Summer 2017

Multi-GPU Accelerated High-Fidelity Simulations of Beam-Beam Effects in Particle Colliders

Naga Sai Ravi Teja Majeti
Old Dominion University, nmaje001@odu.edu

MULTI-GPU ACCELERATED HIGH-FIDELITY SIMULATIONS OF BEAM-BEAM EFFECTS IN PARTICLE COLLIDERS

by

Naga Sai Ravi Teja Majeti

A Thesis submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
August 2017

Approved by:

Mohammad Zubair (Director)

Desh Ranjan (Co-Director)

Balša Terzić (Co-Director)

ABSTRACT

MULTI-GPU ACCELERATED HIGH-FIDELITY SIMULATIONS OF BEAM-BEAM EFFECTS IN PARTICLE COLLIDERS

Naga Sai Ravi Teja Majeti
Old Dominion University, 2017
Director: Dr. Mohammad Zubair

Numerical simulation of beam-beam effects in particle colliders are crucial in understanding and the design of future machines such as electron-ion colliders (JLEIC), linac-ring machines (eRHIC) or LHeC. These simulations model the non-linear collision dynamics of two counter rotating beams in particle colliders for millions of turns. In particular, at each turn, the algorithm simulates the collision of two directed beams propagating at different speeds with different number of bunches each. This leads to non-pair-wise collisions of beams with different number of bunches that results in an increase in the computational load proportional to the number of bunches in the beams. Simulating these collisions for millions of turns using traditional CPUs is challenging due to the complexity in modeling non-linear dynamics of the beams and the need to simulate collision of every bunch in a reasonable amount of time.

In this Thesis, we present a high-performance scalable implementation to simulate the beam-beam effects in electron-ion colliders using a cluster of NVIDIA GPUs. The parallel implementation is optimized to minimize the communication overhead and the performance scales near linearly with number of GPUs. Further, the new code enables tracking and collision of the beams for millions of turns, thereby making the previously inaccessible long-term simulations tractable. As of now, there is no other code in existence that can accurately model

the single particle non-linear dynamics and the beam-beam effects at the same time for a large enough number of turns required to verify the long-term stability of a collider.

ACKNOWLEDGMENTS

This work is funded by a grant from Jefferson National Laboratory. I must acknowledge ODU ITS and Jefferson Laboratory for allowing me to access their computational resources.

Table of Contents

List of Figures

Figure Page

List of Tables

CHAPTER I

INTRODUCTION

Future particle colliders such as the Jefferson Lab Electron-Ion Collider (JLEIC) [13], linac-ring machines (eRHIC) [5] or LHeC [6] are particularly sensitive to beam-beam effects. Their design, construction and operation costs routinely measure in billions of dollars. A non-negligible portion of the cost can be reduced by optimization of the design and performance using computer simulations. The long-term stability of the beams in the collider is the fundamental criterion of the proper design and operation.

In order to simulate accurately the dynamics of the beams in a particle collider, it is necessary to track and collide the beam particles for millions to billions of turns. These long-term simulations are very time-consuming on a single processor system and need to be implemented on the massively parallel computer architectures to reduce the simulation time from the order of months or years to the order of days.

We choose a map-based tracking of the particle transport through the ring. Map generation techniques in application to accelerator lattices are well developed and are available in various codes. Therefore, we rely on existing tools and build upon the well-established verified algorithms of COSY Infinity [7]. The beam-beam interaction requires solving the 3D Poisson equation for each collision, which is computationally very expensive. The Poisson equation can be directly solved via a number of standard techniques, including multi-grid, conjugate gradient [16], or Fourier transform-based approach [10]. But, because of their higher computational load, simulating long-term beam dynamics in colliders becomes difficult. Therefore, we chose to invoke various approximations to alleviate the numerical load. One

approximation is assuming that the beam distribution is Gaussian. Another is the Bassetti-Erskine approximation [2] which further reduces the problem by assuming the interacting bunches to be infinitesimally short.

There are two scenarios for this problem:

1 - When each collider ring has only one bunch in it. In this case, all the interactions happening in this simulation are between these two bunches. For each interaction, we Track both the beams and Collide them. A single turn involves a single interaction in this scenario.

2 - When two rings have different harmonic numbers, each bunch from the first ring will interact with all the bunches present in the second ring. For example, when there are n-1 and n bunches, there is a total of (n-1)n interactions between these bunches. A single turn involves n-1 interactions in this scenario. As there are different harmonic numbers in each ring, each bunch will interact with a different bunch in each turn. So, it takes n turns until all the n-1 bunches from one ring interact with all the n bunches from the other ring which sums up to a total of (n-1)n interactions. In this thesis, one schedule completion of Multi-Bunch implementation refers to the completion of n turns; in practice this schedule repeats from millions to billions number of times.

In this thesis, we propose a new, high-fidelity model for simulation of long-term beam-beam dynamics. The proposed model is optimized to run efficiently on GPU platform which gives us the chance to study efficiently and accurately the long-term dynamics in colliders. Our implementation of the inherently parallelizable computations of beam tracking and collision on GPUs leads to orders-of-magnitude reduction in computational time, thereby making the previously inaccessible physics tractable. On the other hand, to simulate the interactions between the multiple bunches we propose a scheduling algorithm to simulate the interactions between

multiple bunches on a given number of GPUs. The algorithm is optimized to minimize the communication overhead and the performance scales nearly linear with the number of GPUs.

The remainder of this paper is organized as follows. Chapter - 2 provides the background of the physical problem, GPUs, and SIMD Challenges. In Chapter - 3 we outline the steps in numerical simulation of beam-beam dynamics and describe the existing core algorithms for Tracking and Collision. In Chapter - 4, we discuss the GPU implementation of Tracking and Collision algorithms. In Chapter - 5, we discuss the scheduling algorithm used for simulating the interactions between Multi-Bunch on multiple GPUs. Chapter - 6 presents the performance results of the proposed parallel algorithms for Tracking and Collision on NVIDIA Tesla K40 GPU. Also, the results about the scaling of Multi-Bunch implementation on multiple GPUs are also presented in this chapter. Finally, in Chapter - 7, we summarize our findings, conclude and discuss the future work.

CHAPTER II

BACKGROUND AND STATE OF ART

II.1 PHYSICAL PROBLEM

II.1.1 Tracking

Particle tracking for each of the six phase-space coordinates: x, $a \equiv p_x/p_0$, $b \equiv p_y/p_0$, l and δ is done using the equation

$$x = \sum_{\alpha\beta\gamma\eta\lambda\mu} M(x|\alpha\beta\gamma\eta\lambda\mu) x^{\alpha} a^{\beta} y^{\gamma} b^{\eta} l^{\lambda} \delta^{\mu}, \quad (1)$$

where $M(x|\alpha\beta\gamma\eta\lambda\mu)$ is a single turn map that is generated using a readily available accelerator lattice design and tracking codes. x and y are the transverse particle positions, a and b are the associated transverse momentum components p_x and p_y, respectively, normalized to the reference momentum p_0, $l = -(t - t_0)v_0\gamma_0$ and $\delta = (K - K_0)/K_0$. Here t, K, v_0, γ_0 are the time of the flight, kinetic energy, velocity and Lorentz factor, respectively. The subscript 0 indicates the reference value of the variable.

II.1.2 Collision

Beam-beam effects are one of the most dominant effects limiting the luminosity in electron-ion colliders [12]. The interaction between the two colliding beams (or a single particle in the field of particle beam) is described by the Poisson equation:

$$\Delta\emptyset(\boldsymbol{r}) = -\frac{1}{\varepsilon_0}\rho(\boldsymbol{r}), \quad (2)$$

where ρ is the charge distribution, $\emptyset$ the scalar potential, ε_0 the permittivity of free space and $\boldsymbol{r}$ the vector containing spatial coordinates. Solving the Poisson equation can be done directly via a number of techniques, including multigrid, conjugate gradient, or Fourier transform based

approach. These methods provide the exact numerical solution to an arbitrary beam charge distribution; however, their high computational cost makes them inadequate and inefficient for simulating long-term beam dynamics in colliders.

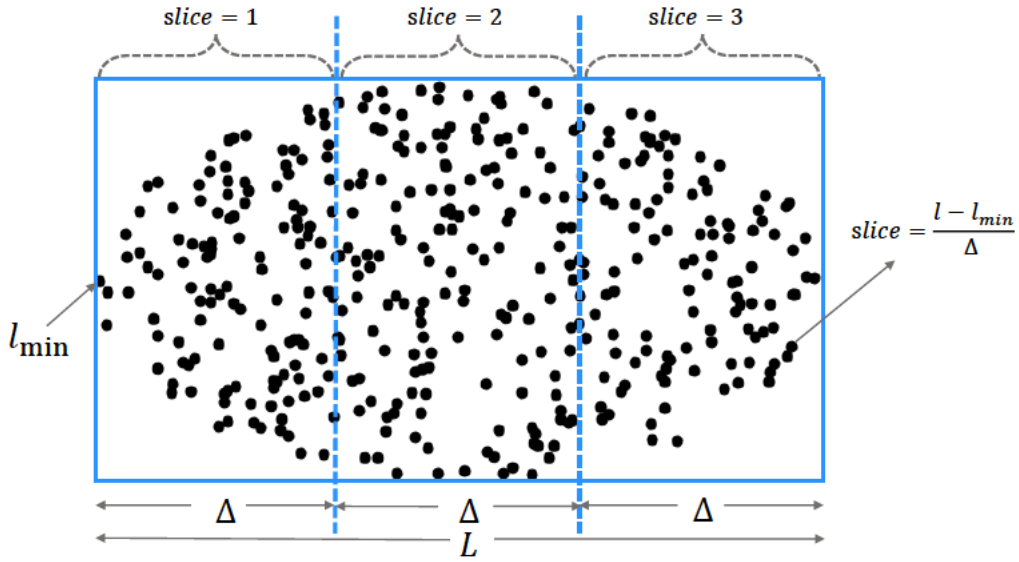


Figure 2.1 - Particles (denoted with black circles) in a beam are partitioned into $m = 3$ slices along longitudinal direction, where L is the maximum length of the beam, Δ is the width of each slice and l_{min} is the longitudinal coordinate of the left most particle.

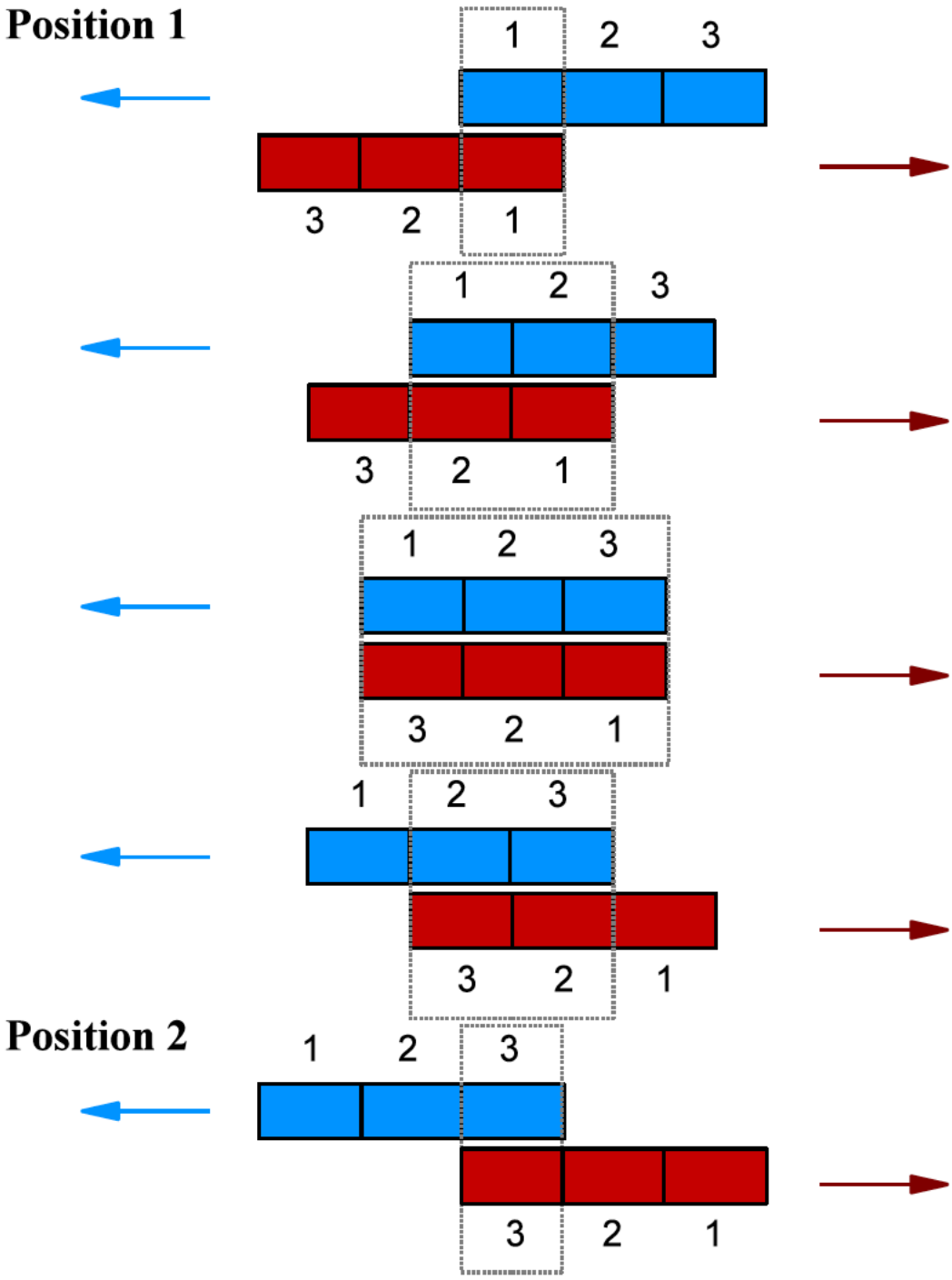


Figure 2.2 - Collisions between two multi-sliced beams, starting at Position 1 and ending with Position 2. After each line, all slices in both beams drift in the direction of the arrow by Δ/2, where Δ is the width of each slice. Grey rectangles denote slices that are colliding at each time.

In this thesis, we use Basetti-Erskine approximation [2] to model efficiently the beam-beam interaction. Our approach assumes the interacting bunches to be infinitesimally short. The finite bunch length is modeled by composing the beam of several infinitesimal slices. Each of these slices can then be treated as an infinitesimally short bunch. Figure 2.1 explains the slicing process, where the beam distribution is divides into 3 slices. L is the total length of the beam, m is the number of slices, $\Delta = L/m$ is the width of the individual slice, and the slice number of the particle is $Slice = (l - l_{min})/\Delta$. The collision between the two beams at the interaction point (IP) is simulated by collisions of individual slices which is illustrated in the Figure 2.2 where each beam is divides into 3 slices. Thus, when the beam is divided into *m* slices, it is evident from the figure-collision that there is a total of m^2 collisions between the slices of two beams. The collision between any two slices with longitudinal positions z^+ and z^-occurs at $s = S(z^+, z^-) \equiv (z^+ - z^-)/2$, taking into the account that the beam sizes are different from those at the IP ($s = 0$). The kicks experienced by both beams can be calculated by:

$$
\begin{aligned}
x_{new}^{\pm} &= x^{\pm} \pm S(z^+, z^-) f_X^{\pm}, \\
p_{x,new}^{\pm} &= p_x^{\pm} - f_X^{\pm}, \\
y_{new}^{\pm} &= y^{\pm} \pm S(z^+, z^-) f_Y^{\pm}, \\
p_{y,new}^{\pm} &= p_y^{\pm} - f_Y^{\pm}, \\
p_{z,new}^{\pm} &= p_z^{\pm} - \frac{1}{2} f_X^{\pm} \left[p_x^{\pm} - \frac{1}{2} f_X^{\pm} \right] - \frac{1}{2} f_y^{\pm} \left[p_y^{\pm} - \frac{1}{2} f_Y^{\pm} \right] - g^{\pm} ,
\end{aligned}
\tag{3}
$$

and

$$
\begin{aligned}
f_X^{\pm} &= \frac{n_i^{\mp} N^{\mp} r_{\pm}}{n^{\mp} \gamma_{\pm}} F_x \left(X^{\pm} - \bar{X}^{\mp}, Y^{\pm} - \bar{Y}^{\mp}; \sigma_x^{\mp}(S), \sigma_y^{\mp}(S) \right), \\
f_Y^{\pm} &= \frac{n_i^{\mp} N^{\mp} r_{\pm}}{n^{\mp} \gamma_{\pm}} F_y \left(X^{\pm} - \bar{X}^{\mp}, Y^{\pm} - \bar{Y}^{\mp}; \sigma_x^{\mp}(S), \sigma_y^{\mp}(S) \right),
\end{aligned}
\tag{4}
$$

$$g^{\pm} = \frac{n_i^{\mp} N^{\mp} r_{\pm}}{n^{\mp} \gamma_{\pm}} [R_{22}(0, z^*) g_x \left(X^{\pm} - \bar{X}^{\mp}, Y^{\pm} - \bar{Y}^{\mp}; \sigma_x^{\pm}(S), \sigma_y^{\pm}(S) \right) + R_{44}(0, z^*) g_y \left(X^{\pm} - \bar{X}^{\mp}, Y^{\pm} - \bar{Y}^{\mp}; \sigma_x^{\pm}(S), \sigma_y^{\pm}(S) \right)] S,$$

$$g_x(x, y, \sigma_x, \sigma_y) = -\frac{1}{2(\sigma_x^2 - \sigma_y^2)} \left\{ x F_x + y F_y + 2 \left[\frac{\sigma_y}{\sigma_x} e^{-\left(\frac{x^2}{2\sigma_x^2} + \frac{y^2}{2\sigma_y^2} \right)} - 1 \right] \right\},$$

$$g_y(x, y, \sigma_x, \sigma_y) = -\frac{1}{2(\sigma_x^2 - \sigma_y^2)} \left\{ x F_x + y F_y + 2 \left[\frac{\sigma_x}{\sigma_y} e^{-\left(\frac{x^2}{2\sigma_x^2} + \frac{y^2}{2\sigma_y^2} \right)} - 1 \right] \right\}$$

where r_- is the electron radius and, r_+ is the proton radius, n_i^- and n_i^+ are the number of simulation particles in the i^{th} slice of the electron and proton beam, respectively, with which the slice containing the particle being advanced is colliding, and $F_{\pm}$ is given below. The $\sigma's$ are evaluated at S as, e.g., $\sigma_x^{\pm}(S) = \sqrt{[(X^{\pm} - \bar{X}^{\pm})^2]}$, where averages are evaluated at $s = 0$. $N^{\pm}$ is the number of electrons $(-)$ and protons $(+)$ in the actual beam bunches, and $n^{\pm}$ is the total number of simulation particles in electrons $(-)$ and protons $(+)$ beam bunches.

The flat beam approximation ($\sigma_x > \sigma_y$), denoted below by subscript *f*, is relaxed by deriving generalized solutions for upright ($\sigma_x < \sigma_y$), given by subscript u, respectively.

$$E_f(x, y; \sigma_x, \sigma_y) \equiv \sqrt{\frac{2\pi}{\sigma_x^2 - \sigma_y^2}} \left[w\left(\frac{x + iy}{\sqrt{2(\sigma_x^2 - \sigma_y^2)}} \right) - e^{-\left(\frac{x^2}{2\sigma_x^2} + \frac{y^2}{2\sigma_y^2} \right)} w\left(\frac{\frac{\sigma_y}{\sigma_x} x + i \frac{\sigma_x}{\sigma_y} y}{\sqrt{2(\sigma_x^2 - \sigma_y^2)}} \right) \right], \qquad (5)$$

$$E_u(x,y;\sigma_x,\sigma_y) \equiv i\sqrt{\frac{2\pi}{\sigma_y^2-\sigma_x^2}}\left[w\left(\frac{y-ix}{\sqrt{2(\sigma_y^2-\sigma_x^2)}}\right) - e^{-\left(\frac{x^2}{2\sigma_x^2}+\frac{y^2}{2\sigma_y^2}\right)} w\left(\frac{\frac{\sigma_y}{\sigma_x}y - i\frac{\sigma_x}{\sigma_y}x}{\sqrt{2(\sigma_y^2-\sigma_x^2)}}\right)\right],$$

where

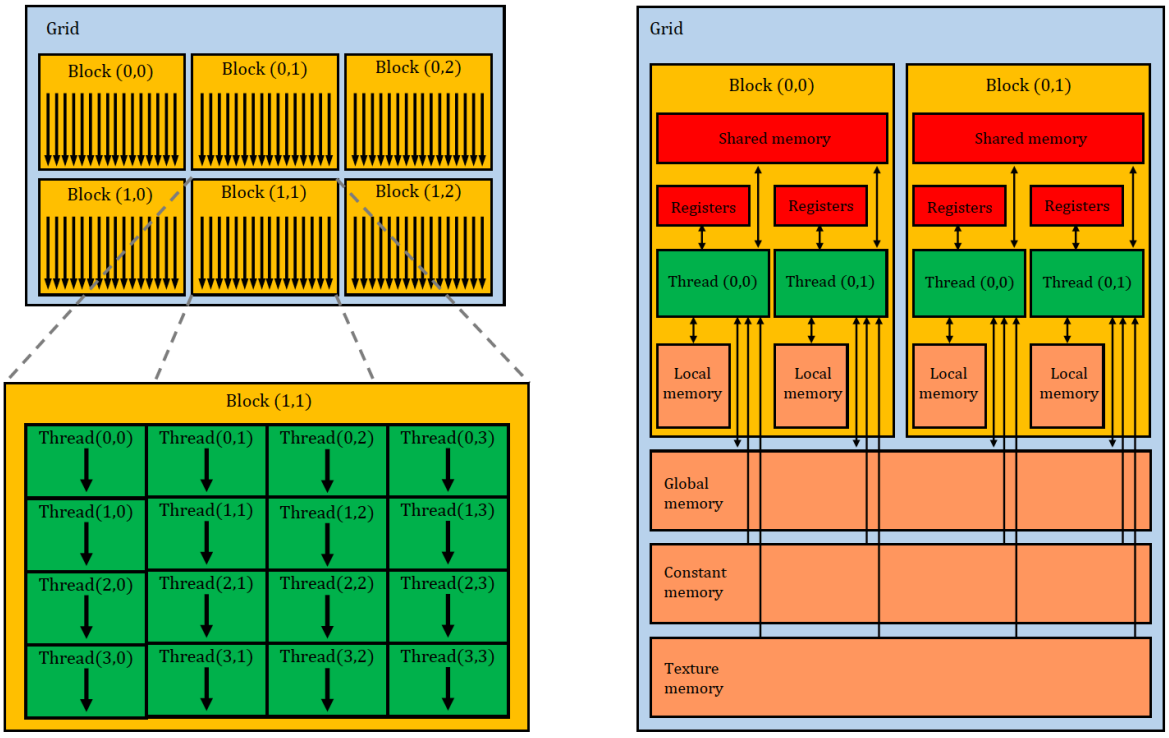
$$E(x,y;\sigma_x,\sigma_y) \equiv F_y(x,y;\sigma_x,\sigma_y) + iF_x(x,y;\sigma_x,\sigma_y), \tag{6}$$

and

$$w(z) \equiv e^{-z^2}\left[1 + \frac{2i}{\sqrt{\pi}}\int_0^z e^{\xi^2}\, d\xi\right] = e^{-z^2} erfc(-iz), \tag{7}$$

is the complex error function (also known as Faddeeva function), and erfc is the complementary error function. Complex error function is implemented using the optimized algorithm reported in [9].

II.2 GPU ARCHITECTURE



(a) Grid of Thread Blocks (b) Memory Hierarchy

Figure 2.3 – CUDA Programming Model.

At the hardware level, NVIDIA GPU architecture can be considered as an array of multithreaded Streaming Multiprocessors (SMs) which are scalable. Each SM comprises of several Streaming Processor (SP) cores, double-precision logic units (DP units), load/store units, special function units (SFU) for transcendental instructions such as sin, cosine, reciprocal, and square root, schedulers and instruction dispatch units, instruction cache, register file, on-chip shared-memory and L1-cache, read-only cache, and texture units.

Each SP core is a fully pipelined integer arithmetic logic unit (ALU) and single-precision floating point unit (FPU). Memory can be shared among all SMs as the GPUs support memory sharing in the form of global, constant and texture memory. 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.

NVIDIA invented Compute Unified Device Architecture (CUDA) [1]. CUDA is a parallel computing platform and programming model used to design parallel computations on NVIDIA GPUs. Any application using CUDA will have an increased computing performance by using the power of GPU. Two important components of CUDA programming are Device and Host. Device is the GPU and Host is the CPU. Kernels are the functions where the logic for the actual computation which is to be run in parallel resides. These kernels are launched by the Host and executed on the Device by 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 size of the thread block varies from one generation of GPUs to another. The thread blocks are combined into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks as illustrated in the Figure 2.3a.

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. The programmer can write the code that may run on any number of cores as the thread blocks are executed independently and can be scheduled in any order across any number of cores as illustrated in CUDA C programming guide. During their execution, CUDA threads can access data from six different memory sources: register memory, constant memory, shared memory, texture memory, local memory, and global memory as illustrated in the Figure 2.3b.

Each thread has private register to hold frequently accessed data, which are not controlled by the programmer. Each thread has private local memory that is used for register spills, function calls, and automatic array variables. Each thread block has a private shared memory generally used for inter-thread communication and is accessible by all the threads of a block with the same lifetime as the block. The global, constant and texture memory can be accessed by all the threads and are available across all the kernel launches through out the execution timeframe of the same application.

When a kernel is compiled and ready to be executed, the thread blocks within the kernel grid are scheduled either concurrently or sequentially on the available SMs as multiple thread blocks can be executed concurrently on a single SM. As the thread blocks terminate at any given time, new blocks are launched on the vacated SM.

A warp is a group of 32 parallel threads and the SM creates, manages, schedules and executes threads in such groups. Even though the individual threads within the warp start their execution from the same program address, they have their own program counter and register state and hence free to branch and execute independently. All the threads within a warp may execute

the same instruction at any given time and which is why SM is considered to be following SIMT architecture to manage and execute hundreds of threads concurrently.

When all the 32 threads within a warp agree to the same control-flow or the same execution path then the full warp efficiency is realized. Warp efficiency is the average percentage of active threads in each executed warp. Often, data-dependent conditional branch causes threads within the same warp to follow different execution paths which is called as branch divergence or control-flow divergence which then prompts the warp to execute each branch path serially, disabling threads that are not on that path. The threads converge back to the same execution path when all the paths are complete.

Note: Branch divergence occurs only within a warp. Threads within different warps execute independently regardless of whether they are following common or disjoint execution paths.

CHAPTER III

EXISTING SERIAL ALGORITHM

In this chapter, we discuss the working of existing serial algorithm that was developed to establish the proof-of-concept of beam-beam interactions in particle colliders using Bassetti-Erskine approximation [2].

III.1 OUTLINE OF THE ALGORITHM

At the top-most level, numerical simulation of beam-beam effects consists of two major steps - Tracking and Collision. These two steps are executed during each turn of the simulation, which in practice, runs for millions to billions of turns to simulate long-term beam-beam dynamics in particle colliders.

1. Tracking - The particles from the two input beams, e- and p-beam, are transported through the ring to bring them to an interaction point using an arbitrary-map generated from readily available accelerator lattice design and tracking codes (e.g. COSY Infinity [7]). This requires solving Equation (1) for all particles in the two input beams.
2. Collision - The simulation of collision (or beam-beam interactions) between the two input beams, e- and p-beam, consist of two consecutive steps.
    a. Slicing - Each input beam is sliced into *m* equal parts along longitudinal direction, as illustrated in Chapter 2. For example, Figure 2.1 illustrates the slicing of a beam into three parts along longitudinal axis.
    b. Apply Kick - The collision of two beams is simulated using slice-to-slice interactions, where each slice from one beam collides with every other slice of the other beam such that the order of collision captures the beams drift along

the collider ring. For example, Figure 2.2 illustrates the collision of two beams that is partitioned into three slices each, where particles from both the colliding beams experience a total of three kicks (or beam-beam effects), one from each slice of the counter-rotating beam. This kick computation between a pair of colliding slices, which is the beam-beam effect of one slice on the other, is calculated using Equations (3)-(7).

Algorithm 1 – $\boldsymbol{function}\ Beam - Beam\ (E, P, M_e, M_p, d, t, m)$

1: $\boldsymbol{for}\ i\ =\ 0\ to\ t\ do$

2: $\quad Track(E, M_e, d)$

3: $\quad Track(P, M_p, d)$

4: $\quad Collide(E, P, m)$

5: $\boldsymbol{end\ for}$

6: $\boldsymbol{end\ function}$

The pseudo code for numerical simulation of the beam-beam effects is illustrated in Algorithm 1. In this algorithm, each beam is represented as a list of particles, where each particle is a six dimensional object denoting the six phase-space coordinates of that particular particle. The map required to transport the particles through the collider ring is given as a list of 2D matrices, where each matrix represents the 2D map along one of the six phase-space coordinates, and each row of the matrix is a 7-tuple object, ($\alpha, \beta, \gamma, \eta, \lambda, \mu, M(x|\alpha\ \beta\ \gamma\ \eta\ \lambda\ \mu)$), denoting the variables with same representation from Equation (1). The procedure $Beam - Beam$ simulating the beam-beam effects takes input E, P, M_e, M_p, d, t and m, where E and P are the list containing particles from e-beam and p-beam, respectively, M_e and M_p are the transport maps for e-beam and p-beam,

respectively, d is the dimension of particles in simulation space (in this case, we have six phase-space coordinates i.e. d = 6), t is the number of turns required for the simulation, and m is the number of slices required for the collision step of the simulation. In this procedure, each iteration of the for loop implements beam-beam effects for a single turn of the simulation, where particles from each beam is first transported through the collider ring using the procedure $Track$ on each beam, and then the collision of the two beams are implemented using $Collide$ procedure. The pseudocode for these two procedures are presented in Algorithms 2 and 4.

III.2 TRACKING ALGORITHM

Algorithm 2 – $\boldsymbol{function}\ Track\ (B, M_b, d)$

1: $\boldsymbol{for}\ i\ =\ 0\ to\ d\ -\ 1\ do$

2: $\quad M \leftarrow M_b[i]$

3: $\quad \boldsymbol{for}\ each\ particle\ p \in B\ \boldsymbol{parallel\ do}$

4: $\quad\quad p[i] \leftarrow Apply - Map(p, M, d, i)$

5: $\quad \boldsymbol{end\ for}$

6: $\boldsymbol{end\ for}$

7: $\boldsymbol{end\ function}$

Algorithm 3 – $function\ Apply - Map(p, M, d, dim)$

1: $x \leftarrow p[dim]$

2: $\boldsymbol{for}\ i\ =\ 0\ to\ M.rows\ -\ 1\ \boldsymbol{do}$

3: $\quad y \leftarrow M[i][M.columns\ -\ 1]$

4: $\quad \boldsymbol{for}\ j\ =\ 0\ to\ d\ -\ 1\ \boldsymbol{do}$

5: $\quad\quad y \leftarrow y.p[i]M[i][j]$

6: ***end for***

7: $\boldsymbol{x} \leftarrow x + y$

8: ***end for***

9: ***return x***

10: ***end function***

The procedure $Track\ (B, M_b, d)$ in Algorithm 2 evaluates Equation (1) for all d-dimensional particles in a input list B using a transport map M_b. In particular, for each dimension in the d-dimensional coordinate space of the particles, transport map of the corresponding dimension is applied to all particles in the list B using an auxiliary procedure $Apply - Map$, where the procedure $Apply - Map(p, M, d, dim)$ called along a dimension dim for a particle p returns the updated value for $p[dim]$ by evaluating Equation (1) using the transport map M. The pseudo code for $Apply - Map$ and other auxiliary methods required in $Beam - Beam$ procedure are illustrated in Algorithm 3.

III.3 COLLISION ALGORITHM

Algorithm 4 – ***function*** $Collide\ (E, P, m)$

1: $S_e \leftarrow Slice(E, m)$

2: $S_p \leftarrow Slice(P, m)$

3: ***for*** $i\ =\ 1\ to\ m$ ***do***

4: ***for*** $j\ =\ 0\ to\ i - 1$ ***parallel do***

5: Let s be the $interaction - point$ for $S_e[j]$ and $S_p[i - j - 1]$ calculated as described in Chapter 2

6: $Apply - Kick(S_e[j], S_p[i - j - 1], s)$

7: **end for**

8: **end for**

9: **for** $i = m + 1$ *to* $2 * m - 1$ **do**

10: **for** $j = i - m$ *to* $m - 1$ **parallel do**

11: Let s be the $interaction - point$ for $S_e[j]$ and $S_p[i - j - 1]$ calculated as described in Chapter 2

12: $Apply - Kick(S_e[j], S_p[i - j - 1], s)$

13: **end for**

14: **end for**

15: $E \leftarrow Merge - Slices(S_e, m)$

16: $P \leftarrow Merge - Slices(S_p, m)$

17: **end function**

Algorithm 5 – ***function*** $Slice(B, m)$

1: let $S[0 \,..\, m - 1]$ be a new array

2: **for** $i = 0$ *to* $m - 1$ **do**

3: make $S[i]$ an empty list

4: **end for**

5: **for** $each\ particle \in$ B **parallel do**

6: $slice \leftarrow Find - Slice(B, p, m)$

7: $List - Insert(S[slice], p)$

8: **end for**

9: **return** S

10: ***end function***

11: ***function*** $Find - Slice(B, p, m)$

12: calculate and return the slice number to which p belongs based on the geometry along longitudinal direction

13: ***end function***

The collision between two counter-rotating beams is implemented in the procedure $Collide\ (E, P, m)$, where E and P represents the list of particles in the two colliding beams and m is the number of slices required per beam. This procedure updates all the particles in E and P to reflect the beam-beam interaction, and it works as follows. Line 1 and 2 divides the list of particles in the input beams into m slices (or sublists) using $Slice$ method on each beam E and P. In particular, procedure $Slice(B, m)$ partitions the input list of particles B into m sublists based on their corresponding slice number which is calculated using $Find - Slice$ method that implements the slicing algorithm described in Chapter 2. Next, the for loops in lines 3 - 14 calculates the beam-beam effects (or kicks) for every pair of colliding slices, where the kick computation on all particles in a pair of colliding slices is implemented using the procedure $Apply - Kick$. The procedure $Apply - Kick(S_e, S_p, s)$ calculates the kick at a interaction point s on all particles in the input list S_e and S_p, where the interaction point is calculated as described in Section 2 and [15], and then it updates all the particles in the input slice with the computed kick. The pseudo code for $Apply - Kick$ is illustrated in Algorithm 6. Finally, in lines 15 and 16, particles from individual slices are merged into a single list using the procedure $Merge - Slices$ on S_e and S_p, respectively. The $Merge - Slices$ procedure returns a sorted list that is the merge of its input array of lists, and the output from the two calls to this procedure is stored in the input lists, E and P respectively. The updated particles in E and P are used for simulating the beam-beam effects in the next turn.

Algorithm 6 - $\boldsymbol{function}\ Apply - Kick(S_e, S_p, s)$

1. $(\bar{x}^e, \bar{y}^e, \bar{\sigma}_x^e, \bar{\sigma}_y^e) \leftarrow Compute - Mean - SD(Se, s)$
2. $(\bar{x}^p, \bar{y}^p, \bar{\sigma}_x^p, \bar{\sigma}_y^p) \leftarrow Compute - Mean - SD(\mathrm{Sp}, \mathrm{s})$
3. $\boldsymbol{for}$ each particle $e \in Se$ $\boldsymbol{parallel\ do}$
4. $(F_x, F_y) \leftarrow Compute - Kick(e, \bar{x}^p, \bar{y}^p, \bar{\sigma}_x^p, \bar{\sigma}_y^p)$
5. $e[0] \leftarrow e[0] - s * Fx$
6. $e[1] \leftarrow e[1] - Fx$
7. $e[2] \leftarrow e[2] - s * Fy$
8. $e[3] \leftarrow e[3] - Fy$
9. $\boldsymbol{end\ for}$
10. $\boldsymbol{for}$ each particle $p \in Sp$ $\boldsymbol{parallel\ do}$
11. $(F_x, F_y) \leftarrow Compute - Kick(p, \bar{x}^e, \bar{y}^e, \bar{\sigma}_x^e, \bar{\sigma}_y^e)$
12. $p[0] \leftarrow e[0] + s * Fx$
13. $p[1] \leftarrow p[1] + Fx$
14. $p[2] \leftarrow p[2] + s * Fy$
15. $p[3] \leftarrow p[3] + Fy$
16. $\boldsymbol{end\ for}$
17. $\boldsymbol{end\ function}$
18. $\boldsymbol{function}\ Merge - Slices(S, m)$
19. B $\leftarrow \varnothing$
20. $\boldsymbol{for}\ i\ =\ 0\ to\ m - 1\ \boldsymbol{do}$
21. $B \leftarrow Merge - List(B, S[i])$

22. ***end for***

23. ***return B***

24. ***end function***

The procedure $Apply - Kick(S_e, S_p, s)$ is the heart of $Beam - Beam$ algorithm, and it works as follows. The two calls to $Compute - Mean - SD$ method calculates and returns the mean and standard deviation along the first and third dimension for the particles in S_e and S_p, respectively, where the two dimensions correspond to the transverse position of particles. Next, for each particle e in S_e, the kick from all the particles in S_p on a particle e is calculated using the procedure $Compute - Kick$, which takes input, a particle e, mean and standard deviation of the particles in S_p, and it returns a pair (F_x, F_y). The output values, F_x and F_y, represents the kick from particles in S_p on e, and it is calculated using equations 3 to 7 (F_x and F_y denotes the variables with same notations from Chapter 2). These computed kicks are used to update the particles in S_e in the first for loop. Similarly, in the next for loop, kick on all the particles in S_p due to the particles from S_e is calculated using the procedure $Compute - Kick$ and the output from this procedure is used to update the particles in S_p.

CHAPTER IV

GPU IMPLEMENTATION

In this chapter, we discuss our GPU implementation of Single-Bunch beam-beam simulation algorithm. We first present a brief overview of the existing GPU algorithm for Tracking and then discuss the GPU algorithm for Collision in detail.

IV.1 PARALLEL ALGORITHM FOR TRACKING

In our implementation, the input lists of particles and the transport map are always stored in the GPU memory. The procedure TRACK, which is highly data-parallel, is implemented on GPU as an independent kernel where the computation of each particle is assigned to parallel threads with one-to-one correspondence. In this kernel, transport map is stored in shared memory and it is accessed efficiently to improve the memory performance.

IV.2 PARALLEL ALGORITHM FOR COLLISION

The essential routines for $Collide$ procedure such as $Slice$, $Apply-Kick$, and $Merge-Slices$ are implemented on GPU. The implementations of all the routines are discussed in the next subsections.

IV.2.1 Slicing in Parallel

The Single-Threaded way of Slicing and Sorting the particles is to compute the slice number of one particle at a time, store the slice numbers of each particle, and then sort them using one of the popular sorting techniques according to their slice number. Although there are some existing libraries for sorting on GPUs, as we have to sort the particles which has 6 dimensions,

none of the libraries were compatible for integration into our implementation. So, we propose a fast and efficient algorithm for Slicing and Sorting the particles of a Beam on GPU.

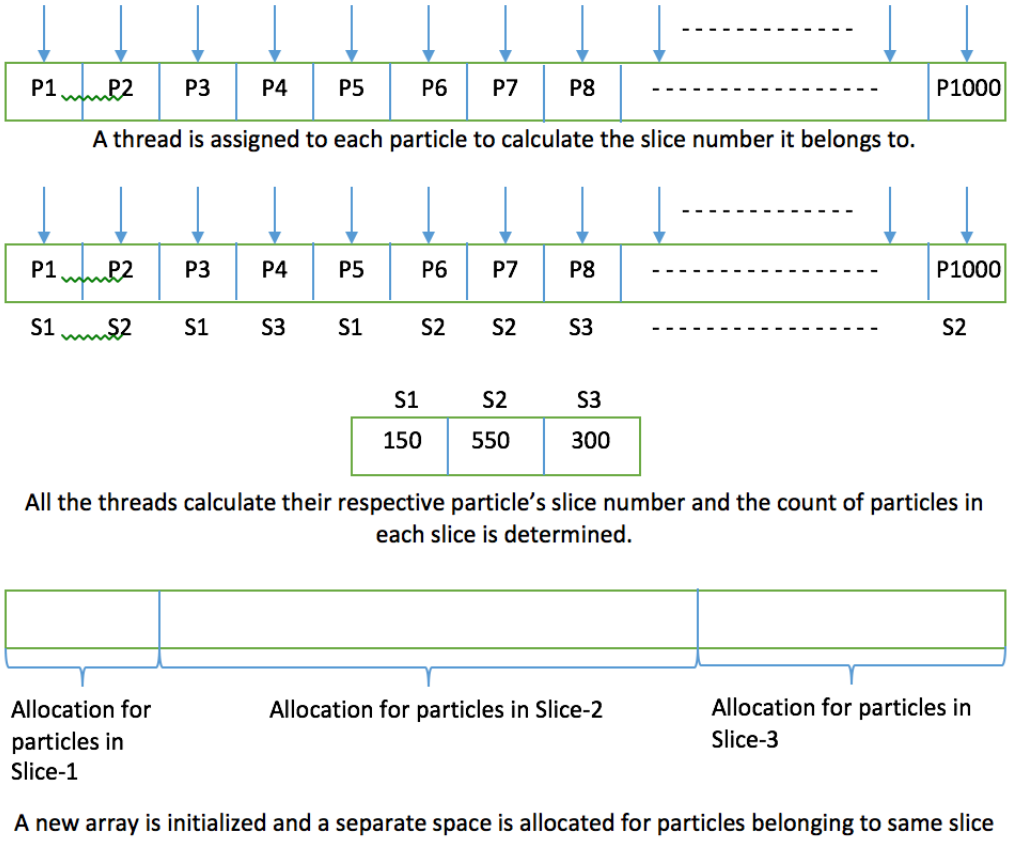


Figure 4.1 – Initial positions of all the particles and global counter stores particles size in each slice. A new array is also initialized to store particles according to their slice number.

During this phase, a thread is assigned to each particle to calculate its slice number. Each thread will find the slice number of the particle and increments the global counter to update the count of particles in each slice. For example, in Figure 4.1 there are a total of 1000 particles in the beam and the beam is to be divided into 3 slices. All the threads calculate their respective particle's slice number and update the global counter. In this case, Slice-1(S1) has 150 particles, Slice-2(S2) has 550 particles, and Slice-3(S3) has 300 particles. Then, an empty list is initialized and the spaces for each slice are allocated separately as shown in the Figure 4.1.

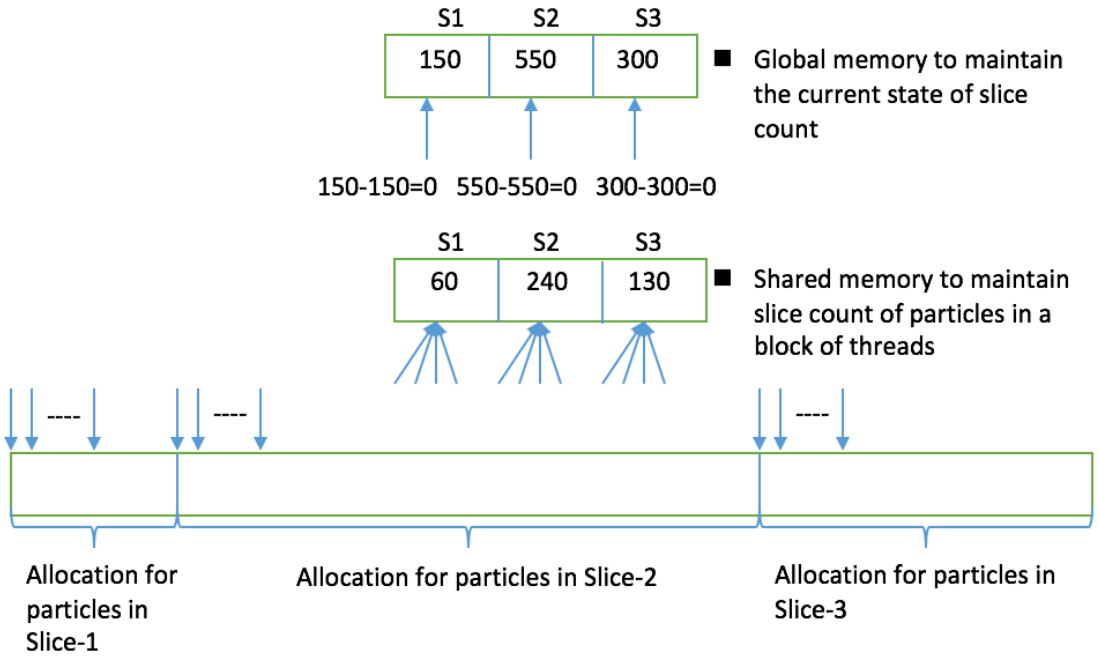


Figure 4.2 – First block accessing the global counter to write the particles in their allocated regions.

Next, a lead thread from each block examines the global slice count status and broadcasts the status to all the particles. For example, in the Figure 4.2, a lead thread from the block (say block-x) examines the status for all the slices and broadcasts the difference between the actual total slice count and the current status. In this case, the difference is 0 as none of the blocks have started sorting their particles according to their slice numbers. So, as there are 60 particles in block-x which belongs to Slice-1 and the first 60 spaces are allocated to the particles belonging to Slice-1 in block-x. A similar procedure is followed for the particles in Slice-2 and Slice-3. After these steps, the global slice count will be 90, 310, 170 respectively for Slice-1, 2, and 3.

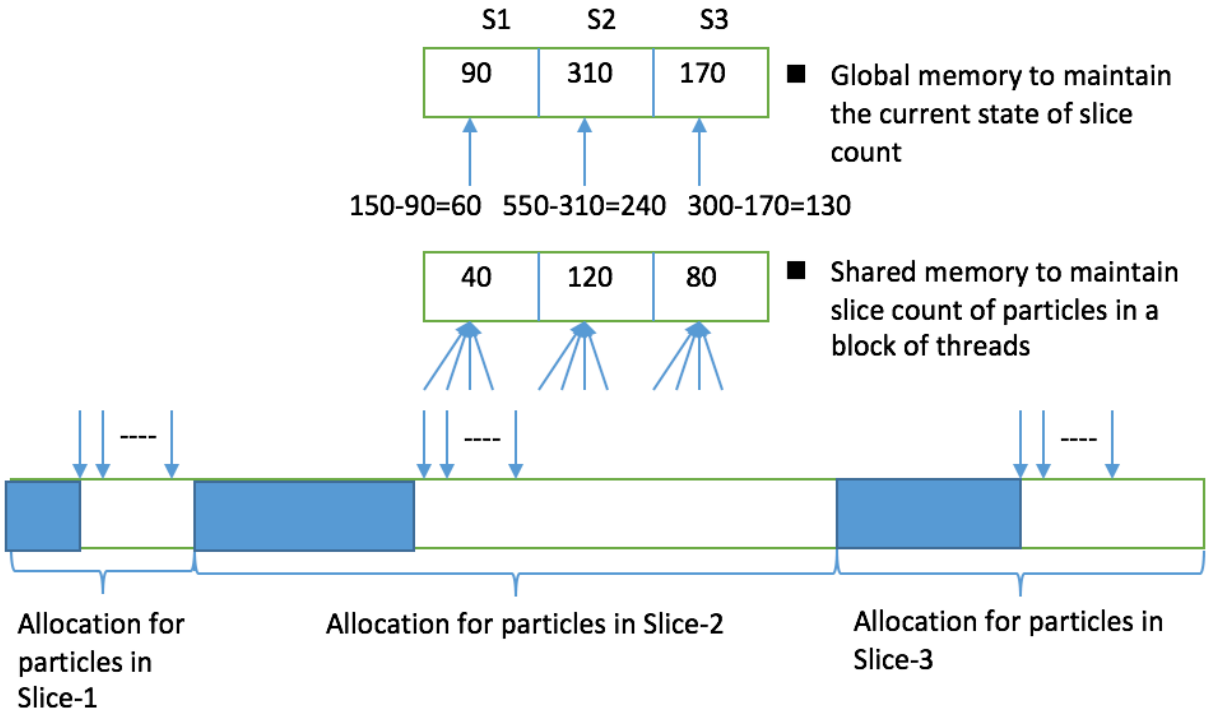


Figure 4.3 - Second block accessing the global counter to write the particles in their allocated regions.

Figure 4.3 illustrates the scenario when another block (say block-y) tries to access the global slice count after block-x. Now the difference between actual total slice count and current status is 60, 240, 330 respectively for Slice-1, 2, and 3. The number of particles in block-y which belong to Slice-1, 2 and 3 are 40, 120 and 80 respectively. So the starting position for particles in block-y that belong to Slice-1 will start from 61 in the area that is specially allocated for Slice-1 particles. Similarly, the starting positions for Slice-2 and 3 particles are 240 and 130 in their respective allocations. Figure 4.4 illustrates the final positions of particles after slicing and sorting them. In this way, all the particles belonging to the same slice are arranged consecutively which

in turn directs the threads to perform a coalesced access memory access in $Compute - Mean - SD$ and $Apply - Kick$ procedures.

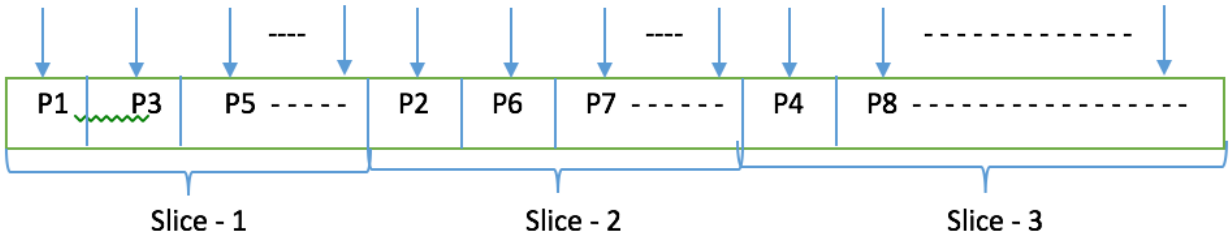


Figure 4.4 – Final positions of particles after slicing and sorting.

IV.2.2 Parallel Apply Kick

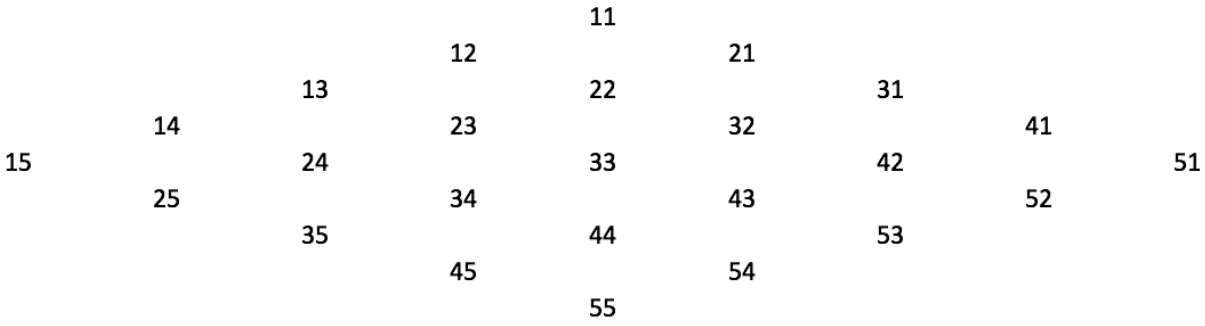


Figure 4.5 – Triangle illustrating the slice-to-slice collisions that can happen in parallel at each stage. In this example, each bunch has 5 slices.

Two main building blocks of $Apply - Kick$ procedure are $Compute - Mean - SD$ and $Compute - Kick$ routines. Figure 4.5 illustrates the slice-to-slice collision process when the beam is divided into 5 slices. When the beam is divided into m slices, there will be a total of $2m$ - 1 stages of slice-to-slice collisions. So, here there are a total 9 stages of slice-to-slice collisions. At each stage, the numbers represent the slices of e- and p-beam that are colliding with each other. For example, in the Stage-2, 1st slice from e-beam is colliding with the 2nd slice of p-beam which is represented as 12, and 2nd slice from e-beam is colliding with the 1st slice from p-beam. In

stage-1 and stage-9, only one pair of slices are colliding with each other where as in all the other stages there are more than one pair of slices that are colliding with each other. We first present the brief outline of what is happening in the $Apply - Kick$ procedure here. When a pair of slices are colliding with each other, the procedure $Compute - Mean - SD$ is called on both the slices which return mean and standard deviations of the slice in x and y directions. Next, the procedure $Compute - Kick$ is called on all the particles of a slice to compute the kick of the opposite slices on each particle and return the forces in x and y directions which are later applied on the particle to get its updated dimensions.

Both routines, $Compute - Mean - SD$ and $Compute - Kick$ are implemented on GPU. At first, we compute the mean and standard deviation (SD) of all the m slices and store them in memory. The routine $Compute - Mean - SD$ is implemented using CUDA based Thrust Library [3] which has powerful and efficient reduction operation implementations. Then at each stage of slice-to-slice collision process, we pass those mean and SDs to the $Compute - Kick$ procedure and calculate the updated mean and SDs of the slices at end of the $Compute - Kick$ procedure. For example, during the Stage-1 the mean and SDs of 1e and 1p are passed to $Compute - Kick$ and at the end of the $Compute - Kick$ procedure we calculate the updated mean and SDs of 1e and 1p and update them in the memory so that we retrieve the correct values of mean and SDs of 1e and 1p when they are participating in collision process again during Stage-2.

During Stage-2 to Stage-8, there are more than one pair of slices participating in the collision process. The sequential way of doing all these pair-wise collisions in a stage is to process each pair-wise collision one after the other. In our GPU implementation, we do all the pair-wise collisions in parallel. We fire up the number of threads that are equal to the number of particles in all the slices and all the thread call the $Compute - Kick$ procedure on their respective particles in

parallel. The advantage of arranging all the particles that belong to the same slice comes into the picture here. As the threads that belong to the same warp access the consecutive memory locations all the memory requests of the threads are completed in at most two transactions.

CHAPTER V

MULTI-GPU IMPLEMENTATION

In this chapter, we present the algorithm for simulating the beam-beam effects in particle colliders where each of the collider rings carry more than one bunch. We refer to the two types of input beams in the following algorithm description as e-beam and p-beam.

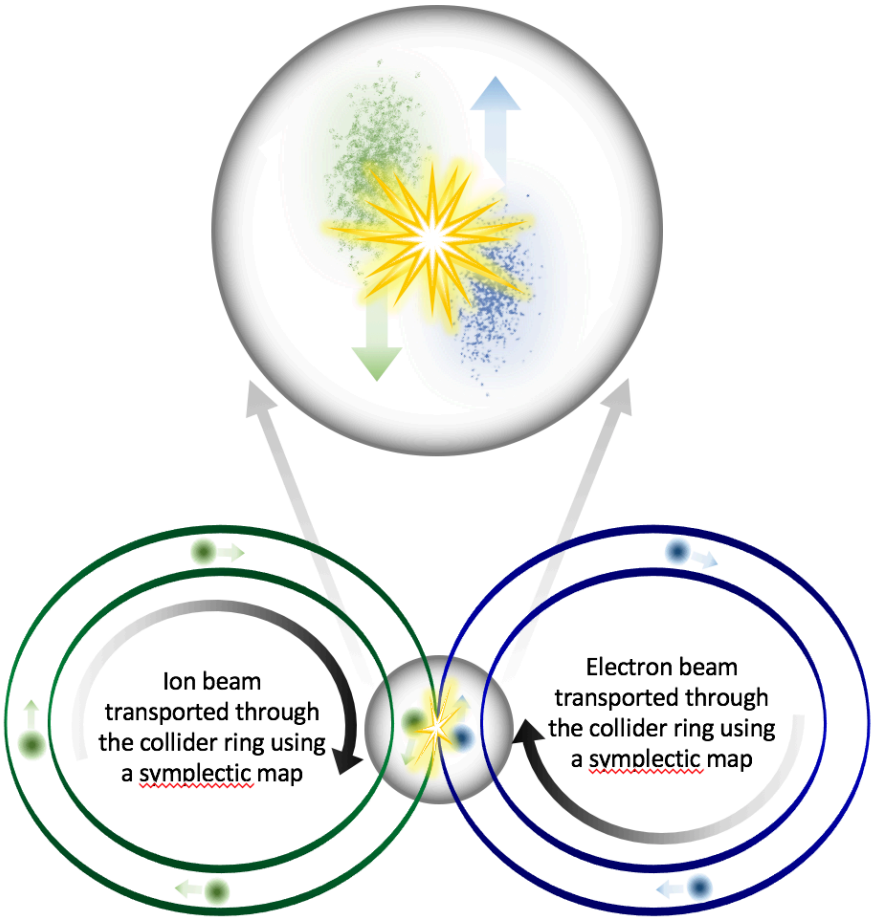


Figure 5.1 – Setup of collider rings with 4p and 3e bunches. Number of e bunches is always one less than number of p bunches.

Figure 5.1 illustrates the setup of two collider rings which has three e-beam and four p-beam bunches. It is to be noted that the number of e-beam bunches is always one less than p-beam bunches. When there are nb p-beam bunches, there will be a total of nb $(nb-1)$ bunch-to-bunch interactions. The interaction between two bunches consists of two major steps - Tracking and Collision as described in Chapter - 3.

Figure 5.2 illustrates the schedule that is repeated for two times for all the $nb(nb-1)$ bunch-to-bunch interactions when $nb = 4$. During iteration – 1, at each turn every e-beam sees a different p-beam and a single iteration is said to be completed after every e-beam sees every p-beam and this also refers to completion of one schedule. Next, from iteration – 2, the same schedule repeats.

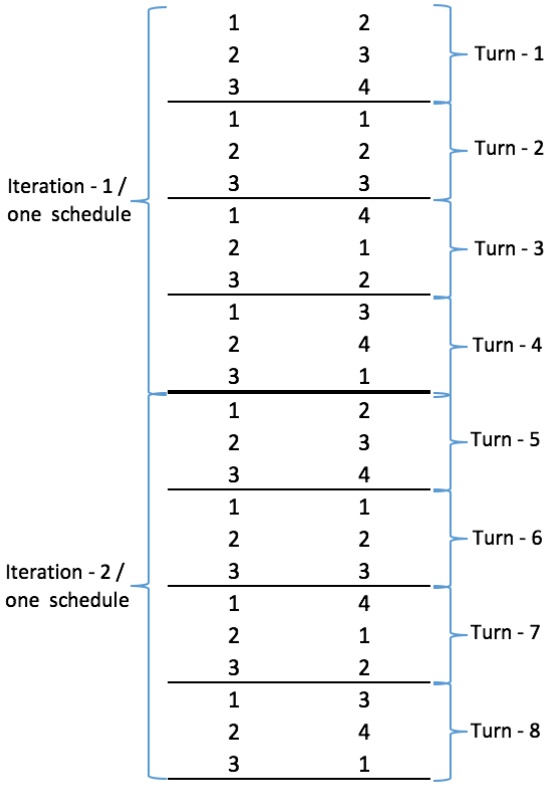


Figure 5.2 – Schedule repeated for two iterations for 3e and 4p bunches. A schedule has 4 turns which is equal to number of p bunches.

Sequential Algorithm

Algorithm 7 - function $Beam - Beam(B_e, B_p, M_e, M_p, d, t, m, nb)$

1. $e\_bunch\_num = 0, p\_bunch\_num = 1$
2. $\boldsymbol{while}\ i < t$:
3. $\quad \boldsymbol{for}\ i = 0\ to\ nb * (nb - 1)\ \boldsymbol{do}$
4. $\quad\quad \boldsymbol{If}\ (e\_bunch\_num >= nb - 1)$:
5. $\quad\quad\quad e\_bunch\_num = 0$
6. $\quad\quad\quad i = i + 1$
7. $\quad\quad \boldsymbol{If}\ (p\_bunch\_num >= nb)$:
8. $\quad\quad\quad p\_bunch\_num = 0$
9. $\quad\quad Track\ (B_e\ [e\_bunch\_num], M_e, d)$
10. $\quad\quad Track\ (B_p\ [p\_bunch\_num], M_p, d)$
11. $\quad\quad Collide\ (B_e\ [e\_bunch\_num], B_p\ [p\_bunch\_num], m)$
12. $\quad e\_bunch\_num++,\ p\_bunch\_num++$
13. $\quad \boldsymbol{end\ for}$
14. $\boldsymbol{end\ while}$
15. $\boldsymbol{end\ function}$

Algorithm - 7 illustrates the pseudo code for the simulation of beam-beam effects when there are multiple bunches. This simulation is driven by the procedure $Beam - Beam$ which takes $(B_e, B_p, M_e, M_p, d, t, m, nb)$ as inputs. Here B_e and B_p are the lists of lists where the outer lists represent the bunches and the inner lists represent the particles in a bunch. As described in Chapter 3, each particle is a six-dimensional object denoting the six phase-space coordinates of that particular particle, M_e and M_p are the transport of maps of e- and p-beams respectively, d is the

dimension of the particles, t is the number of turns required for the simulation which should be the multiple of nb, and m is the number of slices required for the collision step of the simulation. The schedule given above is implemented sequentially using the $for - loop$ in this procedure by passing the respective e- and p-beams to the Track and Collide procedures and their working is described in Chapter 3.

Multi-GPU Approach

The set of interactions happening within each turn of the schedule are independent of each other and the interactions happening in different turns are dependent on each other. For example, in Figure – 5.2, the interaction (1,1) of Turn – 2 cannot happen until interaction (1,2) of Turn-1 finishes, as both the e-beams are same here. So the idea is to simulate all the interactions in a single turn simultaneously on a cluster of GPUs. The pair of beams interacting at each turn is different from the previous iteration. The naive way of running all these turns on multiple GPUs is to move the bunches between GPUs for each turn and perform the simulations. We propose an algorithm where the bunch is stored on a particular GPU throughout the simulation without further moving the bunch data between GPUs for each turn. Once the GPU for all the bunches is decided, during the phase of beam-beam interaction we only move the parameters between the GPUs that are required to apply the interaction effects on the bunches. A robust scheduling algorithm is necessary to schedule all the interactions on the GPUs in a turn without any delay and make neither of the beams wait for the parameters from the opposite GPU. Table – 5.1 depicts the GPU assignment and Figure – 5.3 illustrates the scheduling algorithm for a single turn when there are 6 GPUs and 15 bunches.

GPU 1	1	2	3
GPU 2	4	5	6
GPU 3	7	8	9
GPU 4	10	11	
GPU 5	12	13	
GPU 6	14	15	

Table 5.1 - Distribution of Bunches on GPUs (6 GPUs, 15 Bunches).

V.1 SCHEDULING BUNCHES

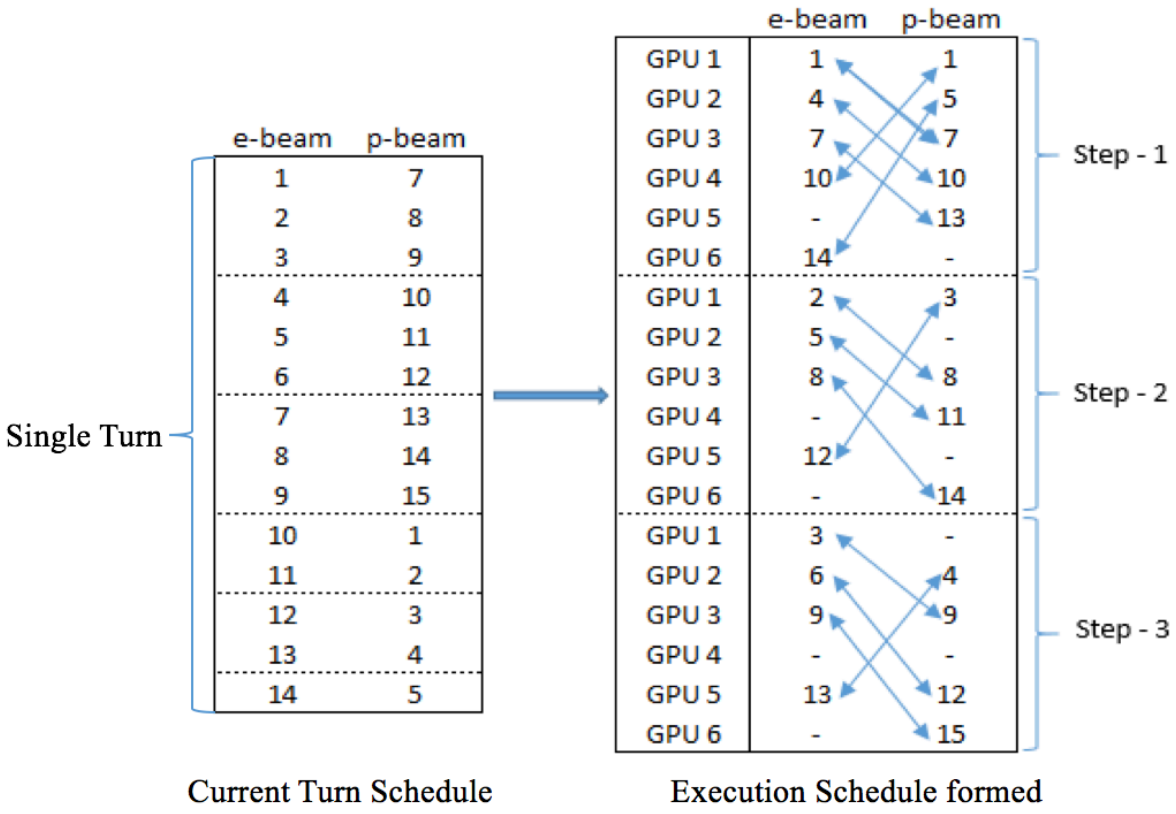


Figure 5.3 – Schedule of a single random turn when there are 15 bunches, and the execution schedule formed by the scheduling algorithm using the Current Turn Schedule.

The first step of this scheduling is to logically divide all the interactions in a turn according to the GPU where the e-beam is stored. For example, (1, 2, 3) e-beams are kept in the same logical

partition (lg) as they are all stored in the same GPU. This division remains constant all over the simulation as the order of the e-beams is same for every turn. As there is a maximum of 3 beams stored in a GPU, this turn is scheduled to finish the execution in 3 steps because a GPU can process only one interaction at a given time. The execution schedule in the Figure-schedule illustrates the order in which all the interactions in the turn are executed in three steps. At each step, a GPU selects a set of e- and p-beams in an order of from a lg for execution. For example, during step – 1, GPU – 2 scans line-1 of lg-2 (as e-beams that belong to GPU – 2 are present in lg-2 in any given turn) and selects 4^{th} e-beam for execution. Next, it scans line-1 of each lg and selects 5^{th} p-beam from lg-6 as it belongs to GPU – 2. The multi-GPU algorithm is capable of halting the execution of one or both the beams for that particular step. The symbol '-' indicates that the particular beam slot for the GPU is empty during that step. There are two possible reasons for that:

Reason: 1. During step - 1, there are two p-beams (1, 3) that appear in line-1 in their logical partition and they belong to the same GPU (GPU - 1). Due to this reason, only one of the beam will be scheduled to execute on GPU - 1 for that particular step. In this case, p-beam - 1 is selected for execution during step - 1, and p-beam - 3 is scheduled to execute in step - 2. Because of this the execution for e-beam - 12 which is interacting with p-beam - 3 is also scheduled to execute in step - 2 which is why the e-beam slot for GPU - 5 is empty. The p-beam slot for GPU - 6 is empty as none of the logical partitions have p-beam that belongs to this GPU that appears in line-1.

Reason: 2. All the GPUs have finished the execution of their e- and p-beams for that particular turn. For example, GPU - 4 has completed the execution of all of its e- and p-beams.

The bi-directional arrows in each step indicate the GPUs that are communicating with each other to execute that respective interaction. For example, during Step - 1, GPU - 2 and GPU - 4 are communicating with each other to execute the interaction 4 - 10 which appears in line-1 of lg-

2. Once this execution schedule shown in Figure – 5.3 is formed, the Track and Collide procedures for e- and p-beams are executed on the GPUs according to the execution schedule.

Algorithm 8 – function $Beam - Beam - MultiGPU(B_e, B_p, M_e, M_p, d, t, m, n_b, GPUofBeam, MaxBunchGPU, numGPU)$

1. ***while*** $i = 1$ *to* t:
2. $total\_ij = 0$
3. ***for*** $i = 1$ *to* n_b ***do***
4. ***for*** $j = 1$ *to* $n_b - 1$ ***do***
5. $cur\_e\_schd[j] = e\_sch[total\_ij]$
6. $cur\_p\_schd[j] = p\_sch[total\_ij]$
7. $total\_ij = total\_ij + 1$
8. ***end for***
9. ***for*** $gpu = 1$ *to* $numGPU$ ***parallel do***
10. ***for*** $step = 1$ *to* $MaxBunchGPU[gpu]$ ***do***
11. ***if*** $step != 1$
12. $e\_b, opp\_p\_b \leftarrow$ pop any remained e-bunch and its opposite p-bunch in line-(step-1) from division-GPU of cur_e_schd
13. ***else***
14. $e\_b, opp\_p\_b \leftarrow$ pop e-bunch and its opposite p-bunch in line-step from any division-GPU of cur_e_schd
15. ***if*** $step != 1$
16. $p\_b, opp\_e\_b \leftarrow$ pop any remained p-bunch and its opposite e-bunch in line-(step-1) from division-GPU of cur_p_schd

17. ***else***

18. $p\_b, opp\_e\_b \leftarrow$ pop p-bunch and its opposite e-bunch in line-step from any division-GPU of cur_p_schd

19. $Track(B_e[e\_b], M_e, d)$

20. $Track(B_p[p\_b], M_p, d)$

21. $Collide(B_e[e\_b], B_p[p\_b], GPUofBeam[opp\_e\_b], GPUofBeam[opp\_p\_b])$

22. ***end for***

23. ***end for***

24. ***end for***

25. ***end while***

26. ***end function***

Algorithm - 8 illustrates the Multi-GPU version of $Beam - Beam$ algorithm that includes the scheduling algorithm and the execution of $Track$ and $Collide$ functions after the formation of execution schedule for each step. Here $GPUofBeam$ is list of list where the outer list represents each GPU and the inner list represents the bunches store in that particular GPU, $MaxBunchGPU$ is the maximum out of count of bunches stored in each GPU, and $numGPU$ is the number of GPUs in a cluster. At first the current turn schedule is formed using the for-loop in the lines 4-10. At each step, every GPU will extract the bunch numbers of e and p from the current turn schedule and executes the $Track$ and $Collide$ functions, and the algorithmic steps related to this are described from lines 9-23. The $Track$ procedure is independent and does not need any communication from other GPUs. Hence the implementation of $Track$ procedure is same as described in Chapter - 3. The $Collide$ procedure has three auxiliary procedures called $Slice, Apply - Kick, Merge$, out of which the $Slice$ and $Merge$ procedures does not need any communication from other GPUs and

their implementation remains same as described in Chapter - 3. Also, the execution flow and parallelism of for-loops calling the procedure $Apply - Kick$ remains the same. As a refresher, the for-loops in the $Collide$ procedure calculates the beam-beam effects (or kicks) for every pair of colliding slices, where the kick computation on all particles in a pair of colliding slices is implemented using the procedure $Apply - Kick$.

V.2 COMMUNICATIONS USING MESSAGE PASSING INTERFACE (MPI)

For our multi-bunch algorithm to simulate multi-bunch collisions, we used MPI [18] to exchange the messages between the GPUs

Algorithm 9 - function $Apply - Kick(S_e, S_p, s, opp\_eGPU, opp\_pGPU)$

25. $(\bar{x}^e, \bar{y}^e, \bar{\sigma}_x^e, \bar{\sigma}_y^e) \leftarrow Compute - Mean - SD(Se, s)$
26. $(\bar{x}^p, \bar{y}^p, \bar{\sigma}_x^p, \bar{\sigma}_y^p) \leftarrow Compute - Mean - SD$(Sp, s)
27. Send $(\bar{x}^e, \bar{y}^e, \bar{\sigma}_x^e, \bar{\sigma}_y^e)$ to opp_pGPU
28. Send $(\bar{x}^p, \bar{y}^p, \bar{\sigma}_x^p, \bar{\sigma}_y^p)$ to opp_eGPU
29. Receive $(\bar{x}^e_{rcvd}, \bar{y}^e_{rcvd}, \bar{\sigma}^e_{x\_rcvd}, \bar{\sigma}^e_{y\_rcvd})$ from opp_eGPU
30. Receive $(\bar{x}^e_{rcvd}, \bar{y}^e_{rcvd}, \bar{\sigma}^e_{x\_rcvd}, \bar{\sigma}^e_{y\_rcvd})$ from opp_pGPU
31. $\boldsymbol{for}$ each particle $e \in Se$ $\boldsymbol{parallel\ do}$
32. $(F_x, F_y) \leftarrow Compute - Kick(e, \bar{x}^p_{rcvd}, \bar{y}^p_{rcvd}, \bar{\sigma}^p_{x\_rcvd}, \bar{\sigma}^p_{y\_rcvd})$
33. $e[0] \leftarrow e[0] - s * Fx$
34. $e[1] \leftarrow e[1] - Fx$
35. $e[2] \leftarrow e[2] - s * Fy$
36. $e[3] \leftarrow e[3] - Fy$

37. ***end for***

38. ***for*** each particle $p \in Sp$ ***parallel do***

39. $(F_x, F_y) \leftarrow Compute - Kick(p, \bar{x}^e_{rcvd}, \bar{y}^e_{rcvd}, \bar{\sigma}^e_{x\_rcvd}, \bar{\sigma}^e_{y\_rcvd})$

40. $p[0] \leftarrow e[0] + s * Fx$

41. $p[1] \leftarrow p[1] + Fx$

42. $p[2] \leftarrow p[2] + s * Fy$

43. $p[3] \leftarrow p[3] + Fy$

44. ***end for***

45. ***end function***

The procedure $Apply - Kick(S_e, S_p, s, opp\_eGPU, opp\_pGPU)$ which is the heart of $Beam - Beam$ algorithm is dependent on the communication from other GPUs. S_e and S_p are the slices of e- and p-beam respectively, s is the interaction point of both the slices, opp_eGPU and opp_pGPU are the GPUs that the particular GPU has to communicate with to receive the mean and standard deviations of opposite e- and p-beams. The modified algorithm of $Apply - Kick$ is described in Algorithm 9. Initially, the procedure $Compute - Mean - SD$ is called to calculate the mean and standard deviations of both e- and p-beam along the first and third dimensions of the particles in S_e and S_p. Then these calculated mean and standard deviations are sent to the respective GPUs where the colliding e- and p-beams are stored. In the next step the GPU receives the mean and standard deviations of colliding e- and p-beams from the same set of GPUs to which it sent the parameters earlier. Then, for each particle e belongs to S_e, the kick from all the particles in the colliding slice of the p-beam which is potentially residing on the other GPU is calculated using the procedure $Compute - Kick$. This procedure takes $\bar{x}^p_{rcvd}$, $\bar{y}^p_{rcvd}$, $\bar{\sigma}^p_{x\_rcvd}$, $\bar{\sigma}^p_{y\_rcvd}$ as inputs and

returns a pair (Fx, Fy). The output values Fx and Fy represents the kick from particles in the slice of the opposite colliding p-beam and it is calculated using Equations (3) to (7) described in Chapter - 2. These computed kicks are used to update the particles in S_e using the first for loop. Similarly, in the next for loop, kick on all the particles in S_p due to the particles in the slice of the colliding e-beam which is potentially residing on other GPU is computed using the $Compute - Kick$ procedure and the output from this procedure is used to update all the particles in S_p.

CHAPTER VI

RESULTS

In this chapter, we discuss the performance of both the implementations. It is to be noted that the Single-GPU implementation is only for Single Bunch simulations on a single GPU and the Multi-GPU implementation is only for Multi-Bunch simulations on Multiple GPUs.

We used NVIDIA Tesla K40 GPUs to run the simulations using Single-Bunch and Multi-Bunch algorithms. Each GPU on a cluster is hosted on a standalone desktop machine with NVIDIA Tesla K40 GPU hosted on a multi-core CPU platform with two Intel® Xeon® E5-2630 v4 processors, where each E5-2630 v4 processor consist of 10 cores, making a total of 20 CPU cores for the multi-core platform. The Tesla K40 used in this study is a GK110B GPU-processor based on the popular Kepler microarchitecture [8]. 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. We use double-precision for all the floating-point arithmetic in our implementation of beam-beam effects simulation. The results reported in this Chapter illustrates the performance for a single turn of the simulation that is averaged over multiple turns of the entire beam-beam dynamics simulation, which in practice runs for millions to billions of turns.

VI.1 SINGLE-GPU PERFORMANCE

The performance of our parallel implementation of $Beam - Beam$ procedure on a single GPU is evaluated against an existing out-of-the-box code that was developed to establish the proof-

of-concept of beam-beam interactions in particle colliders using Bassetti-Erskine approximation [11] [15]. It is important to note that this sequential simulation code is a single threaded implementation, and it is not optimized to take advantage of the multiple cores of CPU architectures. In order to establish a fair comparison, we used OpenMP to develop a naively parallel implementation of this sequential code that uses all the cores of the underlying multi-core CPU architecture and delivers near-linear speedup in the number of cores used. We use this multi-core implementation on 20 CPU cores, along with the sequential implementation on a single CPU core to analyze the performance of our parallel implementation on K40 GPU.

Number of particles per beam	Execution Time (sec.)									Overall Speedup		
	Sequential (Single CPU core)			Multi-core (20 CPU cores)			GPU			$\frac{\text{Sequential}}{\text{Multi-core}}$	$\frac{\text{Sequential}}{\text{GPU}}$	$\frac{\text{Multi-core}}{\text{GPU}}$
	Tracking	Collision	Overall	Tracking	Collision	Overall	Tracking	Collision	Overall			
10000	1.51	0.13	1.64	0.15	0.04	0.18	0.001	0.02	0.02	9	82	9
100000	15.67	1.29	16.96	0.96	0.21	1.17	0.006	0.03	0.04	15	424	30
1000000	154.40	12.10	166.50	7.93	1.59	9.52	0.054	0.15	0.21	17	800	46
10000000	1544.96	131.94	1676.90	77.68	14.33	92.10	0.556	1.35	1.91	18	880	48

Table 6.1 - Single turn performance results of sequential (on a single CPU core), multi-core CPU (on 20 CPU cores), and GPU implementation (on K40 GPU) of beam-beam dynamics simulation that is averaged over multiple turns for varying number of particles with the number of slices fixed to $m = 6$.

Table 6.1 illustrates the single turn performance results of sequential (on a single CPU core), multi-core CPU (on 20 CPU cores), and GPU implementation (on K40 GPU) of the $Beam - Beam$ procedure for varying number of particles with the number of slices fixed to m=6. The tracking time reported in Table 6.1 is the combined execution time of the two $Track$ calls in $Beam - Beam$ procedure, and the collision time is the execution time of the $Collide$ procedure. The results indicate that depending on the number of particles, parallel implementation of beam-beam effects on GPU achieves two to three orders of magnitude speedup when compared against the non-optimized sequential simulation. This speedup behavior is also illustrated in Figure 6.1 with a blue colored plot. On the other hand, GPU implementation achieves two orders-of-magnitude speedup when compared against the multi-core CPU implementation.

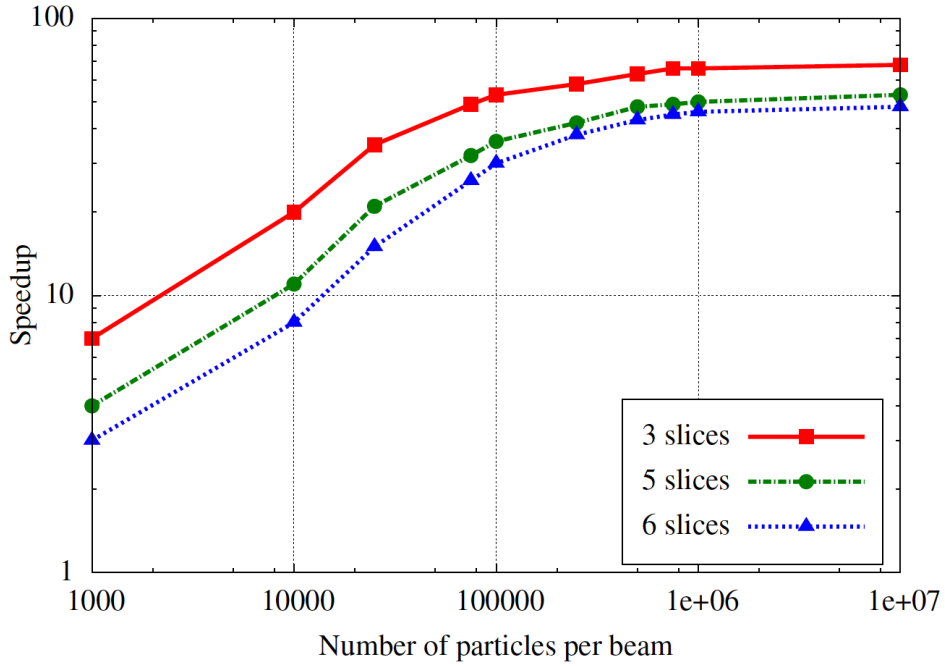


Figure 6.1 - Speedup behavior of the GPU implementation compared against the multi-core CPU implementation (on 20 CPU cores) for different number of slices with varying number of particles.

Figure 6.1 shows the simulation speedup behavior of the GPU implementation for different number of slices with varying number of particles. From Figure 6.1 and Table 6.1, we notice that speedup of the GPU implementation increases near linearly up to one million particles and it saturates beyond that. This behavior is independent of the number of slices considered in the simulation. The reason for this behavior is that amount of thread-level parallelism offered by the GPU implementation and the device utilization has a linear dependence on the number of particles in the simulation. In other words, fewer number of particles per beam leads to a underutilized GPU which results in poor to suboptimal performance, and the device utilization (or occupancy) grows near linearly with the number of particles which leads to a proportional increase in the performance. The current implementations on K40 GPU achieves full occupancy at approximately one million particles, and any increase in the input size beyond this point results in a serialized execution on the GPU, thereby deviating from the linear speedup growth. Note that the point of

saturation for a given implementation depends on the GPU and it often varies with each target architecture.

Tracking Performance

The split execution time in Table 6.1 shows that tracking in the GPU implementation is two to three orders of magnitude faster than the sequential implementation, and it is two orders of magnitude faster than the multi-core CPU implementation. The main reason for such a large performance gain is that the parallel implementation on GPU is highly optimized to take advantage of the data parallel nature of $Track$ procedure, whereas sequential and multi-core implementation is a proof-of-concept code that is not optimized for performance. In particular, data parallelism in $Track$ procedure is exploited in the GPU implementation by mapping the computation of particles to parallel threads with one-to-one correspondence such that it minimizes both branch and memory divergence on GPUs, which leads to the effective utilization of GPU resources. In addition, data reuse is maximized by using shared memory to store the shared transport map. These performance optimizations together with the massive parallelism offered by the GPU architectures results in the large performance gain.

Collision Performance

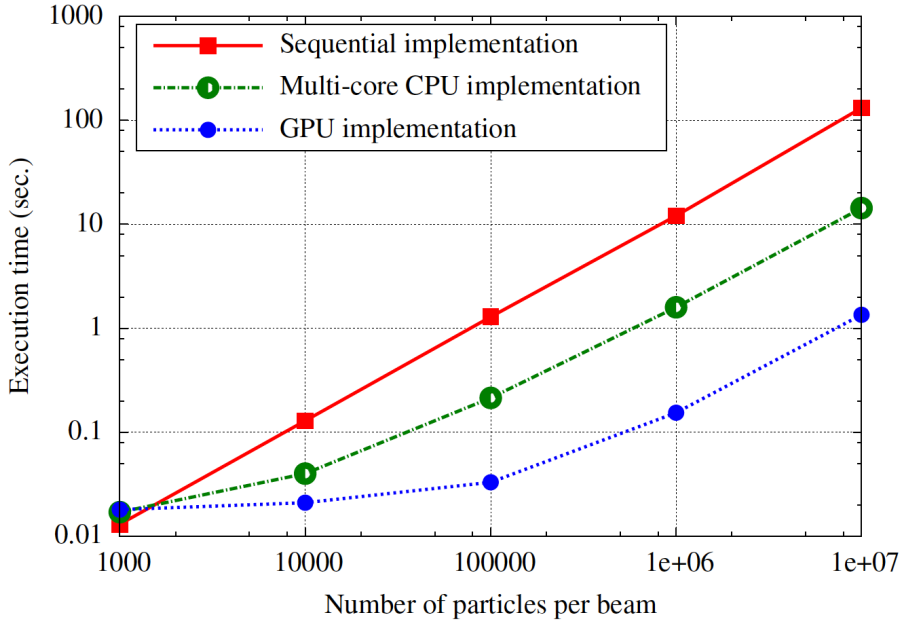


Figure 6.2 - Execution time of *Collide* procedure in sequential (on a single CPU core), multi-core CPU (on 20 CPU cores), and GPU (on Tesla K40) implementation for varying number of particles with the number of slices fixed to $m = 6$.

Figure 6.2 illustrates the execution time of the *Collide* procedure in both sequential and GPU implementations from Table 6.1 for varying number of particles. We notice that the collision time in the sequential implementation is proportional to the number of particles in the simulation, in other words, collision time in sequential code grows linearly with the input size. This behavior of the sequential code is expected, as the number of operations (floating-point and integer) involved in the *Collide* procedure is proportional to the number of particles. On the other hand, collision time in the GPU implementation exhibits a non-linear behavior with the number of particles. The reason for this behavior is that the GPU implementation simulates the collision between two input beams by executing a slice-to-slice collision on a subset of slices at a given time, where the number of threads, operations and data parallelism used on GPU is proportional to the number of particles involved in the current set of colliding slices. This number typically

depends on the particle distribution, and it varies from slice to slice and from turn to turn. As a result, when there are fewer number of particles in the colliding slices, it leads to a underutilized GPU, and the utilization improves as the number of particles increase. For example, for collision in Figure 2.2 each row represents the collision on a subset of slices that is executed on GPU in parallel where the performance depends on the number of particles involved in each row. It is evident from figure that the number of particles participating in the collision increases from the top to the center, and then decreases from the center to the bottom. In other words, occupancy and device utilization starts with a minimum value at the top row and increases as we move to the center row, and then it starts to decrease from the center to the bottom row. This variation in the utilization results in the non-linear increase in the execution time on GPU.

Number of particles per beam	Number of slices per beam	Collision time (sec.)			Speedup		
		Sequential (Single CPU core)	Multi-core (20 CPU cores)	GPU (Tesla K40)	$\frac{\text{Sequential}}{\text{Multi-core}}$	$\frac{\text{Sequential}}{\text{GPU}}$	$\frac{\text{Multi-core}}{\text{GPU}}$
100000	3	0.62	0.114	0.014	5	44	8
	5	1.02	0.194	0.026	5	39	7
	6	1.23	0.214	0.033	6	37	6
1000000	3	6.11	0.863	0.078	7	78	11
	5	10.09	1.378	0.131	7	77	11
	6	12.10	1.594	0.154	8	78	10
10000000	3	61.05	8.045	0.710	8	85	11
	5	110.12	12.014	1.136	9	97	11
	6	131.94	14.328	1.349	9	98	11

Table 6.2 - Single turn performance of the sequential, multi-core CPU, and GPU implementation of COLLIDE procedure in the beam-beam effects simulation for different input configurations.

Table 6.2 illustrates single turn performance of the sequential (on a single CPU core), multi-core CPU (on 20 CPU cores), and GPU implementation (on K40 GPU) of *Collide* procedure in the beam-beam effects simulation for different input configurations. The results indicate that, depending on the number of particles and slices, GPU implementation of *Collide* procedure delivers a speedup gain of up to 98X and 11X when compared to non-optimized sequential and multi-core CPU implementation, respectively.

	With Max-Reg=48				Without Max-Reg			
Number of Particles	10000	100000	1000000	10000000	10000	100000	1000000	10000000
Kernel Time	3.16	8.83	65.24	616.01	4.21	12.76	97.53	947.04
Gflops/Sec	35.19	145.26	198.04	209.73	30.49	100.49	131.54	135.61
Registers	48	48	48	48	82	82	82	82
Global Memory Load Efficiency	88.50%	88.55%	88.56%	88.56%	88.50%	88.55%	88.56%	88.56%
Global Load Transactions Per Request	1.42	1.42	1.42	1.42	1.42	1.42	1.42	1.42
Local Memory Overhead	64.83%	78.68%	79.17%	79.17%	10.25%	4.76%	1.04%	0.60%
Warp Execution Efficiency	40.33%	40.75%	40.86%	40.84%	40.15%	40.16%	40.37%	40.28%
Achieved Occupancy	0.31	0.51	0.60	0.62	0.19	0.28	0.31	0.31

Table 6.3 – Performance results of $Compute - Kick$ kernel with and without maximum registers per thread limitation.

Table – 6.3 illustrates the performance of $Compute - Kick$ Kernel on NVIDIA Tesla K40 GPU for a different number of input particles and with/without the maxregcount option given during compilation. In NVIDIA CUDA Compiler, using the maxregcount option we can limit the number of registers used by a thread to a particular number. These performance metrics are extracted from NVIDIA Profiler.

Analysis Using Roofline Model

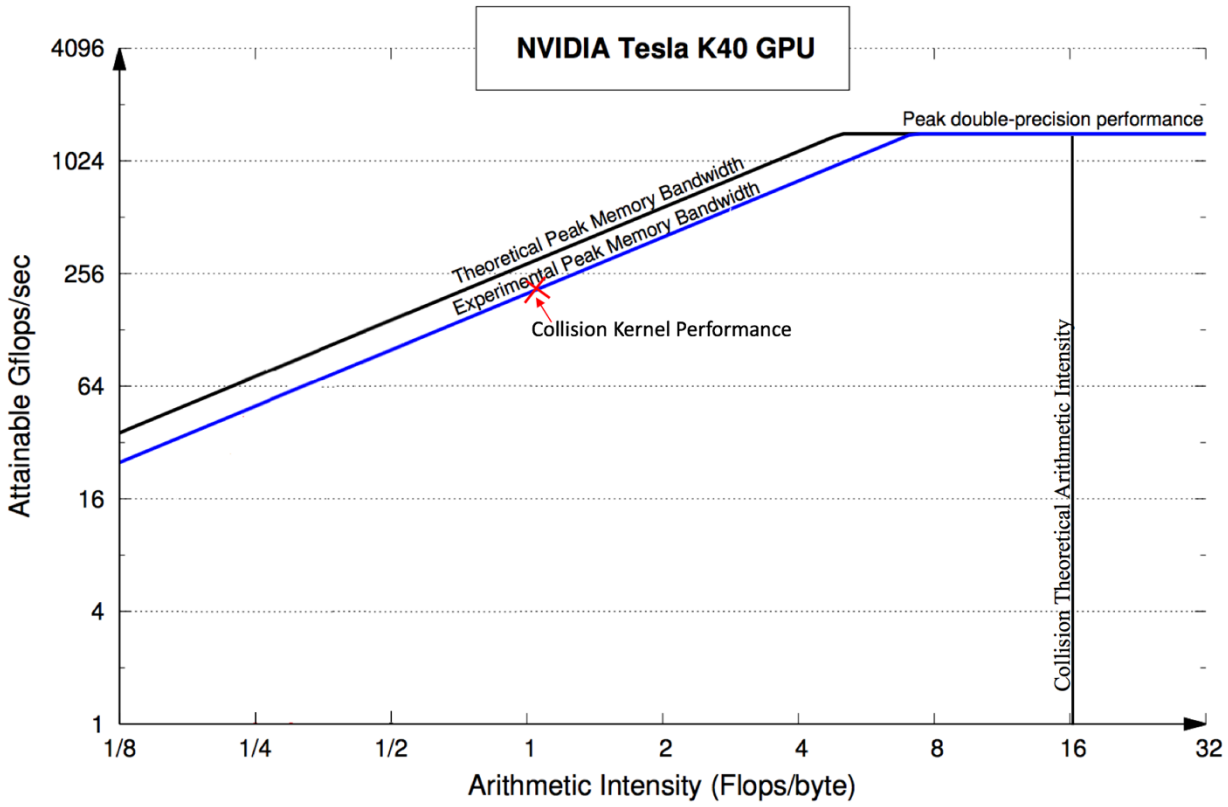


Figure 6.3 – Roofline model analysis for $Compute - Kick$ kernel on NVIDIA Tesla K40 GPU.

Figure 6.3 shows the Roofline model for K40 GPU. The graph is on a log-log scale. The *y*-axis is attainable double-precision floating-point performance in units of Gflops/Sec, and the *x*-axis is arithmetic intensity, varying from 0.125 Flops/DRAM byte-accessed to 32 Flops/DRAM byte-accessed. The system being modeled has a peak double precision floating-point performance of 1.4 Tflops/sec and peak memory bandwidth of $BW_{Theoretical\text{-}Peak}$ = 288 GB/Sec from hardware specifications. The black solid line in Figure 2 indicates the bandwidth ceiling for $BW_{Theoretical\text{-}Peak}$. However, the peak theoretical 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 [19]. Experimental memory bandwidth for K40 is calculated to be $BW_{Experimental\text{-}Peak}$ = 200 GB/sec, and its bandwidth ceiling in roofline model is

shown using the blue solid line plot. The black vertical line indicates the theoretical arithmetic intensity of $Compute - Kick$ kernel.

The theoretical arithmetic intensity of the kernel is around 17 Flops/byte. This indicates that Collision is a compute bound kernel. This is because only the dimensions of the particles, Mean and Standard Deviation of the opposite slice are loaded from the memory in the beginning of the kernel and later there are no other memory accesses performed in between the computations. Our implementation has achieved a performance of around 210 GFlops/sec. This poor performance of the kernel is because of the local memory overhead and Warp Divergence happening inside the kernel. In addition to these two metrics all the other metrics from Table 6.3 are discussed below.

Occupancy

We notice that when maxregcount option is disabled, the number of registers used by each thread are 82 where as the ideal number lies around 32 for K40 GPU. Due to this excessive use of registers, each SM is limited to executing a lower number of blocks simultaneously resulting in low occupancy of GPU. Because of this, the kernel is able to achieve a peak occupancy of only 0.3 and a peak arithmetic intensity of 0.68. When the number of registers used by each thread are limited to 48, the kernel has achieved a peak occupancy of 0.62 and a peak arithmetic intensity of 1.05. Also, we observe that occupancy of the kernel increases with increase in a number of input particles. Hence, occupancy is one of the major contributors of kernel's performance.

Local Memory Overhead

We notice that when the maxregcount option is disabled, there is no local memory overhead. But, when we limit the number of registers per thread to 48, SM runs out registers and starts spilling into Local memory resulting in increased memory traffic. Even though the kernel time here is less when compared to the kernel time for which the maxregcount option is disabled, the local memory overhead remains one of the main factors of the kernel's poor performance in this case.

Control-Flow Divergence

Table 6.3 illustrates the Warp Execution Efficiency for different input configurations. We observe that the Warp Execution Efficiency remains constant and is only 40%. The reason for this poor efficiency is, that we used an existing algorithm for the complex error function in our GPU implementation. We extracted some information from NVIDIA Profiler that shows the information about the divergence happening inside the error function. We see that in Figure 6.4 at line 1195 and 1196 of Collide.cu file which has the source code of $Compute - Kick$ kernel, 100% of the threads are executing the error function (WOFZ). Then, in Figure 6.5, inside the error function, at line 49, we observe that 96% of threads are active, and in Figure 6.6 at line 60, out of that 96%, only 57% of threads are executing the for-loop. Later, in Figure 6.7, the else condition at line 73 is executed by remaining 4% of threads and all the other 96% of threads are inactive during this time. As the computations inside the error function are the major contributors for the overall computations inside the $Compute - Kick$ kernel, the Control-Flow Divergence inside the error function is one of the main reasons for the poor kernel performance.

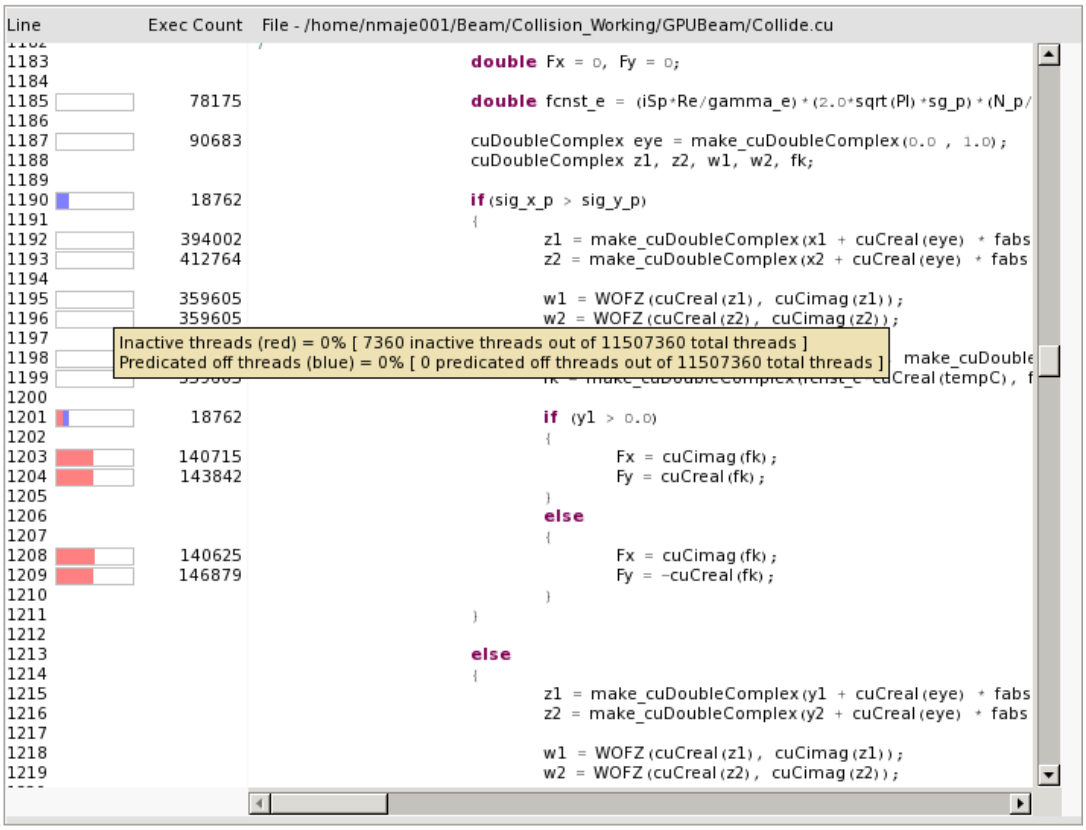


Figure 6.4 – All the threads are calling error function (WOFZ) at lines 1195 and 1196.

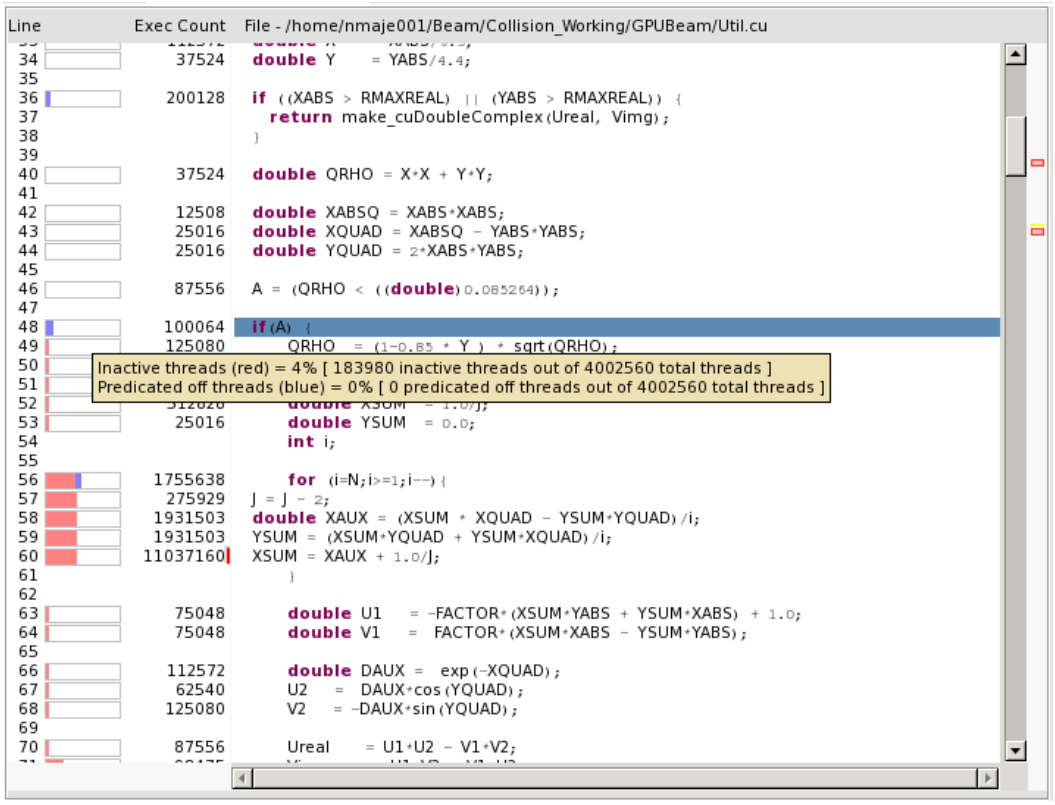


Figure 6.5 – 4% of inactive threads at line 49.

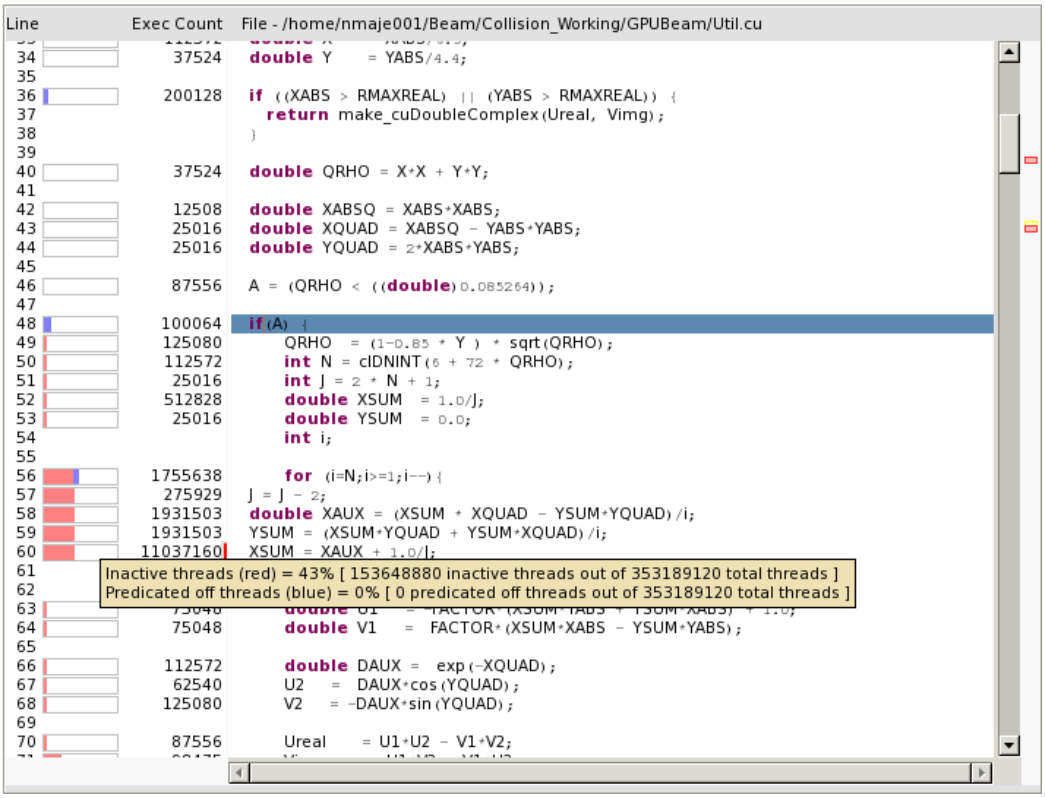


Figure 6.6 – Out of 96% active threads, 43% of threads are inactive inside the for-loop.

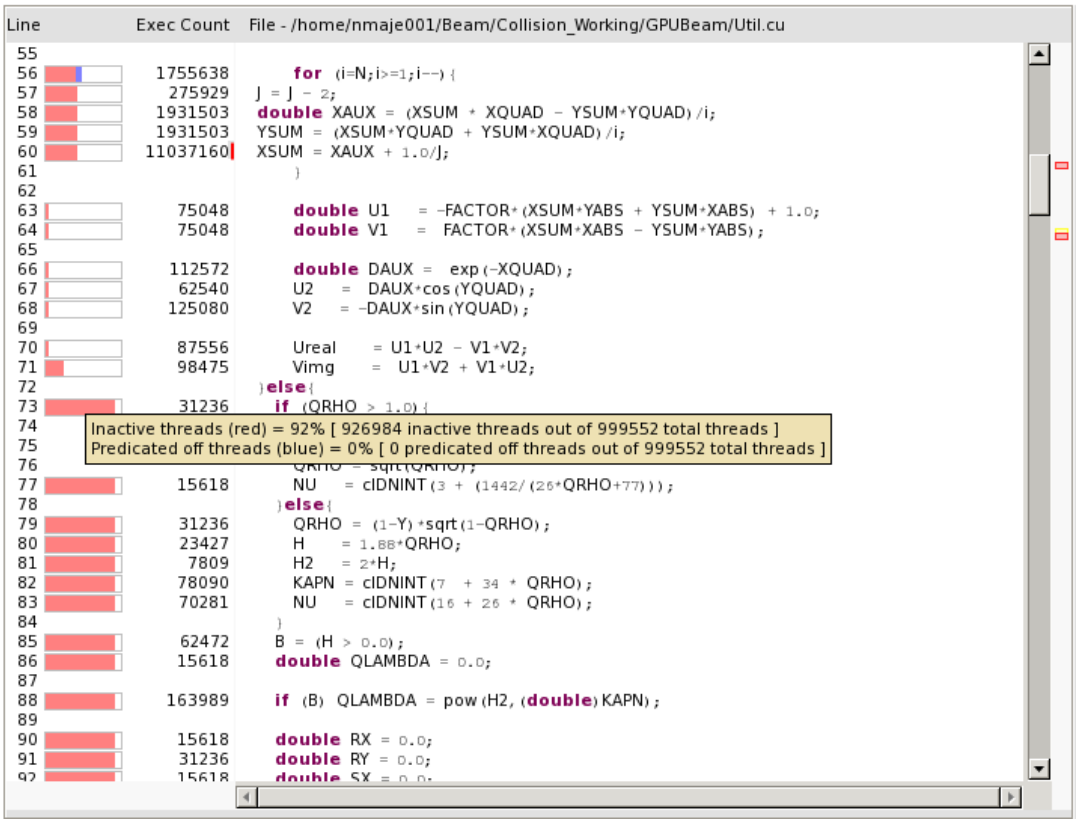


Figure 6.7 – 92% of threads are inactive at line 73.

Memory Performance

The threads inside the $Compute - Kick$ kernel access the global memory only once during the beginning to fetch the particle information. Except these, there are no other global memory fetches performed by the threads inside the kernel. The initial fetches from global memory are perfectly coalesced because of which we observe a global load efficiency of 88% from Table 6.3. Also, the ideal number of global transactions per request is 2 for 8-byte words and we achieved 1.42 which is much closer to the ideal value. This shows that the kernel performs very well in terms of accessing global memory efficiently and is one of the positive contributors for kernel's performance.

From the above analysis, it is clear that the performance of the kernel is limited by the number of registers that each thread is using inside the kernel. Limiting the number of registers

increases the occupancy decreasing the time taken by the kernel. But, as SM runs out of registers and spills the variables into local memory, the traffic created due to the local memory access remains one of the limiting factors of kernel's performance. In addition to the Local Memory overhead, Control-Flow-Divergence also remains as the limiting factor of kernel performance.

VI.2 MULTI-GPU PERFORMANCE

We performed experiments on Multiple-GPUs to see how the Multi-Bunch simulation code scales with the number of GPUs on a cluster. Table – 6.4 illustrates the performance of Multi-GPU algorithm when the number of GPUs and bunches are increased in the powers of 2. All the performance results reported for Multi-GPU algorithm are for one iteration / single schedule, 100K Particles per Bunch, and 3 Slices. We observe that except for a small number of bunches, the Multi-GPU algorithm scales nearly linearly with the number of GPUs. The reasons for the non-linear behavior for a smaller number of bunches and near linear behavior for higher bunch numbers are discussed below.

Time taken in Sec (100K Particles per Bunch, 3 slices)											
	Bunches	2	4	8	16	32	64	128	256	512	1024
Number of GPUs	1	0.05	0.27	1.26	5.46	22.88	93.38	372.36	1479.15	-	-
	2	0.04	0.17	0.70	2.90	11.96	48.11	192.85	762.94	2866.93	-
	4	-	0.09	0.38	1.52	6.13	24.53	98.13	391.47	1474.87	5688.23

Table 6.4 – Performance of Multi-GPU algorithm on a cluster of GPUs. GPUs and bunches are increased in powers of 2.

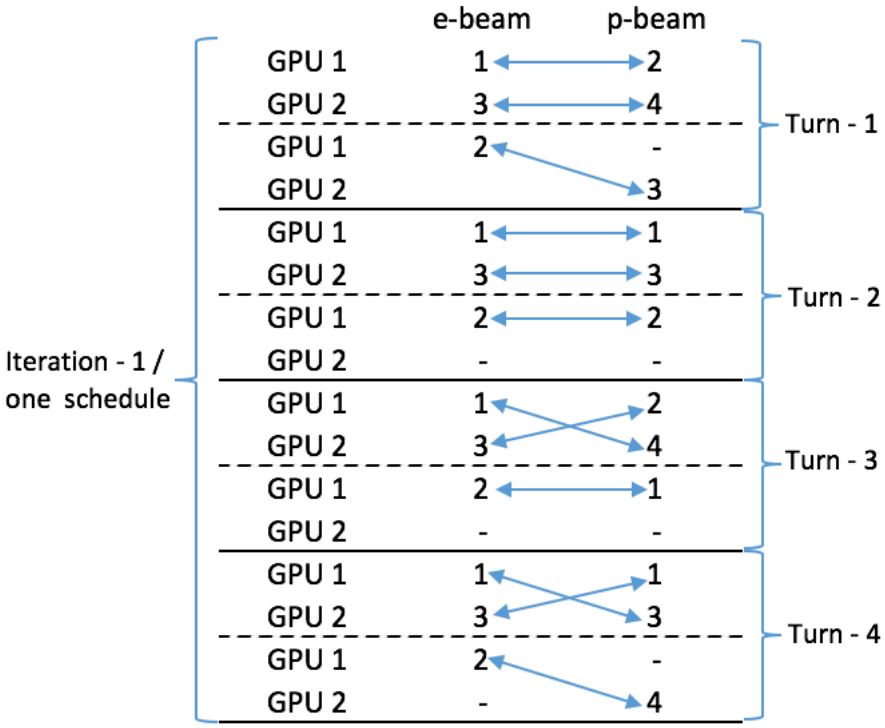


Figure 6.7 – Execution schedule formed by the scheduling algorithm when there are 2 GPUs and 4 Bunches (3e bunches, 4p bunches).

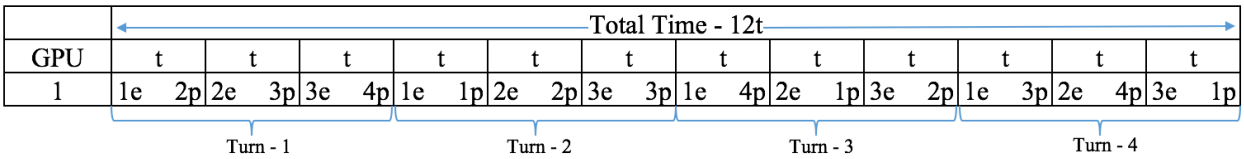


Figure 6.8 – Time slots required to complete all the interactions of a schedule on a Single GPU when there are 3e and 4p bunches.

Reason 1 - The number of time slots required to complete the interactions does not linearly decrease with the number of GPUs. For example, in Figure - 6.8 shows the number of time slots and the total time required to complete all the interactions on a Single GPU when there are 4 bunches. When there are 4 bunches, there is a total of 12 interactions in a schedule. Assuming that the total time taken to complete an interaction on a single GPU when both e- and p-beams are active is 't', it takes a total time of 12t to complete a single schedule on a single GPU.

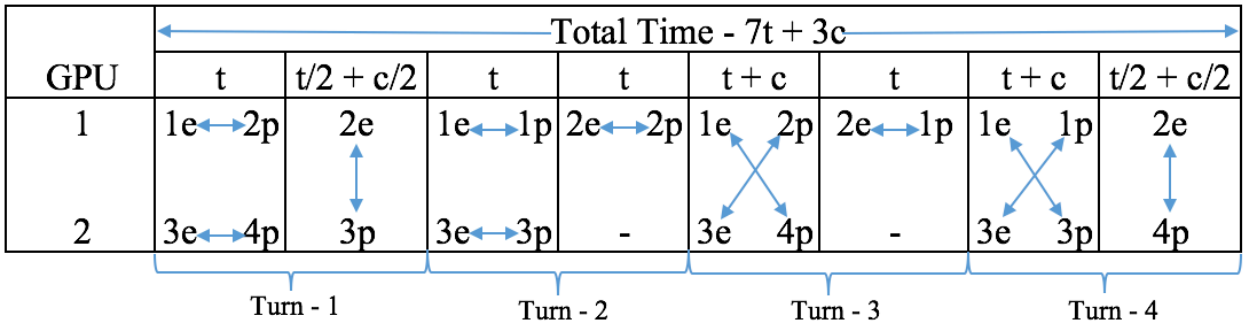


Figure 6.9 - Time slots required to complete all the interactions of a schedule on two GPUs when there are 3e and 4p bunches.

Figure - 6.9 shows the number of time slots and the total time required to complete all the interaction on 2 GPUs when there are 4 bunches. Now each GPU has to handle the work of 2 e-beams and 2 p-beams. If we look at Figure - 6.7 which is execution schedule formed, at first GPU 1 will handle the execution of the beams 1e and 2p and at the same time, GPU 2 will handle the execution of the beams 3e and 4p. As shown in Figure - 6.9, the time taken for these two interactions which are occurring in parallel is t. Now, for the interaction (2e, 3p), GPU 1 will handle the execution of 2e and GPU 2 will handle the execution of 3p. As the GPUs here are not handling the execution of both e- and p-beams, the time required here is only t/2 as shown in Figure - 6.9. Also, as both the beams are residing on different GPUs, there is an extra time taken to exchange the parameters between the GPUs which is called the communication time c. So the total time taken here is t/2 + c/2. In a similar way, both GPUs will execute all the interactions according to the execution schedule and the total time taken for all these interactions is 7t + 3c. Here, we observe that even though we have 2 GPUs, the execution time is not exactly half of the total time taken when we have only 1 GPU because of the dependency on previous turn which is indeed the nature of the problem.

Reason 2 - Most of the times, a GPU has to communicate with other GPUs to get the parameters needed for applying effects on the beam. For example, in figure 6.9, during the 1st time

slot of Turn-3, GPU 1 and GPU 2 are communicating with each other to send and receive the parameters needed to apply effects on their respective beams. Hence, there is an extra overhead added in the form of communication time which is the reason for near-linear speed up of Multi-GPU algorithm.

Reason 3 - The computations performed in the error function are different for each particle. Hence a GPU might have to wait for the other GPU until it finishes the execution of error function and send the required messages.

Timelines similar to Fig - 6.8, 6.9 are shown in the Figures - 6.10, 6.11, 6.12, but only for a single and random turn when there are 8 Bunches and the figures are for 1, 2, 4 GPUs respectively. From these Timelines, we observe that the time taken for single turn scales nearly linear with the number of GPUs.

	Total Time Taken - 0.156 Sec						
Time(Sec)	0.022	0.022	0.022	0.022	0.022	0.022	0.022
GPU 1	0e 4p	1e 5p	2e 6p	3e 7p	4e 0p	5e 1p	6e 2p

Turn - x

Figure 6.10 - Time slots and total time required to complete all the interactions of a single turn on a single GPU when there are 7e and 8p bunches.

	Total Time Taken - 0.082 Sec			
Time(Sec)	0.023	0.023	0.023	0.013
GPU 1	0e 0p	1e 1p	2e 2p	3e
GPU 2	4e 4p	5e 5p	6e 6p	7p

Turn - x

Figure 6.11 - Time slots and total time required to complete all the interactions of a single turn on 2 GPUs when there are 7e and 8p bunches.

	Total Time Taken - 0.046 Sec			
Time(Sec)	0.023		0.023	
GPU 1	0e	0p	1e	1p
GPU 2	2e	2p	3e	-
GPU 3	4e	4p	5e	5p
GPU 4	6e	6p	-	7p

Turn - x

Figure 6.12 - Time slots and total time required to complete all the interactions of a single turn on 4 GPUs when there are 7e and 8p bunches.

Time Slots

For any given number of bunches n_b and given number of GPUs g, the number of time slots required to complete the schedule can be calculated by the equation,

$$Time\ slots\ = n_b(n_b/g) \text{ for } nb > 1$$

$$Time\ slots\ = n_b((n_b - 1)/g) \text{ for } nb = 1 \quad (8)$$

where n_b/g is the number of timeslots required to complete a single turn and n_b is the number of turns required to complete a schedule/one-iteration. It is to be noted that the number of turns required to complete an iteration is always equal to number of p bunches (as e bunches are always $n_b - 1$).

Communications between GPUs

When there are g GPUs and n_b bunches ($n_b - 1$ e-bunches, n_b p-bunches), each of the $n_b - 1$ e-bunches will interact with all the n_b p-bunches. Each GPU has n_b/g ($n_b >> g$) bunches stored in it. So as each bunch has to interact with all the other bunches, a GPU has to communicate $n_b - (n_b/g)$ times to execute all the interactions related to that particular bunch. In the same way, for all the bunches stored on a GPU, it has to communicate $(n_b/g)(n_b - n_b/g)$ times with other GPUs to execute all the interactions that are stored on it. On the whole, when all the GPUs are

considered, a total of $n_b(n_b - n_b/g)$ communications are required in a schedule / one-iteration. But, as all the interactions in a particular time slot are happening in parallel, the communications required during the interactions in a time slot also happen in parallel. In a particular time slot, we observe that (refer to Figure 6.7) either all or none of the GPUs communicate with other GPUs to execute the interaction. Hence the number of communications (ignoring the communications happening in parallel) are less than the number of time slots required to execute a schedule and we observe the ratio of the number of time slots to communications is always equal to $g/(g-1)$. Hence the number of communications (ignoring the communications happening in parallel) required to execute the schedule / one-iteration is given by

$$Communications\ Required = {n_b}^2(g-1)/\ g^2 \quad (9)$$

Predicting the time required for given number of GPUs and Bunches

The time taken by the Multi-GPU algorithm for given number of bunches n_b (> 1) and GPUs g is resolved by the Equation 10 which is the addition of time(t) required by all the time slots (Equation – 8) and the time(c) required for all the communications (Equation – 9).

$$Total\ Time(T_{n_b}) = \frac{{n_b}^2}{g}\left(t + \frac{(g-1)}{g}c\right) \quad (10)$$

7 GPUs, 100K Particles per Bunch, 3 Slices								
Bunches	7	14	21	28	35	42	49	56
Actual Time Taken(sec)	0.171	0.659	1.499	2.666	4.202	6.045	8.212	10.705
Predicted Time(sec)	0.165	0.662	1.489	2.646	4.135	5.956	8.106	10.588

Table 6.5 – Comparison between Actual time taken on 7 GPUs and the predicted time on 7 GPUs when there are bunches that are multiples of 7.

Table 6.5 illustrates the comparison between the actual timings and the predicted timings (using equation 6.3) of the Multi-GPU algorithm. For these predictions, we have taken bunches in the multiples of 7 and fixed the number of GPUs to 7. We observe that the predicted timings are almost closer to the actual timings. The error between those two timings is mainly because the

computations inside the error function are different for different particles. So the time taken by each collision is not exactly the same. But while predicting these timings we assumed a fixed time (t) of 22.5 msec for each interaction on Tesla K40 Architecture. Also, according to our experiments, the average time taken (c) for each communication is 1.2 msec which also often varies in practice depending on the distance between the nodes in cluster. But to predict the timings more accurately, we used Least Squares method to find the best fit of c for a given number of nodes on a cluster. In this way, by knowing the average time taken (t) for an Interaction on a particular GPU, we can calculate the communication time for any cluster size to predict the total time taken taken for a given number of bunches more accurately.

When we apply the Least Squares method to Equation 10 and apply differentiation on it with respect to c, we get the equation below which we can use to find c.

$$\sum_{n_b} 2(A_{n_b} - B_{n_b} c)(-B_{n_b}) = 0 \qquad (11)$$

where $A_{n_b} = T_{n_b} - \frac{{n_b}^2}{g} t$ and $B_{n_b} = \frac{{n_b}^2 (g-1)}{g^2} c$

When we substitute the bunch numbers (n_b) from 7 to 56 in the multiples of 7, we get the c = 1.66 msec as the best fit of communication time. Below are the predicted results when the inputs to the equation 10 are t = 22.5 msec and c = 1.66 msec.

7 GPUs, 100K Particles per Bunch, 3 Slices								
Bunches	7	14	21	28	35	42	49	56
Actual Time Taken(sec)	0.171	0.659	1.499	2.666	4.202	6.045	8.212	10.705
Predicted Time(sec)	0.168	0.673	1.513	2.691	4.204	6.055	8.242	10.766

Table 6.6 – Comparison between Actual time taken on 7 GPUs and the predicted time using Least Squares method on 7 GPUs when there are bunches that are multiples of 7.

Hence, when we have a new cluster setup potentially with different GPU architecture. We first calculate the time taken (t) for an interaction for that particular input configuration and then

run some experiments with different bunch numbers (n_b) to find out the best fit of communication time (c) for Equation 11.

CHAPTER VII

CONCLUSION AND FUTURE WORK

VII.1 CONCLUSION

We presented a high-fidelity, high-performance parallel model for simulation of beam-beam effects in particle colliders using GPUs. This pioneering implementation on modern GPU architectures results in orders-of-magnitude speedup over its serial version, thereby bringing the previously intractable physics within reach for the first time. The parallel implementation of this simulation model on NVIDIA Tesla K40 GPU outperforms the non-optimized sequential simulation and it delivers as much as three orders-of-magnitude reduction in computation time. The development of this advanced new simulation tool will enable carrying out a truly long-term simulations spanning 400 million turns, which in case of the proposed electron-ion collider JLEIC is on the order of an hour of machine operation. This will facilitate fine tuning the collider parameters for more efficient operations which will lead to substantial savings in the design and operation of these expensive machines. Below is the summary of our contributions in this thesis.

- Implemented the simulation algorithm for beam-beam effects when the particles collider carries one or more bunches.
- Achieved an overall speedup of around 880X with our GPU implementation of Single-Bunch beam-beam simulations when compared to the existing sequential implementation.
- Implemented a Multi-GPU algorithm for Multi-Bunch beam-beam simulations with a minimal data movement between the GPUs.
- Our Multi-GPU implementation of Multi-Bunch beam-beam simulations achieved a nearly linear speedup with number of GPUs on a cluster.

VII.2 FUTURE WORK

In the Future, we plan to address the following:

- Analyze the computations happening inside the error function.
- Reduce the control-flow divergence occurring inside the error function.
- Minimize the local memory overhead by reducing the register usage.
- Use OpenMP to taken advantage of Multiple Cores present in the same node to parallelize the CPU computations between kernels.

REFERENCES

[1] NVIDA “Cuda Programming Guide”. Available via http://docs.nvidia.com/cuda/cuda-c-programming-guide.

[2] Bassetti, M., and G. Erskine. 1980. Technical report, CERN Report No. CERN-ISR-TH/80-06.

[3] Bell, N., and J. Hoberock. Thrust library for GPUs. Available via http://docs.nvidia.com/cuda/.

[4] Furman, M. 1991. In Particle Accelerator Conference, pp. 422.

[5] Harrison, M., T. Ludlam, and S. Ozak. 2003. Nuclear Instruments and Methods in Physics Research Section A vol. 499, pp. 235–244.

[6] J. L. Abelleira Fernandez et al. 2012. Journal of Physics vol. G 39 (075001).

[7] Makino, K., and M. Berz. 2002. Nuclear Instrumentation and Methods in Physics Research Section A vol.427, pp. 338.

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

[9] Poppe, G. P. M., and C. M. J. Wijers. 1990. ACM Transactions on Mathematical Software (TOMS) vol. 16,pp. 38–46.

[10] Qiang, J., R. Ryne, and M. Furman. 2002. Physical Review Special Topics - Accelerators and Beams vol. 5 (104402).

[11] Roblin, Y., V. Morozov, B. Terzić, M. Aturban, D. Ranjan, and M. Zubair. 2013. In 4th International Particle Accelerator Conference (IPAC), pp. 1064 – 1066.

[12] S. Abeyratne et al. 2012. arXiv:1209.0757.

[13] S. Abeyratne et al. 2015. arXiv:1504.07961.

[14] Schulte, D. 1996. Ph.D. thesis, University of Hamburg. TESLA-97-08.

[15] Terzić, B., A. Godunov, K. Arumugam, D. Ranjan, and M. Zubair. 2015. In 12th International Computational Accelerator Physics Conference (ICAP'15), pp. 40–43.

[16] Terzić, B., I. Pogorelov, and C. Bohn. 2007. Physical Review Special Topics - Accelerators and Beams vol.10 (034201).

[17] Yokoya, K., and H. Koiso. 1990. Particle Accelerators vol. 27, pp. 181.

[18] MPICH. Available via https://www.mpich.org/

[19] NVIDIA, "CUDA Bandwidth Test." [Online]. Available: http://docs.nvidia.com/cuda/cuda-samples/#bandwidth-test

VITA

Naga Sai Ravi Teja Majeti
Department of Computer Science
Old Dominion University
Norfolk, VA 23529

EDUCATION

M.S. in Computer Science, Old Dominion University, 2017
B.Tech. in Electrical and Electronics Engineering, JNTU Kakinada, India, 2012

EMPLOYMENT

Jan 2016 – May 2017 **Graduate Research Assistant**
Old Dominion University, Norfolk, VA, USA.

Aug 2015 – Dec 2015 **Graduate Teaching Assistant**
Old Dominion University, Norfolk, VA, USA.

Jan 2013 – July 2015 **Assistant Systems Engineer**
Tata Consultancy Services, Bangalore, KA, India.