Petascale Application Improvement Discovery at Blue Waters: Best Practices in Accelerating Gather/Scatter Operations in Particle-in-Cell Algorithms on GPU

Bei Wang * William Tang *

November 9, 2015

Driven by chip- and system power limitations, the HPC community is moving to the era of multi- and many-core architectures with greatly increased thread and vector parallelism on shared memory processors. However, achieving high performance and architectural efficiency is challenging for gather and scatter operations as modern computer architectures rely on single-instruction multiple-data execution to provide high computing capabilities. In this report, we describe optimization techniques to improve the performance of the gather and scatter operations from particle-in-cell (PIC) algorithms on GPU. This includes considerations in choosing appropriate data layouts and memory storage for particle and mesh data, and techniques to enable coalesced access and spatial and temporal locality. For a representative PIC application, we see a performance improvement up to 3.4x between the un-optimized and optimized versions of the code. We hope that this report can serve as a "cook book" for those who plan to port their applications (featuring gather and scatter operations) on GPU architectures.

1 Gather/Scatter Patterns in Particle-in-Cell Algorithms

The basic particle method has long been a well established approach for simulating the behavior of charged particles interacting with each other through pair-wise electromagnetic forces. Unfortunately, the $O(N^2)$ complexity makes a particle method impractical for plasma simulations using millions of particles per process. Rather than calculating $O(N^2)$ forces, the particle-in-cell (PIC) method employs a grid to calculate the long range electromagnetic forces. This reduces the complexity from $O(N^2)$ to $O(N + MlogM)$, where M is the number of grid points ($M \ll N$). Figure 1 shows the particle interpolation to the nearest neighbors (linear interpolation) on a 2D poloidal grid in a PIC algorithm. Similar data access patterns occur in a wide range of applications, such as astrophysics, molecular dynamics and statistical analysis.

In general, for each particle, the neighboring grid points are unknown until run time. For two particles stored at contiguous memory addresses, their neighboring grid data are likely to be stored at non-contiguous addresses far way from each other. Non-contiguous memory access will cause more cache/page misses and less efficiently utilize the single-instruction multiple-data execution units. In multithreading environment, simultaneously writing to the same memory address will lead to data hazard, which must be guarded by synchronization mechanisms in order to guarantee correct results. It is well known that synchronization is expensive as it serializes the instructions.

*Princeton Institute of Computational Science and Engineering, Princeton University, Princeton, NJ 08540 USA

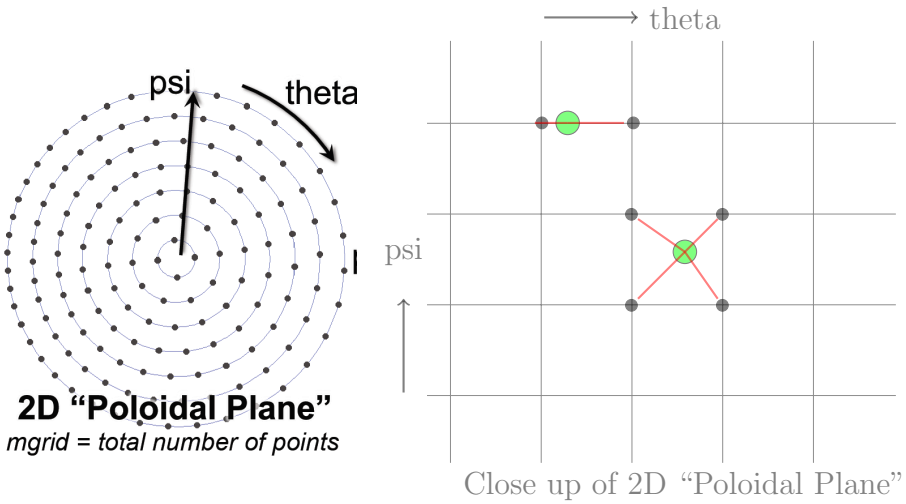

Figure 1: Particle mesh interpolation on a 2D poloidal grid.

```
#if USE_TEXTURE
texture<int2, 1, cudaReadModeElementType> evectorTexRef;
int size = mgrid*3*(mzeta+1); // mgrid*(mzeta+1) is the number of grid points in a 3D torus segment
cudaBindTexture(0, evectorTexRef, (void *)(evector),  size*sizeof(real));
static __inline__ __device__ real fetch_evector(int i)
{
  int2 e = tex1Dfetch(evectorTexRef,i);
  return __hiloint2double(e.y,e.x);
}
#define EVECTOR(i) (fetch_evector(i))
#else
#define EVECTOR(i) (evector[i])
#endif
```

Figure 2: Example: bind double precision *evector* array to texture memory

GPU architectures exhibit multiple distinguishing computational features including high-bandwidth memory, limited coherence support, low-overhead scheduling and high-throughout vector computing. In this report, we share some of the practices we found effective to improve the GPU performance of a PIC code. We start with choosing the appropriate data layouts for grid-based data and particle-based data. Then we use techniques such as *cooperative threading* and *binning* to exploit locality. Finally, we found it is more efficient to put the read only grid-based fields on GPU's texture memory.

2 Choosing Data Layout and Storage

Two of the most common used data layouts in high performance applications are array-of-structures (AoS) and structure-of-arrays (SoA). AoS is generally good for better cache behavior and is also more meaningful to represent physical variables, e.g., particles. However, AoS does not lead to streaming (unit) data access and SIMD instructions. In addition, AoS may cause less efficient utilization of the cached data. For example, reading a single variable from a particle array will result in unnecessary memory transaction. SoA can often lead to streaming access and more efficient utilization of the cached data. However, SoA can lead to poor cache behavior because of the memory distance in accessing multiple struct members of a particle element.

A particle-in-cell code involves two types of data structures, grid-based and particle-based. The grid-based data includes a set of one-dimensional and multi-dimensional arrays, for example, *density* (1D array to store charge density) and *evector* (3D array to store electric field). The particle-based data includes a multi-dimensional array, *zion*, which stores particle's phase space position and weight. Associated with the grid-based and particle-baesd data structures are two types of operations in a PIC algorithm, grid-based one such as the kernel to sovle Poisson's equation, and particle-based one such as charge deposition (scatter) and field interpolation (gather). The computational workload is proportional to the number of grid points and the number of particles for these two operations. Since the number of particles is several order of magnitude larger than the number of grid points, the particle-based kernels are usually the hot spot and consists of 80%- 90% of the total wall-clock time.

Before choosing the appropriate data layouts for field and particles, we will first examine memory access patterns of these two arrays. In the representative PIC code, GTC-P [5, 3, 4], we offload the particle-based kernels such as charge deposition and field interpolation to GPU, while leave the grid-based kernels on CPU. In this case, the multidimensional field variable *evector* will be read only, but features random access pattern (gather). The multidimensional particle array will be read and written, but in a pattern of unit or stride access depending on the choice of data layout.

GPU has a special cache called texture memory. The read-only texture memory is optimized for 2D spatial locality. We thus choose AoS for *evector* and bind it to texture memory. Note the 2D evector data is fattened in 1D. This will significant improve the performance of gather operation in field interpolation. Figure 2 shows a code example of how to bind variable *evector* to texture memory. To enable coalesced data access to particles, *zion* is organized as SoA data layout. On GPU with compute capabilities 3.5 (or newer) architecture, we can utilize texture memory simpler with "__ldg or const restrict".

3 Optimizing Gather/Scatter Patterns

In multithreading environment, two particles operating by two different threads may access the same grid point and cause a read-after-write data hazard. Thus, the data access must be either guarded with a synchronization mechanism or redirected to a private copy depending on the availability of memory resources.

3.1 Private Copy Strategy

One often used approach for GPU based PIC algorithm based on Cartesian grid is to group particles to supercells [2]. Each supercells consists of multiple number of grid points. Then we assign one GPU block to a specified supercell, within which the thread-based data conflict is resolved through fast shared memory atomic operations. The charge in shared memory is added to global memory at once at the end. Given the extensive use of shared memory, we can configure the GPU to favor shared memory.

However, applying the private copy strategy to PIC algorithm based on unstructured grid in magnetic field-line coordinate is challenging. Keeping in mind that particle grouping is based on the particle position. In magnetic field-line coordinate, the final interpolation grid points is not necessary within the associated supercell due to field-line twisting along the toroidal coordinate (see line 35 and 36 in Figure 3a). As such, large number of ghost cells may needed for each supercell. Additionally, it is difficulty to applying the private copy strategy to gyrokinetic PIC algorithm, where each particle is not a single particle,

Processor	Freq. (GHz)	D$/core (KB)	Cores/CPUs	D$/CPU (MB)	CPUs/Node	DP Gflop/s	STREAM (GB/s)	Memory GB
AMD Opteron	2.3	16+2048	8	8	2	140.8	31	32
NVIDIA	0.733	64	14	1.5	1	1314	171	6

Table 1: Blue Waters XK7 nodes configuration

Kernels	Times (ms)			
	baseline	binning	binning+cooperative threading	binning+texture memory
Charge	51.0	51.0	21.5	-
Push	126.9	83.4	-	37.3

Table 2: Performance comparison of two GPU accelerated kernels with different optimization options.

but represents a gyro-ring of varying radius. Since the interpolation is not from a particle to the nearest neighboring grid points, but from multiple points on the gyro-ring of the particle to the their neighboring grid points, up to 8 ghost cells in each dimension is required.

The introduction of the NVIDIA Kepler architecture (employed in Blue Waters) enables fast double-precision atomic increment (implemented via a compare-and-swap operation). As such, we develop a cooperative threading algorithm that uses global atomic write for scatter operations as described in details below. The cooperative threading algorithm can be considered as an alternative for dealing with data hazard on unstructured grid with less organized data.

3.2 Synchronization Strategy

Figure 3a shows the cooperative threading algorithm (highlighted) that uses shared memory for data transpose. Specifically, computation is organized so that CUDA threads access successive particles when reading, but is reorganized so that threads work together on one charge deposition (of up to 32 neighboring grid points) when writing. The reorganization is through transposing the data write in shared memory. Given the limited use of shared memory in this case, the GPU is configured to favor the L1 cache.

As a comparison, the implementation that used global atomic write only is also shown in Figure 3a. For gather operation, the analysis indicates that a simple decomposition of one particle per CUDA thread and 64 threads per block perform well. The key optimization for gather operation is to place the grid-based array, such *evector* on texture memory. Additionally, some variables are also judiciously placed in CUDA's constant memory.

3.3 Data Locality with Binning

Particle binning is often used for PIC algorithm to improve performance by providing better data locality. Here we rely on the CUDA Thrust library to bin the particles according to their radial dimension positions.

4 Performance Evaluation

Blue Waters [1] is a Cray XE6/XK7 MPP at the the National Center for Supercomputing Applications. It contains 22,500 XE6 Interlagos-based nodes argumented by more than 4200 XK7 hybrid nodes containing NVIDIA GK110 Kepler accelerator and connected by Cray's Gemini network into a 3D torus. Each XK7 node includes two 8-core processors running at 2.3GHz in which each core has a private 16KB L1 data cache, each pair of cores shares a 256b-wide SIMD unit, each pair of cores shares a 2MB L2 cache, and all cores share a 32MB L3 cache. Each processor is connected to 16GB of DDR3-1333 DRAM via a 128-bit memory controller. As such, each compute node contains two NUMA nodes and programmers are motivated to run 2 processes of 8 threads per node. The Kepler GK110 includes 14 streaming multiprocessor units (SMX). Each SMX has 64KB on-chip memory that can be configure as 48/16KB shared memory and 16/48KB L1 cache and 1536KB L2 on-chip cache memory. Currently, the NVIDIA accelerator is not directly interact with the Gemini interconnect, so data has to be moved to GPU or accessed as mapped memory. Table 1 lists the system configuration of blue waters XK7 nodes.

We have evaluated the performance of charge deposition and field interpolation operations on GPU with different optimization techniques on one Kepler processor. The total number of grid points in 2D poloidal plane and the total number of particles are 39,693 and 3,959,300, respectively. Table 2 shows the wall-clock time spending on charge deposition (charge) and field interpolation (push) for one time step. Comparing with the baseline performance, optimization techniques such as binning, cooperative threading and texture memory lead to 2.4x and 3.4x speed up for charge and push kernels, respectively.

Acknowledgments

This report is generated as supported by the Petascale Application Improvement Discovery (PAID) program at Blue Waters.

```
const int tid = threadIdx.x; const int bid = blockIdx.x;
const int nblocks = gridDim.x; const int nthreads = blockDim.x;
const int gid = tid+bid*nthreads; const int np = nblocks * nthreads;

#if COOP
extern __shared__ int shared_buffer[];
int *update_idx = shared_buffer;
double *update_val = (real *)&shared_buffer[nthreads*4];
#endif
...

for (int m=gid; m<me; m+=np){
   double psitmp = z0[m];
   double thetatmp = z1[m];
   double zetatmp = z2[m];
   double weight = z4[m];

   double wzt = (zetatmp-zetamin)*delz;
   int kk = abs_min_int(mzeta-1,(int)wzt);

   double wz1  = weight*(wzt-(double)kk);
   double wz0   = weight-wz1;

   double r_diff   = psitmp-a0;
   double rdum = delr*abs_min_real(a_diff,r_diff);
   int ii = abs_min_int(mpsi_max,(int)rdum);

   double wp1 = rdum-(real)ii;
   double wp0  = 1.0-wp1;

   double tflr = thetatmp;
   int im = ii;
   int im2 = ii + 1;

   double tdumtmp = pi2_inv*(tflr-zetatmp*qtinv[im]);
   double tdumtmp2 = pi2_inv*(tflr-zetatmp*qtinv[im2]);
   double tdum = (tdumtmp-(int)tdumtmp)*delt[im];
   double tdum2 = (tdumtmp2-(int)tdumtmp2)*delt[im2];

   int j00 = abs_min_int(mtheta[im]-1,(int)tdum);
   int j01 = abs_min_int(mtheta[im2]-1,(int)tdum2);
   int jtelectron0tmp = igrid[im]+j00;
   int jtelectron1tmp = igrid[im2]+j01;

   double wtelectron0tmp = tdum-(double) j00;
   double wtelectron1tmp = tdum2-(double) j01;

   double wt10 = wp0*wtelectron0tmp;
   double wt00 = wp0-wt10;
   double wt11 = wp1*wtelectron1tmp;
   double wt01 = wp1-wt11;

   int ij1 = kk+(mzeta+1)*jtelectron0tmp;
   int ij2 = kk+(mzeta+1)*jtelectron1tmp;

   /* interpolate from particle to grid - scatter operation */
#if COOP // use shared memory for optimizing scatter operation
   update_idx[4*tid]=ij1;
   update_idx[4*tid+1]=ij1+1;
   update_idx[4*tid+2]=ij1+mzeta+1;
   update_idx[4*tid+3]=ij1+mzeta+2;
   update_val[4*tid]= wz0*wt00;
   update_val[4*tid+1]= wz1*wt00;
   update_val[4*tid+2]= wz0*wt10;
   update_val[4*tid+3]= wz1*wt10;
   __syncthreads();

   atomicDPupdate(density+update_idx[tid], update_val[tid]);
   atomicDPupdate(density+update_idx[stride+tid], update_val[stride+tid]);
   atomicDPupdate(density+update_idx[2*stride+tid], update_val[2*stride+tid]);
   atomicDPupdate(density+update_idx[3*stride+tid], update_val[3*stride+tid]);

   update_idx[4*tid]=ij2;
   update_idx[4*tid+1]=ij2+1;
   update_idx[4*tid+2]=ij2+mzeta+1;
   update_idx[4*tid+3]=ij2+mzeta+2;
   update_val[4*tid]= wz0*wt01;
   update_val[4*tid+1]= wz1*wt01;
   update_val[4*tid+2]= wz0*wt11;
   update_val[4*tid+3]= wz1*wt11;
   __syncthreads();

   atomicDPupdate(density+update_idx[tid], update_val[tid]);
   atomicDPupdate(density+update_idx[stride+tid], update_val[stride+tid]);
   atomicDPupdate(density+update_idx[2*stride+tid], update_val[2*stride+tid]);
   atomicDPupdate(density+update_idx[3*stride+tid], update_val[3*stride+tid]);
#else // use global atomic for scatter operation
   atomicDPupdate(density+ij1,wz0*wt00);
   atomicDPupdate(density+ij1+1,wz1*wt00);
   atomicDPupdate(density+ij1+mzeta+1,wz0*wt10);
   atomicDPupdate(density+ij1+mzeta+2,wz1*wt10);

   atomicDPupdate(density+ij2,wz0*wt01);
   atomicDPupdate(density+ij2+1,wz1*wt01);
   atomicDPupdate(density+ij2+mzeta+1,wz0*wt11);
   atomicDPupdate(density+ij2+mzeta+2,wz1*wt11);
#endif
 }
```

(a) GPU charge deposition (scatter operation)

```
  const int tid = threadIdx.x; const int bid = blockIdx.x;
  const int nblocks = gridDim.x; const int nthreads = blockDim.x;
  const int gid = tid + bid*nthreads; const int np = nblocks * nthreads;
  ...

  for (int m=gid; m<me; m+=np){
      double psitmp   = z0[m];
      double thetatmp = z1[m];
      double zetatmp  = z2[m];

      double wzt = (zetatmp-zetamin)*delz;
      int kk = d_abs_min_int(mzeta-1, (int)wzt);

      double wz1 = wzt - (real) kk;
      double wz0 = 1.0 - wzt;

      double r_diff   = psitmp-a0;
      double rdum = delr * d_abs_min_real(a_diff, r_diff);
      int ii  = d_abs_min_int(mpsi_max, (int) rdum);

      double wp1 = rdum - (real) ii;
      double wp0 = 1.0  - wp1;

      double tflr = thetatmp;
      int im      = ii;
      int im2     = ii + 1;

      double tdumtmp = pi2_inv * (tflr - zetatmp * qtinv[im]);
      double tdumtmp2 = pi2_inv * (tflr - zetatmp * qtinv[im2]);
      double tdum = (tdumtmp - (int) tdumtmp) * delt[im];
      double tdum2 = (tdumtmp2 - (int) tdumtmp2) * delt[im2];

      int j00 = d_abs_min_int(mtheta[im]-1, (int) tdum);
      int j01 = d_abs_min_int(mtheta[im2]-1, (int) tdum2);
      int jtelectron0tmp = igrid[im] + j00;
      int jtelectron1tmp = igrid[im2] + j01;

      double wtelectron0tmp = tdum - (real) j00;
      double wtelectron1tmp = tdum2 - (real) j01;

      double wt10 = wtelectron0tmp;
      double wt11 = wtelectron1tmp;
      double wt01 = 1.0 - wt11;
      double wt00 = 1.0 - wt10;

      int ij1 = kk+(mzeta+1)*jtelectron0tmp;
      int ij2 = kk+(mzeta+1)*jtelectron1tmp;

      int idx1 = 6*ij1;
      int idx2 = 6*ij2;

      double e1=0.0;
      double e2=0.0;
      double e3=0.0;

      /* interpolate from grid to particle - gather operation */
      e1 =e1+wp0*wt00*(wz0*EVECTOR(idx1+0)+wz1*EVECTOR(idx1+3));
      e2 =e2+wp0*wt00*(wz0*EVECTOR(idx1+1)+wz1*EVECTOR(idx1+4));
      e3 =e3+wp0*wt00*(wz0*EVECTOR(idx1+2)+wz1*EVECTOR(idx1+5));

      e1 =e1+wp0*wt10*(wz0*EVECTOR(idx1+6+0)+wz1*EVECTOR(idx1+6+3));
      e2 =e2+wp0*wt10*(wz0*EVECTOR(idx1+6+1)+wz1*EVECTOR(idx1+6+4));
      e3 =e3+wp0*wt10*(wz0*EVECTOR(idx1+6+2)+wz1*EVECTOR(idx1+6+5));

      e1 =e1+wp1*wt01*(wz0*EVECTOR(idx2+0)+wz1*EVECTOR(idx2+3));
      e2 =e2+wp1*wt01*(wz0*EVECTOR(idx2+1)+wz1*EVECTOR(idx2+4));
      e3 =e3+wp1*wt01*(wz0*EVECTOR(idx2+2)+wz1*EVECTOR(idx2+5));

      e1 =e1+wp1*wt11*(wz0*EVECTOR(idx2+6+0)+wz1*EVECTOR(idx2+6+3));
      e2 =e2+wp1*wt11*(wz0*EVECTOR(idx2+6+1)+wz1*EVECTOR(idx2+6+4));
      e3 =e3+wp1*wt11*(wz0*EVECTOR(idx2+6+2)+wz1*EVECTOR(idx2+6+5));

      wpi0[m] = e1;
      wpi1[m] = e2;
      wpi2[m] = e3;
  }
```

(b) GPU field interpolation (gather operation). The code example of using texture memory for evector is given in Figure 2.

References

[1] Blue Waters Cray XE6. http://www.ncsa.illinois.edu/enabling/bluewaters.

[2] V. K. Decyk and T. V. Singh. Particle-in-cell algorithms for emerging computer architectures. *Computer Physics Communications*, 185(3):708 – 719, 2014.

[3] K. Z. Ibrahim, K. Madduri, S. Williams, B. Wang, S. Ethier, and L. Oliker. Analysis and optimization of gyrokinetic toroidal simulations on homogenous and heterogenous platforms. *International Journal of High Performance Computing Applications*, 2013.

[4] K. Madduri, K. Z. Ibrahim, S. Williams, E.-J. Im, S. Ethier, J. Shalf, and L. Oliker. Gyrokinetic toroidal simulations on leading multi- and manycore HPC systems. In *Proc. Int'l. Conf. for High Performance Computing, Networking, Storage and Analysis (SC '11)*, pages 23:1–23:12, New York, NY, USA, 2011. ACM.

[5] B. Wang, S. Ethier, W. Tang, T. Williams, K. Z. Ibrahim, K. Madduri, S. Williams, and L. Oliker. Kinetic turbulence simulations at extreme scale on leadership-class systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 82:1–82:12, New York, NY, USA, 2013. ACM.