High-Fidelity Simulation of Collective Effects in Electron Beams Using an Innovative Parallel Method

Kamesh Arumugam, Desh Ranjan, Mohammad Zubair
Department of Computer Science &
Center for Accelerator Science
Old Dominion University, Norfolk, VA-23529

Alexander Godunov, Balša Terzić
Department of Physics &
Center for Accelerator Science
Old Dominion University, Norfolk, VA-23529

ABSTRACT

Among the most challenging and heretofore unsolved problems in accelerator physics is accurate simulation of the collective effects in electron beams. Electron beam dynamics is 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. Serial, or even naively parallel, implementation of the electron beam's self-interaction is prohibitively costly in terms of efficiency and memory requirements, necessitating simulation times on the order of months or years. In this paper, we present an innovative, high-performance, high-fidelity, scalable model for simulation of collective effects in electron beams using state-of-the-art multicore systems (GPUs, multicore CPUs, and hybrid CPU-GPU platform). Our parallel simulation algorithm implemented on different multicore systems outperforms the sequential simulation, achieving a performance gain of up to 7.7X and over 50X on the Intel Xeon E5630 CPU and GTX 480 GPU, respectively. It scales nearly linearly with the cluster size. Our simulation code is the first scalable parallel implementation on GPUs, multicore CPUs, and on hybrid CPU-GPU platform for simulating the collective dynamical effects of electron beams in accelerator physics.

Author Keywords

Electron Beam Dynamics, High Performance Numerical Simulations, Parallel Simulation Models

1. INTRODUCTION

When electron bunches traveling at nearly the speed of light are forced by accelerator magnets to traverse a curved trajectory, they emit bright ultraviolet or x-ray radiation. If the radiation wavelength is larger than the electron bunch itself, coherent synchrotron radiation (CSR) is produced. CSR leads to a host of deleterious effects, such as emittance degradation and microbunching instability, thereby degrading or entirely erasing the electron beam's experimental usefulness.

There are two main settings in which CSR effects are crucially important. First setting is the synchrotron light sources, a powerful tool for cutting edge research in physics, biology, chemistry, material science, energy, medicine and other fields. Second is the next-generation electron-ion colliders, the future of nuclear physics, which are poised to study the nature of matter at unprecedented depths.

One of the most critical needs for electron machines is to develop efficient codes for simulating collective effects that severely degrade beam quality, such as CSR and CSR-driven microbunching instability [4, 10, 18–20]. The aim of this proposal is to develop an innovative code for high-fidelity simulation of electron beams, which is the essential first step in mitigating the damaging effects of CSR. However, with present tools, such accurate modeling is not possible; it requires new computational models. These new accurate and high-resolution numerical models have to deal with vast computational and memory requirements associated with storing the beam history and computing the beam self-forces. They also have to be robust and efficient in order to address the accuracy and resolution problems, and have to be amenable to massive parallelism.

In this paper, we propose a fundamentally new, high-fidelity, and high-performance model for simulating CSR and other collective effects in an electron beam using state-of-the-art computing platforms. The proposed method is optimized to run efficiently on different computing platforms such as GPUs, multicore CPUs and on hybrid CPU-GPU. Our implementation of the inherently parallelizable computation of beams self-interaction on a multicore platform leads to orders-of-magnitude reduction in computational time, thereby making the previously inaccessible physics tractable. The paper is organized as follows. Section 2 presents an overview of the related work. In Section 3, we present the physical model, and in Section 4 its parallel implementation on GPU, CPU and hybrid GPU-CPU platforms. Section 5 reports on the results of the comparison between the new parallel implementation and the original serial version. Finally, in Section 6 we summarize our finding and conclude.

2. RELATED WORK

Present CSR simulation tools employ a number of approximation in the study of CSR effects. For example, the CSR calculation in *elegant* [10] is based on the analysis of bunch self-interaction for a rigid-line bunch. This code is widely used for accelerator design and is the first to reveal CSR-induced microbunching in bunch compressors. However, in the regime

<SPECTS or SCSC>2015, July 26-29, 2015, Chicago, IL, USA

of extreme bunch compression when the bunch deflection is appreciable, the 1D approximation used in *elegant* may not be appropriate [14]. The earliest 2D CSR simulation is TraFiC^4 [1]. 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 CSR simulation was developed by [11,12]. 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.* [7] 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 [17]. This method can include boundary effect by chamber walls much easier than the Greens function approach. More comprehensive review of the status of CSR simulation can be found in the review article by Bassi *et al.* [6].

3. PROPOSED MODEL FOR NUMERICAL SIMULATION OF COHERENT SYNCHROTRON RADIATION

In this section, we provide an overview of the general equations that numerical CSR simulation model is solving. We then briefly describe the particle tracking approach used in our simulation and followed by a detailed outline of the simulation algorithm. Finally, we describe the crucial and by far the most computationally intensive step of the simulation.

3.1 Physical Problem

The dynamics of electron beams is captured by the Lorentz force [11]:

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

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

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

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

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

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

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

$\boldsymbol{r}$ 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 electron 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 $\boldsymbol{A}$.

Computation of the retarded potentials requires integration over the history of the charge distribution and current density, as can be seen from Equation 3a. This is the main computational bottleneck of the CSR simulations. In particular, the problems to overcome in a successful CSR 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 (iii) accurate and efficient multidimensional integration in the equation for retarded potentials.

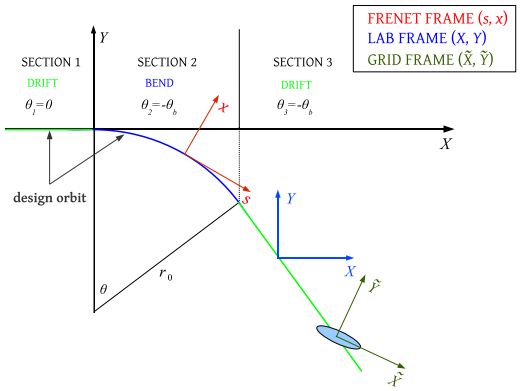

Figure 1: Three coordinate systems along the beamline: Frenet frame (s,x), lab frame (X,Y), and grid frame $(\tilde{X},\tilde{Y})$.

3.2 Frames of Reference

Different calculations in the simulation are best performed in different coordinate frames, shown in Figure 1: beam dynamics (particle pushing) in *Frenet frame* (FF), computation of retarded potentials in *lab frame* (LF), and gridding and interpolation in *grid frame* (GF).

Frenet frame (x,s) is defined so that $x \equiv r - r_0$ is the horizontal offset from the designed orbit, and $s \equiv r_0\theta$ is the longitudinal coordinate:

$$s - s_p = r_0\theta,\quad x = r - r_0, \tag{4}$$

and corresponding momenta

$$p_s = \frac{\gamma\dot{\theta}r}{c} = \gamma\beta_s,\quad p_x = \frac{\gamma\dot{r}}{c} = \gamma\beta_x, \tag{5}$$

where s_p is the position along the beam line at the end of the previous lattice element, r and θ are polar coordinates of the curved orbits, and r_0 is the radial coordinate of the designed orbit.

Lab frame (X,Y) is defined as the Cartesian coordinates in the plane of the beam lattice. The corresponding momenta are defined as

$$p_X = \frac{\gamma\dot{X}}{c} = \gamma\beta_X,\quad p_Y = \frac{\gamma\dot{Y}}{c} = \gamma\beta_Y. \tag{6}$$

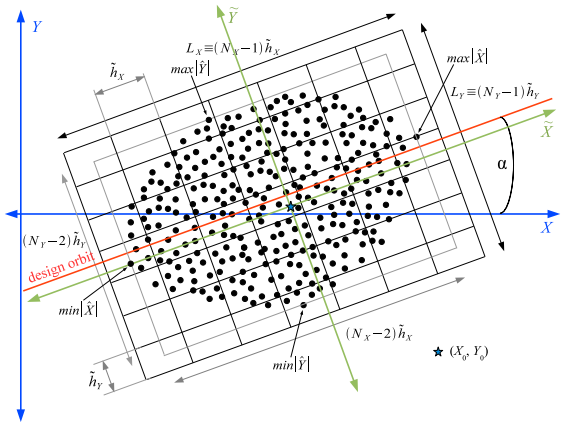

Figure 2: Computational grid tightly envelops the particle distribution. Its size is determined by the outliers of the distribution along the principal axes (along the red line and perpendicular to it). Red line denotes the design orbit.

Grid frame $(\tilde{X}, \tilde{Y})$ is defined as the scaled and rotated LF:

$$\begin{bmatrix} \tilde{X} \\ \tilde{Y} \end{bmatrix} = \begin{bmatrix} \frac{1}{L_X}\cos\alpha & \frac{1}{L_X}\sin\alpha \\ -\frac{1}{L_Y}\sin\alpha & \frac{1}{L_Y}\cos\alpha \end{bmatrix} \begin{bmatrix} X - X_0 \\ Y - Y_0 \end{bmatrix}, \quad (7)$$

where α is the angle between the design orbit and the computational box, center of charge (X_0, Y_0) is the center of the computational box, and L_X and L_Y specify the size of computational box (as in Figure 2).

3.3 Particle Tracking Approach

Equations in Section 3.1 can be solved either directly, by sampling the entire phase space of the DF, either on a grid or in a appropriate basis [5], or by using a particle tracking approach which is most dominant in CSR simulations. Computational requirements associated with sampling the entire phase space limit the direct solvers to low dimensions (usually 1D). Tracking methods are less restrictive owing to the fact that the sampling of the phase space is done only through simulation particles. This allows the study in higher-dimensional systems, which gives them a clear advantage and makes them a preferred method for modeling CSR effects. We use Particle-in-cell (PIC) tracking method to simulate the multiple particle systems, such as charged particle beams, galaxies, or plasma [2,3,15]. PIC codes sample a particle DF with a large number of point-particles, which do not interact directly with each other, but only through a mean-field of the gridded representation (Figure 2).

3.4 Outline of the Algorithm

At the top-most level, algorithm for simulation of CSR and other collective effects in electron beams consist of four consecutive steps that are computed at each timestep:

1. Deposit the DF sampled by N particles onto the computational grid using the PIC deposition scheme [2, 3, 15], thereby yielding the charge (ρ) and current density ($\boldsymbol{J}$) on each grid point. This involves an inverse interpolation from the particle position to the nearest grid points.
2. Compute retarded potentials on the grid via quadratures defined in Equation 3a for all the grid points. This is the crucial and by far the most computationally-intensive step. The details are described in subsection 3.5.

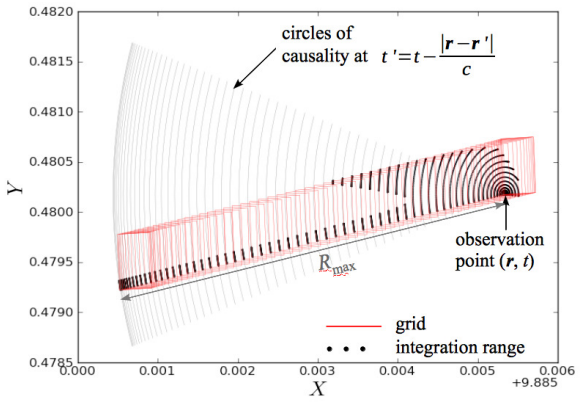

Figure 3: Integration for the retarded potential quadrature in Eq. (8) for a typical grid point. At different retarded times t', the computational boxes are shown in red, circles of causality in light grey and the intersection of the two in black. The black lines represent the limits of integration in θ'. Each continuous line represents a separate "cut". Dark grey line denotes the limit of radial integration $R_{\max}$.

3. Compute the self-forces from Equation 1 on a grid. Next, for each simulation particle compute the self-forces acting on it by interpolating from the grid. It is required 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. Advance particles by a small time step Δt in time by solving the Lorentz equation (given in Equation 1) using a leap-frog scheme [11]. The implementation is identical to that in [11].

The steps 1-4 are repeated until the end of simulations. The coordinates of the rectangular computational grid of resolution (N_X, N_Y) is first tilted through angle α from the design orbit in the (X, Y) plane, so as to account for the X-Y correlations (Figure 2). Computational grid $G_t = \{\tilde{X}_i, \tilde{Y}_j\}_{j=1,N_Y}^{i=1,N_X}$ at time t is constructed to envelope all particles such that the outliers in the tilted plane are binned into the boundary cells. 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. Therefore, at each timestep, the grid is uniquely described 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 the description below, P_t represents the parameters that uniquely describe a grid at time t and $\mathbf{P}$ represents the vector of unique parameters for all timesteps.

3.5 Computing the retarded potentials on the grid

The retarded potentials $\phi(G_{t_k}, t_k)$ and $\boldsymbol{A}(G_{t_k}, t_k)$ for all the grids points on a grid G_{t_k} at time t_k are computed using the quadrature defined in Equation 3a which uses general values of $\rho(G_t, t)$ and $\boldsymbol{J}(G_t, t)$ found by interpolation. In order to avoid singularity at $r' = 0$, the integration in Equation 3a is performed in polar coordinates:

$$\begin{bmatrix} \phi(\boldsymbol{r}, t) \\ \boldsymbol{A}(\boldsymbol{r}, t) \end{bmatrix} = \sum_{i=1}^{M_{int}} \int_0^{R_{max}} dR' \int_{\theta^i_{min}}^{\theta^i_{max}} \begin{bmatrix} \rho(R', \theta', t - \frac{R'}{c}) \\ \boldsymbol{J}(R', \theta', t - \frac{R'}{c}) \end{bmatrix} d\theta', \quad (8)$$

where M_{int} is the number of "cuts" (up to 4) of the grid by the circle of causality $t' = t - R'/c$. R_{max} is computed from

Algorithm 1 COMPUTEPOTENTIAL($\mathbf{G}$, $\mathbf{P}$, $\mathbf{R}$, τ, t_o, $\mathbf{X_o}$, $\mathbf{Y_o}$)

1: **for** all grid point i on grid G_{t_o} **do**
2: $X_o \leftarrow \mathbf{X_o}[i]$, $Y_o \leftarrow \mathbf{Y_o}[i]$
3: $(\phi_i, A_{X_i}, A_{Y_i}) \leftarrow$ QUADRATURE($fout$, [0, R_i], τ, t_o, X_o, Y_o, $\mathbf{G}$, $\mathbf{P}$)
4: **end for**

Algorithm 2 QUADRATURE($fout$, $[a, b]$, τ, t_o, X_o, Y_o, $\mathbf{G}$, $\mathbf{P}$)

1: $(\phi', A'_X, A'_Y, \varepsilon') \leftarrow$ QUADRULE($fout$, $[a, b]$, t_o, X_o, Y_o, $\mathbf{G}$, $\mathbf{P}$)
2: $H \leftarrow \emptyset$
3: PUSH(H, $([a, b], \phi', A'_X, A'_Y, \varepsilon')$)
4: **while** $\varepsilon' > \tau|\phi'|$ **do**
5: $([a, b], \phi, A_X, A_Y, \varepsilon) \leftarrow$ POP(H)
6: $m \leftarrow (a + b)/2.0$
7: $(\phi_L, A_{XL}, A_{YL}, \varepsilon_L) \leftarrow$ QUADRULE($fout$, $[a, m]$, t_o, X_o, Y_o, $\mathbf{G}$, $\mathbf{P}$)
8: $(\phi_R, A_{XR}, A_{YR}, \varepsilon_R) \leftarrow$ QUADRULE($fout$, $[m, b]$, t_o, X_o, Y_o, $\mathbf{G}$, $\mathbf{P}$)
9: $\phi' \leftarrow \phi' - \phi + \phi_L + \phi_R$
10: $A'_X \leftarrow A'_X - A_X + A_{XL} + A_{XR}$
11: $A'_Y \leftarrow A'_Y - A_Y + A_{YL} + A_{YR}$
12: $\varepsilon' \leftarrow \varepsilon' - \varepsilon + \varepsilon_L + \varepsilon_R$
13: PUSH(H, $([a, m], \phi_L, A_{XL}, A_{YL}, \varepsilon_L)$)
14: PUSH(H, $([m, b], \phi_R, A_{XR}, A_{YR}, \varepsilon_R)$)
15: **end while**
16: **return** (ϕ', A'_X, A'_Y)

the circle of causality (Figure 3). We use the tuple (ρ, J_X, J_Y) to denote the integrand value for a given point (R, θ).

In Equation 8, the integrand is tabulated at discrete points given in GF, and it is not available in a functional form. The physical formulation of the problem requires using three different coordinate systems to evaluate the integrand value at off-grid points. Also, the integrand along the outer dimension has regions of high variability as well as regions where change is gradual. In contrast, the inner dimension features only regions where change is gradual. The form of data and the nature of integrand determines the approaches that can be used to evaluate the integral. This necessitates the use of adaptive integration methods to solve the integral along outer dimension and Newton-Cotes rules along inner dimension.

In our description below, QUADRATURE procedure implements the adaptive integration method to solve the outer integral [16, p. 638]. The heart of the QUADRATURE algorithm is the procedure QUADRULE($fout$, $[a, b]$, t_o, X_o, Y_o, $\mathbf{G}, \mathbf{P}$) which outputs a quadruplet $(\phi, A_X, A_Y, \varepsilon)$, where ϕ, A_X, and A_Y are the integral estimate representing the scalar and vector potentials ($\boldsymbol{A}$'s components A_X and A_Y) for an grid point (t_o, X_o, Y_o), ε is an error estimate, $fout$ represents the integrand along outer dimension in Equation 8 with the values of the integrand tabulated in a 3D array $\mathbf{G}$, and $[a, b]$ is the domain of integration along the outer dimension.

We now give a high-level description of the COMPUTEPOTENTIAL algorithm (Algorithm 1). The algorithm input is ($\mathbf{G}$, $\mathbf{P}$, $\mathbf{R}$, τ, t_o, $\mathbf{X_o}$, $\mathbf{Y_o}$), where $\mathbf{R}$ is a vector of radial integration limit R_{max} corresponding to each grid point, τ is the relative error tolerance, t_o is the timestep at which the double integral is to be computed, $\mathbf{X_o}$ and $\mathbf{Y_o}$ are the positions for grids points along the X and Y direction of the grid G_{t_o}. The number of grid points on the grid is $N_X N_Y$, where N_X and N_Y denote the grid resolution along X- and Y-directions. The algorithm executes the QUADRATURE routine to compute the integral value for all the grid points on the grid G_{t_o}. In the QUADRULE routine, the value of the integrand $fout$ for a given point R is computed by first finding the integration range θ in LF, and then evaluating the corresponding inner quadrature in GF. The integration range in θ is computed by finding the intersection between the circles of causality and the computational box at the retarded time $t' = t_o - R/c$ in LF. Finally, each of the inner quadratures are evaluated using the Newton-Cotes rule for tabulated data [16, p. 613] in GF. The integrand values (gridded quantities ρ, J_X and J_Y) at point $(t, \tilde{x}, \tilde{y})$ for the inner integrals are evaluated via 3D interpolation of the integrand data recorded in GF at discrete time steps.

3.6 Algorithm Complexity

Let N denote the number of particles used in the CSR simulation, N_X and N_Y denote the resolution of computational grid along X- and Y-directions. The deposition of particle charge and density onto the grid requires $\Theta(N)$ operations. The retarded potentials are computed for all the grid points, so the total time required to compute these potentials is $\sum_{(x,y)\in G} g(x, y)$, where $g(x, y)$ is the time taken to compute the retarded potential for a point (x, y) on the grid G. The function $g(x, y)$ depends on the position of point (x, y) on the grid, N, N_X, N_Y and on the integration method used to solve the double integral in Equation 8. Function $g(x, y)$ for a point (x, y) is a monotonically decreasing function of N for a fixed value of N_X and N_Y, which is experimentally shown in Figure 9. The reason for this behavior is that increasing the number of particles with a fixed grid resolution reduces the numerical noise associated with the distribution of the integrand values, thereby reducing the number of operations required to compute the integral to within a prescribed accuracy. Numerical noise in PIC simulations is inversely proportional to the square root of the number of particles per cell in the simulation [3].

4. PARALLEL SIMULATION OF CSR

We propose a scalable two-phase parallel algorithm that uses the multicores of underlying architecture to speed up the computations of CSR simulation. The algorithm approximates the integrals (retarded potentials) for each of the $N_X N_Y$ quadratures 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 the algorithm is provided below in the algorithms FIRSTPHASE (Algorithm 3) and SECONDPHASE (Algorithm 4). In the description below, every subregion of a quadrature is identified by the record $([a, b], k)$, where $[a, b]$ denotes the integration domain along the outer dimension and k represents an identifier that uniquely identifies the quadrature for the given grid point. The proposed algorithm is an extension of our new and improved multidimensional numerical integration algorithm proposed in [8,9]. The details of the procedures FIRSTPHASE and SECONDPHASE are provided in [8, 9].

Algorithm 3 FIRSTPHASE ($\mathbf{G}$, $\mathbf{P}$, $\mathbf{R}$, t_o, $\mathbf{X_o}$, $\mathbf{Y_o}$, τ, L_{max})

$L \leftarrow \emptyset$
for $i = 1$ to $|\mathbf{R}|$ **do**
  $\boldsymbol{\phi}[i] \leftarrow 0$, $\boldsymbol{A_X}[i] \leftarrow 0$, $\boldsymbol{A_Y}[i] \leftarrow 0$
  INSERT(L, $([0, R_i], i)$)
end for
while $(|L| < L_{max})$ and $(|L| \neq 0)$ **do**
  for all i in parallel **do**
    $([a_i, b_i], k_i) \leftarrow L[i]$
    $X_o \leftarrow \mathbf{X_o}[k_i]$, $Y_o \leftarrow \mathbf{Y_o}[k_i]$
    $(\phi_i, A_{X_i}, A_{Y_i}, \varepsilon_i) \leftarrow$ QUADRULE($fout$, $[a_i, b_i]$, t_o, X_o, Y_o, $\mathbf{G}$, $\mathbf{P}$)
    INSERT(S, $(L[i], \phi_i, A_{X_i}, A_{Y_i}, \varepsilon_i)$))
  end for
  $L \leftarrow$ PARTITION(S, L_{max}, τ)
  $(\boldsymbol{\phi}, \boldsymbol{A_X}, \boldsymbol{A_Y}) \leftarrow$ UPDATE($S, \tau, \boldsymbol{\phi}, \boldsymbol{A_X}, \boldsymbol{A_Y}$)
end while
return $(L, \boldsymbol{\phi}, \boldsymbol{A_X}, \boldsymbol{A_Y})$

Listing 1: **Procedures** in FIRSTPHASE

function PARTITION((S, L_{max}, τ))
  for $i = 1$ to $|S|$ **do**
    Let $([a_i, b_i], k_i, \phi_i, A_{X\,i}, A_{Y\,i}, \varepsilon_i)$ be the i^{th} record in S
    if $\varepsilon_i \geq \tau$ **then**
      insert $([a_i, b_i], k_i)$ into L_1
    end if
  end for
  $d \leftarrow$ SPLIT-FACTOR($L_{max}, |L_1|$)
  for $i = 1$ to $|L_1|$ **do**
    Let $([a_i, b_i], k_i)$ be the i^{th} record in L_1
    split $[a_i, b_i]$ into d equal parts and insert all these subregions into L_2
  end for
  return L_2
end function

function UPDATE($S, \tau, \boldsymbol{\phi}, \boldsymbol{A_X}, \boldsymbol{A_Y}$)
  for $i = 1$ to $|S|$ **do**
    Let $([a_i, b_i], k_i, \phi_i, A_{X\,i}, A_{Y\,i}, \varepsilon_i)$ be the i^{th} record in S
    if $\varepsilon_i < \tau$ **then**
      $\boldsymbol{\phi}[k_i] \leftarrow \boldsymbol{\phi}[k_i] + \phi_i$
      $\boldsymbol{A_X}[k_i] \leftarrow \boldsymbol{A_X}[k_i] + A_{X\,i}$
      $\boldsymbol{A_Y}[k_i] \leftarrow \boldsymbol{A_Y}[k_i] + A_{Y\,i}$
    end if
  end for
  return $(\boldsymbol{\phi}, \boldsymbol{A_X}, \boldsymbol{A_Y})$
end function

function INITREGIONS($fout$, $[a, b]$, N, t_o, X_o, Y_o, $\mathbf{G}$, $\mathbf{P}$)
  $H \leftarrow \emptyset$
  $\delta \leftarrow (b - a)/N$
  for $i = 0$ to $N - 1$ **parallel do**
    $a_i \leftarrow i \cdot \delta$
    $b_i \leftarrow a_i + \delta$
    $(\phi_i, A_{X\,i}, A_{Y\,i}, \varepsilon_i) \leftarrow$ QUADRULE($fout$, $[a_i, b_i]$, t_o, X_o, Y_o, $\mathbf{G}$, $\mathbf{P}$)
    PUSH(H, $([a_i, b_i], \phi_i, A_{X\,i}, A_{Y\,i}, \varepsilon_i)$)
  end for
  return H
end function

4.1 Implementation on different architectures

In this section, we first describe the implementation of our proposed parallel algorithm to simulate the electron beam dynamics on GPU architectures. Next, we discuss the implementation on multicore CPU architectures and then on a hybrid CPU-GPU architecture which makes use of all the cores of CPU and GPU of the underlying hardware platform.

Algorithm 4 SECONDPHASE ($\mathbf{G}$, $\mathbf{P}$, t_o, $\mathbf{X_o}$, $\mathbf{Y_o}$, L, $\boldsymbol{\phi}, \boldsymbol{A_X}, \boldsymbol{A_Y}$)

for $i = 1$ to $|L|$ **parallel do**
  Let $([a_i, b_i], k_i)$ be the i^{th} record in L
  $X_o \leftarrow \mathbf{X_o}[k_i]$, $Y_o \leftarrow \mathbf{Y_o}[k_i]$
  $(\phi_i, A_{X_i}, A_{Y_i}) \leftarrow$ PARALLELQUADRATURE($fout$, $[a_i, b_i]$, τ, t_o, X_o, Y_o)
  $\boldsymbol{\phi}[k_i] \leftarrow \boldsymbol{\phi}[k_i] + \phi_i$
  $\boldsymbol{A_X}[k_i] \leftarrow \boldsymbol{A_X}[k_i] + A_{X_i}$
  $\boldsymbol{A_Y}[k_i] \leftarrow \boldsymbol{A_Y}[k_i] + A_{Y_i}$
end for
return $(\boldsymbol{\phi}, \boldsymbol{A_X}, \boldsymbol{A_Y})$

Algorithm 5 PARALLELQUADRATURE($fout$, $[a, b]$, τ, t_o, X_o, Y_o, $\mathbf{G}$, $\mathbf{P}$)

$S \leftarrow$ INITREGIONS($fout$, $[a, b]$, N_t, t_o, X_o, Y_o, $\mathbf{G}$, $\mathbf{P}$)
while $|S| \neq 0$ **do**
  $L \leftarrow$ PARALLELPOP(S, N_t)
  for $i = 0$ to $|L|$ **parallel do**
    $([a_i, b_i], \phi_i', A_{X\,i}', A_{Y\,i}', \varepsilon_i') \leftarrow L[i]$
    $m_i \leftarrow (a_i + b_i)/2.0$
    $(\phi_L, A_{X\,L}, A_{Y\,L}, \varepsilon_L) \leftarrow$ QUADRULE($fout$, $[a_i, m_i]$, t_o, X_o, Y_o, $\mathbf{G}$, $\mathbf{P}$)
    $(\phi_R, A_{X\,R}, A_{Y\,R}, \varepsilon_R) \leftarrow$ QUADRULE($fout$, $[m_i, b_i]$, t_o, X_o, Y_o, $\mathbf{G}$, $\mathbf{P}$)
    if $\varepsilon_i' > \tau$ **then**
      PUSH(S, $([a_i, m_i], \phi_L, A_{X\,L}, A_{Y\,L}, \varepsilon_L)$)
      PUSH(S, $([m_i, b_i], \phi_R, A_{X\,R}, A_{Y\,R}, \varepsilon_R)$)
    else
      $\phi_k \leftarrow \phi_k + \phi_L + \phi_R - \varepsilon_i$
      $A_{X\,k} \leftarrow A_{X\,k} + A_{X\,L} + A_{X\,R} - \varepsilon_i$
      $A_{Y\,k} \leftarrow A_{Y\,k} + A_{Y\,L} + A_{Y\,R} - \varepsilon_i$
    end if
  end for
end while
return (ϕ, A_X, A_Y)

Furthermore, for each of these implementations we extend the implementation to a cluster of multicore systems with CUDA-enabled GPUs.

Implementation on GPU Architecture

In the FIRSTPHASE, we divide the subregions list L evenly into a block of subregions each of size B, where B is number threads per block. The number of threads per block depends on the target GPU architecture, shared memory requirement, register utilization, and so on. For our experiments, we have empirically determined the optimal value of B to be 128 for the Fermi architecture. We then assign each block of subregions of size B to a GPU thread block such that a thread from the block operates on one of the subregions from the list L. Each thread then computes the quadruple $(\phi, A_X, A_Y, \varepsilon)$ by evaluating the QUADRULE for an assigned subregion $[a, b]$. The quadruple value computed by each thread is stored in a new global list S along with its subregion $[a, b]$. Likewise, the subregions list L in SECONDPHASE procedure is also evenly divided into blocks of subregions of size B. Each thread from the kernel implementing the SECONDPHASE operates on one of the subregions from the list L and evaluates the integral estimates based on the PARALLELQUADRATURE routine. The PARALLELQUADRATURE is the extension of quadrature routine (Algorithm 2) designed to run efficiently on multicore platforms. The kernel implementing the PARTITION procedure is similar to the partition kernel described in [8].

The accumulation of integral values based on the unique identifier in both the FIRSTPHASE and SECONDPHASE are achieved through the atomic operations in GPU. Atomic updates are considered to be slow in the current NVIDIA hardware. However, it is not the atomic operations that limit the execution speed of the GPU implementation. Instead, the entire routine calculating the retarded potential takes most of the execution time for a single timestep. In the current implementation the tabulated integrand values ($\mathbf{G}$) are stored in double-precision floating-point format in global device memory. Shared memory is used during the update of the scalar and vector potentials as storage for temporary values. Constant memory is used for storing the vector of grid parameters ($\mathbf{P}$) and the vector of observation points ($\mathbf{X_o}, \mathbf{Y_o}$) that do not change during the course of algorithm execution.

For the cluster implementation, general idea is to extend the above mentioned single GPU implementation across a cluster of compute nodes with multiple GPU devices per node. The computations performed under the $while$ loop in Algorithm 3 is distributed equally among the available GPU devices on every iteration. This involves dividing the subregions in list L equally among the available GPU devices on every iteration and implementing the QUADRULE kernel on each of these devices along with the procedures PARTITION and UPDATE. The list L is maintained in the CPU memory for a shared access from the GPU devices. Communication between GPU devices attached to a compute node are handled using OpenMP, whereas the communication between the compute nodes are handled using MPI programming. All the memory transfers between GPU devices at a node are done using the host (compute node) as an intermediary. The implementation starts by creating an MPI process for each compute node in the cluster. One of the MPI process (master) initializes and distributes the constant data and the QUADRULE parameters which do not change during the course of execution using MPI routines. The master process initializes the subregions list L required by the FIRSTPHASE, and partitions the list equally among the available compute nodes. Each of these partitions are distributed to the compute nodes using MPI routines. The process running on every compute node in the cluster receives a set of subregions from the master process. These subregions are further partition among the available GPU devices attached to the compute node. Using OpenMP routines, each process at a compute node creates a thread per GPU device attached to the node. A thread running on a compute node initializes the assigned GPU device and transfers the subregions list to the GPU device memory. Next, all the GPU devices executes the FIRSTPHASE on the assigned subregions in parallel. After the completion of FIRSTPHASE, the results are transferred back to the master process using MPI routines. The master process further partitions and distributes the subregions returned from FIRSTPHASE execution to all the compute nodes in the cluster. The SECONDPHASE is initiated on all the compute nodes by the master process in the same way as it did for FIRSTPHASE. In our implementation, we use CUDA-based THRUST library for common numerical operations such as reduce and scan.

Implementation on Multicore CPU Architecture

For our multicore CPU implementation, we follow the same two-phase approach to simulate the collective effects in electron beams as for GPUs and CPUs. We first implement on a standalone multicore CPU and then extend it to a cluster of multicore CPUs. Each node in the CPU cluster is a multicore system with many-core processors and each core of these processor often supports one or more concurrent threads to be executed in parallel. Efficient implementation for multicore architecture requires the computation to be partitioned into blocks such that multiple cores can work concurrently on different blocks and at the same time effectively utilize the memory hierarchy. In our proposed method, the main computation of Algorithm 3 is done inside the $while$ loop and for the Algorithm 4 the main computation is done inside the for loop. We use OpenMP directives to distribute these computations across different cores.

In FIRSTPHASE, we use the work-sharing directive of OpenMP to distribute the iterations of the for loop among different cores. Likewise, the for loop in SECONDPHASE is distributed equally among different cores using the OpenMP loop directives. This involves dividing the subregions list L equally among the active threads on every iteration. The partitioned subregions are private to each thread. In FIRSTPHASE, each thread essentially identifies the "good" and "bad" subregions from the partition list of subregions by evaluating the QUADRULE on each of them. However, in SECONDPHASE each thread computes the value of integral for each of the assigned subregion using the procedure PARALLELQUADRATURE. The tabulated integrand values ($\mathbf{G}$), vector of grid parameters ($\mathbf{P}$) and the vector of observation points ($\mathbf{X_o}, \mathbf{Y_o}$) are shared among all threads using OpenMP *shared* data scope clause.

The cluster implementation of the multicore CPU implementation follows the same approach as that of GPU cluster implementation. The master process distributes the data evenly among the compute nodes of the cluster using MPI routines. Each compute node has a process running on them, which receives the partitioned data from the master process. Each process at a compute node performs the FIRSTPHASE using the above mentioned multicore CPU implementation on the assigned block of data. After the completion of FIRSTPHASE, the results are transferred back to the master process using MPI routines. The master process further partitions and distributes the subregions returned from FIRSTPHASE execution to all the compute nodes in the cluster. Finally, process running on each compute node implements the SECONDPHASE using the OpenMP directives. The results are accumulated by the master process using the UPDATE routine.

Hybrid CPU-GPU Implementation

The hybrid implementation is designed for a system with multicore CPU with one or more CUDA-enabled GPUs. The implementation utilizes the computational power offered by both multiple cores of the CPU and the GPUs of the underlying hardware to speed up the computation. This involves distributing the computation between the CPU cores and the GPU such that the computational load is evenly distributed between them. In order to determine the amount of work to be

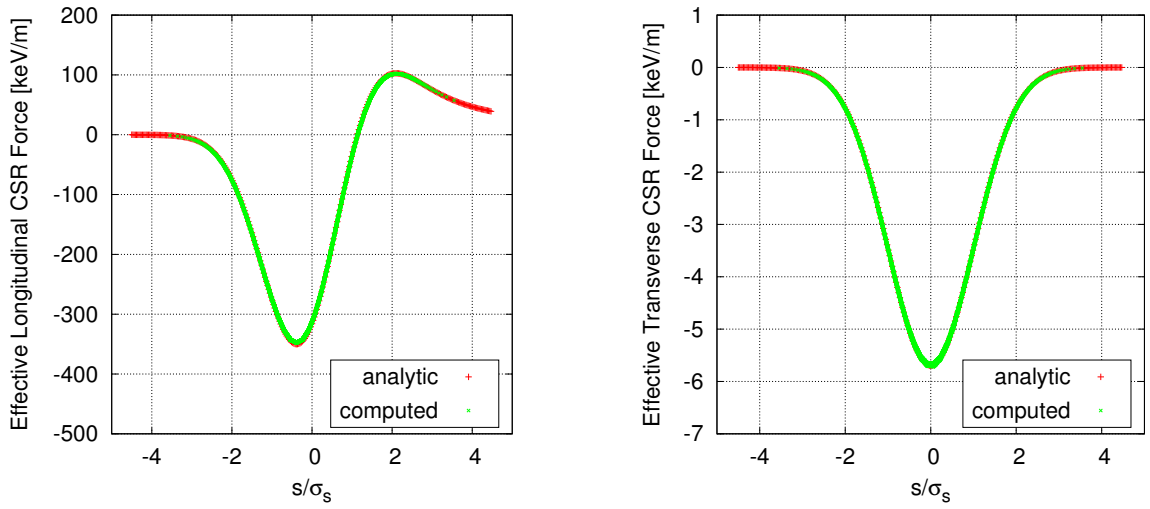

Figure 4: Analytic versus computed effective longitudinal (left) and transverse (right) CSR forces for the LCSL bend [11]: $N = 1024000$ particles on a 64×64 grid, bend radius $R_0 = 25.13$ m, $\theta_b = 11.4°$, longitudinal rms beam size $\sigma_s = 50$ μm, emittance $\epsilon = 1$ nm, and total beam charge of $Q = 1$ nC.

shared between GPUs and CPU cores, we perform empirical analysis of the GPU implementation and the multicore CPU implementation. If for a given set of input parameters, K denotes the ratio of total execution time of CPU implementation and GPU implementation, then the amount of work shared between CPU cores and GPU to even the computational load is $1 : K$. This means, GPU performs K times more work than the multicore CPU for a given amount of time. Once we determine the amount of work to be shared between CPU cores and GPUs, we use the above discussed multicore CPU implementation and the GPU implementation on their respective share of data.

5. RESULTS

5.1 Model Validation

We validate our 2D model for simulation of CSR effects in electron beams by comparing our simulation 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 CSR force for a 1D rigid-line bunch study state model is given in [13, 19]. We benchmark our code against the analytical results described in [13, 19] for the parameters of the LCLS bend [11]: 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 4 it is evident that both longitudinal and transverse CSR forces computed with our code agree perfectly with the exact analytical solution.

5.2 Simulation Performance Analysis

Our numerical simulations were carried out using a cluster of compute nodes with 4 NVIDIA GeForce GTX 480 GPU devices per node. A compute node in the cluster is a multicore system with two Quad-Cores Intel® Xeon® CPU E5630 2.53 GHz processors making a total of 8 cores per node. GeForce GTX 480 is the 11th generation of NVIDIA's GeForce GPU units and is based on the Fermi architecture. Each of the GTX 480 device offers 1.5 GB of GDDR5 on-board memory and 14 Streaming Processors (SMs) with 32 CUDA cores each. The interconnection between the host and the GPU device is via a PCI-Express Gen2 interface. The algorithms described above were first implemented sequentially in C and then the parallel implementations (GPU and multicore CPU) were developed using CUDA 5.0 programming environment.

We use our new model to simulate the collective effects in synchrotron light source and evaluate the performance of our parallel implementations on GPUs, multicore CPUs, and hybrid CPU-GPU architectures with the results of the sequential execution (compiler-optimized) running on a standalone desktop machine using one core. All of the results generated here represent a single timestep of the entire simulation which often runs for a few hundreds or thousands of timesteps. Initial conditions for the simulation are prepared by Monte Carlo sampling of an initial DF of N particles with a total charge of beam bunch $Q = 1$ nC.

Limitations of Sequential Simulation

In Figure 5 and Table 1, we show the limitations of sequential CSR algorithm by comparing the execution time for different stages of the algorithm outlined in subsection 3.4 for a single timestep. The simulation was performed with $N = 1024000$ particles on various grid resolutions. The breakdown of execution time shows that $95 - 99\%$ of the execution time on every timestep is spent in evaluating the double integral to compute the retarded potentials. We observe that the computational requirements associated with the evaluation of the double integral limits the overall performance of the algorithm in sequential implementation.

Comparative Analysis Across Architectures

In this section, we study the performance results of our multicore implementations on a standalone desktop machine with two Quad-Core Intel® Xeon® CPU E5630 processors with 4 NVIDIA GeForce GTX 480 GPU devices connected to it via a PCI-Express Gen2 interface. Table 2 presents our results of (a) the multicore CPU implementation running on the standalone desktop machine using one core, (b) the multicore

Grid Resolution	CPU Execution time (sec.)			
	Deposit Particles	Compute Potential	Compute Forces	Push Particles
32×32	0.70	58.04	0.35	2.10
64×64	0.70	573.87	0.31	2.10
128×128	0.70	7651.47	0.39	2.10

Table 1: Breakdown of CPU computation time for different stages of the CSR simulation with $N = 1024000$ particles on various grid resolutions.

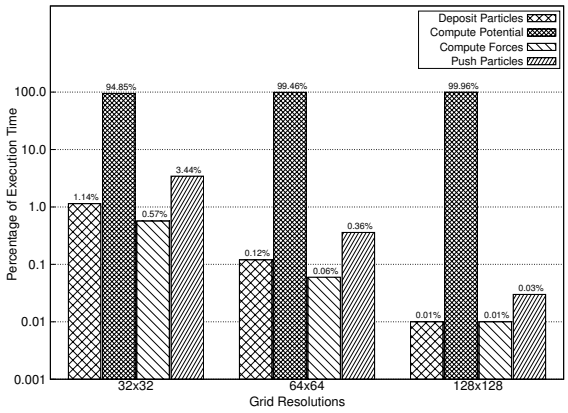

Figure 5: Percentage of CPU execution time spent by different stages of the CSR simulation with $N = 1024000$ particles on various grid resolutions. (Note: y-axis is shown in log-scale).

CPU implementation using 8 cores, (c) the GPU implementation using one GTX 480 device, (d) the GPU implementation using 4 GTX 480 device, and (e) the hybrid implementation using all 8 CPU cores and the 4 GTX 480 devices for different sets of input parameters. The speedup here is with reference to the computer-optimized, auto parallelized code running on a single CPU core of the desktop machine.

Multicore CPU Performance - On the Intel® Xeon® processor with 8 cores, the CSR simulation using all the 8 CPU cores is up to 7.7 times faster than the computer-optimized, auto parallelized code running on a single core of the CPU. Also, we observe that the implementation using multicore CPU architectures achieves a linear speedup with the number of cores on the desktop machine.

GPU Performance - On a single GTX 480 GPU, the CSR simulation archives a speedup of over 50. In terms of absolute performance of multicore implementations, we find that the GPU implementation of the CSR simulation outperforms the multicore CPU implementation. We performed experiments to see the impact on the speedup with the number of GPU devices on a standalone desktop machine with 4 GTX 480 GPU devices. Figure 6 illustrates the speedup plot for the GPU-based simulation with 1024000 particles and on a grid resolution of 128×128 and 64×64. The results for different set of input parameters are consistent with the behaviour shown in Figure 6. We observe a linear speedup with the increase in number of GPU devices. However, the number of GPUs that can be used per node is limited by the hardware capability of the underlying compute node (or host). We choose to use the cluster implementation to scale the performance beyond 4 GPU devices.

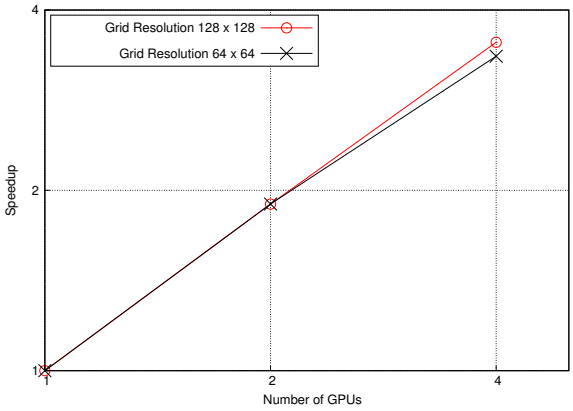

Figure 6: Impact on the speedup of GPU implementation with 1024000 particles with varying number GPU devices on a standalone desktop machine. (speedup is with reference to the GPU implementation using one GPU)

Hybrid CPU-GPU Performance - The performance of hybrid CPU-GPU implementation is evaluated on a desktop machine using all 8 CPU cores and 4 GTX 480 GPU devices. The results reported in Table 2 is computed analytically from the performance of the GPU implementation and the multicore CPU implementation. We observe that the maximum theoretical speedup that could be obtained using the hybrid implementation is nearly same as that of GPU implementation using 4 GPU devices. The performance benefit obtained by using hybrid CPU-GPU implementation is negligible when compared against the GPU implementation. Thus for further analysis we will not consider the hybrid implementation.

Analysis of Cluster Implementation

In this section, we study the performance of our CSR implementation on a cluster of multicore CPU and GPU architectures. Both of the cluster implementations require every node in the cluster to operate with maximum resource utilization. For a multicore CPU cluster, this means we consider each node of the cluster to utilize all the available 8 cores of the underlying architecture. On the other hand, GPU cluster implementation considers each host node to utilize all the 4 GTX 480 GPU devices connected to it.

We performed experiments to see the impact on the speedup with the increase in cluster size. Figure 7 illustrates the speedup plot for the simulation with 1024000 particles and on a grid resolution of 128×128 and 64×64. The speedup for GPU implementation is evaluated by computing the total execution time for the cluster implementation against the time taken by the GPU implementation on a standalone desktop machine with 4 GPU nodes. Likewise, the speedup for multicore CPU cluster implementation is with reference to the multicore CPU implementation on a standalone desktop machine using all the 8 CPU cores. The results for different set of input parameters are consistent with the behavior shown in Figure 7. In the cluster implementation, the overall execution time is a combination of kernel computation time (FIRSTPHASE and SECONDPHASE) and the computational overheads. The overhead includes MPI communication between the compute nodes, device initialization for the GPU

Number of Particles (N)	Grid Resolution	Multicore CPU implementation			GPU implementation on a standalone system with				Hybrid implementation on multicore CPU with 4 GPUs	
		Single Core	8 cores		Single GPU		4 GPUs			
		Time(sec.)	Time (sec.)	Speedup	Time (sec.)	Speedup	Time (sec.)	Speedup	Time (sec.)	Speedup
102400	32×32	73.5	11.1	6.6	1.5	49.0	0.7	105.0	0.7	105.0
	64×64	878.5	116.2	7.6	16.8	52.3	4.7	186.9	4.5	195.2
	128×128	13123.2	1695.3	7.7	246.8	53.2	68.4	191.9	65.8	199.4
1024000	32×32	58.1	12.7	4.6	1.2	48.4	0.6	96.8	0.6	96.8
	64×64	573.9	83.9	6.8	11.1	51.7	3.2	179.3	3.1	185.1
	128×128	7651.5	1000.9	7.6	144.1	53.1	40.1	190.8	38.6	198.2
4096000	32×32	57.8	11.9	4.9	1.3	44.5	0.6	96.3	0.6	96.3
	64×64	452.8	66.5	6.8	9.2	49.2	2.4	188.7	2.3	196.8
	128×128	5307.5	725.3	7.3	101.4	52.3	27.1	195.9	26.1	203.4

Table 2: Performance results of (a) the multicore CPU implementation running on a standalone desktop machine using one core, (b) the multicore CPU implementation using 8 cores, (c) the GPU implementation using one GTX 480 device, (d) the GPU implementation using 4 GTX 480 device, and (e) the hybrid implementation using all 8 CPU cores and the 4 GTX 480 devices for different sets of input parameters.

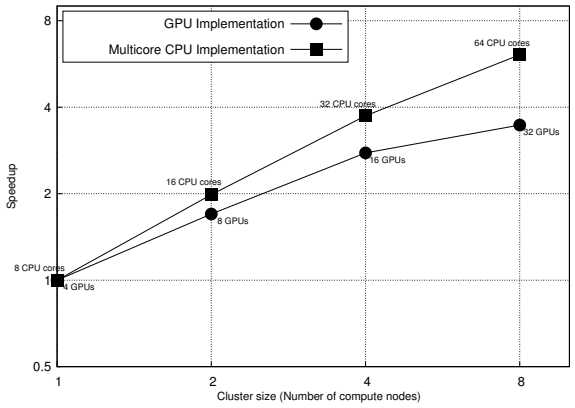

Figure 7: Impact on the speedup of simulation on a cluster of multicore systems with CUDA-enabled GPUs and multicore CPUs for 1024000 particles with increase in cluster size. (Speedup for GPU implementation is with reference to the GPU implementation on a standalone desktop machine with 4 GTX 480 GPU devices. Likewise, the speedup for multicore CPU cluster implementation is with reference to the standalone multicore CPU implementation executing using all the 8 cores.)

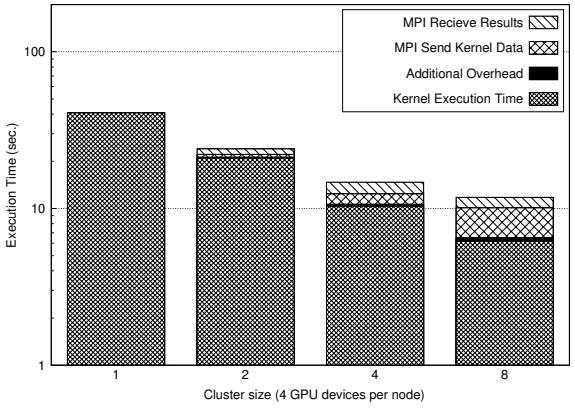

Figure 8: Split computation time for the GPU implementation with different number of GPUs for a simulation with 1024000 particles on a grid resolution of 128×128. (Note the log scale of the y-axis).

implementation and so on. Figure 8 shows the split computation time for the GPU implementation with increase in cluster size for a simulation with 1024000 particles on a grid resolution of 128×128. The results for multicore CPU cluster implementation is consistent with the behavior of GPU-

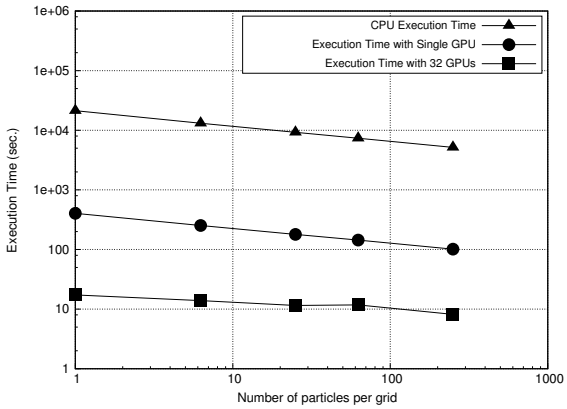

Figure 9: Comparison of execution time for computing the retarded potential for a grid size of 128×128 with varying number of particles per grid using GPU implementation.

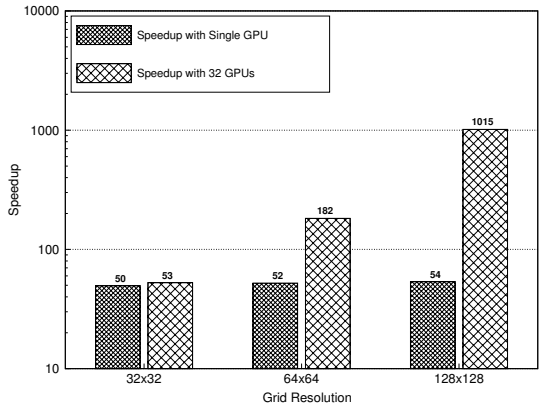

Figure 10: Speedup results for the parallel implementation using multiple GPU devices with 50 particles per grid with varying grid resolution using GPU implementation.

implementation as shown in Figure 8. We observe a near-linear scaling of kernel computation with the cluster size. However, the overall performance deviates from the linear scaling due to the increase in MPI communication overheads with the number of nodes in cluster. Note that in general the speedup scales near linearly with the increase in cluster size until a threshold number of nodes beyond which the performance would degrade due to additional overheads involved.

Effects of Simulation Resolution

In Figure 9 and Figure 10, we illustrate the relationship between the number of particles (N) and the grid resolution using the GPU-based implementation. Figure 9 compares the execution time for the sequential implementation on CPU and the parallel implementation on GPUs for a grid size of 128×128 with varying number of particles per grid. We notice that with the increase in particles to grid ratio the execution time (in both CPU and GPU) for computing the integral decreases. The reason for this behavior is that increasing the number of particles to grid ratio reduces the numerical noise in the distribution of the integrand values ($\rho(\boldsymbol{r}, t)$ and $\boldsymbol{J}(\boldsymbol{r}, t)$ in Equation 3a), thereby reducing the computational load required for computing quadratures to within a prescribed accuracy.

Figure 10 quantifies the performance of the parallel algorithm using one or more GPU devices with fixed number of particles per grid with varying grid resolution. The simulation here is performed with 50 particles per grid (in practice, the number of particle per grid varies from 10-100). The results for different particles per grid values are consistent with the behavior shown in Figure 10. We notice that the increase in grid resolution leads to a non-linear increase in the speedup. The reason for this is that at higher grid resolutions the algorithm generates larger number of subregions, thereby increasing the GPU device occupancy. We also notice a near-linear increase in speedup with number of GPUs for a fixed grid resolution. The behavior is expected because with the increase in number of GPUs the computational load is distributed across a larger set of parallel processors and the processors in each GPU device works independently of the other GPU devices.

6. CONCLUSION

We presented an innovative, high-performance, high-fidelity parallel model for simulation of collective effects, including heretofore prohibitive CSR effects, in electron beams using state-of-the-art multicore systems (GPUs, multicore CPUs, and hybrid CPU-GPU platform). This pioneering implementation on different multicore system results in a orders-of-magnitude speedup over its serial version, thereby bringing the previously intractable physics within reach for the first time. The parallel algorithm outperforms the compiler-optimized sequential simulation and achieves a performance gain of up to 7.7X and over 50X on the Intel Xeon E5630 CPU and GTX 480 GPU respectively. Furthermore we proposed a technique to scale this algorithm on a cluster of multicore systems. The performance gain of the cluster implementation scales nearly linearly with the cluster size.

The development of this advanced new simulation tool will enable unprecedented fidelity and precision in studying all the relevant physics of synchrotron light sources. This will facilitate a fundamental understanding of the adverse collective effects in these machines and their successful mitigation, leading to their improved design and operations.

For the society 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.

Acknowledgments

K. A. and B. T. acknowledge the support of the Jefferson Science Associates Project No. 712336 and the U.S. Department of Energy (DOE) Contract No. DE-AC05-06OR23177.

K. A. acknowledges the generous support of the Modeling and Simulation Graduate Research Fellowship Program provided by Old Dominion University during 2013-2015.

REFERENCES

1. A. Kabel. *PAC Proceedings* (2001).
2. B. Terzić and G. Bassi. *Phys. Rev. ST Accel. Beams 14* (2011), 070701.
3. B. Terzić, I. Pogorelov and C. Bohn. *Phys. Rev. ST Accel. Beams 10* (2007), 034201.
4. C. Bohn. *AIP Conference Proceedings* (2002).
5. C. Bohn and J. Delayen. *Phys. Rev. E 50* (1994), 1516.
6. G. Bassi *et al.* Overview of csr codes. *Nucl. Instrum. Methods Phys. Res. A 557* (2006), 189.
7. G. Bassi *et al. Phys. Rev. ST Accel. Beams 12* (2009), 030704.
8. K. Arumugam, A. Godunov, D. Ranjan, B. Terzić, and M. Zubair. *International Conference on Parallel Processing (ICPP)* (2013).
9. K. Arumugam, A. Godunov, D. Ranjan, B. Terzić, and M. Zubair. *High Performance Computing (HiPC)* (2013).
10. M. Borland *et al. Nucl. Instrum. Methods Phys. Res. A 483* (2002), 268.
11. R. Li. *Nucl. Instrum. Methods Phys. Res. A 429* (1999), 310.
12. R. Li. *Proceedings of the 2nd ICFA Advanced Accelerator Workshop on the Physics of High Brightness Beams* (1999).
13. R. Li. *Phys. Rev. ST Accel. Beams 11* (2008), 024401.
14. R. Li, R. Legg, B. Terzić, J.J. Bisognano, and R.A. Bosh. *Proc. 33rd International FEL Conference* (2011).
15. R. W. Hockney and J. W. Eastwood. *Computer Simulations Using Particles*. Institute of Physics Publishing, London, 1988.
16. S. Chapra and R. Canale. *Numerical Methods for Engineers*, 6 ed. 2009.
17. T. Agoh and K. Yokoya. *Phys. Rev. ST Accel. Beams 7* (2004), 054403,.
18. Y. Derbenev and J. Rossbach and E. L. Saldin and V. Shiltzev. *DESY-Pub-7181* (1995).
19. Y. Derbenev and V. Shiltsev. *SLAC-Pub-7181* (1996).
20. Z. Huang *et al. Phys. Rev. ST Accel. Beams 7* (2004), 074401.