Efficient parallel implementations of scientific applications on multi-core CPUs with accelerators such as GPUs and Xeon Phis is challenging. This requires - exploiting the data parallel architecture of the accelerator along with the vector pipelines of modern x86 CPU architectures, load balancing, and efficient memory transfer between different devices. It is relatively easy to meet these requirements for highly-structured scientific applications. In contrast, a number of scientific and engineering applications are unstructured. Getting performance on accelerators for these applications is extremely challenging because many of these applications employ irregular algorithms which exhibit data-dependent control-flow and irregular memory accesses. Furthermore, these applications are often iterative with dependency between steps, and thus making it hard to parallelize across steps. As a result, parallelism in these applications is often limited to a single step. Numerical simulation of charged particles beam dynamics is one such application where the distribution of work and memory access pattern at each time step is irregular. Applications with these properties tend to present significant branch and memory divergence, load imbalance between different processor cores, and poor compute and memory utilization. Prior research on parallelizing such irregular applications have been focused around optimizing the irregular, data-dependent memory accesses and control-flow during a single step of the application independent of the other steps, with the assumption that these patterns are completely unpredictable. We observed that the structure of computation leading to control-flow divergence and irregular memory accesses in one step is similar to that in the next step. It is possible to predict this structure in the current step by observing the computation structure of previous steps.

In this dissertation, we present novel machine learning based optimization techniques to address the parallel implementation challenges of such irregular applications

on different HPC architectures. In particular, we use supervised learning to predict the computation structure and use it to address the control-flow and memory access irregularities in the parallel implementation of such applications on GPUs, Xeon Phis, and heterogeneous architectures composed of multi-core CPUs with GPUs or Xeon Phis. We use numerical simulation of charged particles beam dynamics simulation as a motivating example throughout the dissertation to present our new approach, though they should be equally applicable to a wide range of irregular applications. The machine learning approach presented here use predictive analytics and forecasting techniques to adaptively model and track the irregular memory access pattern at each time step of the simulation to anticipate the future memory access pattern. Access pattern forecasts can then be used to formulate optimization decisions during application execution which improves the performance of the application at a future time step based on the observations from earlier time steps. In heterogeneous architectures, forecasts can also be used to improve the memory performance and resource utilization of all the processing units to deliver a good aggregate performance. We used these optimization techniques and anticipation strategy to design a cache-aware, memory efficient parallel algorithm to address the irregularities in the parallel implementation of charged particles beam dynamics simulation on different HPC architectures. Experimental result using a diverse mix of HPC architectures shows that our approach in using anticipation strategy is effective in maximizing data reuse, ensuring workload balance, minimizing branch and memory divergence, and in improving resource utilization.

ACKNOWLEDGMENTS

The work presented in this dissertation, as well as other work completed during my graduate career, would have not have been possible without the support of the following people -

- I would like to thank **Dr. Mohammad Zubair** who decided to supervise my doctorate degree and share his experience and knowledge of high performance computing. His ability to ask fundamental questions to understand the given problem has helped me to develop a rational thought process.

- I would like to thank **Dr. Desh Ranjan** who decided to co-supervise my doctorate degree despite his many other academic and administrative commitments. His capability of proposing interesting solutions to problems and proving correctness of the proposed solution always inspired and motivated me.

- I would like to thank **Dr. Balša Terzić** for co-supervising my doctorate degree, and for being a constant source of real world scientific problems that have given my dissertation its purpose with immense practical value. His patience and intuitive explanations of the complex computational physics problems have sparked in me a newfound interest in computational sciences.

- I would like to thank **Dr. Andrey Chernikov** for giving precious advice and participating in my final defense committee.

- I would like to thank my parents, sister, and friends for their continued support and understanding when my PhD has led to periods of reduced social interaction and contact.

This work was supported in part by National Science Foundation through grant 1535641, Jefferson Science Associates Project No. 712336 and the U.S. Department of Energy (DOE) Contract No. DE-AC05-06OR23177, and Old Dominion University's Modeling and Simulation Graduate Research Fellowship during 2013-2016. I would also like to acknowledge the support of NVIDIA Corporation for the donation of Tesla K40 GPUs used in this research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Heterogeneous architectures are now becoming ubiquitous in the computing systems ranging from supercomputers to embedded systems. These architectures integrate different types of processing units (PUs) with different hardware and performance characteristics, for example, multi-core CPUs, GPUs, Intel Many Integrated Core (MIC), and FPGAs. These processing units have the potential to improve performance for many scientific applications [38]. At present, a large fraction of Top500 [72] and Green500 [31] supercomputers now use heterogeneous computing architectures. Accelerators, in particular, have played a significant role in the evolution of these architectures. In fact, systems composed of multi-core CPUs and different types of accelerators, like NVIDIA GPUs and Intel MIC (*e.g.* Xeon Phis), are among the most common type of heterogeneous architectures in world's top supercomputers [72]. For instance, three of top-5 supercomputers in the world are heterogeneous architectures with hardware accelerators: Tianhe-2 (Milkyway-2) employs Intel Xeon Phi many-core accelerators [57], Piz Daint [74] and ORNL Titan [29] employ NVIDIA Tesla GPUs. Table 1 gives a breakdown on the hardware specification of different PUs in these three systems along with their theoretical performance in PFlop/s. The presence of accelerators in these systems boost the overall raw performance as well as the price-to-performance and power-to-performance ratios of these systems when compared to the traditional symmetric CPU architectures. With power consumption as one of the major design constrains in todays computing systems, this trend towards heterogeneous architectures comprising of multi-core CPUs with GPUs, Xeon-Phi and other accelerators will continue.

The vastly different architectures and programming models of multi-core CPUs and accelerators, however, present several challenges in achieving good performance. Optimizing overall application performance requires taking into account the individual characteristics of the PUs. For instance, multi-core CPUs use fewer cores, which are typically out-of-order, multi-instruction issue cores which run at high-frequency and use large-sized caches to minimize the latency of a single thread. This makes

	Tianhe-2	Piz Daint	Titan
CPU cores per node	24	12	16
Accelerators per node	3	1	1
Number of nodes	16000	5320	18688
Total system memory (TB)	1024	340	710
Theoretical Peak Performance (PFlop/s)	54.90	25.33	27.11
Power consumption (MW)	17.81	2.27	8.21

Table 1: Specifications of three supercomputers from world's top-5 (as of June 2017).

CPUs more suitable for latency-critical applications. In contrast, GPUs use thousands of cores, which are in-order cores that share their control unit and are designed for handling multiple tasks simultaneously where the memory access latency is typically hidden with calculations instead of big data caches. This makes GPUs more suitable for throughput-oriented applications which are typically expressed as data-parallel computations - the same program is executed on many data elements in parallel. For this reason, conventional architecture specific optimizations techniques alone may not work well in a heterogeneous system and hence, novel techniques are required to realize the potential and promise of heterogeneous computing. In other words, performance on heterogeneous architecture can only be achieved if the application workload is partitioned and mapped to the PUs such that all the PUs are best utilized and combined to deliver good aggregate performance.

Heterogeneous architectures composed of multi-core CPUs and accelerators are most effective in accelerating applications with dense and highly-structured workloads common in many problem domains ranging from graphics applications to molecular dynamics simulation and climate modeling. This is because many of these applications use regular algorithms that operate on structured data like large vectors or matrices, and access them in statically predictable ways which fit well on these architectures where they can exploit the data-parallel, single-instruction multiple data (SIMD) nature of the accelerators and the vectorization support of x86 CPUs to improve performance. In particular, these algorithms exhibit high computational demands, extensive data parallelism, access memory in a streaming fashion, and require little synchronization. These characteristics in an application make them effective in achieving coalesced memory accesses, minimizing thread divergence and synchronization, improving SIMD and vector pipeline utilizations, etc., which are some of

the important factors to achieve good performance on many parallel architectures. Moreover, there exist a plethora of optimization techniques, methods, and languages models to achieve efficient parallelization of regular algorithms [44, 34, 55], and their implementation on GPUs and Intel MIC can be at least an order of magnitude faster than fine-tuned parallel CPU version [24].

However, a number of scientific and engineering applications are unstructured. Getting performance on accelerators for these applications is extremely challenging because many of these applications employ algorithms which exhibit data-dependent control-flow and memory accesses that are not readily amenable to these architectures. Algorithms with these properties are said to be *irregular*, and pose problems for high-performance parallel implementations due to the following characteristics in them -

- Irregular algorithms often demonstrate significant memory access irregularity which leads to severe performance bottlenecks on SIMD architectures [18, 14]. The data-dependent memory accesses in these programs tend to have less spatial locality compared to traditional graphics and regular general-purpose applications.

- Input values in these algorithms determine the program's runtime behavior, which therefore cannot be statically predicted. These properties in the algorithm 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 core.

- Performance of applications on data-parallel, SIMD 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 [23, 59]. Unfortunately, irregular algorithms tend to present both significant branch and memory divergence which leads to severe performance bottlenecks.

Irregular algorithms are the core of many scientific computing applications that arise from several domains of science and engineering. Some of the well-known applications that employ irregular algorithms are charged particles beam dynamics simulation [77, 39], n-body simulations [9], data mining [75], Boolean satisfiability [17],

social networks [37], system modeling [66], compilers [2], meshing [25], and discrete-event simulation [54]. The irregular nature of the underlying algorithms makes these applications difficult to parallelize and more challenging to map to modern parallel architectures. Several efficient implementations for some of these applications using accelerators and other parallel architectures have been published in recent literature, demonstrating that individually most of these architectures are capable of accelerating at least some irregular applications relative to the CPUs [19, 51, 53]. However, developing parallel implementations for these application using collaborative computing on heterogeneous architectures to deliver the best aggregate performance from all the PUs remains a challenge.

1.1 AIM OF THE THESIS

Efficient parallel implementations of scientific applications on multi-core CPUs with accelerators such as GPUs and Xeon Phis is challenging. This requires - exploiting the data parallel architecture of the accelerator along with the vector pipelines of modern x86 CPU architectures (using 128-bit Streaming SIMD Extensions (SSE) or 256-bit Advanced Vector Extensions (AVX)), load balancing, and efficient memory transfer between different devices. It is relatively easy to meet these requirements for highly-structured scientific applications. In contrast, getting good performance on accelerators for unstructured applications that employ irregular algorithms is extremely challenging, thereby making their efficient parallel implementation a daunting task. Furthermore, these applications are often iterative with dependency between steps, and thus making it hard to parallelize across steps. As a result, the parallelism in these applications is often limited to a single step.

Numerical simulation of charged particles beam dynamics is one such irregular application that has gained increased interest in computational 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. This application simulates the time evolution of charged particles in particle accelerator by computing the collective effects (*e.g.* forces from self-interaction, forces from external magnetic fields, and so on) acting on individual particles of a beam for a few hundreds or thousands of time steps where the computation of collective effects at each time step

is irregular. In particular, distribution of work and data in the accurate computation of collective effects at each time step of the simulation is highly unstructured and cannot be characterized a priori, as these quantities are input-dependent and evolve with the computation itself. To obtain high performance in such irregular algorithms is extremely challenging, and to this end, much effort has been devoted in the development of suitable algorithms that enable unprecedented fidelity and precision in the study of collective effects [10]. However, implementing algorithms with such high-accuracy and resolution have proven to be extremely challenging due to the data- and compute-intensive nature of the underlying numerical methods [47, 46, 42, 16, 49]. Consequently, many of the existing algorithms employ a number of approximations and simplifications to reduce the computational load [16, 49]. This improves the performance while sacrificing the accuracy.

Another well-known irregular application is n-body simulation using the Barnes-Hut algorithm which computes the gravitational forces acting on n different celestial objects for a number of time steps where each time step simulates a particular moment in the time evolution of the celestial bodies. Barnes-Hut algorithm hierarchically decomposes the space around the celestial bodies into successively smaller volumes, called grids, and computes summary information for the bodies contained inside each grid, allowing the algorithm to quickly approximate forces that the n bodies induce upon each other. The hierarchical decomposition is recorded in an octree data structure and the force calculations at each step require tree-building and repeated traversal of the unbalanced octree which is highly irregular. Other examples of such irregular scientific applications include - molecular dynamics simulation, finite elements methods, simulation of wave and sound propagation in 3D objects, etc.

Prior research on parallelizing such applications have been focused around optimizing the irregular, data-dependent memory accesses and control-flow during a single step of the application, independent of the other steps, with the assumption that these patterns are completely unpredictable [18, 9, 5]. Multiple analysis of these applications executing irregular workloads for a few hundreds or thousands of steps show that control-flow and data access patterns made by the irregular algorithm follow a loosely similar pattern between steps. In such situation, one effective approach to reduce the irregularities is to analyze the control-flow and data access patterns at each step of the application 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 these applications, 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 make optimization decisions during application execution which improves the performance at a future step based on the observations from the earlier steps.

In this thesis, we aim to use such predictive analytics and forecasting techniques to optimize irregular scientific computing applications like beam dynamics simulation on emerging high-performance computing (HPC) architectures. In particular, we target on attaining the following two optimization goals while developing efficient parallel implementations on GPUs, Xeon Phis and heterogeneous architectures composed of multi-core CPUs with GPUs or Xeon Phis -

- Performance exploitation of individual PUs of heterogeneous systems - Architecture specific optimizations to reduce the control-flow and memory access irregularities while mapping these applications on to the parallel architectures.
- Effective workload partitioning between the hybrid mix of PUs of the underlying heterogeneous architecture to obtain the best aggregate performance.

To optimize the irregularities, we explore the use supervised learning to adaptively model and track irregular access patterns in the irregular algorithm at each step of the application to anticipate the future control-flow and data access patterns. Access pattern forecasts are then used to formulate runtime decisions that optimize the irregular computations at a future step based on the observations from earlier time steps. For example, forecasts can be used to determine computations to thread mapping that maximize data reuse within a cache sharing thread group and minimize thread divergence, improve data prefetching, linearize the irregularities, etc. Most of these runtime decisions improve the performance of the application on each PU independently. In order to improve the performance in the collective computing environment of the heterogeneous architecture, the forecasts are used to create and distribute sub-problems to different PUs of the heterogeneous architecture which maximizes the resource utilization.

Throughout this thesis, we use numerical simulation of charged particles beam dynamics simulation as a motivating example to develop and illustrate all the optimization techniques. However, the techniques presented here are equally applicable

to a wide range of iterative applications that have characteristics similar to that of beam dynamics simulation.

1.2 THESIS CONTRIBUTIONS

The main contributions of this thesis are -

- We present supervised learning based optimization techniques to address the control-flow and memory access irregularities in the parallel implementation of iterative scientific applications on GPU and Intel MIC architectures. The new optimization technique uses predictive analytics and forecasting techniques to adaptively model and track the irregular memory access patterns at each step of the application to anticipate the future memory access patterns. Access pattern forecasts are used to make optimization and prefetch decisions during application execution which improves the performance at a future step based on observations from earlier steps.

- We present optimization techniques that use machine learning algorithms to divide the original problem into multiple smaller sub-problems and then distribute these sub-problems efficiently between different PUs of the underlying heterogeneous architecture such that it improves the memory performance and resource utilization of all the PUs and delivers a good aggregate performance.

- We demonstrate all our optimization techniques from previous bullets using numerical simulation of charged particle beam dynamics simulation which require execution of irregular workloads for multiple time steps. In particular, we present a cache-aware and memory efficient parallel algorithm that use the proposed machine learning based optimization techniques to address the irregularities in the parallel implementation of beam dynamics simulation on heterogeneous architectures composed of GPUs, Xeon Phis and multi-core CPUs.

1.3 THESIS ORGANIZATION

The structure of this thesis as follows:

- Chapter 2 provides a brief overview of several topics that are essential for understanding the problems addressed in this thesis, namely: overview of different

parallel architectures, implementation challenges on GPUs and Xeon Phis, heterogeneous computing challenges, irregular algorithms and its implementation challenges, physical problem of charged particles beam dynamics simulation and related work, and numerical integration algorithms which is one of the core irregular algorithm used in beam dynamics simulations and in many other scientific computing applications.

- Chapter 3 describes the algorithm to numerically simulate charged particles beam dynamics, its limitation on sequential machines, challenges in developing efficient parallel implementation, and a brief survey of previous research in developing parallel algorithms for this problem.

- Chapter 4 presents the memory efficient parallel algorithm that use machine learning based optimization techniques to address the challenges from irregularities in the parallel implementation of charged particles beam dynamics on GPUs. Further, it presents a quantitative analysis of its performance on NVIDIA Tesla K40 GPU.

- Chapter 5 presents the parallel algorithm and its implementation on heterogeneous architectures for the irregular computations in beam dynamics simulation. In addition to addressing the irregularities, the algorithm presented in this chapter extends the machine learning approach from Chapter 4 to optimize the resource utilization of all the PUs of underlying heterogeneous architecture.

- Chapter 6 provides a concluding discussion on the optimization techniques and algorithms presented in this dissertation.

CHAPTER 2

BACKGROUND

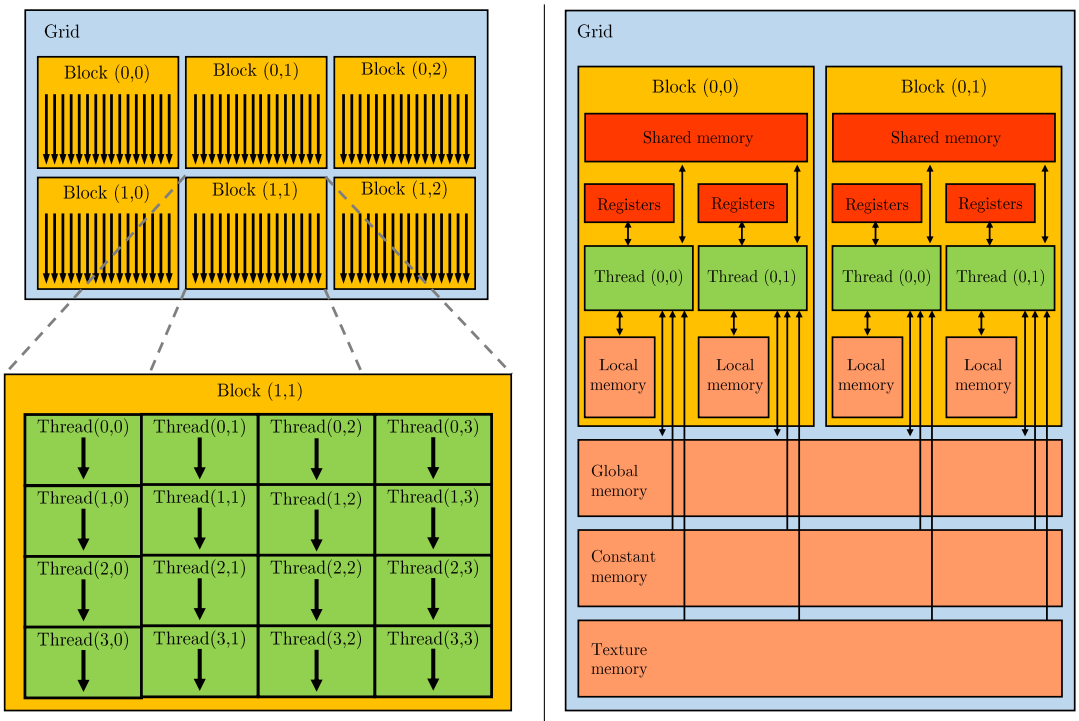
This chapter provides a brief overview of several topics that are essential for understanding the problems addressed in this thesis. In particular, Section 2.1 provides a overview of the parallel architectures used in this study, and Section 2.2 presents the parallel implementation challenges on these architectures. Next, in Section 2.3, we discuss irregular algorithms and its parallel implementation challenges. Section 2.4 presents the physical problem of charged particle beam dynamics simulation and provides a overview of the related work in its numerical simulation methods. Finally, in Section 2.5, we a take brief look at numerical integration algorithms, which is the core irregular algorithm used in beam dynamics simulations and in many other scientific computing applications.

2.1 OVERVIEW OF PARALLEL ARCHITECTURES

In this section, we take a brief look at the typical architecture of two widely used accelerator - NVIDIA's General Purpose GPU and Intel's Xeon Phi coprocessor architecture.

2.1.1 GENERAL PURPOSE GPUS

At the hardware level, NVIDIA's GPU architecture is an scalable array of multithreaded Streaming Multiprocessors (SMs). Each SM features several Streaming Processor (SP) cores and double-precision logic units (DP unit), where each SP core is a fully pipelined integer arithmetic logic unit (ALU) and single-precision floating point unit (FPU). In addition to SP cores and DP units, each SM features (i) load/store units, (ii) special function units (SFU) for transcendental instructions such as sin, cosine, reciprocal, and square root, (iii) schedulers and instruction dispatch units, (iv) instruction cache, (v) register file, (vi) on-chip shared-memory and L1-cache, (vii) read-only cache, and (viii) texture units. The size and number of each unit vary form one generation of GPUs to another. Each core also supports Fused Multiply-Add (FMA) instructions for both single precision and double precision



(a) Grid of thread blocks (b) Memory hierarchy

Figure 1: CUDA programming model for general purpose computing on NVIDIA GPUs.

floating point operations. GPUs also supports larger off-chip global, constant and texture memory that are shared among all SMs. The global/texture memory are often cached and use two-level caching system, where L1-cache is located within each SM, while the L2-cache is located off-chip and is shared among all the SMs.

Compute Unified Device Architecture (CUDA) is the general purpose parallel computing platform and programming model used for designing parallel computations on NVIDIA GPUs. CUDA programming allows the programmer to define functions, called *kernels*, that when called, are executed on the GPUs by many different parallel CUDA threads. The programmer or compiler organizes these CUDA threads into one-dimensional, two-dimensional, or three-dimensional block of threads, called a *thread block* where each thread within a thread block executes an instance of the kernel. The maximum number of threads in a thread block vary from one generation of GPUs to another, for example, a thread block may contain up to 1024 CUDA threads for programming Kepler GPUs. The thread blocks are further organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks

as illustrated by Figure 1a. The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system, which it can greatly exceed. Thread blocks are executed independently which allows it to be scheduled in any order across any number of cores as illustrated in [60], and this enables the programmers to write code that scales with the number of cores.

CUDA threads have access to data from six different memory space during their course of execution: register memory, constant memory, shared memory, texture memory, local memory, and global memory. Figure 1b illustrates how CUDA threads can access data from the different memory spaces. Each thread has private per-thread registers that are often used to hold frequently accessed data, and these are not programmer controlled. Each thread has private local memory that is used for register spills, function calls, and automatic array variables. Each thread block has a per-block shared memory which is visible to all threads of the block and has the same lifetime as the block. Shared memory is often used for inter-thread communication and data sharing in parallel algorithms. The global, constant and texture memory spaces are accessible from all threads and these three memory spaces are persistent across kernel launches by the same application.

When a CUDA kernel is invoked by the host CPU, the blocks of threads that constitute the kernel grid are enumerated and scheduled on the available SMs in any order, concurrently or sequentially, so that the compiled CUDA kernel can execute on any number of SMs. Multiple thread blocks can execute concurrently on a single SM and as the thread blocks terminate, new blocks are launched on the vacated SM. Each SM employs SIMT architecture to manage and execute hundreds of threads concurrently [60]. The SM creates, manages, schedules and executes threads in groups of 32 parallel threads called *warps*. Individual threads composing a warp start their execution at the same program address, but they have their own program counter and register state and are therefore free to branch and execute independently. At any given time, all threads within a warp execute the same instruction in a lockstep. As a result, full warp efficiency is realized when all 32 threads of a warp agree on their execution path, or more formally referred to as control-flow (*warp execution efficiency* is the average percentage of active threads in each executed warp). However, the presence of data dependent conditional branch often cause threads within the same warp to follow different control-flow paths (also known as *branch* or *control-flow divergence*) and this causes the warp to serially execute each branch path taken,

disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. It is important to note that such branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths.

2.1.2 INTEL XEON PHI COPROCESSOR

The Intel Xeon Phi coprocessor is based on the Many Integrated Core (MIC) architecture. This architecture uses previous concepts developed by Intel for the Larrabee many core architecture, as well as the Teraflops Research Chip and the Intel Single-chip Cloud computer. We will discuss one of the Xeon Phi variants, the Knight Corner (KNC). This is the model used in this thesis. The KNC architecture is primarily composed of processing cores (more than 50), caches, memory controllers, PCIe client logic, and a very high bandwidth, bidirectional ring interconnect . Each core is equipped with (i) Vector processing unit (VPU), (ii) scalar processing unit, (iii) Extended Math Unit (EMU) for transcendental instructions such as sin, cosine, reciprocal, and square root, (iv) scalar and vector registers, (v) a L1 cache (data and instruction), and (vi) a unified L2 cache. Caches within a core are fully coherent and implement the x86 memory order model. The L1 and L2 caches provide an aggregate bandwidth that is approximately 15 and 7 times, respectively, faster compared to the aggregate memory bandwidth.

The Xeon Phi is highly optimized for vector processing and it implements SIMD execution model in all VPUs. Each VPU features a 512-bit SIMD instructions, as a replacement for the more commonly found Intel SSE, MMX and AVX instructions. With 512-bit SIMD instructions, VPU provides data parallelism at a very fine grain, working on 512 bits of 16 single-precision floats or 16 integers or 8 double-precision floats at a time. The VPU also supports Fused Multiply-Add (FMA) instructions and hence can execute 32 single-precision or 16 double-precision floating point operations per cycle. A more comprehensive overview of Intel Xeon Phi architecture can be found in [40].

2.2 PARALLEL COMPUTING CHALLENGES

In this section, we first illustrate the impact of data-parallel, SIMD nature of NVIDIA's GPU and Intel's Xeon Phi architecture on application performance. Next, in

Section 2.2.2, we take a brief look at the parallel computing challenges in heterogeneous architectures.

2.2.1 ARCHITECTURAL INFLUENCES ON PERFORMANCE

Effectively exploiting data parallelism using SIMD pipelines is one of the most important aspects in achieving high performance on PUs such as GPU, Intel MIC and modern x86 architectures. For example, Intel MIC has 512-bit VPU units per core to expose SIMD parallelism, GPUs by design employ SIMT architecture within each streaming multiprocessors that execute threads in group of 32 (equivalent to 1024-bit SIMD for single precision floating point), x86 architectures uses 128-bit Streaming SIMD Extensions or 256-bit Advanced Vector Extensions (AVX) to support SIMD parallelism. In these PUs, data parallelism using SIMD pipelines is a power efficient way of boosting the peak performance, and such parallelism is exploited at different granularities: usually, several work items (also called threads) are organized into a work group. These are broken down into several SIMD groups (*e.g.* warps in GPUs) that are executed by a SIMD pipeline. Each PU may have multiple SIMD pipelines to execute SIMD groups in parallel, where effective hardware utilization of the PU relies largely on exploiting the SIMD pipelines.

Programming applications to maximize the utilizations of SIMD pipelines and to deliver high-performance of the application code on the PUs remains a challenge. In particular, the nature of SIMD execution requires that all threads in a SIMD thread group (*e.g.*, a warp in GPUs) to execute the same instruction in lockstep. While this allows the processor design in PUs to be relatively simple, application performance may suffer significantly whenever threads in the same SIMD group behave differently due to control or memory latency divergence [60]. Control divergence results in serialized execution of divergent control paths, leaving execution resources idle and throttling parallelism. Similarly, memory latency divergence causes a SIMD group to stall until the longest memory request for a vector load completes before executing any dependent instructions. In this section, we first briefly overview the effects of control-flow and memory latency divergence on application performance. Then, we look at the SIMD architectural features that affect control-flow and memory latency divergence, and those that try to mitigate them.

Memory Latency Divergence

When a SIMD thread group of size w issues a load instruction (*e.g.*, $w = 32$ threads for GPUs, $w = 8$ double precision loads in Intel Xeon-Phi's 512-bit VPU), the SIMD group will block once an instruction dependent upon the load data becomes the next to issue. As a result, this group of threads is unable to make progress until all the data for the constituent load instructions are available. In particular, for SIMT architectures, a single delinquent load can block forward progress of the SIMD group. This introduces the problem of *memory latency divergence*, where a SIMD group can be stalled until the last memory request from a vector load instruction is returned, potentially long after other memory requests from the vector load have completed. In many workloads, this load latency cannot be hidden by executing other SIMD groups. Several studies have highlighted how memory latency divergence can be a significant performance bottleneck in GPUs and Xeon Phis [52, 68, 22]. This problem of memory latency divergence is not unique to GPUs (or Xeon Phis) and can also manifest itself in other SIMD/vector architectures that support "gather" load operations.

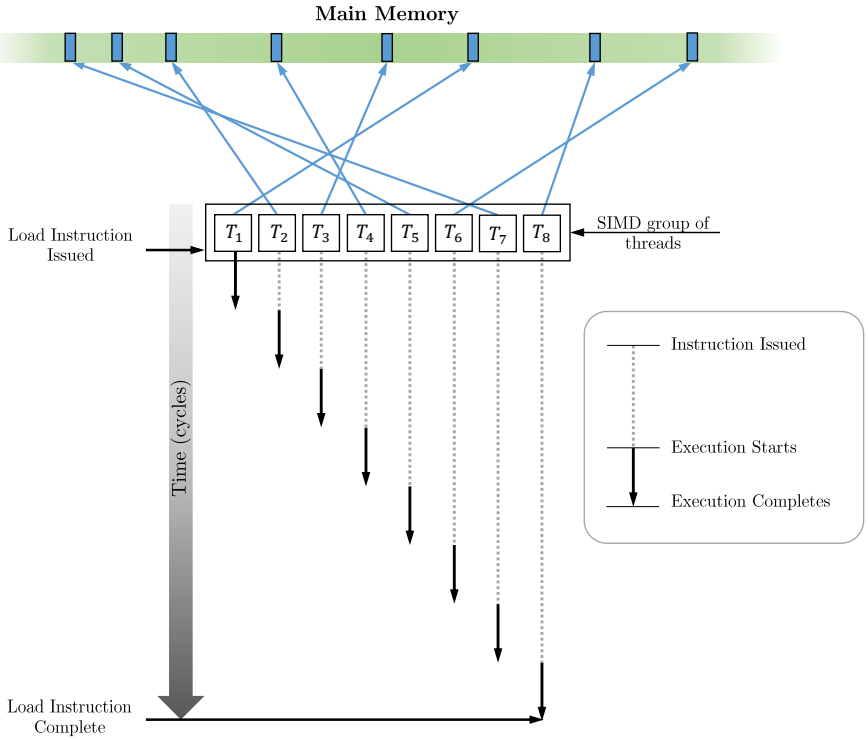


Figure 2: Memory latency divergence when the addresses accessed by SIMD group are scattered in the memory.

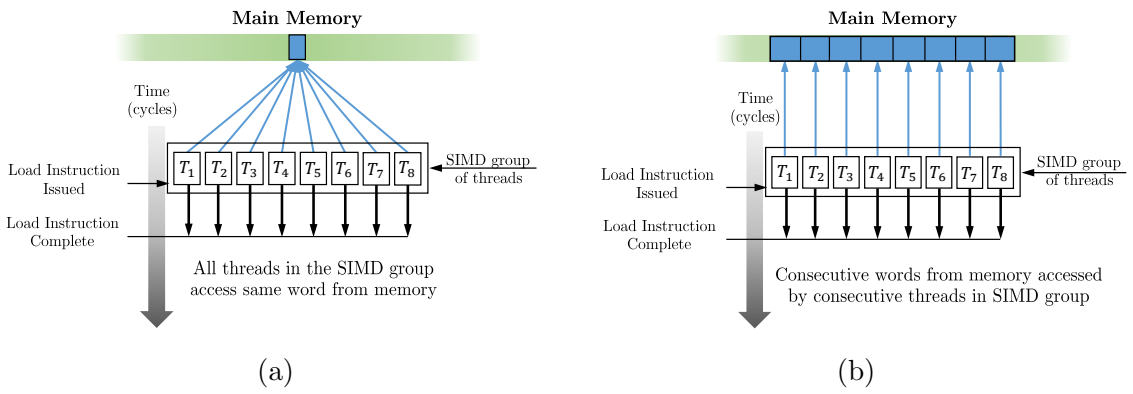
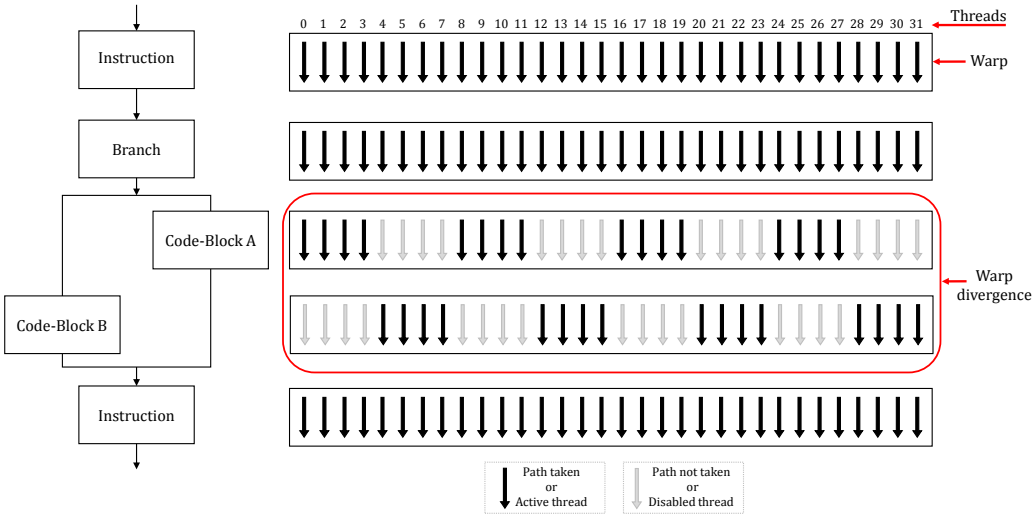


Figure 3: Memory latency divergence when: (a) the addresses accessed by SIMD group are uniform (all equal), (b) the addresses accessed by SIMD group are consecutive.

Memory latency divergence arises from several factors. First, the target data for a set of memory requests from a SIMD group may reside in different levels of memory hierarchy, and the memory hierarchy service different requests at different times due to hits and misses at various levels. Second, the time to complete each memory request depends on several factors, including which DRAM bank the target memory resides in, contention in the interconnect network, and availability of resources in the memory controller. Third, current memory systems are primarily optimized to support traditional, structured workloads with low degree of memory access irregularity. In case of workloads with high degree of irregular or non-coalesced memory accesses, the memory system serializes the execution of memory references from a SIMD thread group, which stalls the SIMD group until the last memory reference is returned. For example, consider a SIMD group of size w that executes a load instruction. In the worst case, all w addresses are scattered in the address space, which results in w data loads from the memory. This serializes the execution of load instructions, which stalls the SIMD group until the last load instruction is returned. Figure 2 illustrates the memory divergence for load instruction within a SIMD group of size $w = 8$ where all the w addresses are scattered in the address space. On the other hand, if the addresses are uniform (*i.e.*, all equal) or consecutive, only a single load has to be issued (see Figure 3a and Figure 3b).

```
__global__ void kernel(){
  ....
  if(condition){
    ....
    Code-Block A
    ....
  }else{
    ....
    Code-Block B
    ....
  }
  ....
}
```

(a) CUDA kernel with $if - else$ construct.



(b) Control-flow for an warp executing the $if - else$ branch condition when half of warps thread evaluate the branch condition to true.

Figure 4: Control-flow divergence in GPUs.

Control-flow Divergence

Consider a SIMD group of threads executing the $if - else$ construct shown in Figure 4a. The threads within the SIMD group works with full efficiency when all the threads execute the if part or all execute the $else$ part. On the other hand, when threads within a SIMD group take different control-flow paths then the execution of the group takes multiple passes through the divergent paths. The first pass executes the threads that follow if part (Code-Block A) and the second pass executes those that follow the $else$ part (Code-Block B). In other words, each thread from the SIMD

group takes both the branch paths sequentially, even though it just executes one of them. This results in poor to suboptimal hardware utilization which often degrades the application performance. For example, Figure 4b illustrates the control-flow in a GPU warp when half of the its thread evaluate the branch condition to true. In the first pass, only half the threads takes the path for *if* condition (denoted by black arrow) and the other half remains disabled (denoted by grey arrow). Similarly, in the second pass, half the threads take the path for *else* condition (denoted by black arrow) and the other half remains disabled (denoted by grey arrow). This reduces the SMID groups or warp execution efficiency to 50%. Such reduction in execution efficiency often leads to poor utilization of the compute resources, which severely degrades the application performance on SIMD architectures [32]. As a result, it is of vital importance to mitigate the control-flow divergence and subsequently increase the execution efficiency of all SIMD thread groups for obtaining good application performance on SIMD architectures.

Mitigating Divergence

The architectural features in GPUs and Intel MIC that affect memory latency divergence (besides the SIMD execution model illustrated in the previous section) and those that try to mitigate divergence are -

- **Memory Coalescing -** A common architectural technique that reduces memory bandwidth demand in SIMD architectures is *memory coalescing*. In this technique, the individual requests from a SIMD group are combined, based on their target address, to form as few cache-line sized (*e.g.* 128B in GPUs) requests as possible. Coalescing is primarily designed to reduce bandwidth requirements by eliminating redundant accesses to the same cache line. However, it also reduces the opportunity for memory latency divergence by minimizing the number of distinct memory requests per SIMD group. Coalescing is not effective, however, if the data accessed by the threads in a SIMD group are not spatially colocated, as is commonly the case in applications that exhibit memory access patterns that are not readily amenable to the SIMD architecture. Previous studies have shown that coalescing is extremely effective for traditional structured workloads with uniform memory access patterns, but its effectiveness falls short for workloads with irregular memory accesses [22].

- **Multithreading -** SIMD architectures leverage thread-level parallelism to hide memory access latency. The effect of long divergence-induced stalls can be mitigated if there are enough ready SIMD thread groups in the system to hide the latency of the slowest request. Previous studies [22, 27, 41] have shown, however, that in spite of having a large number of thread contexts to choose from, a SIMD processor core will frequently sit idle as all the SIMD groups are stalled on pending memory accesses. For instance, recent NVIDIA GPUs support at most 48 to 64 warps within a SP, while the main memory latency have been measured to exceed 400 cycles. It is, therefore, important to note that thread-level parallelism cannot always completely hide main memory latency [8].

- **Caches -** Caches have low latency when compared to the global memory. As a result, memory requests from a SIMD thread group that hit in a cache are returned sooner even with access irregularities or latency divergence when compared to servicing the same requests from global memory. This improves the average memory latency. However, for memory latency divergence to be meaningfully addressed with caches, a substantial fraction of SIMD groups must be able to serve all of their memory requests with cache hits. Otherwise, the cache misses for a SIMD group will be serviced from the global memory, and faces the latency divergence issues described above.

2.2.2 HETEROGENEOUS COMPUTING CHALLENGES

The presence of different type of PUs with different processing capabilities introduces many challenges to the application design. In particular, the level of heterogeneity in these systems introduces non-uniformity in system development, programming practices, and overall system capability. These non-uniformities presents a significant challenge to the programming community in developing applications that can fully exploit the capabilities of the multiple PUs and in approaching their combined theoretical performance. Several studies have been devoted to developing programming models that can exploit these heterogeneous architectures in a unified way, *e.g.*, OpenCL [43], OpenACC [63] and OpenMP [64]. These models simplify the programming effort required in exploiting these architectures, however, achieving good performance on heterogeneous architectures still largely depends on (i)

performance exploitation of each single PU, and (ii) performance aggregation from all the PUs. This requires partitioning the application workload and mapping them efficiently between the PUs such that all the PUs are best utilized and combined to deliver best aggregate performance.

Performance exploitation of the individual PU is achieved using architecture-specific optimization strategies, [60, 59, 80]. Researchers have developed several optimizations techniques and tools on top of the programming models to extract more performance on each type of architecture [70, 69, 79]. On the other hand, vastly different architectures and programming models of different PUs in a heterogeneous system presents several challenges in optimizing applications to deliver good aggregate performance from all the PUs. Several factors relating to both the PU and the application itself, and spanning from microarchitecture-level to system-level need to be taken into account for fully leveraging their potential in heterogeneous computing systems [55]. In particular, designing effective workload partitioning between a hybrid mix of PUs is challenging because -

- Heterogeneity of the platform requires partitioning algorithms to fully consider the PUs differences in processing capability, hardware architecture, and memory model. For example, GPUs and Xeon Phis usually offer higher processing throughput than CPUs (on average 2.5X in [45]), but have separate memory spaces and low data communication bandwidth to the CPU memory (GPUs are connected with the host CPU through a PCI Express bus). Due to the interaction between the PUs in a heterogeneous environment, optimizing performance and energy efficiency requires taking into account the individual characteristics of the PUs. Some of the dominant PU-specific factors that needs to be considered are

  - Current load on PUs and achieving load balancing between them,
  - Memory bandwidth and CPU to accelerator data transfer overhead; avoiding and/or amortizing overhead of launching kernels on accelerators,
  - Overlapping data transfer with computation or CPU computation with computation on accelerators,
  - Taking into account the limitations of PUs, e.g. CPU to GPU/Xeon-Phi memory bandwidth, size of GPU/Xeon-Phi and CPU memory, number

of threads in GPU/Xeon-Phi and CPU cores, reduced performance of GPU/Xeon-Phi for double-precision computations etc. [78]. Additionally, aggressively using CPU for computation affects its ability to act as a host, which harms the performance [30, 73].

- The diversity of platforms, applications, and datasets increases the partitioning complexity [71]. The diversity lies in the fact that heterogeneous platforms exist in different configurations and forms (*e.g.*, multi-core CPUs with accelerators, embedded systems, MPSoC, etc.), and that data parallel applications (with different datasets) present various kinds of performance behavior even on the same platform.

- Application or problem specific factors that influence workload partitioning -

  - Subdividing the workload and selecting suitable work sizes to be allocated to different PUs

  - Accounting for data dependencies, *e.g.* if a task has data dependencies on previous task, where was the previous task executed?

- From programmer's viewpoint, it is challenging to obtain, with limited time and effort, the optimal partitioning that leads to the best performance for a given heterogeneous platform, application, and dataset.

A more comprehensive survey of heterogeneous computing techniques and the challenges involved in heterogeneous computing can be found in [55].

2.3 IRREGULAR ALGORITHMS

The terms *regular* and *irregular* stem from the compiler literature. *Regular algorithms* operate on structured data like large vectors or matrices, and access them in statically predictable ways - *e.g.* dense linear algebra computations, matrix vector multiplications, etc. These algorithms often have high computational demands, exhibit extensive data parallelism, access memory in a streaming fashion, and require little synchronization [50]. This sort of regularity in the algorithm exhibit data independent control-flow and memory references which can be exploited to minimize memory divergence by coalescing memory accesses, and minimize thread divergence and synchronization by maintaining workload balance. In fact, there exists a plethora

of techniques, methods and languages to achieve efficient parallelization of regular algorithms [44, 34], and their implementation on GPUs and Xeon Phis can be at-least an order of magnitude faster than fine tuned parallel CPU version [24].

On the other hand, *irregular algorithms* are characterized by control-flow and memory references that are data dependent. In particular, distribution of work and data in these algorithms cannot be characterized a priori because these quantities are input-dependent and evolve with the computation itself. Irregular algorithms are the core of many high-performance applications, such as n-body simulations [9], data mining [75], Boolean satisfiability [17], social networks [37], system modeling [66], compilers [2], meshing [25], and discrete-event simulation [54]. Efficient parallel implementation of irregular algorithms are challenging because:

- Irregular algorithms often demonstrate significant Memory Access Irregularity (MAI) [18] which leads to severe performance bottlenecks on SIMD architectures [14]. The data dependent memory accesses in these programs tend to have less spatial locality compared to traditional graphics and regular general-purpose applications.

- Input values in these algorithms determine the program's runtime behavior, which therefore cannot be statically predicted. These properties in the algorithm pose problems for high-performance parallel implementations, where equal distribution of work over processors cores and locality of reference are required within each cache sharing processor cores.

- Performance of applications on SIMD 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 [23, 59]. Unfortunately, irregular algorithms tend to present both significant branch and memory divergence which leads to severe performance bottlenecks.

2.4 BEAM DYNAMICS

In this section, we present an overview of charged particle beam dynamics and the general equations that govern the dynamics of charged particles in a *synchrotron* (a type of cyclic particle accelerator).

2.4.1 PHYSICAL PROBLEM

The dynamics of charged particle beams is captured by the Lorentz force [47]:

$$\frac{d}{dt}\left(\gamma m_e \boldsymbol{v}\right) = e\left(\boldsymbol{E} + \boldsymbol{\beta} \times \boldsymbol{B}\right), \tag{1}$$

with the relativistic $\boldsymbol{\beta}$ and γ, velocity $\boldsymbol{v}$, electric field $\boldsymbol{E}$ and magnetic field $\boldsymbol{B}$ specified as, respectively,

$$\boldsymbol{\beta} \equiv \boldsymbol{v}/c, \;\; \gamma = \frac{1}{\sqrt{1+\boldsymbol{\beta}^2}}, \;\; \boldsymbol{v}(\boldsymbol{p}) = \frac{\boldsymbol{p}/m_e}{\sqrt{1+\boldsymbol{p}\cdot\boldsymbol{p}/(m_e c)^2}}, \tag{2a}$$

$$\boldsymbol{E} = -\nabla\phi - \frac{1}{c}\partial_t \boldsymbol{A}, \quad \boldsymbol{B} = \nabla \times \boldsymbol{A}. \tag{2b}$$

ϕ and $\boldsymbol{A}$ the *retarded scalar* and *retarded vector potentials*, respectively. They are obtained by integrating the charge distribution ρ and the charge current density $\boldsymbol{J}$ over the *retarded time* $t' = t - |\boldsymbol{r} - \boldsymbol{r'}|/c$:

$$\begin{bmatrix}\phi(\boldsymbol{r},t)\\ \boldsymbol{A}(\boldsymbol{r},t)\end{bmatrix} = \int_0^\infty \begin{bmatrix}\rho(\boldsymbol{r'}, t-\frac{r-r'}{c})\\ \boldsymbol{J}(\boldsymbol{r'}, t-\frac{r-r'}{c})\end{bmatrix} \frac{d^2\boldsymbol{r'}}{|\boldsymbol{r}-\boldsymbol{r'}|}, \tag{3a}$$

$$\begin{bmatrix}\rho(\boldsymbol{r},t)\\ \boldsymbol{J}(\boldsymbol{r},t)\end{bmatrix} = \int_0^\infty \begin{bmatrix}1\\ \boldsymbol{v}(\boldsymbol{p})\end{bmatrix} f(\boldsymbol{r},\boldsymbol{p},t)d\boldsymbol{p}. \tag{3b}$$

$\boldsymbol{r}$ and $\boldsymbol{p}$ are particle coordinates and momentum, respectively, $f(\boldsymbol{r},\boldsymbol{p},t)$ is the particle distribution function (DF) of the beam in phase space, m_e particle mass, c the speed of light. Both electric and magnetic fields are composed of two components, one due to external fields and the other due to self-fields: $\boldsymbol{E} = \boldsymbol{E}^{\text{ext}} + \boldsymbol{E}^{\text{self}}$, $\boldsymbol{B} = \boldsymbol{B}^{\text{ext}} + \boldsymbol{B}^{\text{self}}$. $\boldsymbol{E}^{\text{ext}}$ and $\boldsymbol{B}^{\text{ext}}$ are external electromagnetic (EM) fields fixed by the accelerator lattice, and $\boldsymbol{E}^{\text{self}}$ and $\boldsymbol{B}^{\text{self}}$ are the EM fields from the beam self-interaction. The beam self-interaction depends on the history of the beam charge distribution ρ and current density $\boldsymbol{J}$ via the retarded potentials ϕ and $\boldsymbol{A}$.

The computation of retarded potentials requires integration over the history of charge distribution and current density, as can be seen from Equation 3a. This is the main computational bottleneck of the beam dynamics simulations. In particular, the problems to overcome in a successful beam dynamics simulation are: (i) data storage for the time-dependent beam quantities (ρ and $\boldsymbol{J}$); (ii) numerical treatment of retardation and singularity in the integral equation for retarded potentials; and

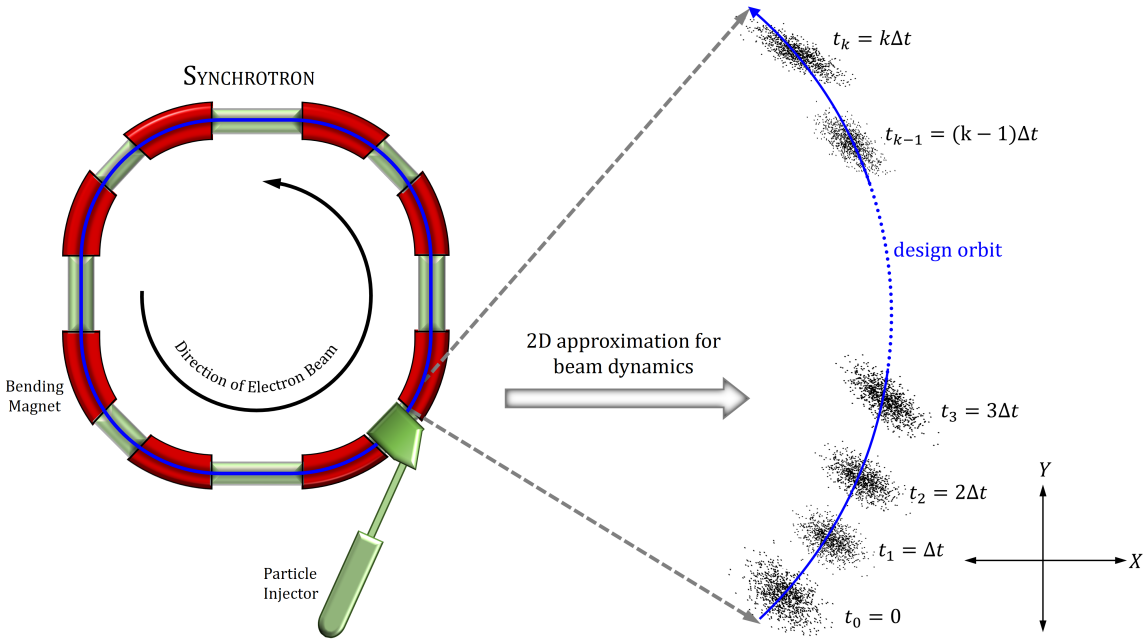


Figure 5: Conceptual view of charged particles beam dynamics on a 2D plane.

(iii) accurate and efficient multi-dimensional integration in the equation for retarded potentials.

Figure 5 and Figure 6 illustrates the conceptual view of charged particle beam dynamics in a *synchrotron* (a type of cyclic particle accelerator) that is approximated on a 2D plane. Figure 5 provides an overall view of the simulation, and Figure 6 illustrates the simulation process in a step-by-step manner. In both these figures, blue line denotes the design orbit along which a beam bunch with N charged particles travel under the influence of a bending force of a magnet. In order to numerically simulate the dynamics of this beam, first the charged particles at time $t_0 = 0$ are generated from Monte Carlo sampling of initial DF of N particles with a total charge of Q. The particles evolve by a small time increment Δt between each time step due to the forces acting on the individual particles. These forces are computed using Vlasov-Maxwell equations that are solved either directly, by sampling the entire phase space of the DF, either on a grid or in an appropriate basis [15], or by using a particle tracking approach [46, 47, 77, 76, 39]. The computational requirements associated with sampling the entire phase space limit the direct solvers to lower dimensions (usually 1D). On the other hand, tracking methods are less restrictive owing to the fact that sampling of the phase space is done only through simulation particles. This

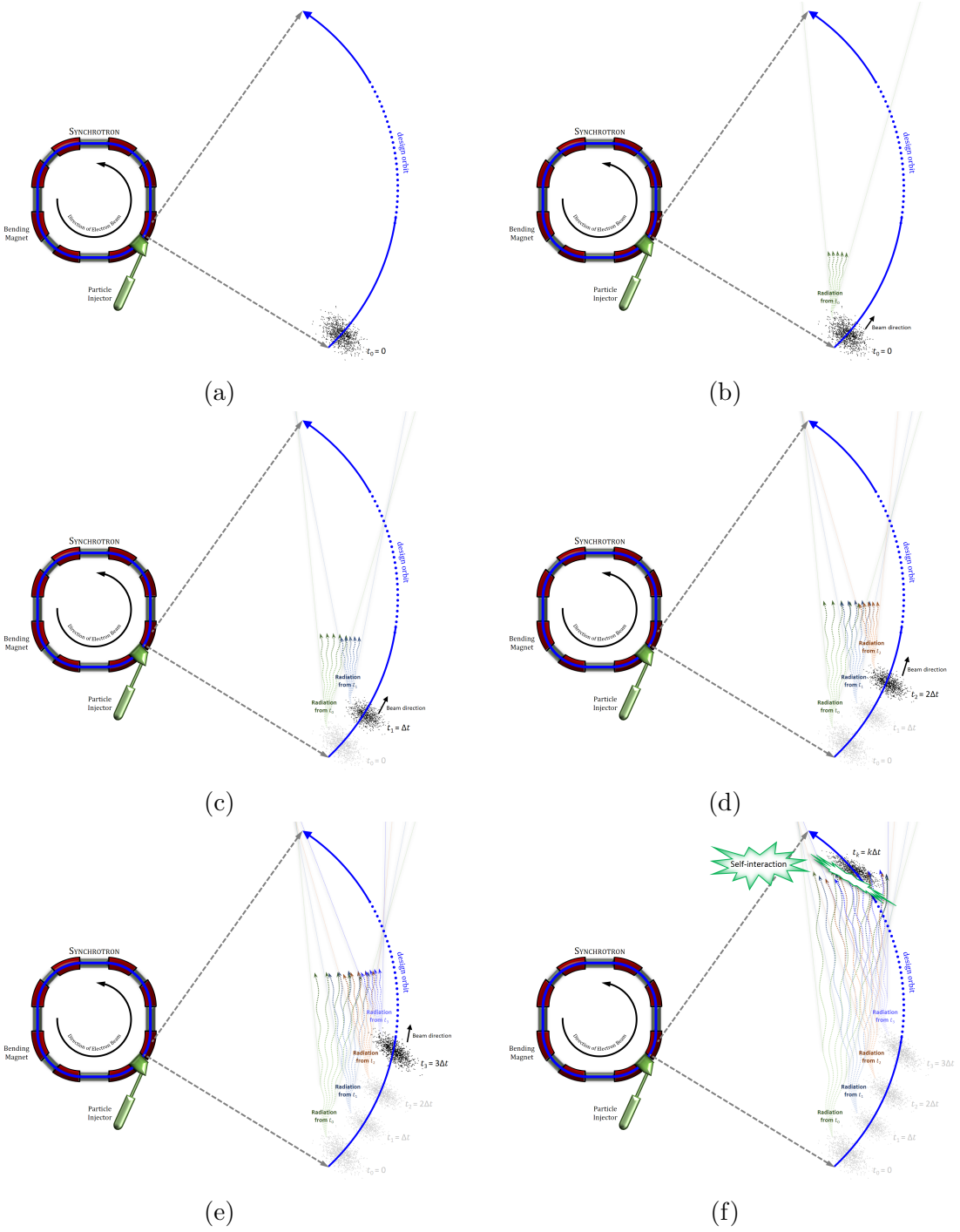


Figure 6: Simulation of charged particles beam dynamics. (a) Particle distribution for charged particle beam at time t_0 is generated from Monte Carlo sampling of initial DF of N particles with a total charge of Q. (b) - (e) The charged particles in the beam emits synchrotron radiation when forced to travel along a curved trajectory under the influence of a bending force of a accelerator magnets (referred to as *Bending Magnet* in the above figure). (f) The radiations emitted from all the earlier time steps catch up with the charged particle beam at time $t_k = k\Delta t$ for all integers $k > 0$, and these synchrotron radiations leads to hazardous self-interactions.

allows the study in higher-dimensional systems, which gives them a clear advantage and makes them a preferred method for modeling collective effects in beam dynamics simulations. The most dominant of the particle tracking approaches is the particle-in-cell method [77, 76, 39], where the individual particles are tracked in continuous phase space, whereas moments of the distribution such as densities and currents are computed simultaneously on discrete spatial grid points. The continuous phase space in the plane of the beam lattice is referred as *Lab Frame (LF)*($XY-$plane in Figure 5).

Furthermore, at each time step, the particle distribution experiences the effects of coherent synchrotron radiation (CSR) and other collective effects. Figure 6 illustrates the collective effects on the particles due to beam's self-interaction. In particular, when a charged particle beam travels along a curved trajectory under the influence of a bending force of an accelerator magnets, it emits synchrotron radiations, as illustrated in Figure 6b - 6e. The radiations emitted from all the earlier time steps catch up with the charged particle beam at time t_k for all integers $k > 0$, and these synchrotron radiations leads to hazardous self-interactions. This self-interaction causes significant emittance degradation, as well as fragmentation and microbunching within the beam bunch, rendering the beam useless for physics research.

2.4.2 RELATED WORK

Present beam dynamics simulation methods employ a number of approximation in the study of collective (including CSR) effects. For example, the beam dynamics calculation in *elegant* [16] is based on the analysis of bunch self-interaction for a rigid-line bunch. This method is widely used for accelerator design and is the first to reveal CSR-induced microbunching in bunch compressors. However, in the regime of extreme bunch compression when the bunch deflection is appreciable, the 1D approximation used in *elegant* may not be appropriate [49]. The earliest 2D beam dynamics simulation is TraFiC^4 [42]. Here electro-magnetic (EM) fields are generated from the source particles moving along prescribed orbit, and CSR effects are calculated from the impact of these EM fields on the dynamics of test particles. An early self-consistent beam dynamics simulation was developed by [46, 47]. This code calculates direct interaction between microparticles, with the retarded potentials obtained by integrating bunch distribution over retarded times. However, the computation efficiency for this code is severely limited by the direct particle-particle

interaction employed in the model. Recently, Bassi *et al.* [11] have developed a highly efficient, high-resolution 2D self-consistent code for simulation of the CSR effects. This simulation has generated interesting results on CSR-induced microbunching in bunch compressors. Currently, the code assumes linear optics, so the effects caused by nonlinear optics are not included. Self-consistent CSR simulations based on finite element method was pioneered by Agoh and Yokoya [1]. This method can include boundary effect by chamber walls much easier than the Greens function approach. More comprehensive review of the status of beam dynamics simulation can be found in the review article by Bassi *et al.* [10].

2.5 NUMERICAL INTEGRATION

Numerical integration (also called as *quadrature*) constitutes a broad family of algorithms for calculating the numerical value of a definite integral. In this section, we describe the two widely used quadrature algorithms to calculate one-dimensional definite integrals: Newton-Cotes formulas and Adaptive quadrature. Then, we extend the description to definite integrals for higher dimensions (commonly called as *multiple or multi-dimensional integrals*).

2.5.1 NEWTON-COTES FORMULAS

Newton-Cotes formulae are a family of numerical integration techniques where the value of a definite integral is approximated as a weighted sum of the integrand values at equally spaced points. More formally, given a function or integrand $f(x)$, Newton-Cotes formula of degree n that approximates the definite integral of $f(x)$ over an interval $[a, b]$ is stated as

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n} w_i f(x_i) \tag{4}$$

where $x_0 = a$, $x_n = b$, and $x_i = x_0 + ih$ with step size $h = \frac{x_n - x_0}{n} = \frac{b-a}{n}$. The weights w_i are derived from Lagrange basis polynomials [21, p. 500], and it depends only on the degree n of the Newton-Cotes formula.

Table 2 lists some of the widely used Newton-Cotes formulae of varying degree. The notation f_i is a shorthand for $f(x_i)$, and the number ξ in error term is between a and b. The exponent of step size h in the error term shows the rate at which the approximation error decreases, and the derivative of f in the error term shows

Degree (n)	Common name	Formula	Error term
1	Trapezoid rule	$\frac{1}{2}h(f_0 + f_1)$	$-\frac{1}{12}h^3 f^{(2)}(\xi)$
2	Simpson's rule	$\frac{1}{3}h(f_0 + 4f_1 + f_2)$	$-\frac{1}{90}h^5 f^{(4)}(\xi)$
3	Simpson's 3/8 rule	$\frac{3}{8}h(f_0 + 3f_1 + 3f_2 + f_3)$	$-\frac{3}{80}h^5 f^{(4)}(\xi)$
4	Boole's rule	$\frac{2}{4}h(7f_0 + 32f_1 + 12f_2 + 32f_3 + 7f_4)$	$-\frac{8}{945}h^7 f^{(6)}(\xi)$

Table 2: Newton-Cotes formulae for different degrees.

which polynomials can be integrated exactly (*i.e.*, with error equal to zero). For the Newton-Cotes rules to be accurate, the step size h needs to be small, which means the domain of integration $[a, b]$ must be small itself, which is not true most of the time. One way to improve the accuracy is to partition the integration interval from a to b into a number of subintervals and apply Newton-Cotes rule on each subinterval, where the integral and error estimate from each subinterval is accumulated to give the final estimates for the original integral. The resulting set of equation is called as *composite integration rule*. For example, the composite integration rule using Trapezoid rule is stated as

$$\int_a^b f(x)dx \approx \frac{h}{2}\left[f(a) + 2\sum_{k=1}^{m-1} f\left(a + kh\right) + f(b)\right] \tag{5}$$

where m is the number of subintervals, and subintervals have the form $[kh, (k + 1)h]$, with $h = \frac{(b-a)}{m}$ and $k = 0, 1, 2, \ldots, m - 1$. The overall error estimate for composite trapezoidal rule can be obtained by accumulating the individual errors for each subinterval. A more comprehensive survey of Newton-Cotes formulae and their corresponding composite rules can be found in [21, p. 601].

2.5.2 ADAPTIVE QUADRATURES

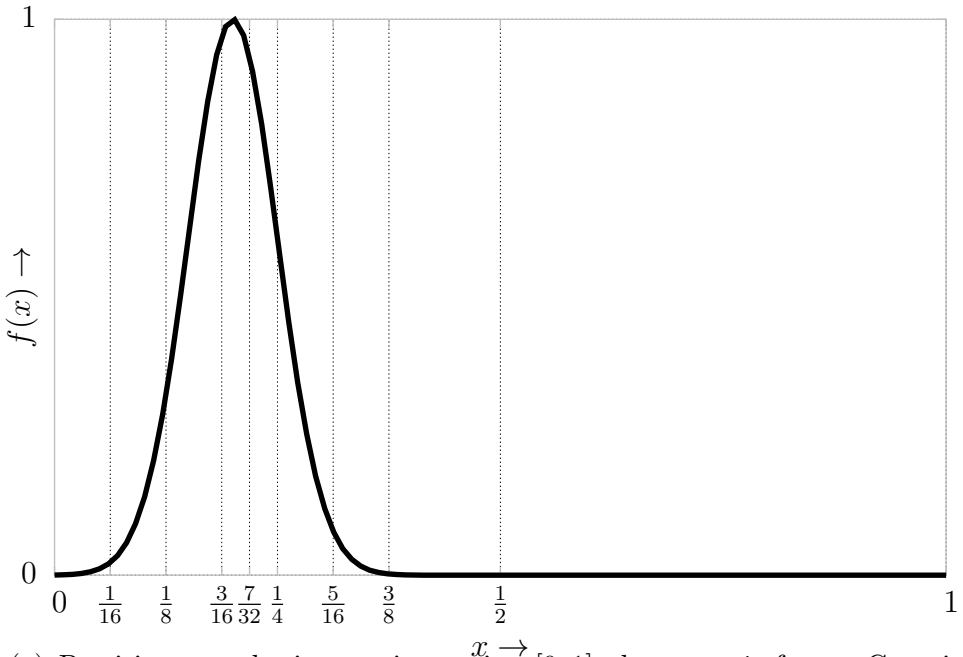
Newton-Cotes formulae and their corresponding family of composite rules are based on evaluating the integrand at evenly spaced points. This global perspective ignores the fact that many functions have regions of high variability along with other sections where change is gradual. As a result, numerical integration using

Newton-Cotes formulae are accurate only when the integrand f is smooth (*i.e.*, if it is sufficiently differentiable or it features only subintervals where change is gradual). On the other hand, when the integrand is highly oscillatory or it lacks derivatives at certain points, then integration methods such as *adaptive quadrature* are used. In adaptive quadrature, step sizes are adaptively adjusted so that small intervals are used in regions of rapid variations and larger intervals are used where the function changes gradually.

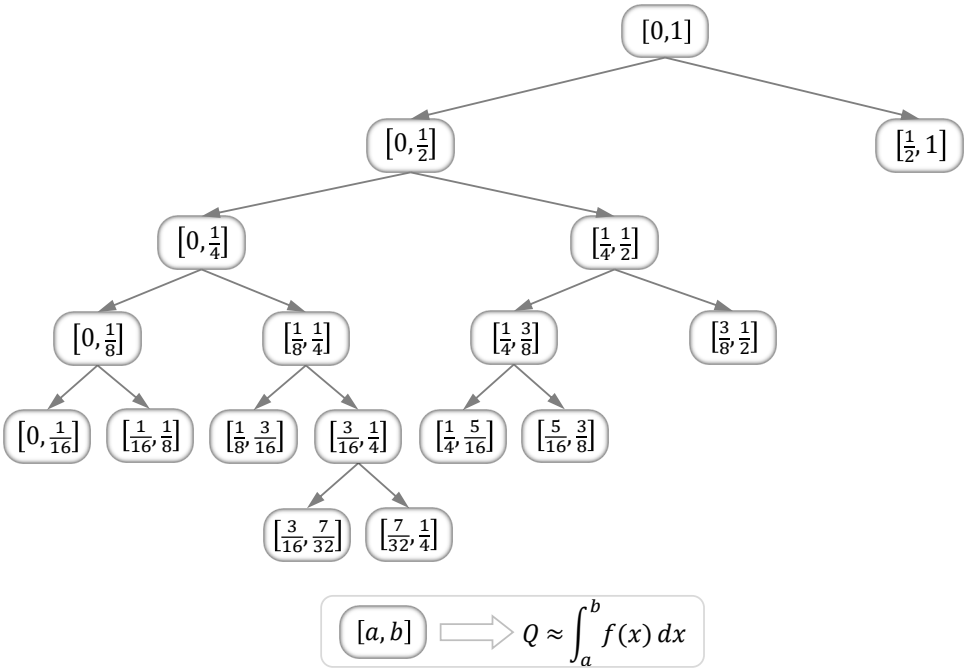
More formally, given a function or integrand $f(x)$, adaptive quadrature approximates the integral $\int_a^b f(x)dx$ to a user-specified error tolerance of τ using static quadrature rules on adaptively refined subregions of the integration domain [21, p. 638], where static quadrature rules are used to compute the integral estimate ($Q(a,b) \approx \int_a^b f(x)dx$) and the error estimate ($\varepsilon(a,b) \approx |Q - \int_a^b f(x)dx|$) on each subregion $[a,b]$. Some of the most commonly used quadrature rules are Simpson's rule, Newton-Cotes formulas, Gauss-Kronrod 7/15-point, and Gauss-Kronrod 10/21-point [21].

Adaptive quadrature is a recursive algorithm which builds a binary recursion tree of subregions as the computation proceeds. This recursion tree of subregions is a visual and conceptual representation of the control-flow in adaptive quadrature. The root of subregion recursion tree is the input integration domain $[a,b]$ over which the initial estimates for integral and error are determined using quadrature rules. When the estimated error is larger than the required error tolerance, the subregion $[a,b]$ is partitioned into two equal halves ($[a, \frac{a+b}{2}]$ and $[\frac{a+b}{2}, b]$). The partitioned subregions constitute the left and right child node of the parent node $[a,b]$. This process is repeated recursively on the left and right nodes until the error estimate on the subregion associated with the tree node is smaller than the error tolerance. Once the recursion terminates, the final estimates of the integral and the error are given by the corresponding estimates from the subregions represented by the leaves of the subregion recursion tree. Moreover, nature of this recursive algorithm adapts to the integrand automatically by partitioning the integration domain into subregions with fine spacing where the integrand is varying rapidly and coarse spacing where the integrand is varying slowly.

Let $P = \langle x_0, x_1, x_2, \ldots, x_{|P|} \rangle$ be the partition on the integration domain $[a,b]$ that is generated by the adaptive quadrature algorithm for an integral $\int_a^b f(x)dx$, where $x_0 = a < x_1 < x_2 < \ldots < x_{|P|} = b$. The final estimate for the integral using the



(a) Partitions on the integration region $[0, 1]$ along *x-axis* for an Gaussian integrand $f(x) = e^{-\frac{(x-0.2)^2}{0.005}}$.



(b) Subregion recursion tree

Figure 7: Adaptive quadrature for the Gaussian integral $\int_0^1 e^{-\frac{(x-0.2)^2}{0.005}} dx$ using Simpson's quadrature rule with error tolerance $\tau = 0.001$.

partition is given by

$$I = \sum_{i=0}^{|P|-1} Q(x_i, x_{i+1}) \tag{6}$$

where $Q(x_i, x_{i+1})$ is the integral estimate given by quadrature rule on the subregion $[x_i, x_{i+1}]$ for all integers i in the range $0 < i < |P|$. Moreover, $\varepsilon(x_i, x_{i+1}) < \tau$, for all integers i in the range $0 < i < |P|$, where $\varepsilon(x_i, x_{i+1})$ is the error estimate given by quadrature rule along the subregion $[x_i, x_{i+1}]$. It is important to note that the set of all subregions $[x_i, x_{i+1}]$ for all integers i in the range $0 < i < |P|$ represents the leaves of subregion recursion tree.

For example, consider a Gaussian function $f(x) = e^{-\frac{(x-0.2)^2}{0.005}}\ \forall x \in [0, 1]$ (see Figure 7a). The subregion recursion tree generated by the adaptive quadrature to approximate the Gaussian integral $\int_0^1 f(x)dx$ to a error tolerance of $\tau = 0.001$ using Simpson's quadrature rule is shown in Figure 7b. The partition generated along the integration region $[0, 1]$ is given by $\langle 0, \frac{1}{16}, \frac{1}{8}, \frac{3}{16}, \frac{7}{32}, \frac{1}{4}, \frac{5}{16}, \frac{3}{8}, \frac{1}{2}, 1\rangle$, where the partitions along the subregion $[0, \frac{3}{8}]$ has fine spacing and has coarse spacing along $[\frac{3}{8}, 1]$ (see Figure 7a).

2.5.3 MULTIPLE INTEGRALS

The multiple integral is a generalization of the definite integral to functions of more than one real variable. For example, multiple integral of a function $f(x_1, x_2, \ldots, x_n)$ in n variables has the following form:

$$\int_{a_1}^{b_1} \int_{a_2}^{b_2} \cdots \int_{a_n}^{b_n} f(x_1, x_2, \ldots, x_n) dx_1 dx_2 \ldots dx_n \tag{7}$$

where $[a_i, b_i]$ is the integration region for the variable x_i, and $[a_1, b_1] \times [a_2, b_2] \times \ldots [a_n, b_n]$ is the hyper-rectangle that represents the integration domain for Equation 7. In general, multiple integral with n variables is referred to as *n-dimensional or n-D integral*. The simplest of all multiple integrals is the 2D integral (also called as double integral), *e.g.*,

$$\int_{a_1}^{b_1} \int_{a_2}^{b_2} f(x_1, x_2) dx_1 dx_2 \tag{8}$$

where $[a_1, b_1] \times [a_2, b_2]$ is a integration region in $\mathbb{R}^2$. Typically, above double integral is computed as repeated one-dimensional integral by applying Fubini's theorem [82],

$$\int_{a_1}^{b_1} \left[\int_{a_2}^{b_2} f(x_1, x_2) dx_1 \right] dx_2 \tag{9}$$

where multiple integral of the function $f(x_1, x_2)$ is first performed over the variable x_1 and then performed over the variable x_2. In other words, numerical approximation of Equation 9 involves applying one-dimensional integration methods like Newton-Cotes formula or Adaptive quadrature along inner dimension with each value of the outer dimension held constant. Then the result of inner dimension is integrated in the outer dimension again by using any one of the standard one-dimensional integration methods. The choice of integration method along each dimension is usually based on the nature of integrand along that particular dimension. In particular, Newton-Cotes formula is used when the integrand is smooth; otherwise adaptive quadrature is used.

Multiple integrals of higher dimensions ($n > 2$) can also be computed as repeated one-dimensional integrals. However, this approach requires the function evaluations to grow exponentially as the number of dimensions increases. For instance, using repeated one-dimensional integral with m function evaluations along each dimension requires a total of m^n functional evaluations to compute n-dimensional integral. In order to overcome this *curse of dimensionality*, methods such as Monte Carlo integration, Sparse Grids, Bayesian Quadrature, or algorithms adaptive on the entire n-dimensional space are used for multiple integrals with higher dimension ($n > 5$) [12, 13, 33, 20]. The fastest known such open source adaptive method is CUHRE [12, 13], which is available as part of CUBA library [35, 36]. A brief overview of CUHRE is illustrated in Appendix A.1, and a more comprehensive survey of other integration methods for multiple integrals can be found in [21, 26, 67, 56].

CHAPTER 3

BEAM DYNAMICS SIMULATION

In this chapter, we first introduce the algorithm to numerically simulate the charged particles beam dynamics and then provide a detailed discussion of the irregular algorithms used in this simulation. Then, in Section 3.2, we present the limitations of implementing this simulation algorithm on sequential machines and present with the irregular properties in the numerical simulation which makes the parallel implementation a challenging task. Finally, Section 3.3 presents our prior work in developing efficient parallel algorithms for this problem.

3.1 OUTLINE OF THE ALGORITHM

Numerical simulation of charged particle 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 8). Formally, for a simulation with step size Δt, 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 -

1. **Particle Deposition** - Deposit the DF sampled by N particles onto a 2D spatial grid of $N_X \times N_Y$ resolution using particle-in-cell method [77, 76, 39], 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.

2. **Compute Retarded Potentials** - The collective effects in the particle distribution due to beam's self-interaction are modeled through retarded potentials, which are computed at each grid-point of the 2D spatial grid using the quadrature defined in Equation 3a.

3. **Compute Self-Forces** - Self-forces are computed on each grid-point of the spatial grid using Equation 1. Next, the self-forces acting on each particle are computed by linear interpolation from the 2D grid of self-forces. It is required

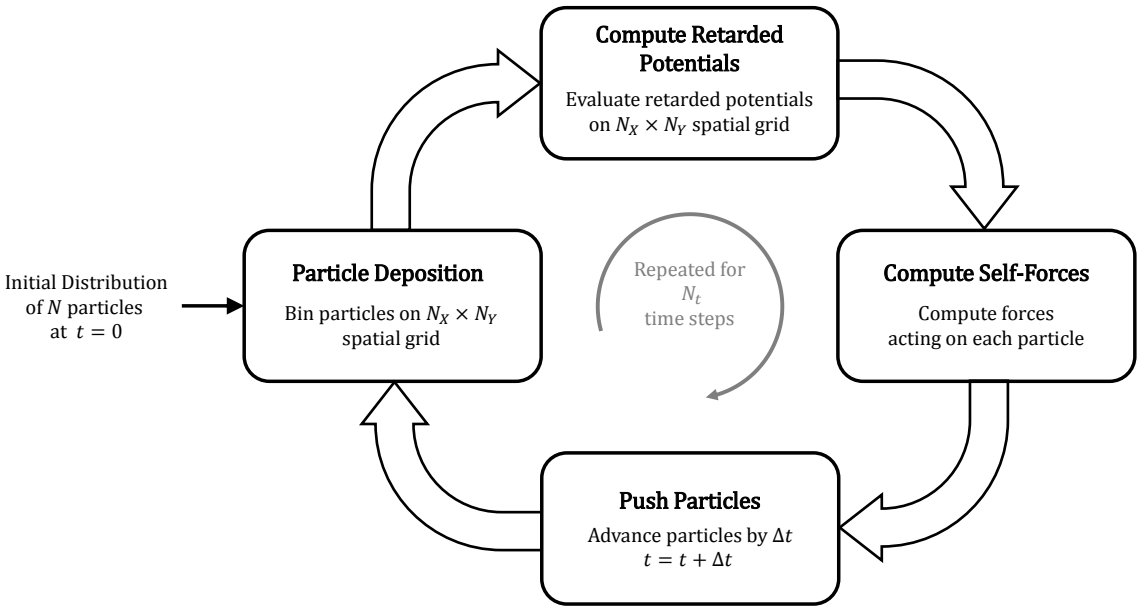


Figure 8: Outline of charged particle beam dynamics simulation algorithm.

that the particle deposition onto the grid and interpolation from the grid onto particles is done in the same manner, so as to avoid "ghost forces".

4. **Push Particles** - The particles are advanced to next time step by a small increment Δt in time by solving Lorentz equation (Equation 1), using leap-frog scheme [47].

The steps 1-4 are repeated for N_t steps, where N_t is usually in the order of few hundreds to thousands. The heart of beam dynamics simulation is the stage at which the retarded potentials are computed, which, as later illustrated in this chapter, is the crucial and by far the most computationally-intensive step of the simulation. As a result, to obtain high performance in the overall beam dynamics simulation, it is important to optimize this step. However, the algorithm required to compute the retarded potentials is highly irregular, making its efficient parallel implementation a daunting task.

3.1.1 PARTICLE DEPOSITION

During particle deposition stage of the simulation, moments of the particle distribution are computed on discrete grid points using PIC approach [77, 76, 39], where

the distribution is first transformed to a normalized representation in which the charged particles are enclosed within a 2D unit-grid. Then, the moments of charged particles are deposited onto the nearest grid points of the unit-grid using inverse interpolation as described in [39]. As a result, during time step k with simulation time $t_k = k\Delta t$ for some integer k in range $0 < k < N_t$, the particle deposition stage generates a 2D grid of moments which denotes the moments of particle distribution at that particular time step. For simplicity, we use the notation D_k to denote the 2D grid of moments at time step k.

Figure 9 and 10 outlines the transformation of particle distribution to a 2D unit-grid representation using PIC approach. The top panel of the Figure 9 shows the charged particle dynamics along a design orbit, where the particle distribution at each time step is tightly enclosed within a spatial grid of resolution $N_X \times N_Y$. The bottom panel of Figure 9 denotes one such spatial grid that tightly envelops the particle distribution at time step k, where the spacing between the extreme points is declared to be L_X and L_Y, respectively, so that the outliers are in the middle of the boundary cells. The spatial grid makes an angle α with the $XY-$plane. Orienting the beam in such a way so as to occupy the smallest volume while containing all the particles yields optimal spatial resolution on a fixed-size rectangular grid. In general, the spatial grid enclosing the particle distribution at time step k is uniquely identified by its tilt angle α, physical size of the grid in $X-$ and $Y-$directions, L_X and L_Y respectively, and the location of its center of charge point (X_0, Y_0). In other words, given the quintuple $\langle \alpha, L_X, L_Y, X_0, Y_0 \rangle_k$ at time step k, the spatial grid that envelops the particle distribution at t_k can be constructed from the quintuple. Next, particles within the spatial grid are rotated through an angle α from the design orbit in $XY-$plane, so as to account for the (X, Y) correlation and is further normalized to be contained within a unit-grid given by $[-0.5, 0.5] \times [-0.5, 0.5]$. The coordinate space in the rotated and normalized plane is referred to as *Grid Frame (GF)* ($\widetilde{X}\widetilde{Y}-$plane in Figure 10). Once the particle distribution is normalized, all the charged particles are deposited onto the nearest grid points using PIC deposition scheme, thereby yielding the moments of the distribution on each grid-point of the unit-grid, which involves inverse interpolation (scatter operation) from the particle position to the nearest grid points. This results in a 2D grid of moments $g_k[1..N_X, 1..N_Y]$ for the particle distribution at t_k, where $g_k[i, j]$ denote the moments deposited on a grid-point $(\widetilde{x}_i, \widetilde{y}_j)$ (in *GF*) located in i^{th} row and j^{th} column of the unit-grid. The moments

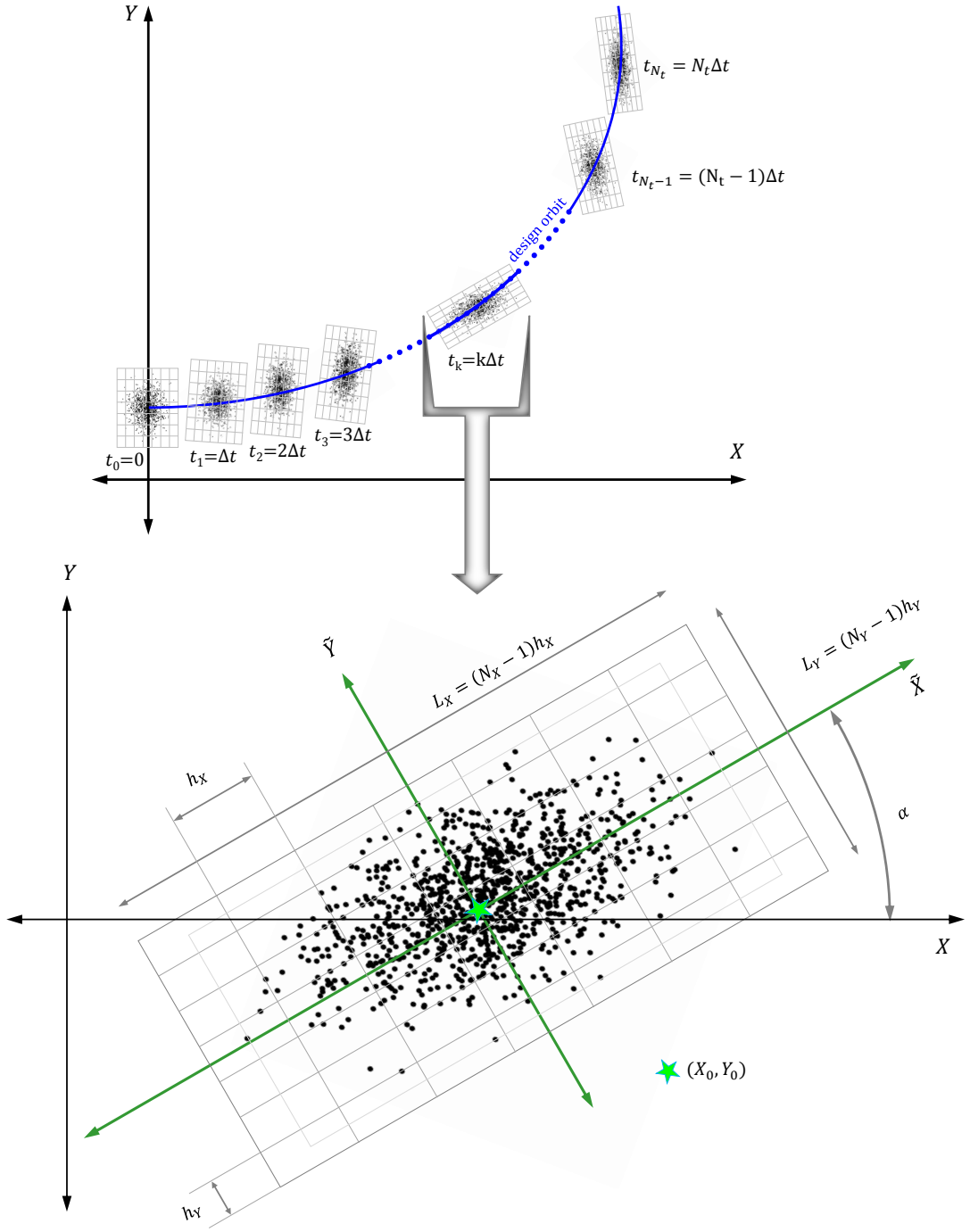


Figure 9: Spatial-grid enclosing the particle distribution at a particular time step of the simulation. Its size is determined by the outliers of the distribution along the principal axes. Blue line denotes the design orbit.

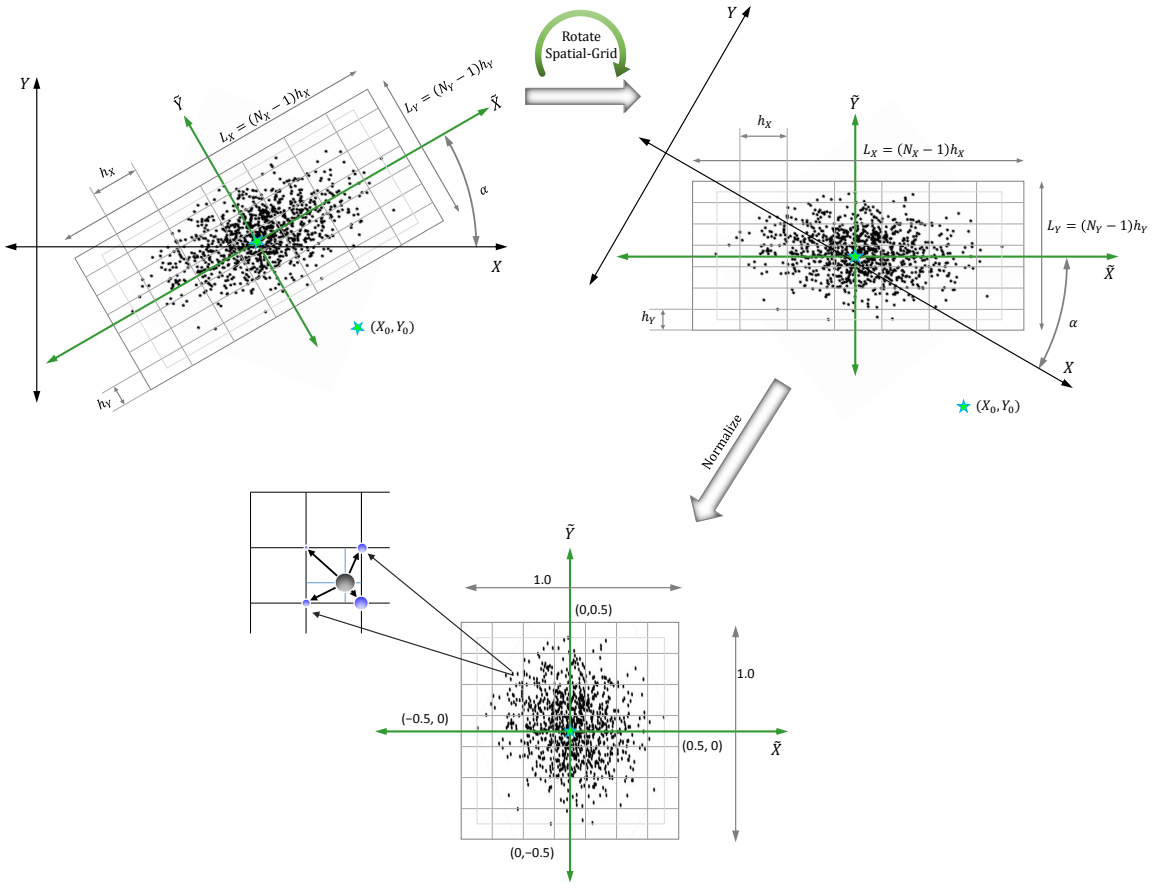


Figure 10: Steps in particle deposition stage of the beam dynamics simulation.

$g_k[i,j]$ is a triplet $\langle \rho, J_X, J_Y \rangle$, where ρ is the deposited charge, J_X and J_Y are the current densities in X- and Y- directions, respectively. It is important to note that the moments deposited on a point $(\widetilde{x}_i, \widetilde{y}_j)$ on the unit-grid (in GF) and on a point (x_i, y_j) on the spatial grid (in LF) at t_k are identical and is given by $g_k[i,j]$, where the transformation between GF and LF using the quintuple $\langle \alpha, L_X, L_Y, X_0, Y_0 \rangle_k$ is given by

$$\begin{bmatrix} x_i \\ y_j \end{bmatrix} = \begin{bmatrix} X_0 \\ Y_0 \end{bmatrix} + \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} L_X \widetilde{x}_i \\ L_Y \widetilde{y}_j \end{bmatrix} \tag{10}$$

$$\begin{bmatrix} \widetilde{x}_i \\ \widetilde{y}_j \end{bmatrix} = \begin{bmatrix} \frac{1}{L_X}\cos\alpha & \frac{1}{L_X}\sin\alpha \\ -\frac{1}{L_Y}\sin\alpha & \frac{1}{L_Y}\cos\alpha \end{bmatrix} \begin{bmatrix} x_i - X_0 \\ y_j - Y_0 \end{bmatrix} \tag{11}$$

3.1.2 COMPUTE RETARDED POTENTIALS

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 of the 2D spatial grid. Formally, suppose V_k denotes the set of grid points on the 2D grid at time step k, such that $|V_k| = N_X N_Y$ and each grid-point $p \in V_k$ is a two-dimensional point (x, y), denoting Cartesian coordinate (or LF-coordinate) of the grid-point on the 2D spatial grid (*i.e.*, (x, y) is the position of p on D_k). Then, the retarded potentials at p are computed using the quadrature in Equation 3a, which, after a variable transformation to avoid singularity, yields

$$I(p) = \int_0^{R(p)} dr' \int_{\theta^{(p)}_{\min}(r')}^{\theta^{(p)}_{\max}(r')} f^{(p)}(r', \theta', t')\ d\theta' \tag{12}$$

where $0 < R(p) \leq \kappa c \Delta t$ for some positive integer $\kappa \leq k$, retarded time $t' = k\Delta t - r'/c$, and $i\Delta t < t' \leq (i+1)\Delta t$ for some integer i in the range $k - \kappa \leq i < k$. The integral estimate $I(p)$ is a triplet $\langle \phi, A_X, A_Y \rangle$, where ϕ is the scalar potential, A_X and A_Y are the vector potentials in $X-$ and $Y-$ directions, respectively. The domain of integration, $\{(r', \theta') \in \mathbb{R}^2 \mid 0 \leq r' \leq R(p), \theta^{(p)}_{\min}(r') \leq \theta' \leq \theta^{(p)}_{\max}(r')\}$, is a subregion of $\mathbb{R}^2$ and it is denoted $S_{0,R(p)}$. For convenience, we shall refer to the integral in Equation 12 that computes retarded potentials as *rp-integral*.

Computation of Integration Limits

Figure 11 illustrates the computation of integration limits, where for all r' sampled along the outer dimension, the portion of circle with radius r' and center at (x, y) that is within the spatial grid enclosing the particle distribution at time t' constitutes the inner integral limits. Moreover, these limits are discontinuous for some r'. The red-line in Figure 11 denotes the integration range in $\theta'-$domain, where $[\theta^{(p)}_{\min}(r'), \theta^{(p)}_{\max}(r')]$ represents the inner integral limits and the circle of radius r' with center at p is referred to as *circle of causality* at p (denoted by grey dotted line). The outer integral limit $R(p)$ is the smallest value of r' such that the circle of causality for all $r' > R(p)$ has no intersection with the spatial grid at t', and for all $0 \leq r' \leq R(p)$ the circle intersects the spatial grid at t'.

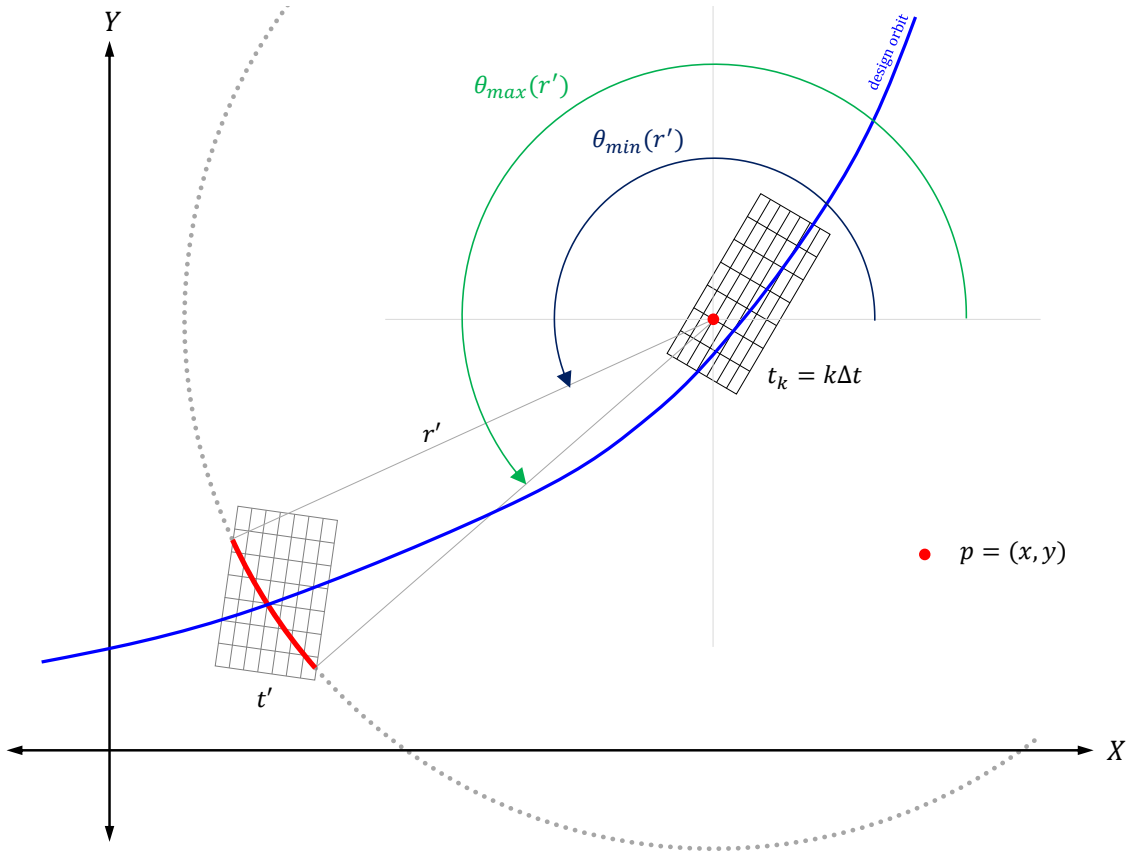


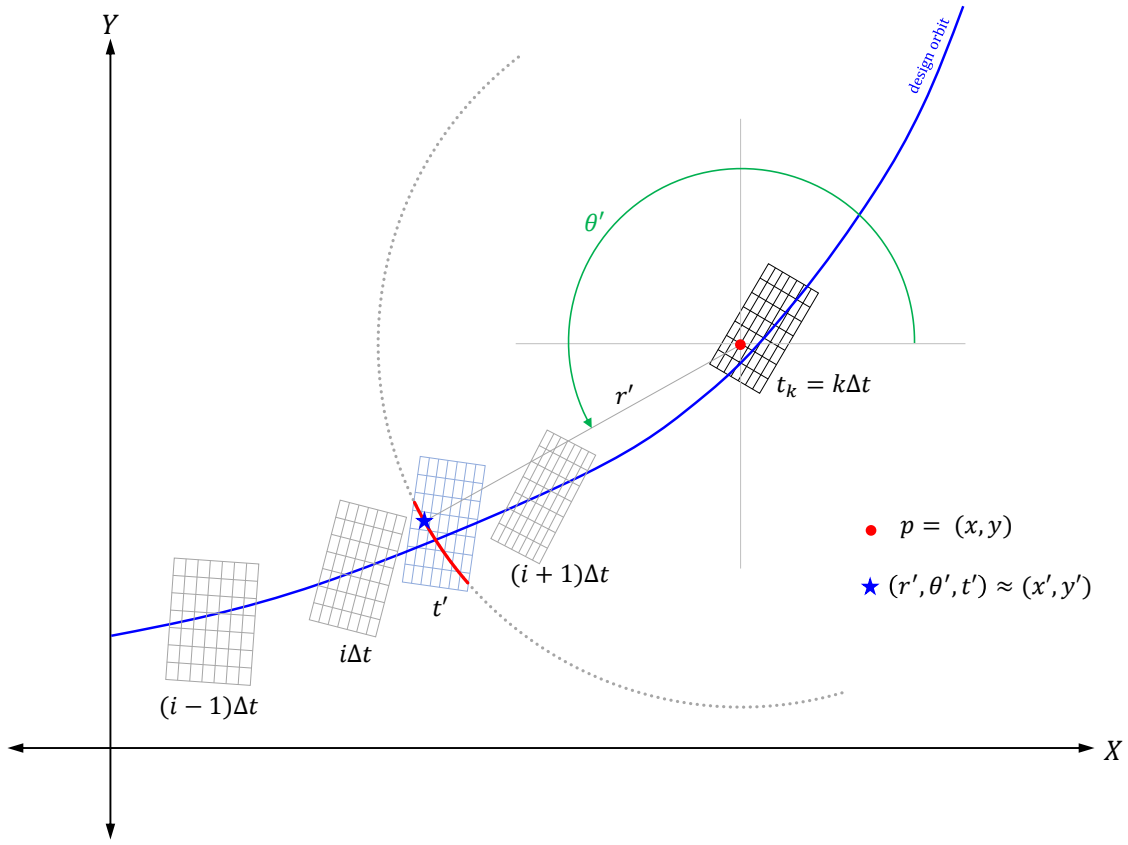
Figure 11: Evaluation of integration limits in $\theta'-$domain for rp-integral.

Integrand Evaluation

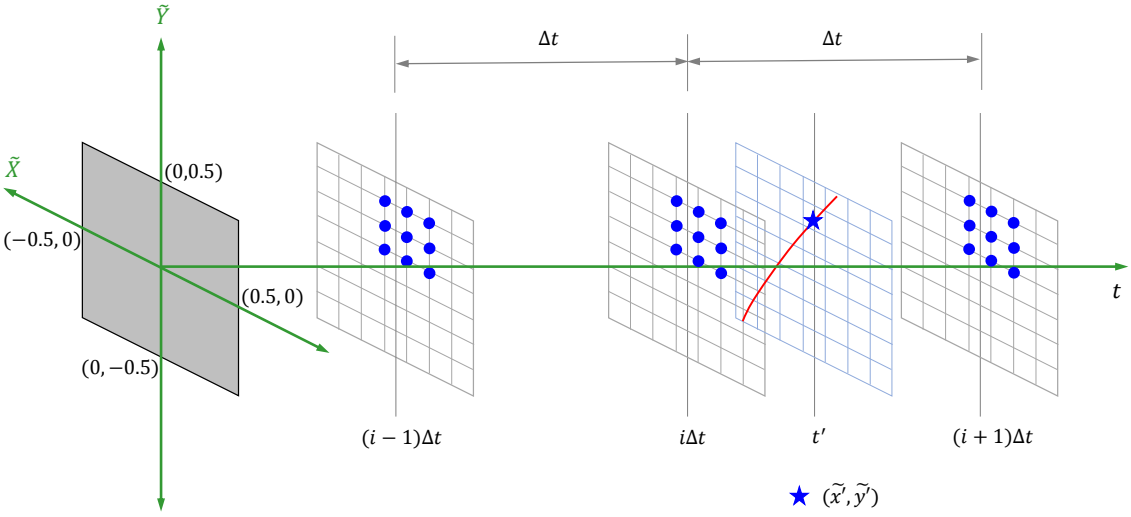
The integrand $f^{(p)}: \boldsymbol{A} \to \mathbb{R}^3$, where $\boldsymbol{A} \in \{(r', \theta', t') \mid 0 \leq r' \leq R(p), t' = k\Delta t - \frac{r'}{c}, \theta' \in [0, 2\pi]\}$ and a point (r', θ', t') on the integrand represents the polar coordinate (r', θ') in LF with center at (x, y) as shown in Figure 12a. Cartesian equivalent for (r', θ', t') is given by

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x & \cos\theta' \\ y & \sin\theta' \end{bmatrix} \begin{bmatrix} 1 \\ r' \end{bmatrix}. \tag{13}$$

The integrand $f^{(p)}(r', \theta', t')$ in rp-integral denote the moments, $\langle \rho, J_X, J_Y \rangle$, deposited on the polar coordinate (r', θ') (or (x', y') in Cartesian system) by the particle distribution at time t'. However, $f^{(p)}$ does not have an analytic form, and as a result, $f^{(p)}(r', \theta', t')$ is numerically approximated using neighboring points from the 2D grid of moments deposited on the spatial grid at time t'. When $t' = i\Delta t$, moments on the spatial grid at t' is given by the 2D grid of moments computed during the particle



(a)



(b)

Figure 12: Numerical approximation of integrand values in rp-integral using interpolation.

deposition stage of time step i. On the other hand, when $i\Delta t < t' < (i+1)\Delta t$, the moments at t' is approximated using interpolation from the 2D grid of moments, D_{i-1}, D_i, and D_{i+1}. Furthermore, for all $t' < t$, the value of $f^{(p)}(r', \theta', t')$ is defined only when (x', y') is within the spatial grid at time t' and is 0 otherwise. For example, Figure 12a and Figure 12b illustrates the approximation of $f^{(p)}(r', \theta', t')$ using 27 neighboring points.

In Figure 12a, spatial grids from earlier time steps $(i-1)$, i, and $(i+1)$ that are computed during the particle deposition stage of the corresponding time steps are shown by grey colored grids, whereas the spatial grid at the current time step k is shown by black colored grid. Further, the spatial grid at retarded time t' which is approximated from the neighboring three spatial grids is shown by blue colored grid. The point (r', θ', t') on the integrand lies on the portion of causality circle that is within the spatial grid at t'. Figure 12b illustrates the approximation of $f^{(p)}(r', \theta', t')$ using the normalized unit-grid (in GF) representation of the moments, where $(\widetilde{x'}, \widetilde{y'})$ is the GF equivalent for the LF-coordinate (x', y'). The moments deposited on $(\widetilde{x'}, \widetilde{y'})$ is approximated using moments from the grid points of neighboring three unit-grids, where the set of grid points required by the interpolation is shown using blue dots.

Formally, for all r' sampled from the interval $[(k-i-1)c\Delta t, (k-i)c\Delta t]$ (*i.e.* for all $t' \in [i\Delta t, (i+1)\Delta t]$), integrand $f^{(p)}(r', \theta', t')$ is approximated using 27 neighboring points from the 2D grids of moments, D_{i-1}, D_i and D_{i+1} (nine points from each of the three data grids). In other words, 2D grids of moments from time steps $i-1$, i, and $i+1$ are required for calculating rp-integral along the subregion, $\{(r', \theta') \in \mathbb{R}^2 \mid (k-i-1)c\Delta t \leq r' \leq (k-i)c\Delta t, \theta^{(p)}_{\min}(r') \leq \theta' \leq \theta^{(p)}_{\max}(r)\}$. Adapting to this relation, rp-integral along the entire integration region $S_{0,R(p)}$ requires grids of moments between time steps $k-\kappa$ and k (*i.e.* $D_{k-\kappa}$ to D_k), where the computation of integral along a subregion $S_{jc\Delta t,(j+1)c\Delta t}$ uses data from D_{k-j}, D_{k-j-1}, and D_{k-j-2}.

Numerical Integration

Numerical approximation of rp-integral use repeated one-dimensional integration algorithm, as illustrated in Section 2.5.3. The choice of integration method along each dimension is based on the nature of integrand along that particular dimension. Empirical analysis on rp-integral behavior shows that the integrand has strongly varying orders of magnitude in different parts of the subregion in r'-domain. In contrast, the inner dimension features only regions where change is gradual (*i.e.*

integrand is smooth in θ'-domain). As a result, outer integral is computed using adaptive quadrature and the inner integral using Newton-Cotes formulas.

More formally, for a grid-point $p \in V_k$ and a subregion $S_{a,b} = \{(r', \theta') \in \mathbb{R}^2 \mid a \leq r' \leq b, \theta^{(p)}_{\min}(r') \leq \theta' \leq \theta^{(p)}_{\max}(r')\}$, integration algorithm results in a partition, $\langle r^{(p)}_0, r^{(p)}_1, \ldots, r^{(p)}_n \rangle$, 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, rp-integral is calculated as follows:

$$I(p) = \sum_{i=0}^{n-1} Q(r^{(p)}_i, r^{(p)}_{i+1}) \tag{14}$$

where $Q(r^{(p)}_i, r^{(p)}_{i+1})$ is the Quadrature rule estimate along the subregion $S_{r^{(p)}_i, r^{(p)}_{i+1}}$, and for every r' sampled along that subregion, inner integral is computed using Newton-Cotes formulae, as illustrated in [21]. For example, $Q(r^{(p)}_i, r^{(p)}_j)$ estimate using adaptive Simpson's rule is given by,

$$\begin{aligned} Q(r^{(p)}_i, r^{(p)}_j) &= S\left(r^{(p)}_i, \tfrac{r^{(p)}_i + r^{(p)}_j}{2}\right) + S\left(\tfrac{r^{(p)}_i + r^{(p)}_j}{2}, r^{(p)}_j\right) - S(r^{(p)}_i, r^{(p)}_j) \\ &\approx \int_{r^{(p)}_i}^{r^{(p)}_j} dr' \int_{\theta^{(p)}_{\min}(r')}^{\theta^{(p)}_{\max}(r')} f^{(p)}(r', \theta', t')\, d\theta' \end{aligned} \tag{15}$$

where $S(r^{(p)}_i, r^{(p)}_j)$, $S\left(r^{(p)}_i, \frac{r^{(p)}_i + r^{(p)}_j}{2}\right)$, and $S\left(\frac{r^{(p)}_i + r^{(p)}_j}{2}, r^{(p)}_j\right)$ are the estimates given by Simpson's quadrature rule and are calculated as follows

$$S(a, b) = \frac{b-a}{6}\Big[f_{in}(a) + 4f_{in}\left(\tfrac{a+b}{2}\right) + f_{in}(b)\Big] \tag{16}$$

where $f_{in}(r') = \int_{\theta^{(p)}_{\min}(r')}^{\theta^{(p)}_{\max}(r')} f^{(p)}(r', \theta', t')\, d\theta'$ *i.e.* $f_{in}(r')$ is the inner integral in $\theta'-$domain and it is approximated as the weighted sum of integrand values at equally spaced points within the domain of integration, where the weights are given by Newton-Cotes formulas [21, p. 613]. For the purpose of this study, we use three-point Newton-Cotes formula to compute the inner integral.

Note that high-fidelity computation of collective effects by calculating retarded potentials at all grid points on the 2D spatial 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.

3.2 SEQUENTIAL SIMULATION

In this section, we first present the limitations of implementing beam dynamics simulation on sequential machines and show that the performance of sequential simulation is severely limited by the retarded potentials computation stage. Next, in Section 3.2.2, we present the empirical analysis of sequential beam dynamics simulation and show that distribution of work and data in the accurate computation of retarded potentials at each time step is irregular and exhibits control-flow and memory access patterns that are not readily amenable to the data-parallel, SIMD nature of GPU and Intel MIC architectures. In particular, we show that the computation of rp-integral is highly irregular, and as a result, naive parallel implementations of such computations tend to present significant branch and memory divergence on SIMD architectures which leads to severe performance bottlenecks.

3.2.1 LIMITATIONS OF SEQUENTIAL IMPLEMENTATION

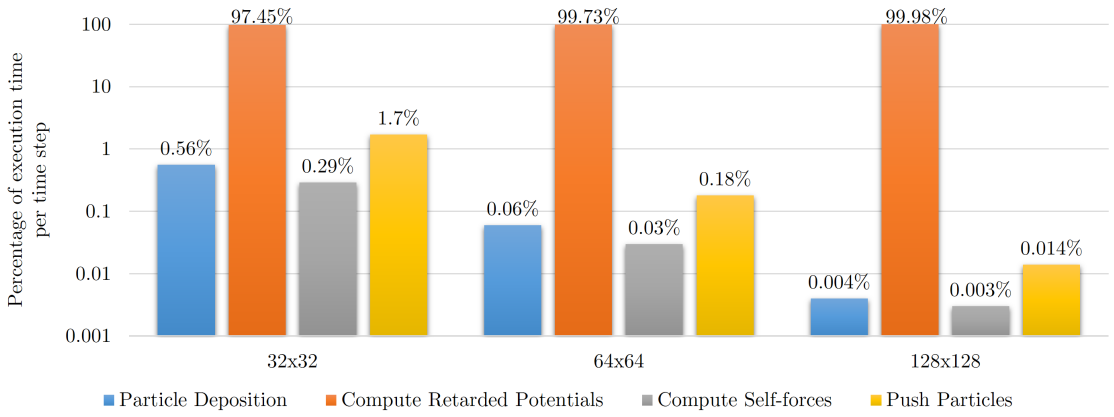


Figure 13: Percentage of sequential execution time spent by different stages of the beam dynamics simulation per time step averaged over all time steps for various grid resolutions.

The heart of beam dynamics simulation is the high-fidelity computation of collective effects which require calculating retarded potentials at all grid points on the 2D spatial grid of each time step. This is crucial and by far the most computationally intensive step of the beam dynamics simulation. We illustrate this empirically by simulating the beam dynamics sequentially with $N = 100000$, where the initial distribution is generated by Monte Carlo sampling of N particles with a total charge of the beam bunch $Q = 1$nC. This simulation is executed sequentially for

three different spatial grid resolutions: (i) 32×32, (ii) 64×64, and (iii) 128×128. Figure 13 illustrates the percentage of execution time required by different stages of this simulation per time step averaged over all time steps. Notice that, during each time step, compute retarded potentials stage of the simulation takes $95 - 99\%$ of the overall execution time. This shows that the efficiency of sequential simulation is severely limited by the computational requirement associated with computing retarded potentials. Empirical study with different simulation configuration generates behavior that is consistent with that shown in Figure 13. In other words, for each time step of the beam dynamic simulation, computing retarded potentials is the most computationally-intensive step for all the valid simulation configurations.

3.2.2 EMPIRICAL ANALYSIS

This section presents the computational requirement and memory access properties of retarded potentials computation stage of the simulation that poses problems for high-performance parallel implementations. We use empirical analysis on sequential beam dynamics implementation to study the computational properties, and then, based on the observed behavior, we make a general inference about the properties that tend to present significant challenge in developing parallel implementation on GPUs and Intel MICs.

The input configuration considered for empirical analysis using sequential simulation is illustrated below and the values are chosen so as to illustrate specific properties of rp-integral and retarded potentials computations that are commonly observed for all valid simulation configurations. It is important to note that individual values of parameters in the chosen configuration are not relevant to the discussion that follows, and the conclusions inferred from the observed properties apply to all valid simulation configurations.

> **Simulation configuration for empirical study:** $N = 100000$, $N_X = 32$, $N_Y = 32$, $N_t = 1000$, $\tau = 0.001$, *and the initial distribution is generated by Monte Carlo sampling of* N *particles with a total charge of beam bunch* $Q = 1nC$.

Note that the resolution of the above described simulation configuration is very low to provide any valuable insights about the physics behind beam dynamics process. However, they are chosen only to study the properties of the computation involved.

Further, high-resolution and high-fidelity beam dynamics simulation is computationally intractable with sequential codes. In practice, to study all the relevant physics of beam dynamics in particle accelerators, N is in the order of few hundred thousands to millions of particles, N_X and N_Y varies from 64 to 1024, error tolerance for rp-integral approximation τ is between 10^{-12} to 10^{-6}, and N_t is of the order of few hundreds to thousands of time steps. Such high-resolution and high-fidelity simulations are intractable with sequential implementations, and thus necessitating the need for high-performance parallel implementations.

Computational Workload Characteristics

Figures 14 and 15 illustrate the computational requirement for beam dynamics simulation using different simulation configurations, where computational requirement or workload is measured in terms of the number of double-precision floating point operations required for the corresponding computation. In particular, Figure 14 shows the workload behavior of compute retarded potentials stage at different time steps of a beam dynamics simulation using the above described configuration, where *x-axis* denote different time steps of the simulation and *y-axis* denote double-precision floating point operations required for computing retarded potentials at a particular time step in GFlops (10^9 Flops). Next, Figure 15 shows the computational workload for rp-integral evaluations at all grid points on a spatial grid at a particular time step of the simulation using above described configuration but with 8×8 spatial grid (We choose 8×8 instead of 32×32 grids for visual presentation convenience). The *x-axis* in figure denotes the set of grid points at a particular time step and *y-axis* denotes double-precision floating point operations required for numerically approximating rp-integral in MFlops (10^6 Flops).

It is evident from Figure 14 and Figure 15 that the computational requirement is not uniform and varies unpredictably between time steps and grid points. In particular, Flops required for rp-integral computation at each grid-point is different from one another which leads to non-uniform workload between time steps. Such non-uniform and input dependent workload poses multiple challenges to develop a high-performance parallel implementation, where equal distribution of work over processor cores and maintaining workload balance is crucial for good performance and scalability.

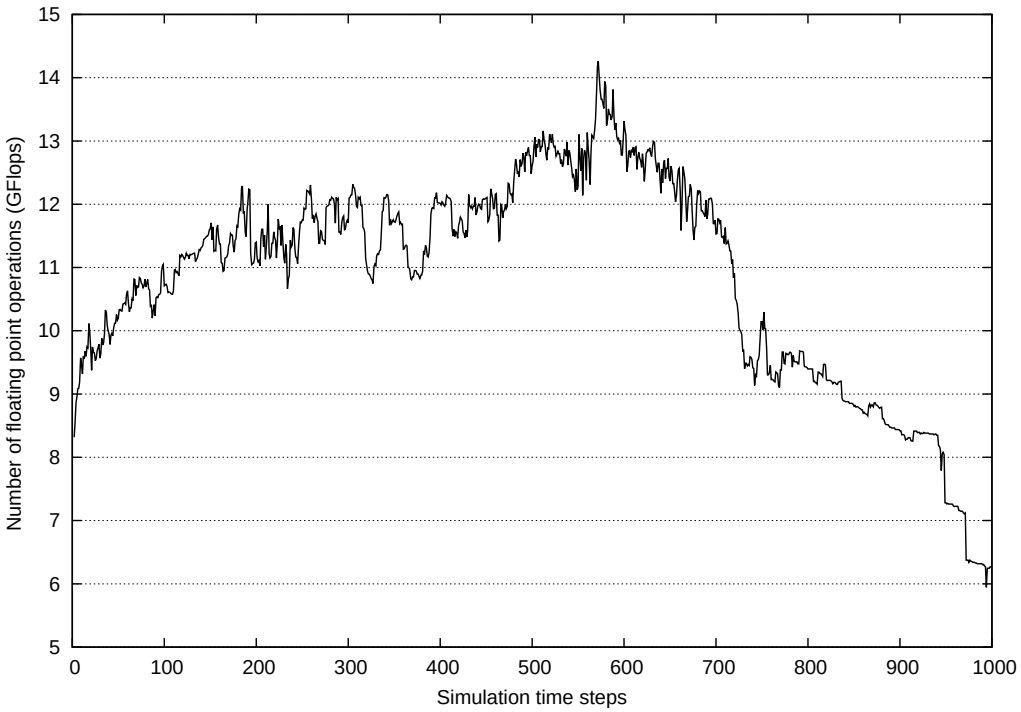


Figure 14: Workload characteristics of compute retarded potentials stage of the beam dynamics simulation for 1000 time steps calculated using the sequential simulation code with $N = 100000$ particles, 32×32 spatial grid resolution, error tolerance for rp-integral computations $\tau = 0.001$, and total beam charge of $Q = 1$nC. (Note that the computational workload is measured as a function of double-precision floating point operations in GFlops)

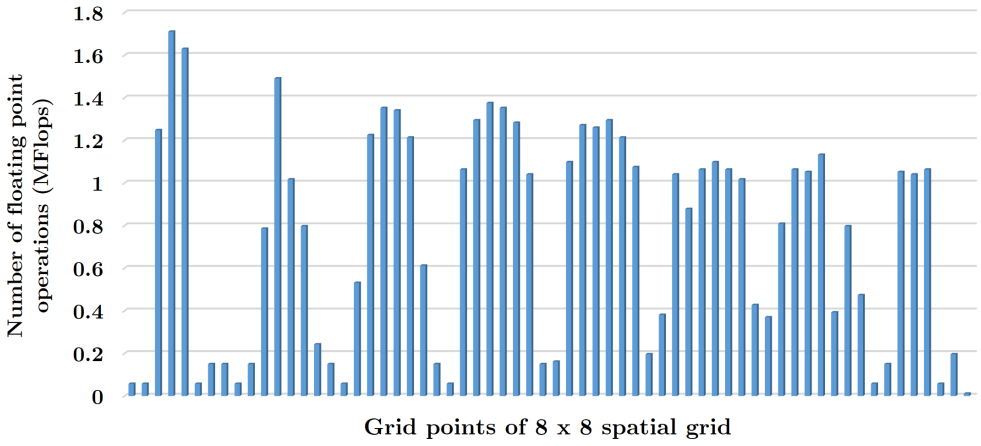


Figure 15: Workload characteristics of rp-integral computation at different grid points on a 8×8 spatial grid at a particular time step of the beam dynamics simulation with $N = 100000$ particles, error tolerance for rp-integral computations $\tau = 0.001$, and total beam charge of $Q = 1$nC. (Note that the computational workload is measured as a function of double-precision floating point operations in GFlops)

Control-flow Properties

Figure 16 illustrates the subregion recursion tree generated by adaptive quadrature along the outer dimension of rp-integral at grid points p_1, p_2, p_3 and p_4 on the spatial grid at time step k, where the values of p_1, p_2, p_3, p_4 and k are empirically selected to illustrate the control-flow properties of rp-integral. In Figure 16, flow of computation or control-flow is determined by the pre-order traversal of the corresponding recursion tree. In particular, the red-arrows on each subregion recursion tree denotes the control-flow path for evaluating rp-integral at the corresponding grid-point, and the order of execution is shown by the number on each arrow. Additionally, the partition generated by the adaptive quadrature for the outer integral is shown at the bottom of each sub-figure.

Now, consider computing retarded potentials at time step k which requires evaluating rp-integral at all grid points $V_k = \{p_1, p_2, \ldots, p_{|V|}\}$, where $|V_k| = N_X N_Y$ and p_i is a point on the spatial grid at t_k, for all integers i in range $0 < i \leq |V_k|$. From Figure 16, and from the empirical analysis on sequential beam dynamics simulation, we make the following general inference about rp-integral properties at any two observation-point p_i and p_j $(i \neq j)$

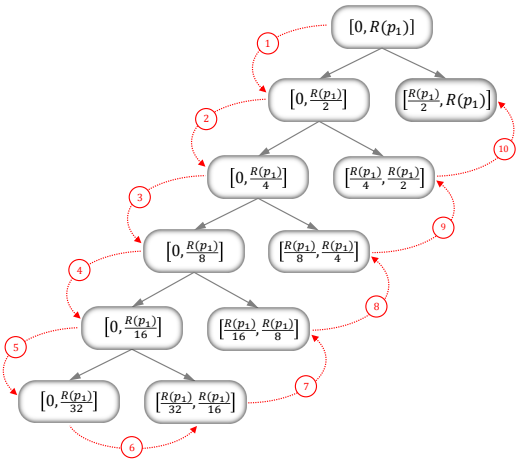
1. Partition generated by adaptive quadrature along the outer dimension of rp-integral is unique for each grid-point *i.e.* outer integral partition for rp-integral at p_i is different form that of rp-integral at p_j. For example, in Figure 16, the partition generated for rp-integral at p_1 and p_4 are

$$P(p_1) = \langle 0, \tfrac{R(p_1)}{32}, \tfrac{R(p_1)}{16}, \tfrac{R(p_1)}{8}, \tfrac{R(p_1)}{4}, \tfrac{R(p_1)}{2}, R(p_1) \rangle$$
$$P(p_4) = \langle 0, \tfrac{R(p_4)}{4}, \tfrac{3R(p_4)}{8}, \tfrac{R(p_4)}{2}, R(p_4) \rangle$$

2. The pre-order traversal for the subregion recursion tree generated by adaptive quadrature along the outer dimension of rp-integral evaluation at p_i is different from that of rp-integral at p_j. In other words, control-flow for rp-integral at p_i and p_j are substantially different from one another.

3. The number of floating-point operations required to calculate rp-integral at p_i is different from that of rp-integral at p_j.

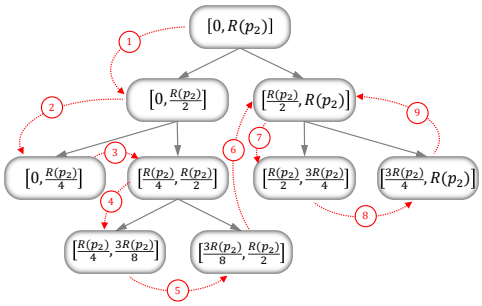
The properties listed above shows that numerical approximation of rp-integral using adaptive quadrature is irregular and exhibits control-flow patterns that are



$$[r_1, r_2] \Rightarrow \int_{r_1}^{r_2} dr' \int_{\theta_{min}(r')}^{\theta_{max}(r')} f^{(p_1)}(r', \theta', t')\, d\theta'$$

$$Partitions = \left\langle 0, \frac{R(p_1)}{32}, \frac{R(p_1)}{16}, \frac{R(p_1)}{8}, \frac{R(p_1)}{4}, \frac{R(p_1)}{2}, R(p_1) \right\rangle$$

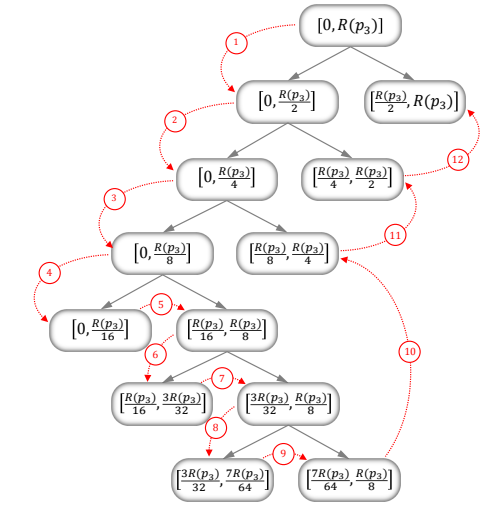
(a) Grid-point p_1



$$[r_1, r_2] \Rightarrow \int_{r_1}^{r_2} dr' \int_{\theta_{min}(r')}^{\theta_{max}(r')} f^{(p_2)}(r', \theta', t')\, d\theta'$$

$$Partitions = \left\langle 0, \frac{R(p_2)}{4}, \frac{3R(p_2)}{8}, \frac{R(p_2)}{2}, \frac{3R(p_2)}{4}, R(p_2) \right\rangle$$

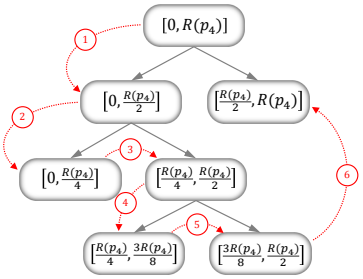
(b) Grid-point p_2



$$[r_1, r_2] \Rightarrow \int_{r_1}^{r_2} dr' \int_{\theta_{min}(r')}^{\theta_{max}(r')} f^{(p_3)}(r', \theta', t')\, d\theta'$$

$$Partitions = \left\langle 0, \frac{R(p_3)}{16}, \frac{3R(p_3)}{32}, \frac{7R(p_3)}{64}, \frac{R(p_3)}{8}, \frac{R(p_3)}{4}, \frac{R(p_3)}{2}, R(p_3) \right\rangle$$

(c) Grid-point p_3



$$[r_1, r_2] \Rightarrow \int_{r_1}^{r_2} dr' \int_{\theta_{min}(r')}^{\theta_{max}(r')} f^{(p_4)}(r', \theta', t')\, d\theta'$$

$$Partitions = \left\langle 0, \frac{R(p_4)}{4}, \frac{3R(p_4)}{8}, \frac{R(p_4)}{2}, R(p_4) \right\rangle$$

(d) Grid-point p_4

Figure 16: Subregion recursion tree generated by adaptive quadrature along the outer dimension of RP-integral at grid points p_1, p_2, p_3 and p_4.

not readily amenable to many modern parallel architectures. Moreover, these irregular properties in an algorithm tend to present multiple challenges in developing high-performance parallel implementations on GPU and Intel MIC architectures, as illustrated in Section 2.3.

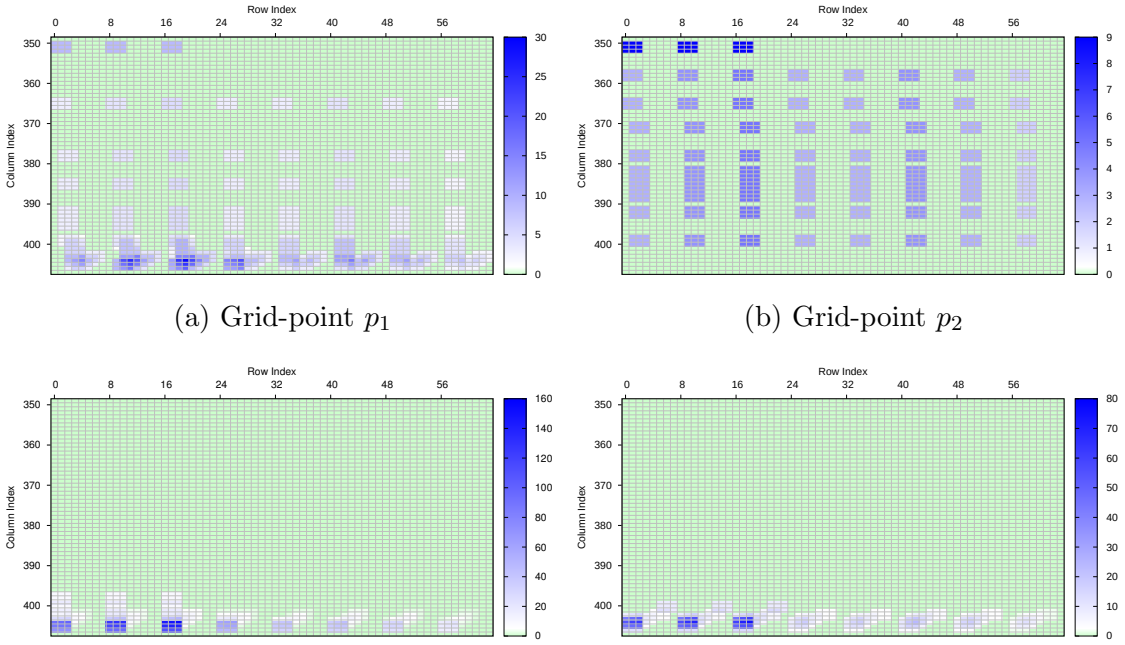
Memory Access Properties

Let the moments computed from all time steps be stored linearly on the host/device memory as a 2D array $M[0..N_t, 1..N_X N_Y]$ in row major order such that moments deposited on a grid-point located in i^{th} row and j^{th} column of the spatial grid during the particle deposition stage at $t_k = k\Delta t$ is stored in $M[k, (iN_X + j)]$, for all integers i, j and k in the range $0 \leq i \leq N_X, 0 \leq j \leq N_Y$, and $0 \leq k \leq N_t$, respectively.

Figure 17 illustrates the memory access on a portion of 2D array M (between row-index $350 - 410$) that is observed during the rp-integral evaluation at each of the four grid points (p_1, p_2, p_3 and p_4), where a grid-cell at m^{th} row and n^{th} column denotes the data element $M[m, n]$. Each cell has a gradient color that represents the number of times the data element corresponding to that cell is accessed while computing rp-integral. When $M[m, n]$ is not accessed during the lifetime of rp-integral evaluation at t_k, then the cell corresponding to $M[m, n]$ is marked green. On the other hand, blue gradient at $M[m, n]$ denotes that the data element is accessed at least once during the course of execution and the gradient magnitude denotes the number of times $M[m, n]$ is accessed during the rp-integral evaluation at the corresponding grid-point.

Now, consider computing retarded potentials at t_k which requires evaluating rp-integral at all grid points $V_k = \{p_1, p_2, \ldots, p_{|V_k|}\}$, where $|V_k| = N_X N_Y$ and p_i for all integers i in range $0 < i \leq |V_k|$ is a point on the spatial grid at t_k. From Figure 17, and from the empirical analysis on sequential beam dynamics simulation, we make the following general inference about the memory-access characteristics of rp-integral computation at any two grid points p_i and p_j $(i \neq j)$

1. *Temporal locality* - A certain subset of data accessed during the computation of rp-integral at p_i is likely to be referenced again in relative temporal proximity. In other words, for rp-integral computation at p_i, there exist one or more data elements $M[m, n]$ that is referenced more than once. In Figure 17, a cell in m^{th} row and n^{th} column with a gradient magnitude of greater than one denotes that the element $M[m, n]$ is referenced and reused gradient magnitude



(a) Grid-point p_1

(b) Grid-point p_2

(c) Grid-point p_3

(d) Grid-point p_4

Figure 17: Memory requests on a portion of 2D array M (between row-index $350 - 410$) for rp-integral at grid points p_1, p_2, p_3, and p_4.

times during the course of corresponding rp-integral evaluation. For example, data element $M[406, 17]$ is accessed and used roughly 160 times during the rp-integral computation at p_3.

2. *Spatial locality* - Value of the integrand $f^{(p_i)}$ that constitutes the rp-integral at p_i is approximated using interpolation from 27 neighboring points, as described in Section 3.1.2. Consequently, memory access to read these 27 points exhibits 3D spatial locality for every integration point that gets sampled during the numerical integration procedure.

3. For each rp-integral evaluation, there exists a small set of data elements that are reused more frequently than others which are denoted by the darkest shade of the blue gradient in Figure 17. For example, consider the memory access on array M for rp-integral computation at grid-point p_1. Notice that the data elements between row index 404 and 407 have higher gradient magnitude *i.e.* they are reused more frequently that others. In other words, the moments computed between t_{404} and t_{407} are reused more frequently. This happens when

the integral computation is focused around the subregion $[(k\Delta t - t_{407})c, (k\Delta t - t_{404})c]$ (since, $t' = k\Delta t - r'/c \Rightarrow r' = (k\Delta t - t')c$ from the conditions in Equation 12) or the partition has fine spacing within that subregion.

4. The memory access footprint for rp-integral at p_i has some overlap with that of rp-integral at p_j for some integer i and j. In other words, there is a significant reuse of data for two grid points. For example, the data elements accessed between the row index 404 and 407 for rp-integral at point p_3 overlaps with that of the rp-integral at point p_4, as illustrated in Figure 17c and Figure 17d. However, the gradient magnitude of the overlapping data access are different *i.e.* the frequency of reuse is different.

5. While computing rp-integral on grid points, there is a significant reuse of data for two nearby grid points. The reuse of data for two grid points is inversely proportional to the distance between the two grid points. More formally, given two points u and v, where $u, v \in V$ and $u \neq v$, the data locality or the number of overlapping memory access between rp-integral evaluation at u and v is inversely proportional to the Euclidean distance between them.

3.3 PRIOR RESEARCH IN PARALLEL SIMULATION

This section presents two parallel algorithms that improve the overall performance of beam dynamics simulation by offloading the retarded potentials computation stage at each time step of the simulation onto GPUs. The first algorithm is Two-phase Algorithm, and Section 3.3.1 presents a brief overview of this algorithm that computes retarded potentials at each time step by focusing on load-balancing, which, due to the uneven computation load associated with rp-integrals evaluation, is critical for good performance and scalability. The second algorithm is Heuristics Algorithm, and Section 3.3.2 presents a overview of this algorithm which uses two different heuristics to maximize data reuse and to balance the workload among threads during the retarded potentials computation stage of the simulation. The heuristics used in this algorithm are based on the properties observed from empirical analysis of sequential simulation illustrated in Section 3.2.2. A more comprehensive review of these two algorithms and their implementation performance on GPU architectures is published in [5, 6].

3.3.1 TWO-PHASE ALGORITHM

The two-phase approach is a globally adaptive algorithm that approximates rp-integral estimates at $N_X N_Y$ grid-points at each time step by adaptively locating subregions in parallel where the error estimate is greater than some user-specified error tolerance. It then calculates the rp-integral estimates on these subregions in parallel.

COMPUTE-POTENTIALS(k, V, τ, M)

1 $L \leftarrow$ RP-PHASEONE(k, V, τ, M)
2 RP-PHASETWO(t, V, L, τ, M)

RP-PHASEONE(k, V, τ, M)

1 $L \leftarrow \emptyset$
2 **for** each grid-point $p \in V$ **in parallel**
3 $\quad p.I \leftarrow 0$, $p.\varepsilon \leftarrow 0$
4 $\quad$ compute outer integral limit $[0, R]$
5 $\quad$ LIST-INSERT$(L, ([0, R], p))$
6 **while** $(|L| < L_{\max})$ and $(|L| \neq 0)$
7 $\quad$ **for** each tuple $([a, b], p) \in L$ **in parallel**
8 $\quad\quad (I, \varepsilon) \leftarrow$ RP-QUADRULE$(([a, b], p), \tau, M)$
9 $\quad\quad$ LIST-INSERT$(S, (([a, b], p), I, \varepsilon))$
10 $\quad L \leftarrow$ PARTITION$(S, L_{\max}, \tau)$
11 $\quad$ **for** each tuple $(([a, b], p), I, \varepsilon) \in S$
12 $\quad\quad$ **if** $\varepsilon < \tau$
13 $\quad\quad\quad p.I \leftarrow p.I + I$
14 $\quad\quad\quad p.\varepsilon \leftarrow p.\varepsilon + \varepsilon$
15 **return** L

RP-PHASETWO(t, V, L, τ, M)

1 **for** each $([a, b], p) \in L$ **parallel**
2 $\quad (I, \varepsilon) \leftarrow$ RP-QUADRATURE$(([a, b], p), \tau, M)$
3 $\quad p.I \leftarrow p.I + I$
4 $\quad p.\varepsilon \leftarrow p.\varepsilon + \varepsilon$

The procedure COMPUTE-POTENTIALS implements this algorithm where RP-PHASEONE and RP-PHASETWO method illustrates the two phases of the algorithm

to compute rp-integral to a error tolerance of τ at all grid points $p \in V$, where p is a grid-point on the spatial grid at time $t = k\Delta t$. Grid-point p is a 5-tuple $(x, y, t, I, \varepsilon)$ element, where $(p.x, p.y)$ denotes the Cartesian coordinate of a point on the spatial grid at time $p.t$, $p.I$ is the integral estimate for the rp-integral at p and $p.\varepsilon$ is the error estimate. The integral estimate $p.I$ is a 3-tuple (ϕ, A_X, A_Y) element which represents the scalar and vector potentials on p. The moments computed from all time steps are stored linearly on the device memory as a 2D array $M[1..N_t, 1..N_X N_Y]$ in row major order such that moments deposited on a grid-point located in i^{th} row and j^{th} column of the spatial grid during the particle deposition stage at $t_k = k\Delta t$ is stored in $M[k, (iN_X + j)]$, for all integers i, j and k in the range $0 \leq i \leq N_X, 0 \leq j \leq N_Y$, and $0 \leq k < N_t$, respectively. In the description below, a subregion is identified by the record $([a, b], p)$, where p denotes a grid-point on the spatial grid at time step k and $[a, b]$ denotes the limits of integration along outer dimension of rp-integral at p. The two-phase approach is an extension of the GPU-accelerated multidimensional numerical integration algorithm described in Appendix A.1.

RP-QUADRATURE$(([a, b], p), \tau, M)$
1 $(I', \varepsilon') \leftarrow$ RP-QUADRULE$(([a, b], p), \tau, M)$
2 $H \leftarrow \emptyset$
3 PUSH$(H, (p, [a, b], I', \varepsilon'))$
4 **while** $\varepsilon' > \tau$
5 $\quad (([a, b], p), I, \varepsilon) \leftarrow$ POP(H)
6 $\quad m \leftarrow \frac{a+b}{2}$
7 $\quad (I_l, \varepsilon_l) \leftarrow$ RP-QUADRULE$(([a, m], p), \tau, M)$
8 $\quad (I_r, \varepsilon_r) \leftarrow$ RP-QUADRULE$(([m, b], p), \tau, M)$
9 $\quad I' \leftarrow I' - I + I_l + I_r$
10 $\quad \varepsilon' \leftarrow \varepsilon' - \varepsilon + \varepsilon_l + \varepsilon_r$
11 $\quad$ PUSH$(H, (([a, m], p), I_l, \varepsilon_l))$
12 $\quad$ PUSH$(H, (([m, b], p), I_r, \varepsilon_r))$
13 **return** (I', ε')

The procedure RP-PHASEONE, RP-PHASETWO and RP-QUADRATURE is identical to QUADRATURE-PHASEONE, QUADRATURE-PHASETWO and SEQUENTIAL-CUHRE, respectively, from Appendix A.1. However, instead of using CUHRE method

Grid Resolution ($N_X \times N_Y$)	64×64	128×128	256×256
Double Precision Performance (GFlops/sec)	60	61	61
Arithmetic Intensity (Flops/DRAM byte)	0.30	0.31	0.31
Effective Bandwidth (GB/sec)	73	75	76
Global Load Efficiency	29%	30%	32%
Global Load Transactions per Request	7.8	8.0	8.1
Warp Execution Efficiency	60%	75%	77%
L1-cache Global Hit Rate	46%	40%	38%
L2-cache Hit Rate	99.6%	97.3%	90%

Table 3: Performance of Two-Phase-RP-Kernel for computing retarded potentials in a beam dynamics simulation with 100000 particles and for different grid resolutions on NVIDIA Tesla K40 GPU.

for numerical integration as described in the appendix, we use repeated one dimensional integral to compute rp-integral estimates, where outer integral is calculated using adaptive quadrature and inner integral using Newton-Cotes rules. In particular, procedure RP-Quadrature implements this repeated one-dimensional integration method to solve rp-integral, where outer integral is computed using adaptive Simpson's rule and inner integral using three-point Newton-Cotes rule. Furthermore, the procedure C-Rule is replaced with RP-QuadRule $(([a, b], p), \tau, M)$, which calculates Simpson's quadrature rule estimates, I and ε, along a subregion $S = \{(r', \theta') \in \mathbb{R}^2 \mid a \leq r' \leq b, \theta^{(p)}_{\min}(r') \leq \theta' \leq \theta^{(p)}_{\max}(r')\}$, for rp-integral at a grid-point p, where I is the integral estimate and ε is the error estimate. In other words, procedure RP-QuadRule calculates Equation 15 from Section 3.1.2. We omit the pseudocode for RP-QuadRule, as it is identical to the standard Simpson's rule (three point Newton Cotes rule) [21]. A more comprehensive review of the two-phase algorithm, its implementation and performance analysis on GPU architectures can be found in [5].

Performance Analysis and Limitations

Table 3 illustrates the double-precision floating-point performance for computing retarded potentials at a particular time step of a beam dynamics simulation with 100000 particles and varying grid resolution on NVIDIA Tesla K40 GPU with global

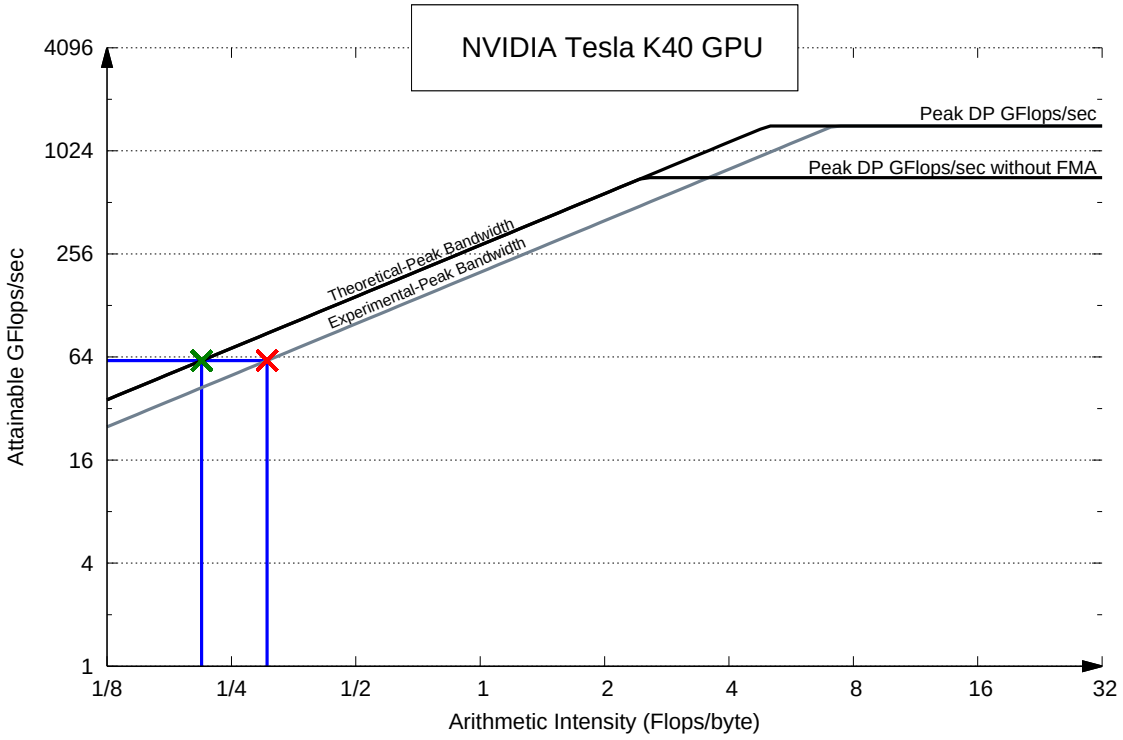


Figure 18: Roofline model analysis for Two-Phase-RP-Kernel on NVIDIA Tesla K40 GPU.

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 illustrated in this section are measured using NVIDIA profiler [62], and a detailed description about each metric and its relation to kernel's performance is illustrated in Appendix B.2. In this study, for convenience, we shall refer to the kernel implementing GPU specific code from Compute-Potentials procedure of the two-phase algorithm as Two-Phase-RP-Kernel.

Roofline Model Analysis - Figure 18 shows the performance of Two-Phase-RP-Kernel illustrated on the Roofline model for K40 GPU. A detailed description about Roofline plot and its use to bound the performance of GPU kernels is illustrated in Appendix B.1. The performance achieved by Two-Phase-RP-Kernel for the

simulation with 100000 particles and 128 × 128 grid resolution is 61 GFlops/sec (see Table 3), and it is indicated by a blue horizontal line in Figure 18. The point where this blue horizontal line intersects the bandwidth ceiling marks the achieved arithmetic intensity. In particular, the green and red X marks achieved arithmetic intensity for Two-Phase-RP-Kernel when the system being modeled delivers a memory bandwidth of $BW_{\text{Theoretical-Peak}} = 288$ GB/sec and $BW_{\text{Experimental-Peak}} = 200$ GB/sec, respectively. Table 3 illustrates the arithmetic intensity (with the bandwidth assumed to be $BW_{\text{Experimental-Peak}}$) achieved by Two-Phase-RP-Kernel measured for different simulation configuration. We notice that arithmetic intensity achieved for the kernel is approximately 0.31 Flops/DRAM byte-accessed when the memory the system being modeled delivers a memory bandwidth of $BW_{\text{Experimental-Peak}}$. It is clear from Figure 18 and the achieved arithmetic intensity in Table 3 that the kernel is memory-bound and it performs poorly because it does not make good use of the available bandwidth, which, due to low arithmetic intensity of the implementation, is the main bottleneck.

Branch Divergence - The warp execution efficiency Two-Phase-RP-Kernel is far less than 100% for all simulation configurations, as illustrated in Table 3. This indicates that the kernel implementation has large number divergent branches that results in poor utilization of the GPU's hardware resources, thereby leading to poor execution performance.

Memory Performance - The memory performance of Two-Phase-RP-Kernel on GPU is analyzed using profiler metrics - *Global load efficiency*, *Global load transaction per requests*, *Cache hits*, etc. These metrics and its relation to the kernel's performance is described in Appendix B.2. The following key observations about the memory performance of Two-Phase-RP-Kernel are inferred from the metrics in Table 3:

1. Global load efficiency is far less than 100% which indicates scattered and non-coalesced memory access, and such accesses waste off-chip bandwidth by over-fetching unnecessary data. Typically, caching in L2 only (non-caching mode) is enabled to reduce such over-fetch. However, as illustrated in [6], this kernel has poor global load efficiency even with non-caching mode indicating high bandwidth waste in both caching and non-caching mode.

2. The global load transactions per request is approximately 4X times greater than the ideal value; in other words, the global load transaction is replayed 8 times for each global memory load. This shows that substantial number of memory request from the kernel are non-coalesced that results in transaction replays.

3. Global memory requests from the kernel almost always hits in L2-cache which is evident by near 100% L2-cache hit rate. This indicates that (i) kernel exhibits high data locality and/or (ii) the problem fits entirely in L2-cache. Even though nearly 100% of the memory request is serviced from L2-cache, which in practice has a bandwidth much greater than $BW_{\text{Theoretical-Peak}}$, the achieved or effective bandwidth is still far less than $BW_{\text{Theoretical-Peak}}$.

It is clear from the above observations that Two-Phase-RP-Kernel performs poorly because of irregular, and non-coalesced global memory accesses, which, together with low arithmetic intensity of the kernel, is the main performance bottleneck. Moreover, poor data locality and massively multithreaded nature of the kernel with little cache capacity per thread results in high L1-cache miss rates. Such behavior significantly over-fetches off-chip data for the application, wasting memory bandwidth, and on-chip storage. It is, therefore, clear that optimizing the global memory access and improving the effective bandwidth is most important for the best performance of the beam dynamics simulation on GPUs. On the other hand, this also suggests that other optimization methods such as improving the floating point performance through the optimization of the arithmetic instructions or hiding the latency of global memory access through maximizing the multiprocessor occupancy are not effective.

3.3.2 HEURISTICS ALGORITHM

The heuristics algorithm presented in this section use two different heuristics to maximize data reuse and to minimize divergence in the parallel implementation of compute retarded potentials stage of the beam dynamics simulation on GPUs. The heuristics used in the algorithm helps in effectively utilizing the bandwidth at different levels of the memory hierarchy by coalescing the memory accesses, and maximize the data reuse (i.e. increase the probability of a data block to be reused more extensively before the block is replaced) by improving data locality. Furthermore, the

algorithm uses techniques that effectively balance the workload among threads, minimize their divergence, and reduce the overall floating point operations required to compute the retarded potentials when compared to prior implementations.

The procedure COMPUTE-POTENTIALS-H implements the heuristics based compute retarded potential stage of the simulation that approximates rp-integral to a error tolerance of τ at all points $p \in V$, where p is a grid-point on the spatial grid at time step k with simulation time $t = k\Delta t$ and $|V| = N_X N_Y$. Grid-point p is a 5-tuple $(x, y, t, I, \varepsilon)$ element, where $(p.x, p.y)$ denotes the Cartesian coordinate of a point on the spatial grid at time step $p.t$, $p.I$ is the integral estimate for the rp-integral at p and $p.\varepsilon$ is the error estimate. The integral estimate $p.I$ is a 3-tuple (ϕ, A_x, A_y) element which represents the scalar and vector potentials on p. The moments computed from all time steps are stored linearly on the device memory as a 2D array $M[1..N_t, 1..N_X N_Y]$ in row major order such that moments deposited on a grid-point located in i^{th} row and j^{th} column of the spatial grid during the particle deposition stage at $t_k = k\Delta t$ is stored in $M[k, (iN_X + j)]$, for all integers i, j and k in the range $0 \leq i \leq N_X, 0 \leq j \leq N_Y$, and $0 \leq k < N_t$, respectively.

In COMPUTE-POTENTIALS-H, lines 1-2 initialize the estimates $p.I$ and $p.\varepsilon$ to 0 for all points $p \in V$. Next, RP-CLASSIFIER procedure at line-3 partitions the set of grid points into a small number of clusters based on the data locality properties of the corresponding rp-integrals, using standard clustering techniques like k-means. More formally, given a set of grid points V and an integer $k > 0$, RP-CLASSIFIER partitions the $|V|$ grid points into $k(\leq |V|)$ classes, $C = \{C_1, C_2, \ldots, C_k\}$, such that the sum of distance functions of each point in the cluster to its center is minimum,

$$\arg\min_{C} \sum_{i=1}^{k} \sum_{p \in C_i} d(p, \mu_i) \tag{17}$$

where μ_i is the center of cluster C_i, and the distance function $d(p, \mu_i)$ is the measure of similarity between the grid-point $p \in C_i$ and its cluster center μ_i for all integer i in range $0 < i \leq k$. A value of zero to the distance function implies strong similarity, and the similarity decreases with the increase in distance value. In RP-CLASSIFIER, two points u and v ($u, v \in V$, $u \neq v$) are considered to have strong similarity when the rp-integral evaluations at u and v exhibit high data locality against one another, in such a case, $d(u, v) \approx 0$. One of the simplest measure of data locality is the number of overlapping memory locations required to evaluate rp-integral at u and v. Consequently, the distance function $d(u, v)$ can be expressed as the inverse of

data locality *i.e.* inverse of the number of overlapping memory locations required to evaluate rp-integral at u and v.

COMPUTE-POTENTIALS-H(k, V, τ, M)

```
1  for each grid-point p ∈ V in parallel
2      p.I ← 0, p.ε ← 0
3  C ← RP-CLASSIFIER(V, k) // implements k-means clustering
4  for each class c ∈ C in parallel
5      P ← RP-INTEGRALPARTITION(c, t, M)
6      for each grid-point p ∈ c in parallel
7          for i = 0 to P.length − 1
8              a ← P[i], b ← P[i + 1]
9              (I, ε) ← RP-QUADRULE(p, [a, b], τ, M)
10             if ε < τ
11                 p.I ← p.I + I
12                 p.ε ← p.ε + ε
13             else
14                 LIST-INSERT(L, ([a, b], p))
15 for each ([a, b], p) ∈ L in parallel
16     (I, ε) ← RP-QUADRATURE(p, [a, b], τ, M)
17     p.I ← p.I + I
18     p.ε ← p.ε + ε
```

The main motivations behind such locality based classification is that when a set of rp-integrals that exhibit high data locality between each other are mapped to parallel CUDA threads with one-to-one correspondence, they exhibit strong inter-thread locality. Such data locality among the threads can be exploited by grouping them into one or more thread blocks, where the memory performance is improved due to the benefit form data locality by using L1-cache or shared-memory. In addition, these thread blocks when scheduled on a single core or simultaneously on different cores can also benefit from locality using shared L2-cache. However, accurate prediction of data locality without actually evaluating the integral is non-trivial and computationally challenging due to the data-dependent, irregular, and statically unpredictable (i.e. unknown until run time) memory accesses patterns of rp-integral

evaluations at different grid points. As a result, we propose a heuristic to approximate $d(u, v)$, referred to as *locality heuristic*. The key observation behind locality heuristic is that while computing rp-integral on grid points, there is a significant reuse of data for two nearby grid points. The reuse of data for two grid points is inversely proportional to the distance between the two grid points. More formally,

> **Locality heuristic:** *Given two points u and v, where $u, v \in V$ and $u \neq v$, the data locality or the number of overlapping memory access between rp-integral evaluation at u and v is inversely proportional to the Euclidean distance between them.*

We performed empirical analysis using sequential beam dynamics simulation for different input configurations to validate the above assumed heuristics, and all our experiments confirm the locality heuristic. Consequently, in RP-Classifier, Euclidean distance is used as the distance function to measure the data locality between two points. (Note that other distance metrics such as L1-distance will do fine also.)

Next, for each class $c \in C$, we require to calculate the rp-integral estimate at all grid points $p \in c$. Typically, numerical approximation of rp-integral at each grid-point results in a partition that is independent of the rp-integral at other points, and methods such as adaptive quadrature are often used to compute these partitions, as is the case in [5]. Once the partition is computed, integral estimate is calculated using Equation 14. However, in our proposed approach, a single unique partition per class that combines the partition of rp-integral at all grid points of that particular class is calculated using heuristics instead of using traditional adaptive quadrature methods on each point.

The main motivations for calculating such unique partition for a group of points instead of individual grid-point is that it eliminates the need for adaptive quadrature or data-dependent control-flow on each integral evaluation, which, as illustrated in [3, 4, 5], is the main performance bottleneck for such adaptive computations on SIMD architectures. The procedure RP-IntegralPartition implements this heuristics approach, where for each class $c \in C$, it generates a unique partition $P[1..P.length]$ that denotes a rp-integral partition along the outer integration domain (r'-domain). Ideally, P should be a combination of the partitions generated by rp-integral at all $p \in c$. However, computing such partition per class instead of individual grid-point is computationally challenging due to the data-dependent, and irregular control-flow

behavior of different rp-integrals. Moreover, it requires prior understanding of the integrand being integrated. As a result, we propose a heuristic to compute the partition for each class $c \in C$, referred to as *control-flow heuristic*. The key observation behind control-flow heuristic is that while working on a set of grid points, we can use partial results of the same set of grid points at an earlier time step. More formally,

> **Control-flow heuristic:** *Given a class of grid points $c \in C$, where c is one of the k-classes generated by* RP-Classifier. *Then, the unique partition required to evaluate the rp-integral at all $p \in c$ is approximated by combining the partition for rp-integral at the class center and the partition for same set of grid points from earlier time step.*

In the heuristics, partition for class center is computed sequentially using traditional adaptive quadrature method, whereas the partition for each grid-point from previous time step is computed during the Compute-Potentials-H procedure of that particular time step. We performed empirical analysis using sequential beam dynamics simulation for different input configurations to validate the above assumed heuristics, and all our experiments confirm the control-flow heuristics. Like locality heuristics, when rp-Integral computations at all grid points of a particular class are mapped to parallel CUDA threads with one-to-one correspondence, they exhibit uniform flow of computation. Such uniform control-flow will eliminate the branch divergence and load balancing complexities that are introduced when adaptive quadrature method is used, as is the case in [5]. Moreover, uniform control-flow, combined with inter-thread data locality from locality heuristics will further increase the memory performance when the threads accessing same cache line are grouped in a warp.

Once the procedure RP-IntegralPartition at line-5 outputs the partition $P[1..P.length]$ for a class $c \in C$, then the rp-integral estimate for all grid points $p \in c$ is calculated as

$$p.I = \sum_{i=1}^{P.length-1} \int_{P[i]}^{P[i+1]} dr' \int_{\theta^{(p)}_{\min}(r')}^{\theta^{(p)}_{\max}(r')} f_p(r', \theta', t') d\theta' \tag{18}$$

where for all i, integral estimate along the subregion $[P[i], P[i+1]]$ is computed using Simpsons quadrature rule when the error estimate is less than τ; otherwise, adaptive quadrature method is used to compute the integral. Ideally, for all $p \in c$, all the subregions in Equation 18 should have error estimate less than τ, as defined

by the control-flow heuristics. However, there may exist some grid-point $p' \in c$ for which the error estimate along the subregion $[P[j], P[j+1]]$ is larger than τ, for some j in range $0 < j < P.length$ (*i.e.* the heuristic fails for some subregion). We hypothesize that this particular scenario will seldom occur, and even if it should, the number of subregions that fail the heuristics will be insignificant. However, in order to maintain the correctness of the integral estimate, we use adaptive quadrature on these subregions.

The procedure RP-QuadRule at line-9 of Compute-Potentials-H denotes the Simpsons quadrature rule computation for rp-integral at p along the subregion $[a, b]$, and it outputs a pair I and ε which correspond to the integral and error estimate for that particular subregion. Lines 10-13 accumulates the integral estimate when error estimate for the subregion is less than τ and all other subregions are inserted into a list L (line-14) for later application using parallel adaptive quadrature method (line 15-18). The computation between line 15-18 is identical to that of RP-PhaseTwo procedure from Section 3.3.1. A more comprehensive review of the heuristics algorithm, its implementation and performance analysis on GPU architectures can be found in [6].

Performance Analysis and Limitations

Table 4 illustrates the double-precision floating-point performance for computing retarded potentials at a particular time step of a beam dynamics simulation with 100000 particles and varying grid resolution 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 $Q = 1$nC, and the rp-integral at all grid points are approximated to a error tolerance of $\tau = 10^{-6}$. The performance metrics illustrated in this section are measured using NVIDIA profiler [62], and a detailed description about each metric and its relation to kernel's performance is illustrated in Appendix B.2. In this study, for convenience, we shall refer to the kernel implementing GPU specific code from Compute-Potentials-H procedure of the heuristics algorithm as Heuristics-RP-Kernel.

Roofline Model Analysis - Figure 19 shows the performance of Heuristics-RP-Kernel illustrated on the Roofline model for K40 GPU. A detailed description

Grid Resolution ($N_X \times N_Y$)	64×64	128×128	256×256
Double Precision Performance (GFlops/sec)	401	420	440
Arithmetic Intensity (Flops/DRAM byte)	2.00	2.10	2.22
Effective Bandwidth (GB/sec)	398	446	476
Global Load Efficiency	105%	115%	120%
Global Load Transactions per Request	1.8	1.8	1.8
Warp Execution Efficiency	92%	96%	97%
L1-cache Global Hit Rate	100%	99%	99%
L2-cache Hit Rate	100%	100%	97%

Table 4: Performance of Heuristics-RP-Kernel for computing retarded potentials in a beam dynamics simulation with 100000 particles and for different grid resolutions on NVIDIA Tesla K40 GPU.

about Roofline plot and its use to bound the performance of GPU kernels is illustrated in Appendix B.1. The performance achieved by Heuristics-RP-Kernel for the simulation with 100000 particles and 128×128 grid resolution is 420 GFlops/sec (see Table 4), and it is indicated by a blue horizontal line in Figure 19. The point where this blue horizontal line intersects the bandwidth ceiling marks the achieved arithmetic intensity. In particular, the green and red X marks achieved arithmetic intensity for Heuristics-RP-Kernel when the system being modeled delivers a memory bandwidth of $BW_{\text{Theoretical-Peak}} = 288$ GB/sec and $BW_{\text{Experimental-Peak}} = 200$ GB/sec, respectively. Table 4 illustrates the arithmetic intensity (with the bandwidth assumed to be $BW_{\text{Experimental-Peak}}$) achieved by Heuristics-RP-Kernel measured for different simulation configuration. We notice that arithmetic intensity achieved for the kernel is approximately 2.10 Flops/DRAM byte-accessed when the memory the system being modeled delivers a memory bandwidth of $BW_{\text{Experimental-Peak}}$, which is 7X more than Two-Phase-RP-Kernel. Furthermore, the effective bandwidth for the kernel is greater than the experimental peak, $BW_{\text{Experimental-Peak}}$. The increase in effective bandwidth indicates that Heuristics-RP-Kernel is effective in utilizing the caches to filter the number of accesses that go to memory, thereby increasing the arithmetic intensity.

Branch Divergence - The warp execution efficiency Heuristics-RP-Kernel is nearly 100% for all simulation configurations, as illustrated in Table 4. This indicates

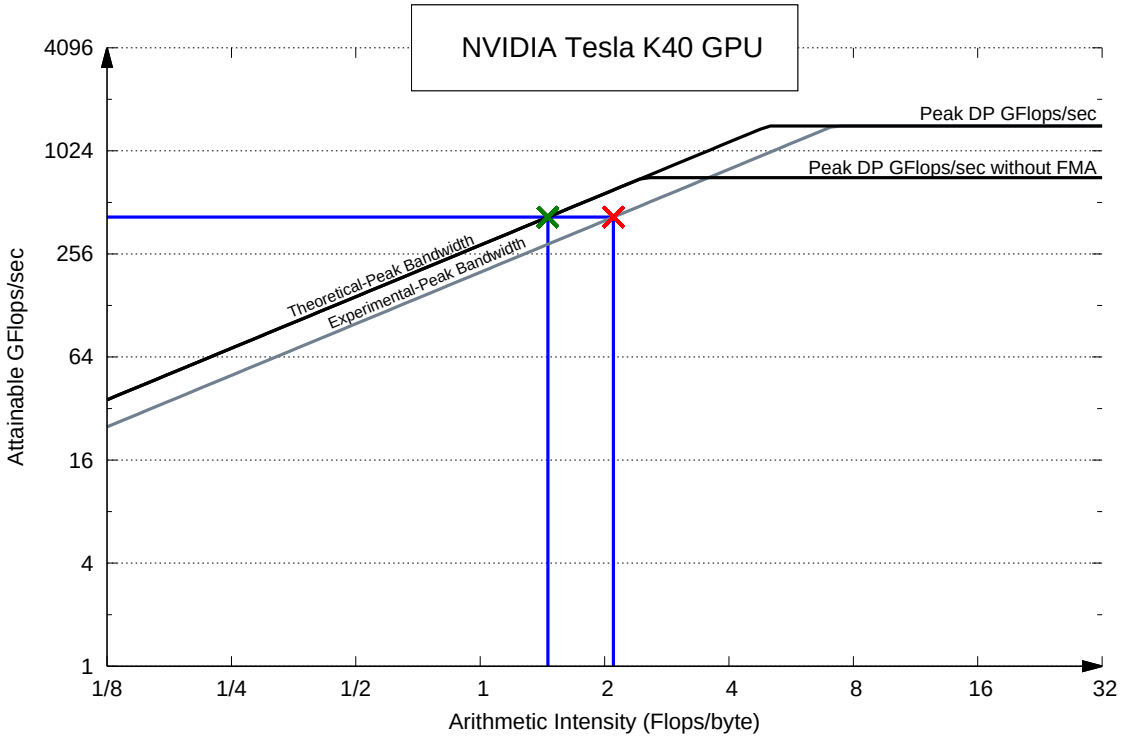


Figure 19: Roofline model analysis for HEURISTICS-RP-KERNEL on NVIDIA Tesla K40 GPU.

that GPU kernel has fewer divergent branches and has near uniform control-flow. In other words, locality heuristics in the proposed algorithm is effective in reducing the control-flow irregularity among threads and minimizing their divergence, which, as illustrated in Section 2.2 and [60, 59], is one of the most important performance consideration in programming for CUDA-capable GPU architectures.

Memory Performance - The memory performance of HEURISTICS-RP-KERNEL on GPU is analyzed using profiler metrics - *Global load efficiency*, *Global load transaction per requests*, *Cache hits*, etc. These metrics and its relation to the kernel's performance is described in Appendix B.2. The following key observations about the memory performance of HEURISTICS-RP-KERNEL are inferred from the metrics in Table 4:

- Global load efficiency for the kernel, which is the ratio of number of bytes requested by the kernel to number of bytes transferred, is greater than 100.

This indicates that, on average, the load requests of multiple threads in a warp are fetched from the same memory address.

- Global load transactions per request value is much closer to the ideal value of 2.0 for transactions with 8-byte words. This shows that most of the memory requests from within a warp are coalesced, and the memory accesses are within at most two cache lines.

- Global memory requests from the kernel almost always hits in L2-cache which is evident by near 100% L2-cache hit rate. This shows that the kernel fits entirely in L2-cache, and the behavior is identical to that of Two-Phase-RP-Kernel.

- The L1-cache hit rate is nearly 100%. Typically, increased cache hit from a kernel reduces the DRAM bandwidth which contributes to the increase in effective bandwidth of that particular kernel, as is the case for Heuristics-RP-Kernel.

It is clear from the above observations that the performance of Heuristics-RP-Kernel is substantially improved when compared to that of Two-Phase-RP-Kernel. In particular, heuristcs based algorithm is effective in improving data locality, maximizing data reuse, coalescing the memory accesses, and in increasing the effective bandwidth of the kernel. Additionally, effective utilization of the bandwidth at different levels of the memory hierarchy results in an increase in arithmetic intensity, which is shown in Figure 19. The heuristics algorithm and its implementation on GPU is currently the fastest known method for high-fidelity computation of retarded potentials in a charged particle beam dynamics simulations.

CHAPTER 4

EFFICIENT PARALLEL SIMULATION ON GPUS

In Section 3.2, we showed that the distribution of work and data in the accurate computation of retarded potentials 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. As a direct consequence of these properties, obtaining high-performance in such algorithms is extremely challenging, and to this end, we have developed two parallel algorithms to accurately calculate the retarded potentials at each time step of the simulation using GPUs, Two-phase Algorithm and Heuristics Algorithm. Two-phase Algorithm is illustrated in Section 3.3.1 and it targets equal distribution of work over processor to reduce control-flow irregularity. Heuristics Algorithm is illustrated in Section 3.3.2 and it uses heuristics to maximize data reuse and to balance the workload among threads, thereby reducing both control-flow and memory access divergence on GPUs. Both 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 retarded potentials 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 chapter presents the use of predictive analytics and forecasting techniques to optimize the computation of retarded potentials 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 retarded potentials at a future time step based on the computations and access patterns observed from earlier time steps. The remainder of the chapter is organized as follows. Section 4.1 presents the machine learning approach to model irregular data access patterns in the computation of retarded potentials 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 retarded potentials using predictive analytics and its implementation on GPU architecture is illustrated in Section 4.2. Section 4.3 validates the parallel implementation and then illustrates the performance of parallel algorithm on NVIDIA Tesla K40 GPU.

4.1 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 using application-specific supervised learning algorithms described in this section. First, we outline the representation of data access pattern in the numerical approximation of rp-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 rp-integral evaluations at a future time step using the prediction model learned at an earlier time step.

4.1.1 REPRESENTATION OF DATA ACCESS PATTERN

The computation of rp-integral at a grid-point $p \in V_k$ during time step k requires referencing data from the 2D grids of moments computed from one or more earlier time steps. In particular, calculating rp-integral along the subregion $S_{ic\Delta t,(i+1)c\Delta t}$ requires referencing data from the 2D grids computed during time steps $k-i$, $k-i-1$, and $k-i-2$ (*i.e.*, 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 $k < N_t$. Further, when $S_{ic\Delta t,(i+1)c\Delta t}$ is subdivided into n_i partitions then rp-integral evaluation within that subregion will result in αn_i memory references to each of the three data grids, where α 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 rp-integral evaluation using the partition generated along subregions, $S_{ic\Delta t,(i+1)c\Delta t}$, for integers i such that $0 \leq i < N_t$. More formally, data access pattern observed during rp-integral evaluation at a grid-point $p \in V_k$ is represented by a list, $[n_0^{(p)}, n_1^{(p)}, \ldots, n_{N_t-1}^{(p)}]$, where $n_i^{(p)}$ denotes the number of partitions along the subregion $S_{ic\Delta t,(i+1)c\Delta t}$ 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 $\alpha(n_i^{(p)} + n_{i-1}^{(p)} + n_{i-2}^{(p)})$.

Data access patterns are extracted during regular execution of the beam dynamics simulation with negligible overhead except for additional storage required to log the access patterns. These 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.

4.1.2 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 $[n_0^{(q)}, n_1^{(q)}, \ldots, n_{N_t}^{(q)}]$ 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.

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 illustrated in Section 3.3.2 and in [6] is an intuitive choice.

4.1.3 FORECASTING MEMORY ACCESS

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 algorithm illustrated in Section 4.2, 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.

4.1.4 FORECASTING 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, $\langle r_0^{(p)}, r_1^{(p)}, \ldots, r_n^{(p)} \rangle$, along the outer dimension. Adaptive quadrature is traditionally used to compute such partitions, which, as illustrated in [3, 4], 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-14. 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 $[n_0^{(p)}, n_1^{(p)}, \ldots, n_{N_t}^{(p)}]$ corresponding to a grid-point $p \in V_k$, the forecasting algorithm computes a partition list, $\langle r_0^{(p)}, r_1^{(p)}, \ldots, r_m^{(p)} \rangle$, 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 $S_{ic\Delta t,(i+1)c\Delta t}$ is divided into $(n_i^{(p)} - 1)$ finer subregions of equal size (*i.e.*, $n_i^{(p)}$ partitions along $S_{ic\Delta t,(i+1)c\Delta t}$), for all integers i in range $0 < i < N_t$. This generates a global partition of size $\sum_{i=0}^{N_t} n_i^{(p)}$ 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 Section 3.3.2, 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 $S_{ic\Delta t,(i+1)c\Delta t}$ 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 $\sum_{i=0}^{N_t} n_i^{(p)}$ on the entire integration region $S_{0,R(p)}$.

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 4.2 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.

4.2 PARALLEL ALGORITHM

The procedure COMPUTE-POTENTIALS-ML 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 grid for a given time step. The procedure takes input k, V, τ, g, and D, where k is the current time step of simulation, V is a set of grid points on the 2D grid at k^{th} time step such that $|V| = N_X N_Y$, τ 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, $(x, y, t, I, \varepsilon, access\_pattern, partition)$, where $(p.x, p.y)$ denote the Cartesian coordinate of the grid-point on the 2D grid at time step k, $p.t$ is the simulation time of the corresponding time step, $p.I$ is the rp-integral estimate, $p.\varepsilon$ is the rp-integral error estimate, $p.access\_pattern$ is a list containing the data access pattern for rp-integral computation, and $p.partition$ holds a list containing the partition for rp-integral computation.

The procedure COMPUTE-POTENTIALS-ML works as follows. Line 1-4 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 4.1.4 to convert access pattern forecast to rp-integral partition. Next, RP-CLUSTERING procedure at line-5 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,

$$\arg\min_{C} \sum_{i=1}^{m} \sum_{p \in C_i} ||p.access\_pattern - \mu_i||^2 \tag{19}$$

where μ_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.

Compute-Potentials-ML(k, V, τ, g, D)

```
1  for each grid-point p ∈ V in parallel
2      p.I ← 0, p.ε ← 0
3      p.access_pattern ← g(p.x, p.y, p.t)
4      p.partition ← Compute-Partition(p.access_pattern)
5  C ← RP-Clustering(V)
6  L ← ∅
7  for each cluster c ∈ C in parallel
8      P ← ∅
9      for each grid-point p ∈ c in parallel
10         P ← Merge-Lists(P, p.partition)
11     for each grid-point p ∈ c in parallel
12         L′ ← Compute-RP-Integral(p, P, τ, D)
13         L ← Merge-Lists(L, L′)
14 for each ([a, b], p) ∈ L in parallel
15     (I, ε, P, A) ← RP-Quadrature(([a, b], p), τ, D)
16     p.access_pattern ← Merge-Lists(p.access_pattern, A)
17     p.partition ← Merge-Lists(p.partition, P)
18     p.I ← p.I + I
19     p.ε ← p.ε + ε
20 g ← Online-Learning(V, g)
```

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 at u and 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 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 Compute-Potentials-ML is similar to the one used in Section 3.3.2, however, algorithm in Section 3.3.2 uses heuristics to measure the data reuse between two grid points, whereas in here, 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 4.3 shows that this approach is effective in improving the memory performance.

The *for* loop in lines 7-13 evaluates rp-integral at all grid points using the partition approximated in line-4. First, for each cluster $c \in C$, line-8 initializes a list P, and for each grid-point $p \in c$, the *for* loop in line 9 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 9-10 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-11, which aids in minimizing the thread divergence when computations of this loop are mapped to GPU threads.

The procedure Compute-RP-Integral(p, P, τ, D) 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 τ, and the integral and error estimates along each subregion is approximated using Simpson's quadrature rule. We use an auxiliary procedure RP-QuadRule$([a, b], p, D)$ to compute Simpson's quadrature rule estimates, (I, ε), along a subregion $[a, b]$ for rp-integral at a point p, where I is the integral estimate

and ε is the error estimate. We omit the pseudocode for RP-QuadRule, as it is identical to the standard Simpson's quadratue rule with the inner integral approximated using Newton-Cotes formulae, as illustrated in [5]. In i^{th} iteration of the *for* loop in Compute-RP-Integral procedure, integral and error estimates along an integration region $S_{P[i],P[i+1]}$ is calculated by calling RP-QuadRule. Next, when the error estimate returned from RP-QuadRule is less than τ, both integral and error estimates are accumulated to the input 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 τ 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 Compute-RP-Integral procedure.

```
Compute-RP-Integral(p, P, τ, D)
 1  L ← ∅
 2  for i = 0 to P.length − 1
 3       a ← P[i], b ← P[i + 1]
 4       (I, ε) ← RP-QuadRule(([a, b], p), τ, D)
 5       if ε < τ
 6            p.I ← p.I + I
 7            p.ε ← p.ε + ε
 8       else
 9            List-Insert(L, ([a, b], p))
10  p.partition ← P
11  update p.access_pattern with the observed data access pattern
12  return L
```

In the pseudocode for Compute-Potentials-ML, the method Compute-RP-Integral is called on for each grid-point $p \in c$ inside the *for* loop at line-11, 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 τ. 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 14-19. We use

the procedure RP-Quadrature 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, RP-Quadrature$([a, b], p, \tau, D)$ outputs a tuple, (I, ε, P, A), where I and ε 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 RP-Quadrature is merged with the corresponding grid-point's access pattern and partition, respectively, that is seen during Compute-RP-Integral procedure. Furthermore, rp-integral estimates from adaptive quadrature are accumulated to the grid-point's global estimates.

Online-Learning(V, g)

1 $X \leftarrow \emptyset$
2 $Y \leftarrow \emptyset$
3 **for** each grid-point $p \in V$
4 List-Insert$(X, (p.x, p.y, p.t))$
5 List-Insert$(Y, p.access_pattern)$
6 update the predictor function $g : X \rightarrow Y$ using supervised learning on the inputs X and Y
7 **return** g

Next, in Online-Learning procedure, 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-ML procedure during the next time step.

4.2.1 GPU IMPLEMENTATION

Initialization steps between lines 1-4 are implemented on CPU, where the *for* loop is parallelized using OpenMP, and the grid-point attributes: *access_pattern* and *partition*, are stored only in CPU memory. The procedure RP-Clustering implementing the *k-means* clustering algorithm is also implemented on CPU, using *scikit-learn* library [65]. Further, in the implementation of RP-Clustering, we choose number of clusters to be $m = \max(N_X, N_Y)$, and since *k-means* algorithm prefers clusters of approximates similar size, each cluster size is approximately $\min(N_X, N_Y)$.

Lines 7-13 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-7 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-11 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-11 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 14-19 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-QUADRATURE procedure in parallel and independent of other threads. This implementation is identical to the globally adaptive algorithm illustrated in Section 3.3.1. 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 CPU cores to speed up the training process.

4.3 EVALUATION AND EXPERIMENTAL RESULTS

Simulation experiments for studying beam dynamics and performance analysis of the parallel implementation of COMPUTE-POTENTIALS-ML 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*), unless specified otherwise. Initial

distribution for all the simulations are generated by Monte Carlo sampling of N particles with a total charge of beam bunch $Q = 1$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 [62], and the results are averaged over multiple runs. For convenience, we shall refer to the kernel implementing GPU specific code from COMPUTE-POTENTIALS-ML procedure as PREDICTIVE-RP-KERNEL.

4.3.1 VALIDATION OF PARALLEL SIMULATION

The correctness and accuracy of beam dynamics simulation depends on the fidelity of PREDICTIVE-RP-KERNEL 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 [28, 48]. The parallel implementation presented here is benchmarked against the analytical results described in [28, 48] for the parameters of the Linac Coherent Light Source (LCLS) bend [47]: bend radius $R_0 = 25.13$ m, $\theta_b = 11.4°$, longitudinal rms beam size $\sigma_s = 50$ μm, emittance $\epsilon = 1$ nm, total beam charge $Q = 1$ nC. From Figure 20, 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 21, which shows the mean-square error, defined as

$$\epsilon = \frac{1}{N} \sum_{i=1}^{N} \left(F_i - F_i^{\text{exact}} \right)^2 ,$$

with N the number of particles, F_i the computed force and F_i^{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 [76].

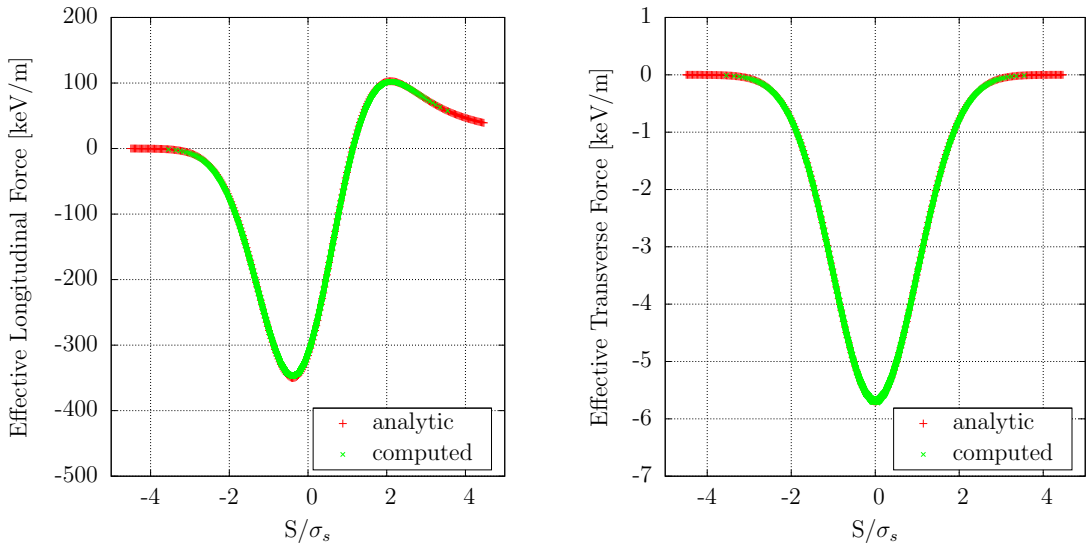


Figure 20: Analytic versus computed effective longitudinal (left) and transverse (right) forces for the LCSL bend [47]: $N = 1000000$ particles on a 128×128 grid, bend radius $R_0 = 25.13$ m, $\theta_b = 11.4°$, longitudinal rms beam size $\sigma_s = 50$ μm, emittance $\epsilon = 1$ nm, and total beam charge of $Q = 1$nC.

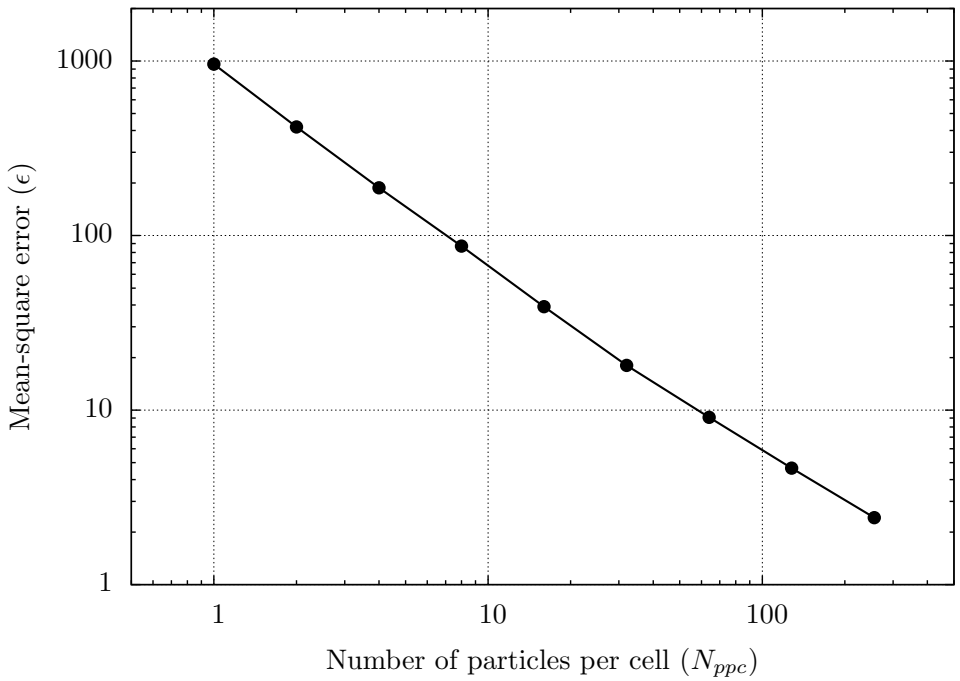


Figure 21: Mean-square error for the longitudinal force, as defined in the text, as a function of the number of particles per cell $N_{\rm ppc} = N/N_{\rm grid}$, for a fixed grid of 128×128 (or $N_{\rm grid} = 128^2$).

Grid Resolution ($N_X \times N_Y$)	64×64	128×128	256×256
Double precision performance (GFlops/sec)	460	480	485
Arithmetic Intensity (Flops/DRAM byte)	2.30	2.40	2.43
Effective Bandwidth (GB/sec)	532	567	580
Global Load Efficiency	135%	150%	160%
Global Load Transactions per Request	1.9	1.9	1.9
Warp Execution Efficiency	97%	99%	99%
L1-cache Global Hit Rate	100%	100%	99%
L2-cache Hit Rate	100%	100%	100%

Table 5: Performance of PREDICTIVE-RP-KERNEL for computing retarded potentials in a beam dynamics simulation with 100000 particles and for different grid resolutions on NVIDIA Tesla K40 GPU.

4.3.2 PERFORMANCE ANALYSIS

Table 5 illustrates the double-precision floating-point performance of PREDICTIVE-RP-KERNEL at a particular time step of a beam dynamics simulation with 100000 particles and varying grid resolution on NVIDIA Tesla K40 GPU. The results from Table 5 are used to provide a quantitative analysis on the effects of using predictive analytics and forecasting techniques in improving the performance of compute retarded potentials stage of the beam dynamics simulations on GPUs. The performance metrics illustrated in this section are measured using NVIDIA profiler [62], and a detailed description about each metric and its relation to kernel's performance is illustrated in Appendix B.2.

Roofline Model Analysis

Figure 22 shows the performance of PREDICTIVE-RP-KERNEL illustrated on the Roofline model for K40 GPU. A detailed description about Roofline plot and its use to bound the performance of GPU kernels is illustrated in Appendix B.1. The performance achieved by PREDICTIVE-RP-KERNEL for the simulation with 100000 particles and 128×128 grid resolution is 480 GFlops/sec (see Table 5), and it is indicated by a blue horizontal line in Figure 22. The point where this blue horizontal line intersects the bandwidth ceiling marks the achieved arithmetic intensity. In particular, the green X (1.67 Flops/byte) and red X (2.40 Flops/byte) marks

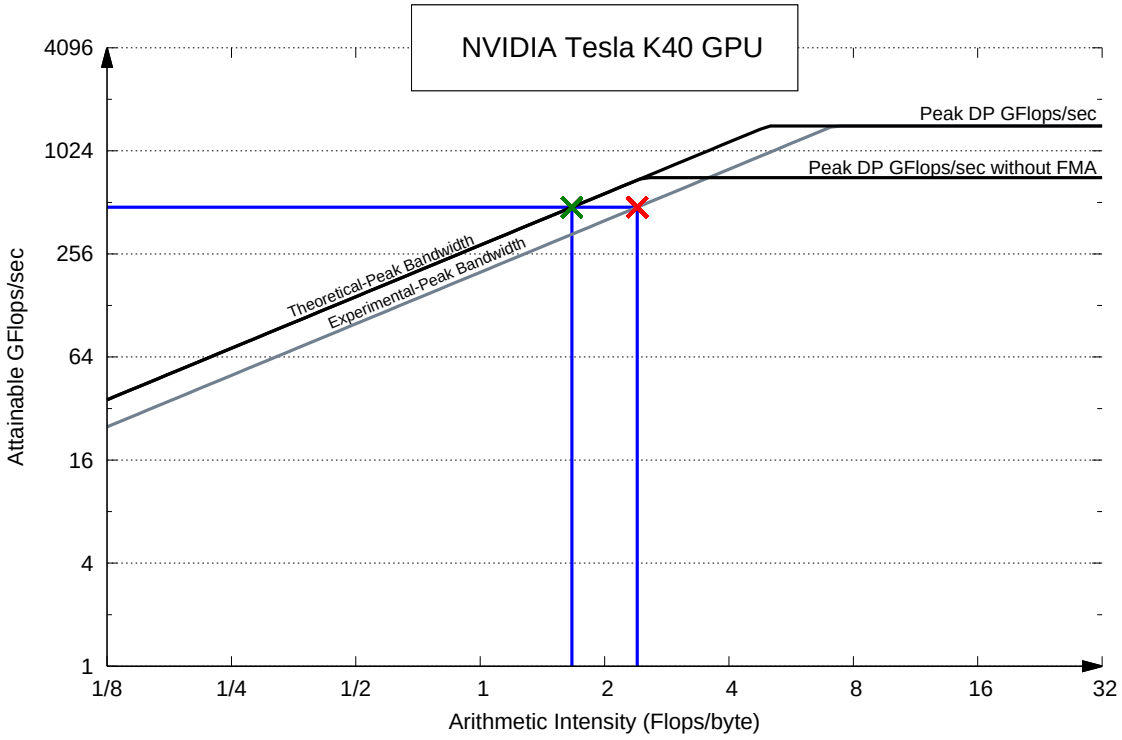


Figure 22: Roofline model analysis for Predictive-RP-Kernel on NVIDIA Tesla K40 GPU.

achieved arithmetic intensity for Predictive-RP-Kernel when the memory bandwidth attained by the kernel is assumed to be $BW_{\text{Theoretical-Peak}} = 288$ GB/sec and $BW_{\text{Experimental-Peak}} = 200$ GB/sec, respectively.

Figure 23 compares the performance of Predictive-RP-Kernel against existing two kernels - Two-Phase-RP-Kernel and Heuristics-RP-Kernel, illustrated on the Roofline model for K40 GPU. The vertical lines indicate the achieved arithmetic intensity when the attained memory bandwidth is assumed to be $BW_{\text{Experimental-Peak}}$ and the X marks performance attained for that particular GPU kernel. It is clear from Figure 22 and Figure 23 that the parallel algorithm and its implementation on GPUs based on predictive analytics and forecasting technique has sufficiently high arithmetic-intensity when compared to the previous two implementations. In particular, Predictive-RP-Kernel delivers 480 GFlops/sec of double precision floating-point performance on K40 GPU and this translates to achieved arithmetic intensity of 2.40 Flops/DRAM-byte accessed which is 8X and 1.2X

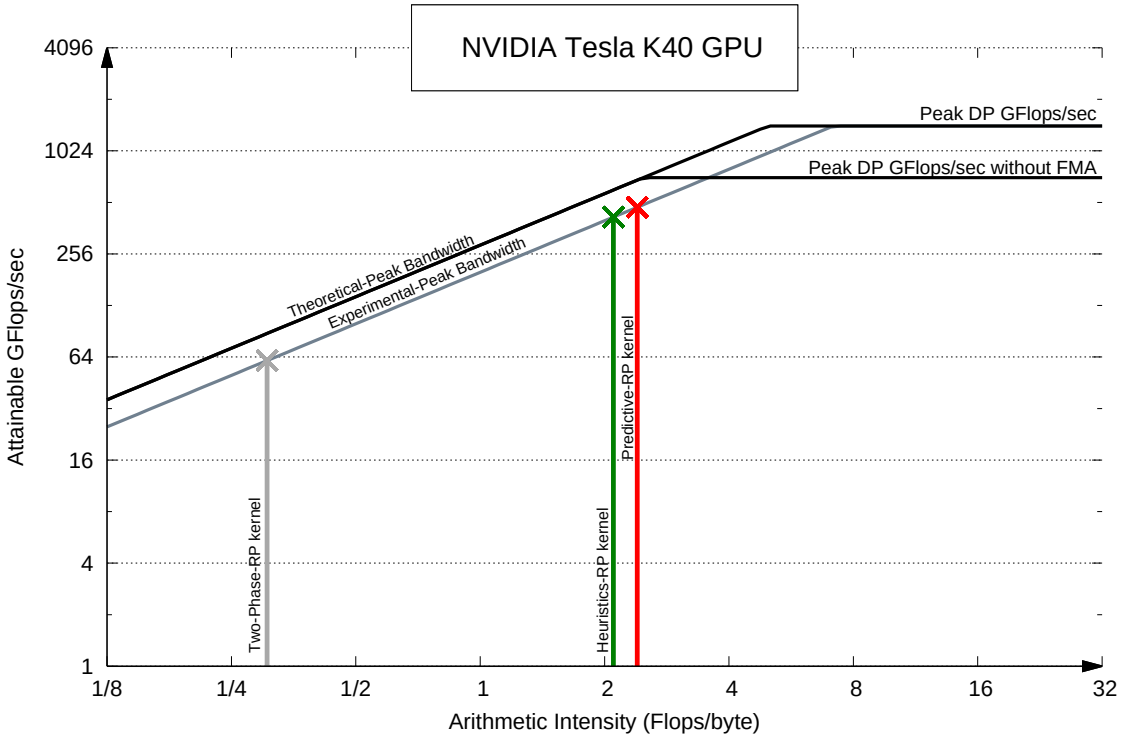


Figure 23: Roofline model analysis of Predictive-RP-Kernel (red line) compared against Heuristics-RP-Kernel (green line) and Two-Phase-RP-Kernel (grey line) on NVIDIA Tesla K40 GPU.

more than the Two-Phase-RP-Kernel and Heuristics-RP-Kernel, respectively. Furthermore, the effective bandwidth for Predictive-RP-Kernel kernel is greater than $BW_{\text{Experimental-Peak}}$ (see Table 5). The increase in effective bandwidth indicates that the kernel implementation is effective in utilizing the caches to filter the number of accesses that go to memory, thereby increasing the arithmetic intensity.

Branch Divergence

The warp execution efficiency for Predictive-RP-Kernel is nearly 100%, illustrated in Table 5. 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 Section 2.2 and [60, 59], is one of the most important performance consideration in programming CUDA-capable GPU architectures. Moreover, such higher

value of warp execution efficiency for PREDICTIVE-RP-KERNEL indicates that this kernel implementation is better at utilizing the GPU device to its full potential when compared to the other two kernels .

Memory Performance

The memory performance of PREDICTIVE-RP-KERNEL on GPU is analyzed using profiler metrics - *Global load efficiency*, *Global load transaction per requests*, *Cache hits*, etc. These metrics and its relation to the kernel's performance is described in Appendix B.2. The following analysis about the memory performance of PREDICTIVE-RP-KERNEL is inferred from studying different profiler metrics illustrated in Table 5 -

- Global load efficiency of PREDICTIVE-RP-KERNEL is greater than 100%, which indicates that on average, the load requests of multiple threads in a warp are fetched from the same memory address and are also coalesced. In other words, implementation is effective in taking advantage of the memory coalescing feature of the GPU's architecture.

- Global load transactions per request for PREDICTIVE-RP-KERNEL is much closer to the ideal value of 2.0 for transactions with 8-byte words. This shows that most of the memory requests from within a warp are coalesced, and the memory accesses are within at most two cache lines.

- Global memory requests from PREDICTIVE-RP-KERNEL almost always hits in L2-cache which is evident by near 100% L2-cache hit rate. This indicates that kernel exhibits high data locality and/or the problem fits entirely in L2-cache.

- 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 for the GPU implementation of PREDICTIVE-RP-KERNEL on Tesla K40 GPU is substantially improved when compared to TWO-PHASE-RP-KERNEL and HEURISTICS-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. Moreover, optimizations based on predictive analytics and forecasting is effective in coalescing the memory accesses and in increasing the effective bandwidth of the kernel. This improves the overall utilization of the bandwidth at different levels of the memory hierarchy, thereby increasing the arithmetic intensity, as shown in Figure 22.

Speedup

Table 6 and Table 7 illustrates the performance of compute retarded potentials stage of the simulation using Predictive-RP-Kernel compared against Two-Phase-RP-Kernel and Heuristics-RP-Kernel for different simulation configurations. In Table 6, GPU time refers to the kernel execution time on the GPU together with the time spent in memory operations (allocations, initialization, and memory copy between CPU and GPU), Clustering time and Online-training time refers to the execution time of procedures RP-Clustering and Online-Learning, respectively, and the Overall time is the total execution time of the compute retarded potentials stage of the simulation.

The results indicate that depending on the grid resolution and the number of particles in the simulation, computing retarded potentials using Predictive-RP-Kernel achieves a speedup gain of up to 6.4X and 2.8X compared to the Two-Phase-RP-Kernel and Heuristics-RP-Kernel, respectively. Further, the split execution time in Table 6 shows that the time spent in online-training and clustering are negligible when compared to the GPU time. In other words, the time spent in using the access pattern forecast to optimize the computation to thread mapping and in formulating the runtime decisions are negligible when compared to the performance benefit achieved from it.

Number of particles	Grid Resolution	Execution Time (sec.)			
		Clustering	GPU (Compute and Memory)	Online-training and Prediction	Overall
100000	64×64	0.08	1.04	0.05	1.17
	128×128	0.40	4.77	0.10	5.27
	256×256	1.50	53.40	0.40	55.30
1000000	64×64	0.08	0.93	0.05	1.06
	128×128	0.16	3.41	0.10	3.67
	256×256	1.18	30.10	0.40	31.68

Table 6: Execution time of compute retarded potentials stage of the simulation using Predictive-RP-Kernel for different simulation configurations on NVIDIA Tesla K40 GPU.

Number of Particles (N)	Grid Resolution ($N_X \times N_Y$)	Two-Phase-RP Execution Time (sec.)	Heuristics-RP Execution Time (sec.)	Predictive-RP		
				Execution Time (sec.)	Speedup with respect to	
					Two-Phase-RP	Heuristics-RP
100000	64×64	3.55	1.75	1.17	3.0	1.5
	128×128	28.56	14.20	5.27	5.5	2.7
	256×256	293.63	149.55	55.30	5.3	2.7
1000000	64×64	2.97	1.40	1.06	2.9	1.3
	128×128	22.82	10.15	3.67	6.2	2.8
	256×256	201.29	88.35	31.68	6.4	2.8

Table 7: Speedup of Predictive-RP-Kernel compared against Two-Phase-RP-Kernel and Heuristics-RP-Kernel for different simulation configurations on NVIDIA Tesla K40 GPU.

CHAPTER 5

EFFICIENT PARALLEL SIMULATION ON HETEROGENEOUS ARCHITECTURE

This chapter presents the parallel algorithm and its implementation on heterogeneous architectures for the retarded potentials computation stage of the beam dynamics simulation which uses machine learning algorithms to create and distribute sub-problems to different PUs of the underlying heterogeneous architecture. In particular, supervised learning algorithm illustrated in the previous chapter and in [7] is used to adaptively model and track irregular data access patterns in the computation of retarded potentials at each time step of the simulation to anticipate the future data access patterns. Then, at some future time step, access pattern forecast is used to approximately divide the original problem of evaluating the rp-integral into multiple smaller sub-problems that are defined only on a subset of data (a subproblem is also referred to as *tasks* in this chapter). These tasks created from subdividing all the $N_X N_Y$ rp-integrals of that particular time step are distributed between multiple PUs of the heterogeneous architectures such that each PU is scheduled with tasks localized to a portion of the entire data at a given time. In particular, data required for computing retarded potentials is first partitioned into multiple smaller data blocks, and these data blocks are mapped to PUs dynamically such that no two PU shares the same data block. Then, tasks localized to a data block is scheduled on a PU based on data block to PU mapping which is determined dynamically. This approach of task creation and distribution based on the data access pattern has the advantage of assigning tasks with different memory footprint to different PUs which improves the memory performance on the heterogeneous architectures by ensuring that only a portion of data is required to be transferred to a particular PU.

The remainder of this chapter is organized as follows. Section 5.1 presents the algorithm to compute retarded potentials on heterogeneous architecture composed of one or more PUs of distinct hardware characteristics and processing capabilities. In Section 5.1, we first provide a brief overview of the data layout (Section 5.1.1) and the access pattern representation (Section 5.1.2) considered for designing efficient parallel

algorithm on heterogeneous architecture. Next, in Section 5.1.3, we illustrate the task creation algorithm that divides the original problem of evaluating the rp-integral at a grid-point into multiple smaller sub-problems that are defined only on a subset of data. Section 5.1.4 presents the algorithm for calculating retarded potentials using the sub-problems created from task creation algorithm, where the set of sub-problems are distributed across different PUs of the underlying architectures based on their processing capabilities. Finally, Section 5.2 presents the implementation performance of the algorithm on two different heterogeneous architectures and show that the use of machine learning algorithms is effective in partitioning and mapping the workload to different PUs such that all the PUs are best utilized and combined to deliver good aggregate performance.

5.1 HETEROGENEOUS ALGORITHM

5.1.1 DATA PARTITION

The heterogeneous algorithm illustrated here divides the original problem of evaluating the rp-integral at different grid points into multiple smaller sub-problems or tasks that are defined only on a subset of data, and it later maps these tasks to different PUs based on their memory access and computational load requirement. In order to create such tasks, we first partition the memory logically into multiple smaller data blocks. In particular, the array of data grids containing the moments from all the time steps is logically partitioned into multiple smaller data blocks, B_0, B_1, ..., $B_{N_t/m}$ where a data block B_i corresponds to the data generated between time steps $mi - 1$ to $m(i + 1)$ ($i.e. D_{mi-1}$ to $D_{m(i+1)}$), for all integers i in range 0 to N_t/m, and $m \geq 3$ is an integer which defines the granularity of data partition. The value of m also controls the granularity of the sub-problems generated by the task creation algorithm, which is illustrated in later sections of this chapter. It is important to note that the rp-integral approximation algorithm is such that even the smallest possible sub-problem require data from at-least three data grids which is why the value of m must be at-least 3.

The task creation and distribution algorithm presented in this chapter use the partition among the data grids to identify tasks that are localized around individual data blocks. Then, the distribution algorithm maps these tasks to PUs such that each PU computes the tasks localized to one or more data-block, where a data-block

is not shared between multiple PUs. The motivation for distributing the tasks based on their data block requirement is that such mapping has the advantage of assigning tasks with different memory footprint to different PUs. This ensures that only a portion of data is required to be transferred to a particular PU and this is crucial for improving the memory performance on heterogeneous architectures.

5.1.2 DATA ACCESS PATTERN

The representation of data access pattern forecast in the numerical approximation of rp-integrals is identical to the one described in Section 4.1 of Chapter 4. In particular, data access pattern for rp-integral evaluation at a grid-point $p \in V_k$ is represented by a list, $[n_0^{(p)}, n_1^{(p)}, \ldots, n_{N_t-1}^{(p)}]$, where $n_i^{(p)}$ denotes the number of partitions along the subregion $S_{ic\Delta t,(i+1)c\Delta t}$ 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 $\alpha(n_i^{(p)} + n_{i-1}^{(p)} + n_{i-2}^{(p)})$.

5.1.3 TASK CREATION

The task creation algorithm divides the original problem of evaluating the rp-integral at a grid-point into multiple smaller sub-problems that are defined only on a subset of data. Formally, suppose the current time step of the simulation is k and g_{k-1} be the predictor function learned by the supervised learning algorithm using data access patterns observed from one or more time step up to time step $k-1$. Then, data access pattern for rp-integral evaluation at a grid-point $p \in V_k$ for time step k is approximated as, $g_{k-1}(p) = [n_0^{(p)}, n_1^{(p)}, \ldots, n_{N_t}^{(p)}]$. This forecast is used to approximate the partition $P^{(p)} = \langle r_0^{(p)}, r_1^{(p)}, \ldots, r_n^{(p)} \rangle$, along the outer dimension, which is required to numerically approximate rp-integral at p within the required error tolerance (illustrated in Section 4.1.3). Next, for each grid-point $p \in V_k$, the partition array $P^{(p)}$ is sliced into m_p non-overlapping sub-arrays, $P_1^{(p)}, P_2^{(p)}, \ldots, P_{m_p}^{(p)}$, such that rp-integral computation using the partitions from a sub-array $P_i^{(p)}$ requires data from at-most one data block B_j for some integer $0 < j < N_t/m$ and $0 < i \leq m_p$. This divides the original problem of integrating along the region $S_{0,R(p)}$ using the partition array $P^{(p)}$ into m_p sub-problems, where a sub-problem refers to the rp-integral computation using a partition sub-array $P_i^{(p)}$ for some integer $i \leq m_p$ (a sub-problem is also referred to as a *task*). Once subdivided, the solution to the original problem is given by sum of rp-integral estimates of the individual sub-problems.

Each sub-problem or task is represented using a tuple, $\langle p, P, j, M \rangle$, where p denotes a grid-point on the 2D spatial grid at the current time step, P denotes a partition sub-array such that rp-integral at p using the partitions from P is localized to at-most one data block, an integer j that identifies the data block required for computing the sub-problem (*i.e.* B_j), and M denotes the data access pattern expected from the computation of rp-integral at p using partitions from P (Note that the access pattern M falls into data block B_j).

5.1.4 ALGORITHM TO COMPUTE RETARDED POTENTIALS

The procedure COMPUTE-POTENTIALS-HT implements the second step of the four step beam dynamics simulation algorithm on heterogeneous architecture where it approximates the rp-integral at all grid points on a 2D grid for a given time step. The procedure takes input k, V, τ, g, and D, where k is the current time step of simulation, V is a set of grid points on the 2D grid at k^{th} time step such that $|V| = N_X N_Y$, τ 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, $(x, y, t, I, \varepsilon, access\_pattern, partition)$, where $(p.x, p.y)$ denote the Cartesian coordinate of the grid-point on the 2D grid at time step k, $p.t$ is the simulation time of the corresponding time step, $p.I$ is the rp-integral estimate, $p.\varepsilon$ is the rp-integral error estimate, $p.access\_pattern$ is a list containing the data access pattern for rp-integral computation, and $p.partition$ holds a list containing the partition for rp-integral computation.

The procedure COMPUTE-POTENTIALS-HT works as follows. Line 2-7 initializes different attributes of the grid-point object. In particular, for each grid-point $p \in V$, line-3 initializes integral and error estimates to 0, line-4 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-5 calls a procedure that implements the algorithm described in Section 4.1.4 to convert access pattern forecast to rp-integral partition. Line-6 divides the original problem of calculating rp-integral at a grid-point p using the predicted partition array $p.partition$ into multiple smaller sub-problems such that each sub-problem is localized to at-most one data block B_j, for some integer j (a sub-problem is also referred to as a *task*). The procedure

CREATE-TASKS implements this algorithm which is described in Section 5.1.3.

COMPUTE-POTENTIALS-HT(k, V, τ, g, D)

```
1  T ← ∅
2  for each grid-point p ∈ V in parallel
3       p.I ← 0, p.ε ← 0
4       p.access_pattern ← g(p.x, p.y, p.t)
5       p.partition ← COMPUTE-PARTITION(p.access_pattern)
6       T′ ← CREATE-TASKS(p)
7       T ← MERGE-LISTS(T, T′)
8  bins ← TASK-BINNING(T)
9  for each available processing unit in parallel
10      while bins is not empty
11           get list of tasks T′ from a bin i for some integer i based on the bin
             weight and the processing unit where it is scheduled
12           V′ ← ∅
13           for each task ⟨p, j, P, M⟩ ∈ T′
14                q ← p
15                q.I ← 0, q.ε ← 0
16                q.access_pattern ← M
17                q.partition ← P
18                LIST-INSERT(V′, q)
19           copy data generated between time steps mi − 1 to m(i + 1) from D to D′
20           move D′ to the device memory
21           RP-COMPUTATION-PU(k, V′, τ, g, D′)
22           update the estimates from V′ to the global list of grid points V
23 g ← ONLINE-LEARNING(V, g)
```

The procedure CREATE-TASKS for a grid-point p returns a list of tasks created by subdividing the rp-integral at p into multiple smaller sub-problems or tasks. The individual tasks list corresponding to each grid-point $p \in V$ is merged to the global list T. Next, TASKS-BINNING procedure at line-8 implements a binning algorithm which partitions the list of tasks into multiple disjoint clusters based on the data block required to compute the tasks. Formally, given the set of tasks T and an integer $k = N_t/m$, where each task is a tuple $\langle p, j, P, M \rangle$, the binning algorithm

partitions the $|T|$ tasks into at-most k disjoint clusters or bins, $C = \{c_1, c_2, \ldots, c_k\}$, such that the tasks in a bin c_i require data from data-block B_i. The TASKS-BINNING procedure returns a pool of tasks, *bins*, which contain the tasks grouped into multiple bins based on their data-block requirement. These tasks in *bins* data structure is shared between multiple PUs of the underlying heterogeneous architectures. Further, each bin in *bins* is assigned a weight which denotes the collective computational workload of the tasks in that particular bin. The weight for a bin is calculated by reducing all the access vectors of the tasks in that bin. This weight is equivalent to the total memory reference made by the tasks of that particular bin.

RP-COMPUTATION-PU(k, V, τ, g, D)

```
1  C ← RP-CLUSTERING(V)
2  L ← ∅
3  for each cluster c ∈ C in parallel
4       P ← ∅
5       for each grid-point p ∈ c in parallel
6            P ← MERGE-LISTS(P, p.partition)
7       for each grid-point p ∈ c in parallel
8            L' ← COMPUTE-RP-INTEGRAL(p, P, τ, D)
9            L ← MERGE-LISTS(L, L')
10 for each ([a, b], p) ∈ L in parallel
11      (I, ε, P, A) ← RP-QUADRATURE(([a, b], p), τ, D)
12      p.access_pattern ← MERGE-LISTS(p.access_pattern, A)
13      p.partition ← MERGE-LISTS(p.partition, P)
14      p.I ← p.I + I
15      p.ε ← p.ε + ε
```

Lines 9-22 implements the task distribution algorithm where tasks from *bins* are mapped to PUs such that each PU computes the tasks from one or more bins, where a particular bin is not shared between multiple PUs. The motivation for distributing the tasks based on their data block requirement is that such mapping has the advantage of assigning tasks with different memory footprint to different PUs. This ensures that only a portion of data is required to be transferred to a particular PU instead of moving the entire data which is crucial for improving the memory performance. Each PU dynamically selects the bins from the shared pool,

one at a time according to the weights assigned to them and then computes the tasks of that particular bin in parallel. GPU and Xeon Phi selects the bins starting with larger weight and work towards the bins with smaller weight, and CPU selects the bins starting with smaller weight and work towards the bins with larger weight. This sort of scheduling results in tasks with heavy computational load to be scheduled on accelerators and the tasks with lighter computational load to be scheduled on CPUs. The main motivation for such weighted scheduling is because accelerators offer larger raw compute power than the CPUs.

Each PU implements RP-Computation-PU procedure which evaluates the tasks assigned to them from the bin (*i.e.* evaluates rp-integral for the sub-problems from the bin). The procedure RP-Computation-PU is similar to the pseudo-code between lines 5 - 19 of Compute-Potentials-ML. 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 RP-Computation-PU procedure during the next time step.

5.2 PERFORMANCE RESULTS

The performance analysis of the heterogeneous implementation of compute retarded potentials stage of the simulation is carried out on two different heterogeneous system -

- Machine-G - Heterogeneous architecture composed of multi-core CPU with four NVIDIA Tesla K40 GPUs
  - *Multi-core CPU* - Dual socketed Intel® Xeon® CPU E5-2650 v3 @ 2.30GHz with 10 cores per socket and supports 2 threads per core, making a total of 20 CPU cores for the multi-core CPU platform.
  - *GPU* - The Tesla K40 used here is a GK110B GPU-processor based on the popular Kepler micro-architecture [61]. The GK110B processor in K40 offers 12 GB of GDDR5 on-board memory with a peak memory bandwidth of 288 GB/sec, and it contains 15 streaming multiprocessors (SMs) each with 192 single-precision CUDA cores and 64 double-precision units clocked at 745 MHz. These cores in SMs collectively delivers a peak floating-point performance of 4.29 Tflops and 1.43 Tflops in single-precision and

double-precision, respectively. The global memory accesses in all the GPU kernels are configured to be cached in both L1 and L2 (commonly called the *Caching mode*), unless specified otherwise.

- MACHINE-X - Heterogeneous architecture composed of multi-core CPU with two Intel Xeon Phi 5110P coprocessor
  - *Multi-core CPU* - Dual socketed Intel® Xeon® CPU E5-2670 v2 @ 2.50 GHz with 10 cores per socket and supports 2 threads per core, making a total of 20 CPU cores for the multi-core CPU platform.
  - *Xeon Phi* - The Xeon Phi 5110P used in this study is based on the Knights Corner architecture. Each 5110P coprocessor consist of 60 cores and offers 8GB of GDDR5 on-board memory with a peak memory bandwidth of 320 GB/sec. The cores collectively delivers a peak floating-point performance of 2.02 Tflops and 1.01 Tflops in single-precision and double-precision, respectively.

The global memory access for the implementation on GPU architecture is configured to be cached in both L1 and L2 (commonly called the *Caching mode*), unless specified otherwise. The initial distribution for all the simulations used to analyze the performance of heterogeneous implementation is 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 results presented in this section are generated by compiling the heterogeneous code using GCC 5.4 compiler for multi-core portion of the code, Intel's ICC 16 compiler for Xeon Phi portion of the code, and NVCC compiler with CUDA 8.0 environment for GPU portion of the code.

5.2.1 PERFORMANCE ON DIFFERENT ARCHITECTURES

GPU-only Execution Performance

Table 8 illustrates the performance of parallel implementation of compute retarded potentials stage of the simulation on MACHINE-G using only GPUs for different simulation configuration with varying number of Tesla K40 GPUs. The speedup reported in Table 8 is measured by comparing the multi-GPU execution against single GPU. We notice that performance scales near linearly with the increase in number

Number of particles	Grid Resolution	One GPU Time (sec.)	Two GPUs Time (sec.)	Two GPUs Speedup	Four GPUs Time (sec.)	Four GPUs Speedup
100000	64 × 64	1.03	0.62	1.7	0.47	2.2
	128 × 128	4.71	2.68	1.8	1.51	3.1
	256 × 256	55.96	29.61	1.9	16.48	3.4
1000000	64 × 64	0.94	0.58	1.6	0.45	2.1
	128 × 128	3.31	1.87	1.8	1.12	3.0
	256 × 256	34.53	18.14	1.9	10.15	3.4

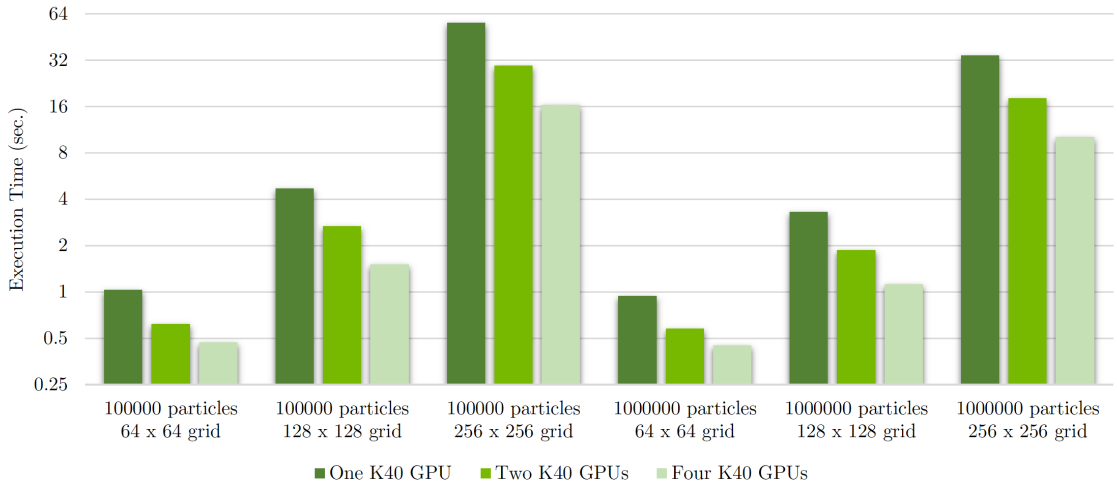


Table 8: Execution time of the parallel implementation of compute retarded potentials stage of the simulation on MACHINE-G in GPU-only execution mode for different simulation configurations with varying number of Tesla K40 GPUs.

of GPUs and achieves up to 1.9X and 3.4X speedup with two and four GPUs, respectively. This shows that the task creation and distribution algorithm is effective in partitioning the computational load nearly uniformly between multiple GPUs with minimal overhead.

Xeon Phi-only Execution Performance

Table 9 illustrates the performance of parallel implementation on MACHINE-X using only Xeon Phis for different simulation configuration with varying number of KNC Xeon Phi accelerators. The speedup reported in Table 9 is measured by comparing the execution on multiple Xeon Phi against single Xeon Phi. We notice that performance scales near linearly with the increase in number of Xeon Phi and achieves up to 1.7X speedup with two Xeon Phis. This shows that the task creation

Number of particles	Grid Resolution	One Xeon Phi Time (sec.)	Two Xeon Phis Time (sec.)	Two Xeon Phis Speedup
100000	64 × 64	1.73	1.0	1.7
	128 × 128	11.02	6.37	1.7
	256 × 256	149.34	91.73	1.6
1000000	64 × 64	1.54	0.92	1.7
	128 × 128	8.06	4.68	1.7
	256 × 256	93.03	57.03	1.6

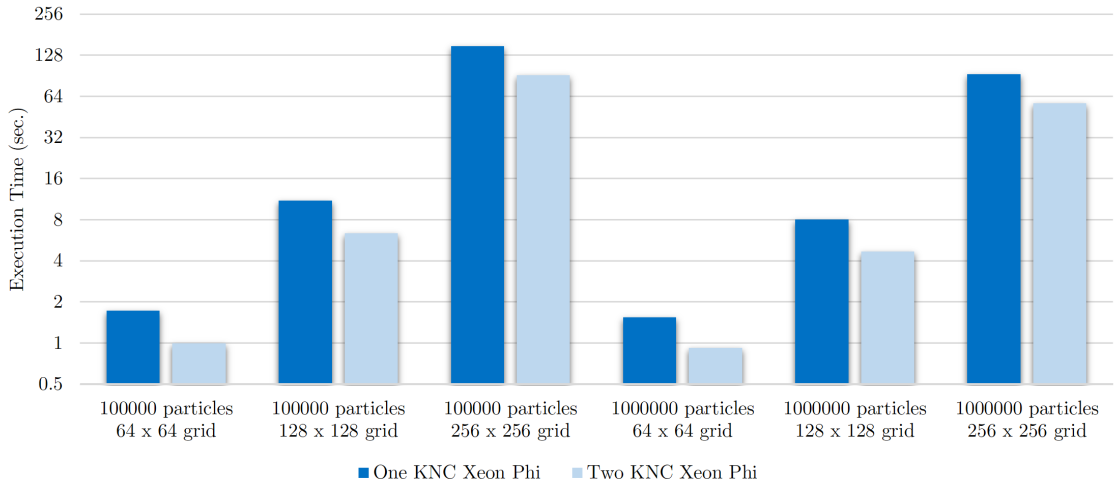


Table 9: Execution time of the parallel implementation of compute retarded potentials stage of the simulation on MACHINE-X in Xeon Phi-only execution mode for different simulation configurations with varying number of KNC Xeon Phi coprocessor.

and distribution algorithm is effective in partitioning the computational load nearly uniformly between multiple Xeon Phis.

Performance Comparison Between Different Architectures

Table 10 illustrates the performance of parallel implementation of compute retarded potentials stage of the simulation on different parallel architectures. The speedup reported for the implementation on accelerators (GPU and Xeon Phi) in Table 10 is calculated by comparing the execution on accelerator against the multi-core CPU execution. It is evident from Table 10 that the implementation on KNC Xeon Phi is up to 2.3X faster than the multi-core CPU implementation, and the GPU implementation on Tesla K40 is up to 5.2X and 2.7X faster than the multi-core CPU and

#. of particles	Grid Resolution	20 CPU cores	Tesla K40 GPU		KNC Xeon Phi	
		Time (sec.)	Time (sec.)	Speedup	Time (sec.)	Speedup
100000	64 × 64	4.00	1.03	3.9	1.73	2.3
	128 × 128	23.50	4.71	5.0	11.02	2.1
	256 × 256	279.97	55.96	5.0	149.34	1.9
1000000	64 × 64	3.57	0.94	3.8	1.54	2.3
	128 × 128	17.20	3.31	5.2	8.06	2.1
	256 × 256	176.15	34.53	5.1	93.03	1.9

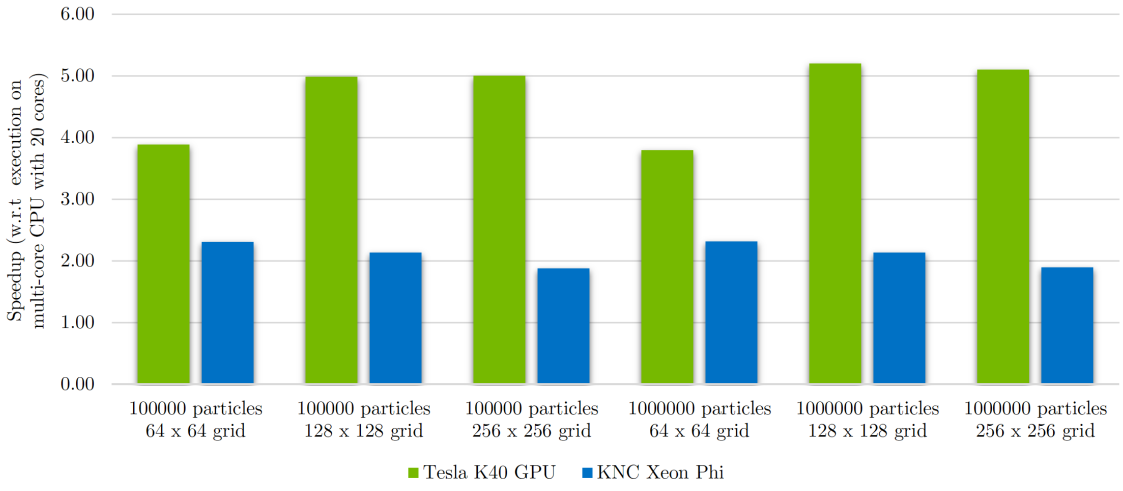


Table 10: Performance comparison of the parallel implementation of compute retarded potentials stage on different HPC architectures - (a) Multi-core CPU with 20 cores (using the CPU from MACHINE-X which is the fastest of the two available multi-core CPUs), (b) Tesla K40 GPU from MACHINE-G, and (c) KNC Xeon-Phi from MACHINE-X.

Xeon Phi implementations, respectively.

5.2.2 PERFORMANCE ON HETEROGENEOUS ARCHITECTURES

Consider a heterogeneous system with n processing units, $P = \{P_1, P_2 \ldots, P_n\}$, and an application A which requires W floating point operations where W represents the computational workload of A. Let t_p and f_p represent the execution time in seconds and the achieved performance in Flops/sec, respectively, for the parallel implementation of A using processing unit p, where $t_p = W/f_p$, for all $p \in P$. Then, the ideal execution time (without any overheads) for the parallel implementation of

A with workload W using a subset of processing units, $S \subset P$, is given by

$$t_{ideal} = \frac{W}{\sum\limits_{p \in S} f_p} = \frac{1}{\sum\limits_{p \in S} \frac{1}{t_p}} \tag{20}$$

We analyze the performance of all our heterogeneous implementations by comparing the execution time observed experimentally (denoted by $t_{observed}$) against the ideal execution time (t_{ideal}).

Table 11 and Table 12 illustrates the performance of the parallel implementation of compute retarded potentials stage of the simulation on MACHINE-G and MACHINE-X, respectively. In particular, Table 11 illustrates the performance with two different heterogeneous system configurations on MACHINE-G: multi-core CPU with one Tesla K40 GPU, and multi-core CPU with all four Tesla K40 GPUs. Likewise, Table 12 illustrates the performance with two different heterogeneous system configurations on MACHINE-X: multi-core CPU with one KNC Xeon Phi, and multi-core CPU with two KNC Xeon Phis. In both these tables, execution time of the implementation using only multi-core CPU is denoted by t_{cpu}. In Table 11, execution time using GPU-only mode with one and four Tesla K40 GPU(s) is denoted by t_{1gpu} and t_{4gpu}, respectively. Similarity, in Table 12, execution time using Xeon Phi-only mode with one and two KNC Xeon Phi(s) is denoted by t_{1phi} and t_{2phi}, respectively. In Table 11 and Table 12, execution time observed on the collective computing environment of the corresponding heterogeneous architectures is denoted by $t_{observed}$, and the ideal execution time on the corresponding heterogeneous architectures is denoted by t_{ideal}, which is calculated using Equation 20.

The efficiency of the heterogeneous implementation on MACHINE-G and MACHINE-X is shown in Figure 24 and Figure 25, respectively. The results indicate that depending on the grid resolution and the number of particles in the simulation, the parallel algorithm for heterogeneous architectures achieves 70-97% execution efficiency. This efficiency is the direct consequence of the effectiveness of heterogeneous algorithm in partitioning the workload efficiently between different PUs of the underlying architecture. In particular, high-efficiency here indicates that the task creation and distribution approach used in the heterogeneous algorithm is effective in breaking down the problem into multiple smaller sub-problems and efficiently mapping them to different PUs based on their individual processing capabilities such that together they deliver a good aggregate performance.

It is important to note that the task creation and distribution approach used in

the current implementation has a overhead associated with creating and maintaining the shared pool of tasks between the PUs. In particular, the constant communication from the shared pool to fetch tasks by each PU contributes to the overhead, and increase in this overhead reduces the achieved execution efficiency. This overhead is more prominent for simulations with smaller resolutions where the computation time is not sufficiently large enough to hide the cost of the overhead, as is the case for simulations with 64×64 grid-resolution. In such scenarios, accelerator only implementation delivers good performance than the corresponding heterogeneous implementation. This is evident from the results in Table 11, where for the simulations with 64×64 grid resolution, the execution on GPU-only mode with four GPUs is faster than the corresponding heterogeneous execution.

Number of particles (N)	Grid Resolution ($N_X \times N_Y$)	Execution Time (sec.)						
		CPU	GPU-only		Heterogeneous Architectures			
		20 cores	One K40	Four K40	20 CPU cores + One K40 GPU		20 CPU cores + Four K40 GPUs	
		t_{cpu}	t_{1gpu}	t_{4gpu}	$t_{observed}$	$t_{ideal} = \frac{t_{cpu}t_{1gpu}}{t_{cpu}+t_{1gpu}}$	$t_{observed}$	$t_{ideal} = \frac{t_{cpu}t_{4gpu}}{t_{cpu}+t_{4gpu}}$
100000	64×64	4.80	1.03	0.47	0.97	0.85	0.51	0.43
	128×128	24.72	4.71	1.51	4.30	3.96	1.50	1.42
	256×256	282.95	55.96	16.48	50.11	46.72	16.42	15.57
1000000	64×64	4.36	0.94	0.45	0.84	0.77	0.50	0.41
	128×128	18.35	3.31	1.12	3.01	2.81	1.11	1.06
	256×256	178.13	34.53	10.15	31.23	28.92	10.12	9.60

Table 11: Performance of the heterogeneous implementation of compute retarded potentials stage of the simulation on Machine-G which is composed of multi-core CPU and four NVIDIA Tesla K40 GPUs.

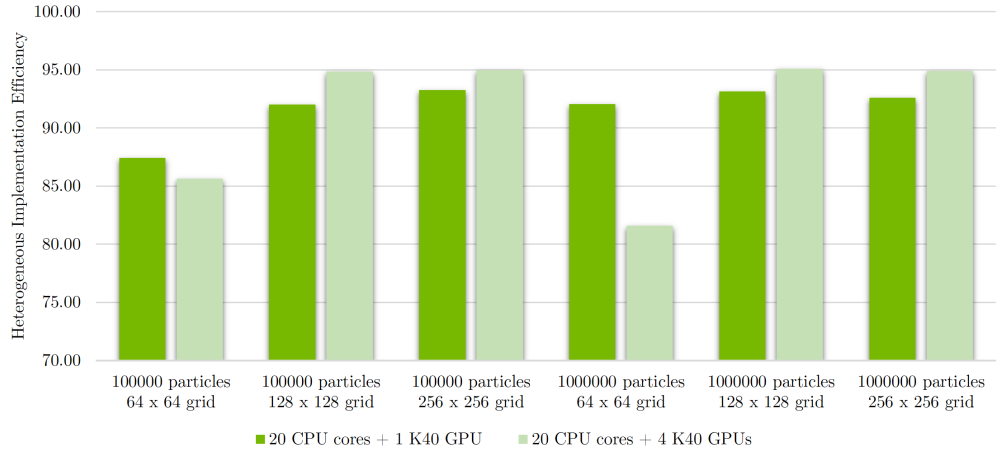


Figure 24: Efficiency of the heterogeneous implementation of compute retarded potentials stage of the simulation on Machine-G which is composed of multi-core CPU and four NVIDIA Tesla K40 GPUs.

Number of particles (N)	Grid Resolution ($N_X \times N_Y$)	Execution Time (sec.)						
		CPU	GPU-only		Heterogeneous Architectures			
		20 cores	One KNC	Two KNC	20 CPU cores + One Xeon Phi		20 CPU cores + Two Xeon Phi	
		t_{cpu}	t_{1phi}	t_{2phi}	$t_{observed}$	$t_{ideal} = \frac{t_{cpu}t_{1phi}}{t_{cpu}+t_{1phi}}$	$t_{observed}$	$t_{ideal} = \frac{t_{cpu}t_{2phi}}{t_{cpu}+t_{2phi}}$
100000	64×64	4.00	1.73	1.00	1.39	1.21	1.10	0.80
	128×128	23.50	11.02	6.37	7.75	7.51	5.81	5.01
	256×256	279.97	149.34	91.73	99.37	97.40	71.13	69.09
1000000	64×64	3.57	1.54	0.92	1.36	1.08	1.05	0.73
	128×128	17.20	8.06	4.68	5.96	5.50	4.61	3.68
	256×256	176.15	93.03	57.03	63.45	60.88	45.53	43.08

Table 12: Performance of the heterogeneous implementation of compute retarded potentials stage of the simulation on Machine-X which is composed of multi-core CPU and two KNC Xeon Phi coprocessor.

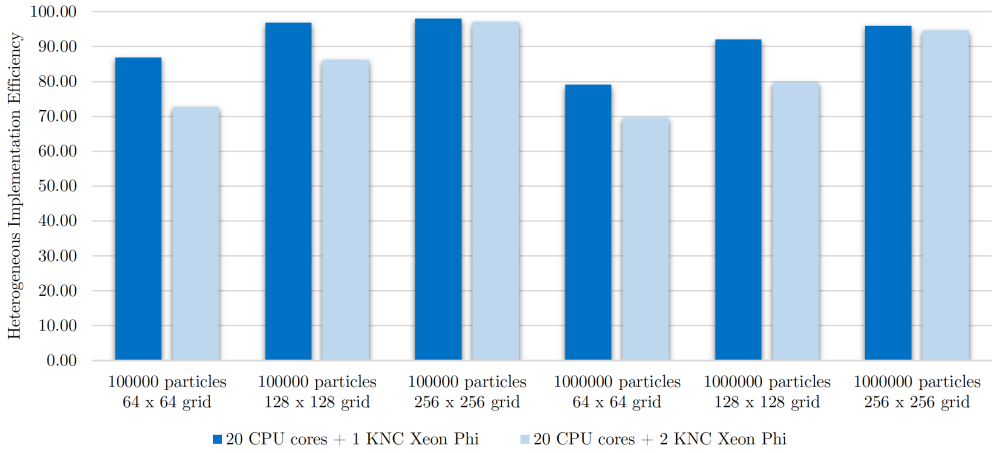


Figure 25: Efficiency of the heterogeneous implementation of compute retarded potentials stage of the simulation on Machine-X which is composed of multi-core CPU and two KNC Xeon Phi coprocessor.

CHAPTER 6

CONCLUSIONS

This dissertation targeted several goals relating to optimizing the performance of irregular scientific applications on emerging HPC architectures like GPUs, Xeon Phis, and heterogeneous architectures composed of multi-core CPUs with GPUs or Xeon Phis as accelerators. In particular, the dissertation primarily focuses on optimizing the irregular workloads in scientific applications which require execution of one or more irregular algorithms for multiple time steps (in the order of few hundred thousand to millions), where the irregular algorithm at each time step exhibit control-flow and memory access pattern that are not readily amenable to most parallel architectures. Using numerical simulation of charged particles beam dynamics simulations as a motivating example, this dissertation presented novel machine learning based optimization techniques to address the computational challenges in the efficient parallel implementation of such irregular applications on HPC architectures.

The machine learning approach presented here relies on supervised learning algorithms to adaptively model and track irregular access patterns observed during the computation of irregular workloads at each time step of the simulation to anticipate the future control-flow and data access patterns. We demonstrated that the access pattern forecasts from anticipation strategies can be successfully used to formulate optimization decisions that improve the application performance at a future time step based on the observation from earlier time steps. In particular, we successfully used the forecasts to minimize both branch and memory divergence, thereby reducing the control-flow and memory access irregularities in its parallel implementation on GPU and Intel MIC architecture. We also showed that the forecast can be used to optimize the computation-to-thread mapping which maximize the data reuse by improving data locality. For implementation on heterogeneous architectures, we demonstrated that the forecast can be used to approximately divide the original problem into multiple smaller sub-problems, and once divided, we showed that they can be efficiently mapped between the hybrid mix of processing units of a heterogeneous architecture to deliver a good aggregate performance.

We further quantified the impact of using machine learning approach in resolving the computational challenges in the parallel implementation of beam dynamics simulation using NVIDIA GPUs and two different heterogeneous architectures. Then, 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 Heuristics-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. Heuristics-RP algorithm addressed the memory inefficiencies in Two-Phase-RP algorithm, which further provided a substantial boost in the performance. Now, with the new and improved algorithm presented in this work that can run on a wide variety of computing platforms compared to the existing algorithms, it enables unprecedented efficiency in numerical simulation of all the relevant physics of synchrotron light sources and electron-ion particle colliders. The newly improved efficiency, coupled with high-fidelity and precision of our earlier implementations, makes the previously inaccessible physics tractable. For accelerator physics community in general, this research is a step forward in developing ultra-bright light sources which are essential tools for discoveries and innovations in physical, biological, energy and medical sciences.

Though the reported work has focused on addressing the irregularities in parallel implementation of charged particles beam dynamics simulation, the work presented has the potential to be much wider reaching. The optimizations should be equally applicable to other irregular applications which suffers from similar implementation challenges as that of beam dynamics simulation. For instance, the approach used to optimize the computation-to-thread mapping based on the predicted access pattern can be adapted by other applications while developing parallel implementations on GPUs. As an example, in the parallel implementation of n-body simulation on GPUs using Barnes-Hut algorithm, we can use this approach to determine the particles-to-thread mapping to minimize the divergence, improve data reuse and to make prefetch decisions which improves the overall application performance. Other applications where this approach can be adapted are molecular dynamics simulation, finite elements methods, simulation of wave and sound propagation in 3D objects, etc.

REFERENCES

[1] T. Agoh and K. Yokoya. Calculation of coherent synchrotron radiation using mesh. Physical Review Special Topics: Accelerators and Beams, 7:054403, May 2004.

[2] A. Aho, R. Sethi, and J. Ullman. Compilers: principles, techniques, and tools. Addison Wesley, 1986.

[3] K. Arumugam, D. Ranjan, M. Zubair, A. Godunov, and B. Terzić. An efficient deterministic parallel algorithm for adaptive multidimensional numerical integration on GPUs. In 42nd International Conference on Parallel Processing, ICPP'13, pages 486–491, Washington, DC, USA, October 2013. IEEE Computer Society.

[4] K. Arumugam, D. Ranjan, M. Zubair, A. Godunov, and B. Terzić. A memory-efficient algorithm for adaptive multidimensional integration with multiple GPUs. In 20th Annual International Conference on High Performance Computing, HiPC'13, pages 169–175, December 2013.

[5] K. Arumugam, D. Ranjan, M. Zubair, A. Godunov, and B. Terzić. High-fidelity simulation of collective effects in electron beams using an innovative parallel method. In Summer Computer Simulation Conference (SCSC), SummerSim'15, pages 1–10, San Diego, CA, USA, 2015. Society for Computer Simulation International.

[6] K. Arumugam, D. Ranjan, M. Zubair, A. Godunov, and B. Terzić. Memory-efficient parallel simulation of electron beam dynamics using GPUs. In 23rd International Conference on High Performance Computing, HiPC'16, pages 212–221, December 2016.

[7] K. Arumugam, D. Ranjan, M. Zubair, B. Terzi, A. Godunov, and T. Islam. A machine learning approach for efficient parallel simulation of beam dynamics on GPUs. In 46th International Conference on Parallel Processing (ICPP), pages 462–471, August 2017.

[8] S. S. Baghsorkhi, I. Gelado, M. Delahaye, and W.-m. W. Hwu. Efficient performance evaluation of memory hierarchy for highly multithreaded graphics processors. In Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '12, pages 23–34, 2012.

[9] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. Nature, 324:446–449, December 1986.

[10] G. Bassi, T. Agoh, M. Dohlus, L. Giannessi, R. Hajima, A. Kabel, T. Limberg, and M. Quattromini. Overview of CSR codes. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, 557(1):189–204, 2006. Energy Recovering Linacs 2005.

[11] G. Bassi, J. A. Ellison, K. Heinemann, and R. Warnock. Microbunching instability in a chicane: Two-dimensional mean field treatment. Physical Review Special Topics: Accelerators and Beams, 12:080704, August 2009.

[12] J. Berntsen, T. O. Espelid, and A. Genz. An adaptive algorithm for the approximate calculation of multiple integrals. ACM Transactions on Mathematical Software (TOMS), 17(4):437–451, December 1991.

[13] J. Berntsen, T. O. Espelid, and A. Genz. DCUHRE: An adaptive multidemensional integration routine for a vector of integrals. ACM Transactions on Mathematical Software (TOMS), 17(4):452–456, December 1991.

[14] E. Blem, M. Sinclair, and K. Sankaralingam. Challenge benchmarks that must be conquered to sustain the GPU revolution. In Proceedings of the 4th Workshop on Emerging Applications for Manycore Architecture, 2011.

[15] C. L. Bohn and J. R. Delayen. Fokker-Planck approach to the dynamics of mismatched charged-particle beams. Physical Review E, 50:1516–1534, August 1994.

[16] M. Borland, Y. C. Chae, P. Emma, J. Lewellen, V. Bharadwaj, W. M. Fawley, P. Krejcik, C. Limborg, S. V. Milton, H. D. Nuhn, R. Soliday, and M. Woodley. Start-to-end simulation of self-amplified spontaneous emission free electron lasers from the gun through the undulator. Nuclear Instruments and

Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, 483(1):268–272, 2002. Proceedings of the 23rd International Free Electron Laser Confere nce and 8th FEL Users Workshop.

[17] A. Braunstein, M. Mézard, and R. Zecchina. Survey propagation: An algorithm for satisfiability. Random Structures and Algorithms, 27(2):201–226, September 2005.

[18] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on GPUs. In 2012 IEEE International Symposium on Workload Characterization (IISWC), pages 141–151, November 2012.

[19] M. Burtscher and K. Pingali. An efficient CUDA implementation of the tree-based barnes hut n-body algorithm, pages 75–92. Morgan Kaufmann, 2011.

[20] R. E. Caflisch. Monte carlo and quasi-monte carlo methods. Acta Numerica, 7:1–49, 1998.

[21] S. Chapra and R. Canale. Numerical Methods for Engineers. McGraw-Hill, 6 edition, 2009.

[22] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonian. Managing DRAM latency divergence in irregular GPGPU applications. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC'14, pages 128–139, 2014.

[23] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron. Pannotia: Understanding irregular GPGPU graph applications. In 2013 IEEE International Symposium on Workload Characterization (IISWC), pages 185–195, September 2013.

[24] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using CUDA. Journal of Parallel and Distributed Computing, 68(10):1370–1380, 2008. General-Purpose Processing using Graphics Processing Units.

[25] L. P. Chew. Guaranteed-quality mesh generation for curved surfaces. In Proceedings of the Ninth Annual Symposium on Computational Geometry, SCG '93, pages 274–280, 1993.

[26] G. Dalquist and A. Björck. Numerical Methods in Scientific Computing, volume 1. Society for Industrial and Applied Mathematics, 2008.

[27] G. Dasika, A. Sethia, T. Mudge, and S. Mahlke. PEPSC: A power-efficient processor for scientific computing. In 2011 International Conference on Parallel Architectures and Compilation Techniques, pages 101–110, October 2011.

[28] Ya. S. Derbenev and V. D. Shiltsev. Transverse effects of microbunch radiative interaction. SLAC-Pub-7181, 1996.

[29] DOE/SC/Oak Ridge National Laboratory. Titan - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x. `http://www.olcf.ornl.gov/titan/`.

[30] T. Endo, S. Matsuoka, A. Nukada, and N. Maruyama. Linpack evaluation on a supercomputer with heterogeneous accelerators. In 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS), pages 1–8, April 2010.

[31] W. Feng and T. Scogland. Green Top 500 Supercomputers. `https://www.top500.org/green500/lists/2017/06/`.

[32] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007), pages 407–420, December 2007.

[33] A. C. Genz and A. A. Malik. An adaptive algorithm for numerical integration over an n-dimensional rectangular region. Journal of Computational and Applied Mathematics, 6(4):295–302, 1980.

[34] G. Golub and C. V. Loan. Matrix Computations. Johns Hopkins University Press, 3 edition, 1996.

[35] T. Hahn. CUBA-A library for multidimensional numerical integration. Computer Physics Communications, 168(2):78–95, 2005.

[36] T. Hahn. The CUBA library. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, 559(1):273–277, 2006. Proceedings of the X International Workshop on Advanced Computing and Analysis Techniques in Physics Research.

[37] K. Hildrum and P. S. Yu. Focused community discovery. In Fifth IEEE International Conference on Data Mining (ICDM'05), November 2005.

[38] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. Computer, 41(7):33–38, July 2008.

[39] R. W. Hockney and J. W. Eastwood. Computer simulations using particles. Institute of Physics Publishing, London, 1988.

[40] Intel. Intel Xeon Phi Core Micro-architecture. `https://software.intel.com/en-us/articles/intel-xeon-phi-core-micro-architecture`.

[41] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. OWL: Cooperative thread array aware scheduling techniques for improving GPGPU performance. In Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, pages 395–406, 2013.

[42] A. Kabel. Coherent synchrotron radiation calculations using TraFiC4: Multiprocessor simulations and optics scans. In PACS2001. Proceedings of the 2001 Particle Accelerator Conference (Cat. No.01CH37268), volume 4, pages 2988–2990 vol.4, 2001.

[43] Khronos Group. The OpenCL Specification v2.0. `https://www.khronos.org/opencl/`.

[44] D. B. Kirk and W. W. Hwu. Programming Massively Parallel Processors: A Hands-on Approach, volume 1. Morgan Kaufmann Publishers Inc., 2010.

[45] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU. In Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA '10, pages 451–460, New York, NY, USA, 2010. ACM.

[46] R. Li. Progress on the study of CSR effects. In 2nd ICFA Advanced Accelerator Workshop on the Physics of High Brightness Beams, pages 369–390, November 1999.

[47] R. Li. Self-consistent simulation of the CSR effect on beam emittance. In Nuclear Instruments and Methods in Physics Research Section A, volume 429, pages 310–314, June 1999.

[48] R. Li. Curvature-induced bunch self-interaction for an energy-chirped bunch in magnetic bends. Physical Review ST Accelerators and Beams, 11:024401, February 2008.

[49] R. Li, R. Legg, B. Terzić, J. Bisognano, and R. Bosh. Two-dimensional effects on the behavior of the CSR force in a bunch compressor chicane. In 33rd International Free Electron Laser Conference, Shanghai, 2011.

[50] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. IEEE Micro, 28(2):39–55, March 2008.

[51] M. Mendez-Lojo, M. Burtscher, and K. Pingali. A GPU implementation of inclusion-based points-to analysis. In Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '12, pages 107–116, 2012.

[52] J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA '10, pages 235–246, 2010.

[53] D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU graph traversal. In Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '12, pages 117–128, 2012.

[54] J. Misra. Distributed discrete-event simulation. ACM Computing Surveys (CSUR), 18(1):39–65, March 1986.

[55] S. Mittal and J. S. Vetter. A survey of CPU-GPU heterogeneous computing techniques. ACM Computing Surveys, 47(4):69:1–69:35, July 2015.

[56] NAG. Fortran 90 library. Numerical Algorithms Group Inc., Oxford, U.K., 2000.

[57] National Super Computer Center in Guangzhou. Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P. https://www.top500.org/system/177999.

[58] NVIDIA. CUDA Bandwidth Test. http://docs.nvidia.com/cuda/cuda-samples/#bandwidth-test.

[59] NVIDIA. CUDA C Best Practices Guide. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/.

[60] NVIDIA. CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[61] NVIDIA. Next Generation CUDA Compute Architecture: Kepler GK110. https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[62] NVIDIA. Profiler:CUDA Toolkit Documentation. http://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[63] OpenACC.org. The OpenACC Application Programming Interface. https://www.openacc.org/.

[64] OpenMP Architecture Review Board. OpenMP Application Program Interface v4.0. http://www.openmp.org/.

[65] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12:2825–2830, 2011.

[66] J. L. Peterson. Petri Nets. ACM Computing Surveys (CSUR), 9(3):223–252, September 1977.

[67] R. Piessens, E. de Doncker-Kapenga, C. Überhuber, and D. Kahaner. QUADPACK:A Subroutine Package for Automatic Integration. Springer-Verlag, Berlin, 1983.

[68] J. Sartori and R. Kumar. Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications. In Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, PACT '12, pages 427–428, 2012.

[69] S. Seo, G. Jo, and J. Lee. Performance characterization of the NAS parallel benchmarks in OpenCL. In IEEE International Symposium on Workload Characterization (IISWC), pages 137–148, November 2011.

[70] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu. An application-centric evaluation of OpenCL on multi-core CPUs. Parallel Computing, 39(12):834–850, 2013. Programming models, systems software and tools for High-End Computing.

[71] J. Shen, A. L. Varbanescu, Y. Lu, P. Zou, and H. Sips. Workload partitioning for accelerating applications on heterogeneous platforms. IEEE Transactions on Parallel and Distributed Systems, 27(9):2766–2780, September 2016.

[72] E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer. Top 500 Supercomputers. `https://www.top500.org/lists/2017/06/`.

[73] E. Sun, D. Schaa, R. Bagley, N. Rubin, and D. Kaeli. Enabling task-level scheduling on heterogeneous platforms. In 5th Annual Workshop on General Purpose Processing with Graphics Processing Units, GPGPU-5, pages 84–93, 2012.

[74] Swiss National Supercomputing Centre (CSCS). Piz Daint - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100. `http://www.cscs.ch/computers/piz_daint/index.html`.

[75] P. Tan, M. Steinbach, and V. Kumar. Introduction to Data Mining. Pearson Addison Wesley, 2005.

[76] B. Terzić and G. Bassi. New density estimation methods for charged particle beams with applications to microbunching instability. Physical Review Special Topics: Accelerators and Beams, 14:070701, July 2011.

[77] B. Terzić, I. Pogorelov, and C. Bohn. Particle-in-cell beam dynamics simulations with a wavelet-based poisson solver. Physical Review Special Topics: Accelerators and Beams, 10:034201, March 2007.

[78] J. S. Vetter and S. Mittal. Opportunities for nonvolatile memory systems in extreme-scale high-performance computing. Computing in Science Engineering, 17(2):73–82, March 2015.

[79] M. Viñas, Z. Bozkus, and B. B. Fraguela. Exploiting heterogeneous parallelism with the heterogeneous programming library. Journal of Parallel and Distributed Computing, 73(12):1627–1638, December 2013.

[80] A. Vladimirov, R. Asai, and V. Karpusenko. Parallel Programming and Optimization with Intel Xeon Phi Coprocessors. Colfax International, 2nd edition, 2015.

[81] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. Communications of the ACM - A Direct Path to Dependable Software, 52(4):65–76, April 2009.

[82] Wolfram MathWorld. Fubini Theorem. `http://mathworld.wolfram.com/FubiniTheorem.html`.

APPENDIX A

ADAPTIVE MULTI-DIMENSIONAL INTEGRATION

A.1 OVERVIEW OF CUHRE

This section describes the sequential CUHRE algorithm for n-dimensional integration, where the integrals have the form

$$\int_{a_1}^{b_1}\int_{a_2}^{b_2}\ldots\int_{a_n}^{b_n} f(\mathbf{x})d\mathbf{x}, \tag{21}$$

where $\mathbf{x}$ is an n-vector, and f is an integrand. We use $[\mathbf{a},\mathbf{b}]$ to denote the hyper rectangle $[a_1,b_1]\times[a_2,b_2]\ldots\times[a_n,b_n]$.

The heart of CUHRE algorithm is the procedure C-RULE($[\mathbf{a},\mathbf{b}],f,n$) which outputs a triple (I,ε,κ) where I is an estimate of the integral over hyper rectangle $[\mathbf{a},\mathbf{b}]$ ($[a_1,b_1]\times[a_2,b_2]\ldots\times[a_n,b_n]$), ε is an error estimate for I, and κ is the axis along which $[\mathbf{a},\mathbf{b}]$ should be split if needed. An important feature of C-RULE is that it evaluates the integrand only for $2^n+p(n)$ points where $p(n)$ is $\Theta(n^3)$ [12]. This is much fewer than 15^n function evaluations required by a simple repeated one-dimension integration scheme based on 7/15-point Gauss-Kronrod method.

The procedure SEQUENTIAL-CUHRE implements the CUHRE method to compute n-dimensional multiple integral. The procedure takes input n, $\mathbf{a}$, $\mathbf{b}$, f, a relative error tolerance τ_{rel} and an absolute error tolerance τ_{abs}, where $\mathbf{a}=(a_1,a_2,...,a_n)$ and $\mathbf{b}=(b_1,b_2,...,b_n)$. In the description provided below, H is a priority queue of 4-tuples $([\mathbf{x},\mathbf{y}],I,\varepsilon,\kappa)$ where $[\mathbf{x},\mathbf{y}]$ is a subregion, I is an estimate of the integral over this region, ε an estimate of the error and κ the dimension along which the subregion should be split if needed. The parameter ε determines the priority for extraction of elements from the priority queue. The algorithm maintains a global error estimate ε^g and a global integral estimate I^g. The algorithm repeatedly splits the region with greatest local error estimate and updates ε^g and I^g. The algorithm terminates when the $\varepsilon^g\leq\max(\tau_{abs},\tau_{rel}|I^g|)$ and outputs integral estimate I^g and error estimate ε^g.

SEQUENTIAL-CUHRE($n, \mathbf{a}, \mathbf{b}, f, \tau_{rel}, \tau_{abs}$)

1 $(I^g, \varepsilon^g, \kappa) \leftarrow$ C-RULE($[\mathbf{a}, \mathbf{b}], f, n$)
2 $H \leftarrow \emptyset$
3 INSERT($H, ([\mathbf{a}, \mathbf{b}], I^g, \varepsilon^g, \kappa)$
4 **while** $\varepsilon^g > \max(\tau_{abs}, \tau_{rel}|I^g|)$
5 $([\mathbf{a}, \mathbf{b}], I, \varepsilon, \kappa) \leftarrow$ EXTRACT-MAX(H)
6 $a' \leftarrow (a_1, a_2, \ldots, (a_\kappa + b_\kappa)/2, \ldots, a_n)$
7 $b' \leftarrow (b_1, b_2, \ldots, (a_\kappa + b_\kappa)/2, \ldots, b_n)$
8 $(I_{left}, \varepsilon_{left}, \kappa_{left}) \leftarrow$ C-RULE($[\mathbf{a}, \mathbf{b}'], f, n$)
9 $(I_{right}, \varepsilon_{right}, \kappa_{right}) \leftarrow$ C-RULE($[\mathbf{a}', \mathbf{b}], f, n$)
10 $I^g \leftarrow I^g - I + I_{left} + I_{right}$
11 $\varepsilon^g \leftarrow \varepsilon^g - \varepsilon + \varepsilon_{left} + \varepsilon_{right}$
12 INSERT($H, ([\mathbf{a}, \mathbf{b}'], I_{left}, \varepsilon_{left}, \kappa_{left})$)
13 INSERT($H, ([\mathbf{a}', \mathbf{b}], I_{right}, \varepsilon_{right}, \kappa_{right})$)
14 **return** I^g and ε^g

A.2 GPU-ACCELERATED PARALLEL ALGORITHM

The sequential adaptive quadrature for multi-dimensional integral is poorly suited to GPUs, as it does not take advantage of the GPU's data or task parallelism. This section presents a parallel quadrature algorithm that can utilize the parallel processors of GPUs to speed up the computation. The parallel algorithm approximates the integral by adaptively locating the subregions in parallel where the error estimate is greater than some user-specified error tolerance. It then calculates the integral and error estimates on these subregions in parallel. The pseudocode for this algorithm is provided below in QUADRATURE-PHASEONE and QUADRATURE-PHASETWO procedures.

In QUADRATURE-PHASEONE, L_{max} is a parameter that is based on target GPU architecture. The goal of this procedure is to create a list of subregions from the whole region $[\mathbf{a}, \mathbf{b}]$, with at least L_{max} elements for which further computation is necessary for estimating the integral to desired accuracy. This list is later passed on to QUADRATURE-PHASETWO. The algorithm maintains an list L of subregions, stored as $[\mathbf{a_j}, \mathbf{b_j}]$. Initially the whole integration region is split into roughly L_{max} equal parts through the procedure INIT-PARTITION. In each iteration of the while loop in QUADRATURE-PHASEONE, first the CUHRE rules are applied to all subregions in L

in parallel to get the integral estimate, error estimate, and the split axis. A list S is created to store the intervals with these values. Thereafter the algorithm essentially identifies the "good" and the "bad" subregions in S – the good subregions have error estimate that is below a chosen threshold, whereas bad subregions have error estimates exceeding this threshold. The bad subregions need to be further divided, while the integral and error estimates for the good regions can simply be accumulated. This is accomplished through the procedures PARTITION and UPDATE.

QUADRATURE-PHASEONE(n, $\mathbf{a}, \mathbf{b}$, f, d, τ_{rel}, τ_{abs}, L_{max})

1 $I^p \leftarrow 0$, $I^g \leftarrow 0$, $\varepsilon^p \leftarrow 0$, $\varepsilon^g \leftarrow \infty$
  // I^p, ε^p keep sum of integral and error estimates for the "good" subregions
  // I^g, ε^g keep sum of integral and error estimates for all subregions
2 $L \leftarrow$ INIT-PARTITION($\mathbf{a}, \mathbf{b}, L_{max}, n$)
3 **while** $(|L| < L_{max})$ and $(|L| \neq 0)$ and $(\varepsilon^g > \max(\tau_{abs}, \tau_{rel}|I^g|)$
4 $S \leftarrow \emptyset$
5 **for** all j in **parallel**
6 $(I_j, \varepsilon_j, \kappa_j) \leftarrow$ C-RULE($L[j], f, n$)
7 INSERT($S, (L[j], I_j, \varepsilon_j, \kappa_j)$)
8 $L \leftarrow$ PARTITION($S, L_{max}, \tau_{rel}, \tau_{abs}$)
9 $(I^p, \varepsilon^p, I^g, \varepsilon^g) \leftarrow$ UPDATE($S, \tau_{rel}, \tau_{abs}, I^p, \varepsilon^p$)
10 **return** $(L, I^p, \varepsilon^p, I^g, \varepsilon^g)$

QUADRATURE-PHASETWO(n, $\mathbf{f}$, τ_{rel}, τ_{abs}, L, I^g, ε^g)

1 **for** $j = 1$ to $|L|$ **parallel**
2 Let $[\mathbf{a_j}, \mathbf{b_j}]$ be the j^{th} record in L
3 $(I_j, \varepsilon_j) \leftarrow$ SEQUENTIAL-CUHRE(n, $\mathbf{a_j}$, $\mathbf{b_j}$, f, τ_{rel}, τ_{abs})
4 $I^g \leftarrow I^g + \sum_{[\mathbf{a_j}, \mathbf{b_j}] \in L} I_j$
5 $\varepsilon^g \leftarrow \varepsilon^g + \sum_{[\mathbf{a_j}, \mathbf{b_j}] \in L} \varepsilon_j$
6 **return** $I^g$and ε^g

It is worth noting that the original CUHRE algorithm always divides selected subregion into two parts along the chosen axis where the integrand has the largest fourth divided difference [12].The proposed algorithm here uses this strategy of choosing the axis,with the distinction that the selected subregion is divided into d pieces

INIT-PARTITION($\mathbf{a}, \mathbf{b}, L_{max}, n$)

1 $l \leftarrow \max\{j | j^n \leq L_{max}\}$
2 split $[\mathbf{a}, \mathbf{b}]$ along each dimension into l equal parts and
save these l^n subregions into L
3 **return** L

UPDATE($S, \tau_{rel}, \tau_{abs}, I^p, \varepsilon^p$)

1 $t_1 \leftarrow I^p, t_2 \leftarrow \varepsilon^p, t_3 \leftarrow 0, t_4 \leftarrow 0$
// t_1, t_2 accumulates integral and error estimates for the "good" subregions
// t_3, t_4 accumulates integral and error estimates for all the subregions
2 **for** $j = 1$ to $|S|$
3 Let $([\mathbf{a_j}, \mathbf{b_j}], I_j, \varepsilon_j, \kappa_j)$ be the j^{th} record in S
4 **if** $\varepsilon_j < \max(\tau_{abs}, \tau_{rel}|I_j|)$
5 $t_1 \leftarrow t_1 + I_j$
6 $t_2 \leftarrow t_2 + \varepsilon_j$
7 **else** $t_3 \leftarrow t_3 + I_j$
8 $t_4 \leftarrow t_4 + \varepsilon_j$
9 $t_3 \leftarrow t_3 + t_1$
10 $t_4 \leftarrow t_4 + t_2$
11 **return** (t_1, t_2, t_3, t_4)

PARTITION($S, L_{max}, \tau_{rel}, \tau_{abs}$)

1 $L_1 \leftarrow \emptyset$, $L_2 \leftarrow \emptyset$
// L_1 stores the "bad" subregions before subdivision
// L_2 stores the subregions after subdivision of "bad" subregions
2 **for** $j = 1$ to $|S|$
3 Let $([\mathbf{a_j}, \mathbf{b_j}], I_j, \varepsilon_j, \kappa_j)$ be the j^{th} record in S
4 **if** $\varepsilon_j \geq \max(\tau_{abs}, \tau_{rel}|I_j|)$
5 insert $([\mathbf{a_j}, \mathbf{b_j}], \kappa_j)$ into L_1
6 $d \leftarrow$ SPLIT-FACTOR($L_{max}, |L_1|$)
7 **for** $j = 1$ to $|L_1|$
8 Let $([\mathbf{a_j}, \mathbf{b_j}], \kappa_j)$ be the j^{th} record in L_1
9 split $[\mathbf{a_j}, \mathbf{b_j}]$ into d equal parts along the axis κ_j and
insert all these subregions into L_2
10 **return** L_2

Listing: Auxiliary procedures in QUADRATURE-PHASEONE and QUADRATURE-PHASETWO

along the chosen axis instead of two. The parameter d is dynamically calculated using a heuristic SPLIT-FACTOR based on the target architecture and on the number of bad intervals.Subdivision of a region refines the resolution of that region along with generating enough subregions to balance the computational load for second phase.

First phase continues until (i) a long enough list of "bad" subregions is created in which case we proceed to the second phase or (ii) there are no more "bad" subregions in which case we can return the integral and error estimates I^g and ε^g as the answer or (iii) I^g, ε^g satisfy the error threshold criteria in which case we also return I^g and ε^g as the answer. Note that, in case (ii) or (iii) second phase of the algorithm is not used.

The algorithm continues with the second phase when the global error estimate is still larger than the required global tolerance. In second phase, on every subregion $[\mathbf{a_j}, \mathbf{b_j}]$ in the list L the algorithm calls sequential CUHRE routine (SEQUENTIAL-CUHRE) to compute global integral and error estimate for the selected subregion (Line 3). Line 4 and 5 update the global integral and error estimate. Second phase implements a modified version of CUHRE to run in parallel for each of the subregions in the list L returned from first phase. The modified version of CUHRE implemented for GPU take advantage of state-of-the art GPU architectures to speed-up the computations. Our approach combines the original features of CUHRE with the improved algorithm efficiency afforded by massive parallelism on a GPU platform.

APPENDIX B

PERFORMANCE ANALYSIS TOOLS

B.1 ROOFLINE PERFORMANCE MODEL

Roofline is a visually intuitive performance model used to bound the performance of various applications running on multicore, manycore, or accelerator processor architectures. Rather than simply using percent-of-peak estimates, the model can be used to assess the quality of attained performance by combining locality, bandwidth, and different parallelization paradigms into a single performance figure. One can examine the resultant Roofline figure in order to determine both the implementation and inherent performance limitations.

Figure 26 shows the Roofline model for NVIDIA Tesla K40 GPU [81]. 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 $BW_{\text{Theoretical-Peak}} = 288$ GB/sec from hardware specifications. Additionally, when fused multiply and add (FMA) instructions are not utilized then the peak double precision floating-point performance drops to 0.7 Tflops/sec. The two horizontal lines denotes the performance ceilings for peak double-precision (DP) floating point operations with and without FMA. The black solid diagonal line in Figure 26 indicates the bandwidth ceiling for $BW_{\text{Theoretical-Peak}}$. 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 [58]. Experimental memory bandwidth for K40 is calculated to be $BW_{\text{Experimental-Peak}} = 200$ GB/sec, and its bandwidth ceiling in roofline model is shown using the blue solid diagonal line.

The Roofline sets an upper bound on performance of a kernel depending on the kernel's arithmetic intensity and if we think of arithmetic intensity as a column or

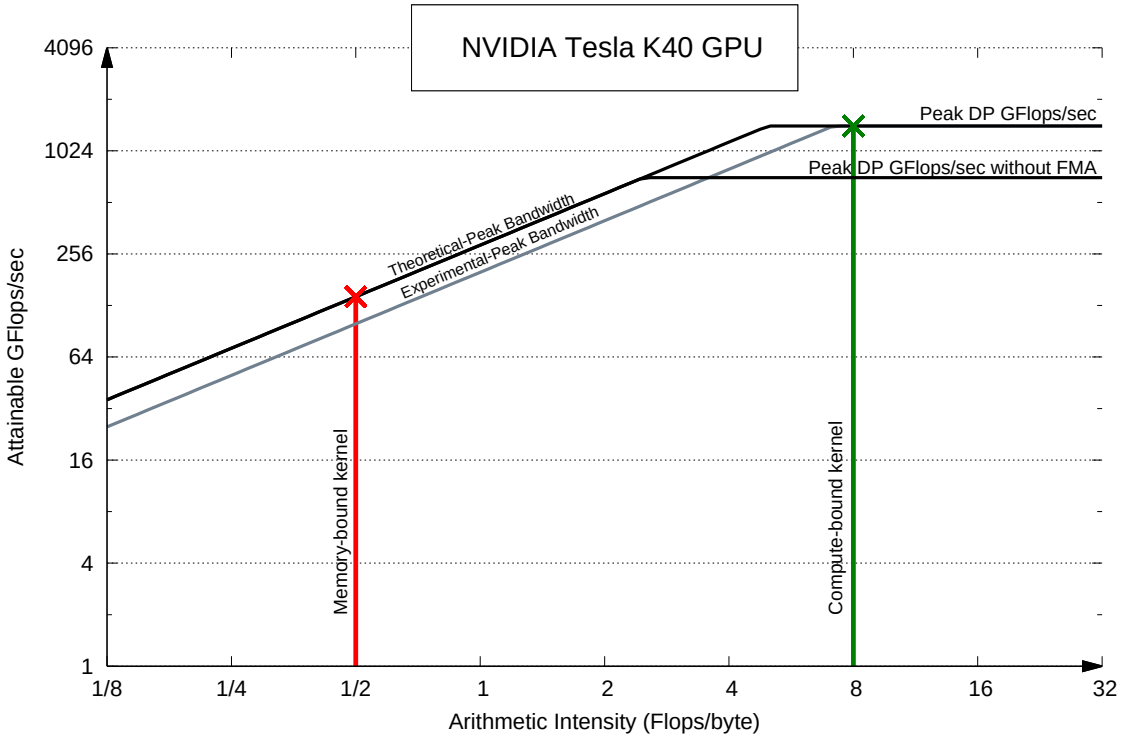


Figure 26: Roofline model for NVIDIA Tesla K40 GPU.

vertical line that hits the roof then it either hits the slanted part or the flat part of the roof. When the column hits slanted part of the roof, kernel is memory-bound and the only way to reach peak performance is to increase the arithmetic intensity. For example, the red vertical column indicates the arithmetic intensity of a memory-bound kernel and red X marks performance achieved for that particular kernel when the achieved bandwidth is $BW_{\text{Theoretical-Peak}}$. On the other hand, when the column hits the flat part of the roof, memory bandwidth is not the limiting factor and the kernel is compute-bound. For example, the green column indicates the arithmetic intensity for a compute-bound kernel and green X marks performance achieved for that particular kernel when the achieved bandwidth is $BW_{\text{Theoretical-Peak}}$.

B.2 PROFILER METRICS

This section contains detailed descriptions of the profiler metrics that are used to analyze the performance of GPU kernels in this thesis report. These metrics are

collected using NVIDIA's nvprof and the Visual Profiler [62].

- **Metric name:** Warp execution efficiency
  Profiler metric: warp_execution_efficiency
  Description: Ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor expressed as percentage. Values for warp_execution_efficiency indicate the following -
  - Values of less than 100% indicate the presence of threads with different control-flow paths which leads to performance bottlenecks on GPU architectures, as illustrated in Section 2.2.1.
  - 100% warp execution efficiency indicate that the kernel has no divergent threads.

- **Metric name:** Global load efficiency
  Profiler metric: gld_efficiency
  Description: Ratio of number of bytes requested by the kernel to number of bytes transferred from the global memory expressed as percentage (*or* ratio of requested global memory load throughput to required global memory load throughput expressed as percentage). Values for gld_efficiency indicate following performance behavior of a kernel -
  - Efficiency of 100% indicates perfect coalescing.
  - 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.
  - Values less than 100% indicates scattered and non-coalesced memory access, and such accesses waste off-chip bandwidth by over-fetching unnecessary data.

- **Metric name:** Global load transactions per request
  Profiler metric: gld_transactions_per_request
  Description: Average number of global memory load transactions performed for each global memory load.

  - Ideal value for transaction per requests is (32 threads per warp × word size in bytes / 128 bytes per line), which for 8-byte words is 2.0. The ideal value is obtained by perfect coalescing.
  - Values greater than the ideal value shows that substantial number of memory request from the kernel are non-coalesced and only a fraction of cache line is used , thereby resulting in transaction replays.

- **Metric name:** Global L1-cache hit rate
  Profiler metric: l1_cache_global_hit_rate
  Description: Hit rate in L1 cache for global loads

- **Metric name:** L2-cache hit rate
  Profiler metric: l2_l1_read_hit_rate
  Description: Hit rate at L2 cache for all read requests from L1 cache.

VITA

Kamesh Arumugam Karunanithi
Department of Computer Science
Old Dominion University
Norfolk, VA 23529

Kamesh Arumugam Karunanithi received his Bachelor of Engineering degree in Computer Science and Engineering from Visvesvaraya Technological University, India in 2010. During his bachelor studies, he worked on various research projects in the field of parallel computing, bioinformatics, and wireless sensor networks. After receiving his bachelor degree, he worked as Software developer and Consultant for web-based startup industries in Bangalore, India. In Fall 2011, Kamesh joined the Computer Science Department of Old Dominion University to pursue a PhD degree. He is currently working with Dr. Mohammad Zubair and Dr. Desh Ranjan of Department of Computer Science, and Dr. Balša Terzić of Department of Physics as research assistant, developing high-performance parallel algorithms and its implementation on emerging HPC architectures for a wide variety of scientific applications. Most of his research work are interdisciplinary which is a result of close collaboration with the computational scientists and physicists from ODU and Jeffersons Lab. His work is published in several peer reviewed international conferences - ICPP (2013 & 2017), HiPC (2013 & 2016), GTC (2013), SCSC (2015 & 2017), IPAC (2015 & 2017), and ICAP (2017). One of his research work titled "High-Fidelity Simulation of Collective Effects in Electron Beams Using an Innovative Parallel Method" won the best paper award at the 47th Summer Computer Simulation Conference (SCSC) held by the Society for Modeling & Simulation International (SCS). This work has been well received within the computational physics community and has enabled unprecedented fidelity and precision in studying all the relevant physics of synchrotron light sources. Further, he received the Modeling and Simulation Research Fellowship from Virginia Modeling, Analysis & Simulation Center of ODU to support his research for 3 years (maximum allowance). He also serves as a reviewer for BMC Bioinformatics Journal.

Typeset using LaTeX.