Heterogeneous Programming and Optimization of Gyrokinetic Toroidal Code and Large-Scale Performance Test on TH-1A

Xiangfei Meng[1], Xiaoqian Zhu[2], Peng Wang[3], Yang Zhao[1], Xin Liu[2], Bao Zhang[4], Yong Xiao[5], Wenlu Zhang[6], and Zhihong Lin[7]

[1] National Supercomputer Center in Tianjin, Tianjin, China
{mengxf,zhaoyang}@nscc-tj.gov.cn
[2] School of Computer, National University of Defense Technology, Changsha, China
zhu_xiaoqian@sina.com, liuxin@nscc-tj.gov.cn
[3] NVIDIA, Beijing, China
penwang@nvidia.com
[4] Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an, China
123zhangbao456@163.com
[5] Institute for Fusion Theory and Simulation, Zhejiang University, Hangzhou, China
yxiao@zju.edu.cn
[6] CAS Key Laboratory of Plasma Physics, University of Science and Technology of China, Hefei, Anhui, China
wzhang@iphy.ac.cn
[7] Department of Physics and Astronomy, University of California, Irvine, California, USA
zhihongl@uci.edu

Abstract. In this work, we discuss the porting to the GPU platform of the latest production version of the Gyrokinetic Torodial Code (GTC), which is a petascale fusion simulation code using particle-in-cell method. New GPU parallel algorithms have been designed for the particle push and shift operations. The GPU version of the GTC code was benchmarked on up to 3072 nodes of the Tianhe-1A supercomputer, which shows about 2x–3x overall speedup comparing NVIDIA M2050 GPUs to Intel Xeon X5670 CPUs. Strong and weak scaling studies have been performed using actual production simulation parameters, providing insights into GTC's scalability and bottlenecks on large GPU supercomputers.

Keywords: particle-in-cell, hybrid programming, TH-1A.

1 Introduction

The global gyrokinetic toroidal code (GTC) [1] is a massively parallel particle-in-cell code for first-principles, integrated simulations of the burning plasma experiments such as the International Thermonuclear Experimental Reactor (ITER) [2], the crucial next step in the quest for the fusion energy. The GTC code has

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 81–96, 2013.

grown over the years from a single-developer code to a prominent code being developed by an international collaboration with many users and contributors from the magnetic fusion energy and high performance computing communities. GTC is the key production code for the multi-institutional U.S. Department of Energy (DOE) Scientific Discovery through Advanced Computing (SciDAC) project, Gyrokinetic Simulation of Energetic Particle Turbulence and Transport (GSEP) Center, and the National Special Research Program of China for ITER. GTC is currently maintained and developed by an international team of core developers who have the commit privilege, and receives contributions through the proxies of core developers from collaborators worldwide [3].

The GPU version of the GTC code reported in this paper, GTC-GPU, has the same physics capability and parallel scalability of the current production version of GTC originally written in Fortran. GTC continuously pushes the frontiers of both physics capabilities and high performance computing. At present, GTC is the only fusion code capable of integrated simulations of key physical processes that underlie the confinement properties of fusion plasmas, including micro-turbulence [4–7], energetic particles instabilities [8–11], magnetohydrodynamic modes [12], and radio-frequency heating and current drive.

GTC is the first fusion code to reach the teraflop in 2001 on the Seaborg computer at NERSC and the petaflop in 2008 on the jaguar computer at ORNL in production simulations, and to fully utilize the computing power provided by CPU and GPU heterogeneous architecture on Tianhe-1A.

There have been some other recent works on porting PIC (particle-in-cell) codes with simplified physics models to GPU for proof of principles. Decyk et al. [13] discussed a new 2D PIC code with data structures optimized for GPU. Burau et al. [14] discussed the PIConGPU code, a GPU PIC code developed for fast-response simulations of laser-plasma interaction. Stanchev et al. [15] focused on optimizing the particle-to-grid interpolation kernel. Rossinelli et al. [16] focused on optimizing the grid-to-particle interpolation kernel. In contrast, the GTC-GPU reported in this paper is a current production version and the benchmark simulations use actual physics simula-tion parameters [6,7].

The most relevant to the current work is the work of Madduri et al. [17,18], who discussed the porting of an earlier version of GTC to GPU. They concluded that GPU was slower than CPU for their version of GTC, which only include kinetic ions with adiabatic electrons. However, we use the latest version of GTC including new important features for a realistic turbulence simulation, such as kinetic electrons and general geometry. We also employ a set of realistic experiment parameters suitable for simulating plasma turbulence containing both kinetic ions and kinetic electrons [6,7]. As a result, our version contains more routines, especially the computing-intensive module for the electron physics. We also designed new GPU parallel algorithms for the PUSH and SHIFT operations, which are the two most dominant operations in our profiling. The GTC-GPU shows a 2x–3x overall speedup comparing NVIDIA M2050 GPUs to Intel Xeon X5670 CPUs on up to 3072 nodes of the Tianhe-1A supercomputer. Preliminary test run of the GTC-GPU on a small number of nodes of the Titan at ORNL shows a similar speedup.

2 Experimental Platform: Tianhe-1A

All the benchmark runs in the following sections were performed on the Tianhe-1A supercomputer. In this section, we introduce the Tianhe-1A supercomputer which is a hybrid massively parallel processing (MPP) system with CPUs and GPUs. Tianhe-1A is the host computer system of National Supercomputer Center in Tianjin (NSCC-TJ).

The hardware of Tianhe-1A consists of five components: computing system, service system, communication system, I/O storage system, monitoring and diagnostic system. Its software consists of operating system, compiling system, parallel programming environment and the scientific visualization system. Tianhe-1A includes 7168 computing nodes. Currently, each computing node is equipped with two Intel Xeon X5670 CPUs (2.93 GHz, six-core) and one NVIDIA Tesla "Fermi" M2050 GPU (1.15 GHz, 448 CUDA cores). The GPU offers 3 GB GDDR5 memory on board, with the bus width of 384 bits and peak bandwidth of 148 GB/s. The total memory of Tianhe-1A is 262 TB, and the disk capacity is 4 PB. All the nodes are connected via a fat tree network for data exchange. This communication network is constructed by high-radix Network Routing Chips (NRC) and high-speed Network Interface Chips (NIC). The theoretical peak performance of Tianhe-1A was 4.7 PFlops, and its LINPACK test result reached 2.566 PFlops. Moreover, the power dissipation at full load is 4.04 MW and the power efficiency is about 635.1 MFlops/W, which was ranked the fourth highest according to the Green 500 list released in 2010. The whole system power of Tianhe-1A is 4.04 MW with 7168 nodes. So the average power consumption per node is 564 W, which has 2 CPUs and 1 GPU. Using the single CPU power 95 W and single GPU power 200 W, we can then roughly estimate the per node power of using only the CPU or the GPU by subtracting the unused processor power from the total power data.

3 GPU Acceleration

To simulate electron turbulence, we need many subcycles to track the fast electron motion [6,7]. As a result, typically the electron push routine PUSHE takes about 50–70% of the total time while electron shift routine SHIFTE takes about 10–20% of the total time. So those two routines were our focus in the GPU porting so far. We used the CUDA C programming language for the GPU port [19]. The strategy of incremental migration to GPU also allow physics users to immediately utilize the new GPU acceleration while further upgrades of physics capabilities and computing power are continuously implemented in the code. This ensures that the GTC-GPU always remains as the current production version shared by all developers and users.

3.1 Pushe

There are two time-consuming loops in PUSHE. The first loop gathers fields at grid point to every electron's positions (electric fields in electrostatic case

and electromagnetic fields in electromagnetic case). The second loop updates electrons' positions based on the gathered fields. Those two loops were mapped to two kernels in the CUDA version: *gather_fields* and *update_gcpos*.

The parallelization scheme and optimization of those two kernels are actually very similar so we will focus on discussing the *gather_fields* kernel below.

The field gathering loop is highly parallel: the calculations of every particle are completely independent of other particles. Thus a straightforward one thread per particle scheme was used to parallelize the loop. As a result, the CUDA porting of the gather loop is fairly straightforward: just replace the loop statement over all electrons by a thread index calculation and then put all the loop body into a CUDA kernel, which won't need much change itself.

```
e1=e1+wt00*(wz0*gradphi(1,0,ij)+wz1*gradphi(1,1,ij))
e2=e2+wt00*(wz0*gradphi(2,0,ij)+wz1*gradphi(2,1,ij))
e3=e3+wt00*(wz0*gradphi(3,0,ij)+wz1*gradphi(3,1,ij))
e4=e4+wt00*(wz0*phit(0,ij)+wz1*phit(1,ij))
```

→

```
float4 tmp1 = tex1Dfetch(texGradphiPhit, ij*2);
float4 tmp2 = tex1Dfetch(texGradphiPhit, ij*2+1);
e1 += wt00*(wz0*tmp1.x + wz1*tmp1.w);
e2 += wt00*(wz0*tmp1.y + wz1*tmp2.x);
e3 += wt00*(wz0*tmp1.z + wz1*tmp2.y);
e4 += wt00*(wz0*tmp2.z + wz1*tmp2.w);
```

Fig. 1. Field gathering code in the original CPU (left) and CUDA version using texture prefetch (right)

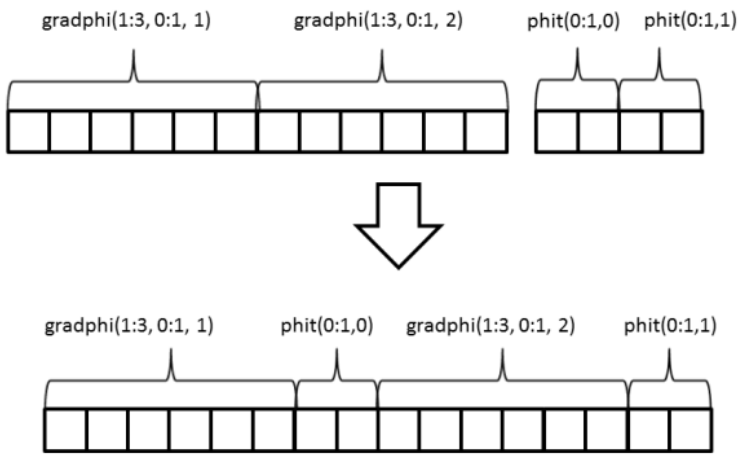

Fig. 2. Reorganizing gradphi and phit arrays into a single array

In this initial CUDA version, kernel profiling showed that the bottleneck is in the reading of fields, part of which is shown in Fig. 1. In this code segment, each

particle gathers the *gradphi* and *phit* fields surrounding its hosting cells. As neighboring particles in the particle array do not necessarily belong to the same cell, this gathering operation is potentially highly uncoalesced, which may lead to a lot of wasted memory transactions on GPU [19]. Furthermore, the linear memory layout of GPU's L1 cache is not a good fit to the higher dimensional locality nature of 3D particle position. GPU has a special cache called texture which is optimized for 2D spatial locality. As a result, binding the *gradphi* and *phit* arrays to texture will improve the performance by about 20%. However, if just doing this code change, the texture cache hit rate turns out to be fairly low: about 8%. Analyzing the field gathering code suggests that if both arrays can be prefetched to the texture cache, the cache hit rate may be significantly increased. To achieve this, *gradphi* and *phit* arrays are first reorganized into a single array (Fig. 2) so that the data needed for a thread is laid out continuously in memory. Second, as each gathering operation needs to load 6 floats in *gradphi* and 2 floats in *phit*, the reorganized array was bound to a float4 texture. In this way, 2 float4 texture fetch to 2 float4 register variables would be enough for all the data a field gathering operation needs. This texture prefetch technique increased the texture cache hit rate from 8% to 35%, which led to 3x kernel performance increase. The final kernel is memory-bound and achieves about 65% of the peak memory band-width. It worth commenting that, as the *gather_fields* kernel is called many times every time *gradphi* and *phit* are updated, the data reorganization overhead is negligible.

Another key optimization technique in PUSHE is minimizing CPU-GPU data transfer. As a complicated plasma physics code, several dozen arrays are needed for the PUSHE calculation. If all the arrays were transferred from CPU to GPU at the beginning of PUSHE and back at the end, CPU-GPU data transfer time would totally dominate the PUSHE time. To avoid this, careful analysis of the data flow in PUSHE was performed. Based on the analysis, there are several optimizations one can do to minimize CPU-GPU data transfers. First, for temporary arrays that are used only inside PUSHE, they can be allocated, used and deallocated directly on the GPU. Second, some temporary arrays in the CPU version are used only within a single kernel. In this case, they can be replaced by register variables, instead of explicitly allocated. Third, there are some arrays whose values are initialized at simulation initialization time and will never change during the whole simulation, e.g. arrays describing grid geometries. Thus those arrays were declared as static variables in PUSHE and data are transferred from CPU to GPU only when calling the routine for the first time. After those optimizations, as can been seen in the next section, CPU-GPU data transfer time will not be a significant fraction.

3.2 Shifte

SHIFTE contains both computation and MPI part. The computation part contains two major steps:

1. Figure out which particles need to be transferred outside of the current MPI process and then copying outgoing particles to separate send buffers.

2. Fill in the "holes" in the particle array left by outgoing particles so that particles are always stored continuously in memory.

The two computational part is actually not very expensive on CPU: both steps basically needs only a single *O(*N*)* transversal of the particle array. The operation itself has also nontrivial sequential part: only after all the previous outgoing particles are processed, one can know where to put the next outgoing particle. Thus it's expected that the porting of SHIFTE to GPU is nontrivial and probably won't see the significant kernel speedup as compared to PUSHE due to the low compute intensity and nontrivial parallelism.

However, porting SHIFTE to GPU is important for the overall performance of the GPU code. If SHIFTE stays on CPU, one needs to transfer all the particle positions to CPU as we don't know which particles are outgoing. This was actually our first implementation. It turns out that in this case data transfer will be a significant overhead. On the other hand, if one can figure out the outgoing particles on GPU, then only outgoing particles need to be transferred back to CPU, which is typically a small fraction. Thus if SHIFTE was not ported to GPU, it will be much slower than the CPU version. So the major purpose of porting SHIFTE is really to get a GPU version that is not slower than the CPU version so SHIFTE will not become a bottleneck.

Flagging which particles are outgoing is straightforward to do in parallel. One can just assign one thread per particle for this calculation. After this step, a flag array is produced whose elements will be one if the corresponding particle is outgoing and zero otherwise. However, writing outgoing particles to a continuous buffer is nontrivial to implement in parallel: the key difficulty is that each outgoing particle needs to know the number of outgoing particles are before itself, which will be used as its output position in the send buffer. However, the number of outgoing particles is determined during runtime. So the key is to figure out a parallel algorithm for finding out the number of outgoing particles before each outgoing particles. Scan is the usual parallel primitive to solve this problem [20]. If one performs an exclusive scan of the flagging array, the corresponding scan result will be just the output positions in send buffer. This was used as the first version in our GPU SHIFTE implementation. However, this version turns out to be about 2x slower than the CPU version. Profiling shows that the bottleneck is in the scan operations as expected. A parallel scan needs to traverse the particle array many times [20]. So the GPU version turns out to be slower than the CPU version since the CPU version only needs to traverse the array once.

To overcome this problem, we adapted the hierarchical scan solution to stream compaction problem [21]. This approach can avoid the need to traverse the particle array multiple times. The basic idea is dividing the operation into 3 steps:

1. First the __ballot warp vote function is combined with the intrinsic integer instruction __popc to implement a very fast intra-warp scan routine. Then the position array is divided to groups of 32 in size (the same size as a CUDA warp). Next the intra-warp scan routine is applied to each group and it writes the number of outgoing particles of each group into an *outgoing_count* array.

2. Performing a scan of the *outgoing_count* array, writing the results to a *global_offset* array. The key now is to realize that the i-th element of the *global_offset* array now stores the number of outgoing particles before the i-th group.
3. Applying the intra-warp scan routine to each group again to calculate the intra-warp offset. Finally, adding the *global_offset* to the intra-warp offset would give the global output positions of each outgoing particles.

Note that in step two, as the *global_offset* array is only 1/32 of the size of the particle array, this scan will be much faster than our first scan-based implementation, which is the key for the higher performance. It is also worth mentioning that as the outgoing particles are divided into left and right directions, these steps need to be performed for both the left and right outgoing particles simultaneously. The hole filling part of SHIFTE is handled in a similar scan-based way, so it won't be discussed in detail here.

4 Results and Analysis

In this section, we present the performance of GTC on Tianhe-1A.

4.1 Medium Scale Benchmark Problem

We carried out the performance study using the typical collisionless trapped electron (CTEM) turbulence [6,7] parameters in a production run for a medium size tokamak, $\alpha = 250\rho_i$, where α is the tokamak minor radius and ρ_i is the ion gyroradius. There are in total 2 billion ions and 0.83 billion electrons for this medium size problem.

For this benchmark, we launch 128 MPI processes on 128 nodes of Tianhe-1A. In the CPU case, each MPI process launches 6 OpenMP threads to utilize all the 6 cores of a single CPU within a node. In the GPU case, each MPI process still launches 6 OpenMP threads for the CPU computation while uses the M2050 within each node to accelerate the electron calculation.

Table 1 shows the total time ("loop") and profile of different modules in GTC. The first two columns show the time and fraction of the CPU version using 128 CPUs while the next two columns show the corresponding numbers for the 128 CPUs + 128 GPUs runs. The last column reports the overall speedup, as well as speedup of the modules that has been ported to GPU. The GTC GPU version gets about 3.1x speedup comparing 128 CPUs + 128 GPUs to 128 CPUs. The results demonstrate that our GPU acceleration of GTC can deliver significant application performance increases. The GPU accelerated routine PUSHE actually got about 8x speedup. It is just because of Amdahl's law the overall speedup is lower.

To understand in more details the GPU performance, Table 2 shows the profile of the GPU PUSHE routine. In this table, "h2d" refers to the CPU-GPU transfer time; "d2h" refers to the GPU-CPU transfer time; "init" refers to memory allocation, free and initialization time; "cpu" refers to computations that

Table 1. Profile of the CPU and GPU version for 128 MPI processes run

	128 CPUs (second)	%	128 CPUs + 128 GPUs (second)	%	Speedup
loop	522.25	100	168.28	100	3.1
field	0.63	0.12	0.66	0.39	
ion	54.4	10.42	54.6	32.45	
shifte	84.6	16.20	51.8	30.78	1.6
pushe	365.4	69.97	44	26.15	8.3
poisson	4.4	0.84	4.4	2.61	
electron other	12	2.30	12	7.13	
diagnosis	0.82	0.16	0.82	0.49	

Table 2. Profile of the GPU version PUSHE for 128 MPI processes run

	elapsed time (second)	%
total	43.99	100
kernel	32.5	73.9
h2d	2.75	6.3
d2h	3.16	7.2
init	1.11	2.5
cpu	4.35	9.9
mpi	0.12	0.3

are still performed on CPU. It can be seen that about 74% of the PUSHE time is spent in the kernel computations. This shows that PUSHE spends most of its running time doing actual computations instead of communication and other overheads. Within the kernel time, 50% of the time is spent in the *gather_fields* kernel and 36% of the time is spent in the *update_pos* kernel. So those two kernels will be the primary target for future optimizations. About 13.5% of the PUSHE time is spent in CPU-GPU data transfer. This shows that our optimizations in minimizing data transfer were successful and it is not a bottleneck of the GPU implementation.

For SHIFTE, with the new hierarchical scan method, the GPU version performance is about 1.6x of the CPU version. As our initial purpose of porting SHIFTE is to make it not slower than the CPU version so we can avoid the expensive data-transfer cost, this result exceeded our initial goal.

4.2 Strong Scaling Results

For the strong scaling study, we use the same medium size problem as in section 4.1, but vary the number of cores in the simulation.

As Tianhe-1A has 2 CPUs and 1 GPU within each compute node, 2 sets of strong scaling studies were performed. The first set uses 1 CPU and 1 GPU within a node (1 CPU + 1 GPU) while the other set uses 2 CPUs and 1 GPU within a node (2 CPUs + 1 GPU). We think both sets are interesting because the first set is the typical way of discussing GPU acceleration result while the second set would be the practical benefit that a Tianhe-1A user will get.

For both sets, the number of MPI processes launched is equal to the number of nodes used. Six OpenMP threads were launched for the first set while twelve OpenMP threads were launched for the second set.

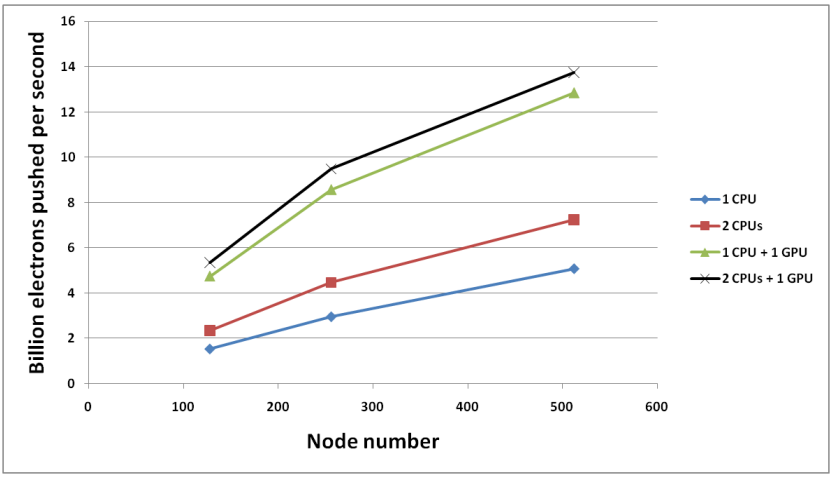

Fig. 3. Performance of CPU and GPU GTC on Tianhe-1A. From bottom to top, the lines correspond to each nodes use 1 CPU, 2 CPUs, 1 CPU + 1 GPU and 2 CPUs + 1 GPU

Fig. 3 shows the total time for those runs. The x-axis shows node number while the y-axis shows billion electrons pushed per second. It is worth commenting that the reason strong scaling experiment used up to 512 nodes is because weak scaling is more relevant in practices than the strong scaling since we typically use more cores for simulations of a larger problem size (i.e., roughly constant wall-clock time for each simulation). For the modest problem size in the strong scaling plot of Fig. 3, typically less than 512 nodes are used in the production runs. It can be seen that for a fixed node count, if comparing the CPU version performance in those two sets, going from 1 CPU to 2 CPUs in each node leads to about 1.5x speedup. On the other hand, the differences in the GPU performance are much smaller because most of the application time is spent on GPU so additional CPU computation power won't help much in improving overall performance.

Fig. 4 shows the GPU speedup factor in those two sets. One thing to notice is that the speedup factor will decrease at large node counts: e.g. from 3.1x at 128

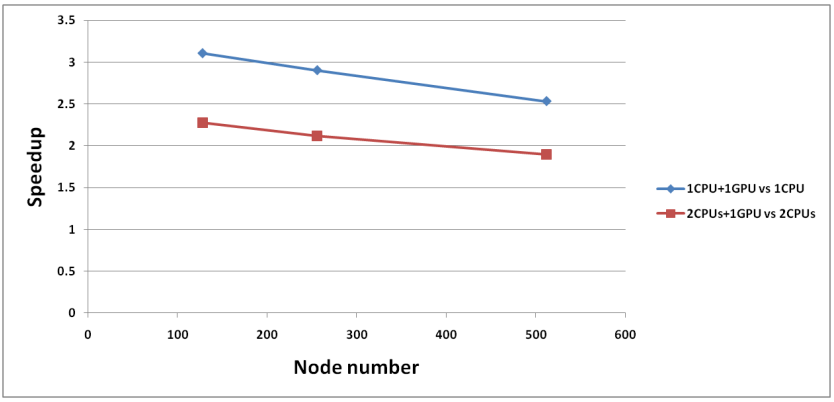

Fig. 4. GTC GPU speedup on Tianhe-1A. The blue line is comparing 1 CPU + 1 GPU to 1 CPU per node while the red line is comparing 2 CPUs + 1 GPU to 2 CPUs per node.

Table 3. Profile of the CPU and GPU version for 512 MPI processes run

	512 CPUs (second)	%	512 CPUs + 512 GPUs (second)	%	Speedup
loop	158.09	100	62.05	100	2.5
field	0.63	0.40	0.69	1.11	
ion	16.3	10.31	16.2	26.11	
shifte	31.7	20.05	17.5	28.20	1.8
pushe	94.3	59.65	12.5	20.15	7.5
poisson	4.9	3.10	4.9	7.90	
electron other	10	6.33	10	16.12	
diagnosis	0.26	0.16	0.26	0.42	

nodes to 2.5x at 512 nodes for the 1 CPU + 1 GPU set. There are two possibilities causing this phenomenon. First, as a strong scaling study, at large node count the problem size for each MPI process will decrease. A smaller computational problem may lead to a lower GPU routine speedup. Second, the remaining CPU part doesn't scale well so the total time spent in GPU-accelerated routines will decrease at a large node count. As a result, the GPU speedup factor also decreases because of Amdahl's law.

To find out which is the case, the profiling details of the 512 nodes case are shown in Table 3. It can be seen that with the smaller problem per MPI processes, the PUSHE speedup indeed decreases from 8.3x to 7.5x. However, larger node also leads to more computation in SHIFTE and correspondingly SHIFTE speedup increases from 1.6x to 1.8x. Those two effects slightly offset each other. As a result, GPU acceleration in PUSHE + SHIFTE drops from 4.7x

to 4.2x. On the other hand, the percentage of PUSHE + SHIFTE decreases from 86.3% to 80%. This is mainly because the Poisson and electron charge deposition part do not scale as well as PUSHE. With those data, we can do two thought experiments. First, if PUSHE + SHIFTE was still 86.3% of the total time and GPU speedup is 4.2x, the overall GPU speedup would be 2.9x. Second, if GPU speedup was still 4.7x and PUSHE + SHIFTE is 80% of the total time, the overall GPU speedup would be 2.7x. This analysis shows that *for strong scaling, both reduced GPU acceleration and CPU scaling contribute to the reduced GPU speedup at large node counts but the CPU scaling plays a bigger role.*

Energy efficiency is one of the most important issues in current and future supercomputer systems. So in addition to absolute performance, performance per watt would also be an interesting metric. Using the method discussed in the end of section 2, the performance per W of different runs can be estimated and plotted in Fig. 5.

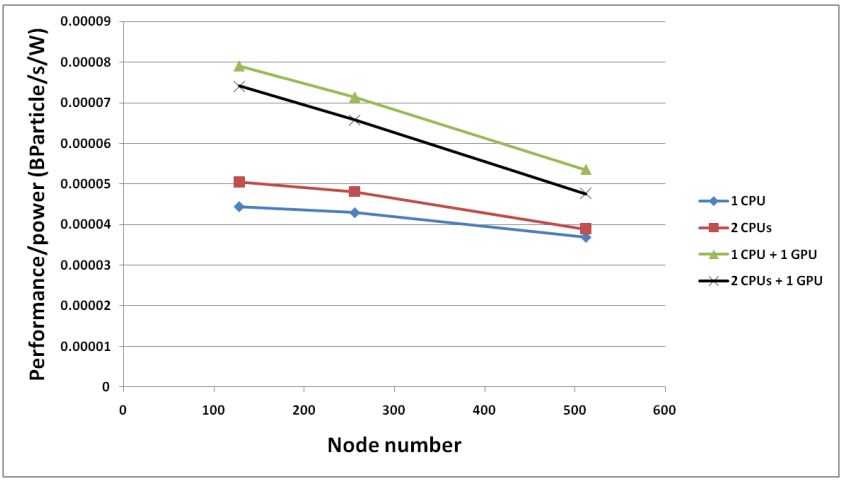

Fig. 5. Performance/W for the strong scaling runs

Fig.5 shows the interesting fact that while 2 CPUs + 1 GPU has the highest performance, 1 CPU + 1 GPU actually has the highest energy efficiency. This is easy to understand. The GPU is more energy efficient for GTC's computations. So using one more less energy-efficient CPU will decrease the overall energy efficiency.

4.3 Weak Scaling Results

It is even more important to test the weak scaling of the GTC code, since more cores are usually desired for a larger size problem in production simulation.

For weak scaling study on Tianhe-1A, we still use the two sets of comparison methods in the strong scaling study, i.e. comparing 1 CPU to 1 CPU + 1 GPU and comparing 2 CPUs to 2 CPUs + 1 GPU.

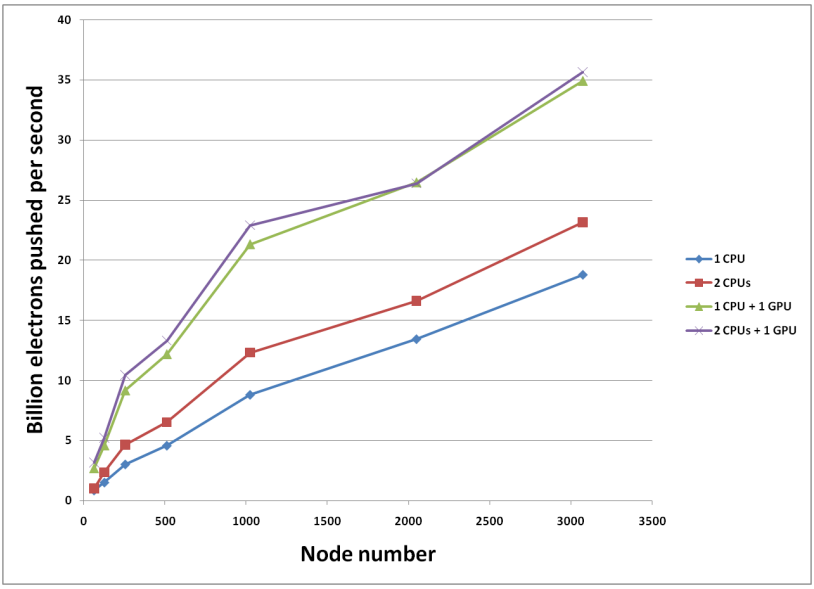

Fig. 6. GTC weak scaling performance on Tianhe-1A

Fig. 6 shows the GTC performance in this weak scaling study while Fig. 7 shows the speedup factor. Fig. 6 shows that in the CPU code, using 2 CPUs within a node will give a significant performance increase compared to using just 1 CPU. However, for the GPU version, 2 CPUs does not give any significant performance increase compared to the 1 CPU case. This is because most of the computation is now in the GPU and the additional CPU power does not lead to further performance gains. As a result, the speedup factor is lower in the second comparison set as can be seen in Fig. 7. In either case, it can be seen that using the GPU in each Tianhe-1A's node, large scale GTC simulations can get a 1.5–2x performance gain even at large node counts. This demonstrates the advantage of using GPU for large scale fusion simulations on Tianhe-1A.

Fig.7 also shows that the GPU speedup decreases with increasing node count. Similar to our analysis in the strong scaling case, Table 4 shows the detailed profiling for the 1 CPU vs 1 GPU set at 3072 nodes. From the speedup column, it can be seen that the GPU speedup factor is about the same for PUSHE, as expected from a weak scaling study. The speedup factor for SHIFTE decreases from 1.6x to 1.2x. This is primarily because MPI communication takes a larger percentage of SHIFTE time at larger node count and thus the effect of GPU

acceleration is smaller. The CPU profiling result shows that PUSHE + SHIFTE decreases from 86% at 128 nodes to 61%, which leads to a lower GPU speedup because of Amdahl's law. Thus *for weak scaling, CPU scaling is the main reason for the reduced GPU speedup at a large node count.*

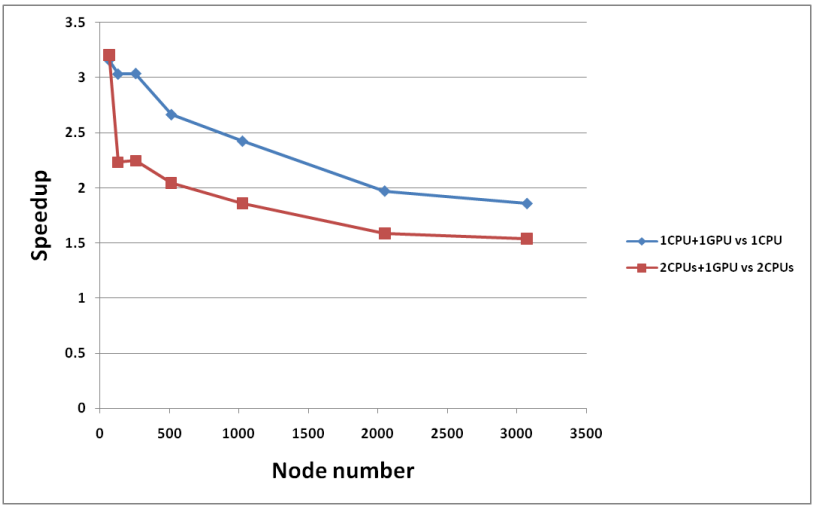

Fig. 7. GTC weak scaling speedup on Tianhe-1A

Table 4. Profile of the CPU and GPU version for 3072 MPI processes run

	3072 CPUs (second)	%	3072 CPUs + 3072 GPUs (second)	%	Speedup
loop	699.5	100	375.4	100	1.9
field	9.3	1.33	8.9	2.37	
ion	79.5	11.37	79.3	21.12	
shifte	67.3	9.62	55	14.65	1.2
pushe	359.5	51.39	44.2	11.77	8.1
poisson	53	7.58	53	14.12	
electron other	98.3	14.05	102.8	27.38	
diagnosis	32.6	4.66	32.2	8.58	

Similar to the strong scaling case, we also plot the performance per watt of the weak scaling runs in Fig.8.

Similar to the strong scaling case, Fig.8 also shows that 1 CPU + 1 GPU would be the most energy efficient way to run GTC simulations.

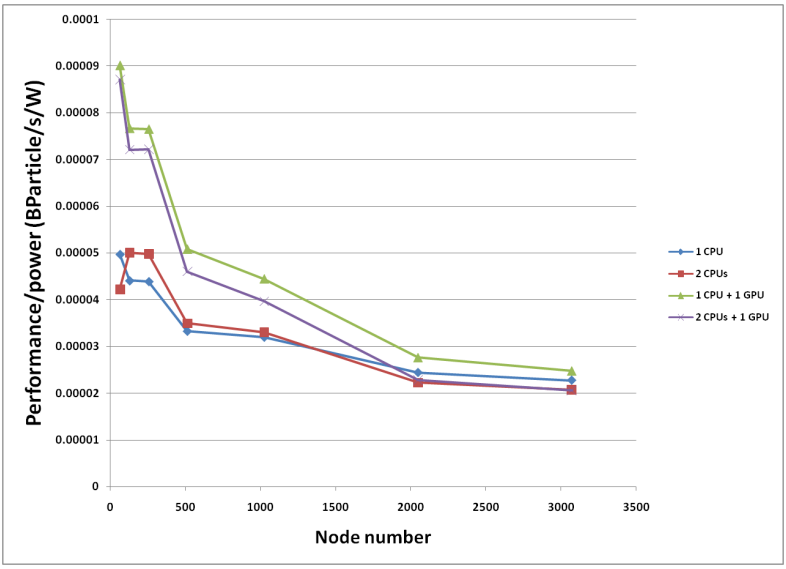

Fig. 8. Performance per Watt of the weak scaling runs

5 Conclusions and Discussions

As far as we know, this is the first work to demonstrate the advantage of GPU for large-scale production fusion simulations. We showed that the texture cache specific to the GPU architecture, combined with some data reorganization, is particularly suitable for the data locality pattern of the PUSH operations, which led to significant kernel speedup. We also presented new hierarchical scan-based implementation of the SHIFT operation on GPU, which is faster than the CPU version.

To further advance our understanding of the microphysics of burning plasma, larger scale simulations will be needed whose computational requirements far exceed the capabilities of today's petascale machines. This work shows that GPU has the great potential to enable those next generation large scale fusion simulations. Our experience with GTC-GPU demonstrates again the implications of the large-scale heterogeneous cluster computing: both the CPU and GPU parts need to get good performance and scaling in order to get good overall application performance and scaling.

As for future development, we have seen that the primary reason for reduced GPU speedup at large node counts is the CPU scaling. Thus further work is needed to improve the performance and scaling of the other CPU part. Furthermore, after electron part is ported to GPU, the next most time-consuming module is ion, which will be our next porting target. Finally, with OpenACC and CUDA Fortran becoming mature recently, we are also evaluating reporting PUSHE using OpenACC and CUDA Fortran. As the main difficulty of porting to

GPU is choosing the proper parallelization scheme and optimization techniques, which are solved in this work, using OpenACC and CUDA Fortran will be just applying those techniques with a different syntax. So we expect the performance to be basically the same as the CUDA C version as reported in this paper. However, OpenACC and CUDA Fortran should make the GPU code easier to maintain as GTC's CPU code is Fortran-based.

Acknowledgments. This research is supported by National Magnetic Confinement Fusion Science Program under grant NO. 2009GB105000 and the National Natural Science Foundation of China under grant NO. 11105033. Thanks to Jinghua Feng, Bin Xu of National Supercomputer Center in Tianjin, they help us solve some problems when we do large-scale test on TH-1A.

References

1. Lin, Z., Hahm, T.S., Lee, W.W., Tang, W.M., White, R.B.: Turbulent Transport Reduction by Zonal Flows: Massively Parallel Simulations. Science 281, 1835 (1998)
2. `http://www.iter.org`
3. `http://phoenix.ps.uci.edu/GTC`
4. Lin, Z., Holod, I., Chen, L., Diamond, P.H., Hahm, T.S., Ethier, S.: Wave-particle decorrelation and transport of anisotropic turbulence in collisionless plasmas. Phys. Rev. Lett. 99, 265003 (2007)
5. Zhang, W., Lin, Z., Chen, L.: Transport of Energetic Particles by Microturbulence in Magnetized Plasmas. Phys. Rev. Lett. 101, 095001 (2008)
6. Xiao, Y., Lin, Z.: Turbulent transport of trapped electron modes in collisionless plasmas. Phys. Rev. Lett. 103, 085004 (2009)
7. Xiao, Y., Lin, Z.: Convective motion in collisionless trapped electron mode turbulence. Phys. Plasmas 18, 110703 (2011)
8. Holod, I., Zhang, W.L., Xiao, Y., Lin, Z.: Electromagnetic formulation of global gyrokinetic particle simulation in toroidal geometry. Phys. Plasmas 16, 122307 (2009)
9. Zhang, H.S., Lin, Z., Holod, I., Wang, X., Xiao, Y., Zhang, W.L.: Gyrokinetic particle simulation of beta-induced Alfven eigenmode. Phys. Plasmas 17, 112505 (2010)
10. Zhang, W., Holod, I., Lin, Z., Xiao, Y.: Global Gyrokinetic Particle Simulation of Toroidal Alfven Eigenmodes Excited by Antenna and Fast Ions. Phys. Plasmas 19, 022507 (2012)
11. Deng, W., Lin, Z., Holod, I., Wang, Z., Xiao, Y., Zhang, H.: Linear properties of reversed shear Alfven eigenmodes in DIII-D tokamak. Nuclear Fusion 52, 043002 (2012)
12. Deng, W., Lin, Z., Holod, I.: Gyrokinetic simulation model for kinetic magnetohydrodynamic processes in magnetized plasmas. Nuclear Fusion 52, 023005 (2012)
13. Decyk, V.K., Singh, T.V.: Adaptable particle-in-cell algorithms for graphical processing units. Computer Physics Communications 182(3), 641–648 (2011)
14. Burau, H., Widera, R., Honig, W., Juckeland, G., Debus, A., Kluge, T., Schramm, U., Cowan, T.E., Sauerbrey, R., Bussmann, M.: PIConGPU: A Fully Relativistic Particle-in-Cell Code for a GPU Cluster. IEEE Transaction on Plasma Science 38(10), 2831–2839 (2010)

15. Stantchev, G., Dorland, W., Gumerov, N.: Fast parallel particle-to-grid interpolation for plasma PIC simulations on the GPU. Journal of Parallel and Distributed Computing 68(10), 1339–1349 (2008)
16. Rossinelli, D., Conti, C., Koumoutsakos, P.: Mesh-particle interpolations on graphics processing units and multicore central processing units. Philosophical Transactions of the Royal Society 369, 2164–2175 (2011)
17. Madduri, K., Ibrahim, K.Z., Williams, S., Im, E.J., Ethier, S., Shalf, J., Oliker, L.: Gyrokinetic toroidal simulations on leading multi- and manycore HPC systems. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (2011)
18. Madduri, K., Im, E.J., Ibrahim, K.Z., Williams, S., Ethier, S., Oliker, L.: Gyrokinetic particle-in-cell optimization on emerging multi- and manycore platforms. Parallel Computing 37(9), 501–520 (2011)
19. NVIDIA Corporation, CUDA Programming Guide. In: CUDA Development Toolkit (2011)
20. Sengupta, S., Harris, M., Zhang, Y., Owens, J.: Scan Primitives for GPU Computing. In: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware (2007)
21. Billeter, M., Olsson, O., Assarsson, U.: Efficient Stream Compaction on Wide SIMD Many-Core Architectures. In: High Performance Graphics (2010)