



# GASPRNG: GPU accelerated scalable parallel random number generator library<sup>☆</sup>

Shuang Gao, Gregory D. Peterson<sup>\*</sup>

Electrical Engineering and Computer Science, University of Tennessee, Knoxville, United States

## ARTICLE INFO

### Article history:

Received 27 January 2012

Received in revised form

30 November 2012

Accepted 3 December 2012

Available online 12 December 2012

### Keywords:

GPGPU

Pseudorandom number generator

SPRNG

Monte Carlo

## ABSTRACT

Graphics processors represent a promising technology for accelerating computational science applications. Many computational science applications require fast and scalable random number generation with good statistical properties, so they use the Scalable Parallel Random Number Generators library (SPRNG). We present the GPU Accelerated SPRNG library (GASPRNG) to accelerate SPRNG in GPU-based high performance computing systems. GASPRNG includes code for a host CPU and CUDA code for execution on NVIDIA graphics processing units (GPUs) along with a programming interface to support various usage models for pseudorandom numbers and computational science applications executing on the CPU, GPU, or both. This paper describes the implementation approach used to produce high performance and also describes how to use the programming interface. The programming interface allows a user to be able to use GASPRNG the same way as SPRNG on traditional serial or parallel computers as well as to develop tightly coupled programs executing primarily on the GPU. We also describe how to install GASPRNG and use it. To help illustrate linking with GASPRNG, various demonstration codes are included for the different usage models. GASPRNG on a single GPU shows up to 280x speedup over SPRNG on a single CPU core and is able to scale for larger systems in the same manner as SPRNG. Because GASPRNG generates identical streams of pseudorandom numbers as SPRNG, users can be confident about the quality of GASPRNG for scalable computational science applications.

### Program summary

*Program title:* GASPRNG

*Catalogue identifier:* AEOL\_v1\_0

*Program summary URL:* [http://cpc.cs.qub.ac.uk/summaries/AEOL\\_v1\\_0.html](http://cpc.cs.qub.ac.uk/summaries/AEOL_v1_0.html)

*Program obtainable from:* CPC Program Library, Queen's University, Belfast, N. Ireland

*Licensing provisions:* UTK license.

*No. of lines in distributed program, including test data, etc.:* 167900

*No. of bytes in distributed program, including test data, etc.:* 1422058

*Distribution format:* tar.gz

*Programming language:* C and CUDA.

*Computer:* Any PC or workstation with NVIDIA GPU (Tested on Fermi GTX480, Tesla C1060, Tesla M2070).

*Operating system:* Linux with CUDA version 4.0 or later. Should also run on MacOS, Windows, or UNIX.

*Has the code been vectorized or parallelized?:* Yes. Parallelized using MPI directives.

*RAM:* 512 MB~ 732 MB (main memory on host CPU, depending on the data type of random numbers.) / 512 MB (GPU global memory)

*Classification:* 4.13, 6.5.

<sup>☆</sup> This paper and its associated computer program are available via the Computer Physics Communication homepage on ScienceDirect (<http://www.sciencedirect.com/science/journal/00104655>).

<sup>\*</sup> Corresponding author. Tel.: +1 8659746352; fax: +1 8659745483.

E-mail addresses: [sgao@utk.edu](mailto:sgao@utk.edu) (S. Gao), [gdp@utk.edu](mailto:gdp@utk.edu) (G.D. Peterson).

*Nature of problem:*

Many computational science applications are able to consume large numbers of random numbers. For example, Monte Carlo simulations are able to consume limitless random numbers for the computation as long as resources for the computing are supported. Moreover, parallel computational science applications require independent streams of random numbers to attain statistically significant results. The SPRNG library provides this capability, but at a significant computational cost. The GASPRNG library presented here accelerates the generators of independent streams of random numbers using graphical processing units (GPUs).

*Solution method:*

Multiple copies of random number generators in GPUs allow a computational science application to consume large numbers of random numbers from independent, parallel streams. GASPRNG is a random number generators library to allow a computational science application to employ multiple copies of random number generators to boost performance. Users can interface GASPRNG with software code executing on microprocessors and/or GPUs.

*Running time:*

The tests provided take a few minutes to run.

© 2012 Elsevier B.V. Open access under [CC BY-NC-ND license](#).

## 1. Introduction

A pseudorandom number generator with good statistical properties is a key component for a set of computational science applications such as Monte Carlo simulations. To be practically useful in massively parallel codes, the ability to create a large number of independent pseudorandom number streams represents another basic requirement. A variety of random number generating algorithms and libraries are available [1]. In particular, the Scalable Parallel Random Number Generators Library (SPRNG) provides high quality pseudorandom number generation for large numbers of independent, parallel threads of execution [2,3].

Over the past fifteen years, graphical processing units (GPUs) evolved from specialized custom hardware, to programmable graphical processing engines, and recently to generally applicable computational engines, particularly for streaming floating point calculations [4,5]. Since the introduction of GPUs to high performance computing (HPC), many HPC applications and libraries exploit GPU accelerators to obtain performance improvements. To facilitate the use of GPUs for scientific computing, developers currently can choose between two primary programming languages: CUDA and OpenCL. Nvidia developed its CUDA programming language for use with its graphics hardware, particularly with its Tesla and Fermi families of GPUs [6,7]. OpenCL is an open standard supporting a broad spectrum of multi-core processors and computational accelerators such as GPUs [8,9]. The code developed for this paper and our GASPRNG library employs CUDA.

Monte Carlo simulation is a statistical method; the GPU architecture and programming model fit well for this type of application [10,11]. To meet the requirements of these GPU accelerated codes, many parallel random number generators are available [10,12–26]. These generators work in two ways:

- (1) Providing interface functions to CPU routines, and creating numerous generators on the GPU to generate random numbers in the GPU's global memory [13,14]; or,
- (2) Providing a device level interface to tightly integrate random number generation into the caller's GPU kernel [10,12,15–26].

This paper presents the GPU Accelerated Scalable Parallel Random Number Generators (GASPRNG) library which extends the Scalable Parallel Random Number Generators (SPRNG) library for CPU+GPU hybrid simulations. SPRNG is a commonly used random number generator library [2,3] with good statistic properties. It includes six types of generators: Modified Lagged Fibonacci Generator (LFG), Linear Congruential Generator (LCG), 64-bit LCG (LCG64), Multiplicative Lagged Fibonacci Generator (MLFG), Combined Multiple Recursive Generator (CMRG), and Prime

Modulus Linear Congruential Generator (PMLCG). GASPRNG has identical statistical properties as the original SPRNG.

To reflect the diverse ways in which GPUs can be employed with applications, GASPRNG supports several different approaches for interfacing. First, applications executing with CPU cores can employ GASPRNG as a co-processor for generating pseudorandom numbers on demand, referred to herein as the *host GASPRNG interface*. The host interface allows developers to exploit GPUs with very little change to their original application, particularly for legacy users of SPRNG. In this case we focus on managing the buffers and data movement from the GPU. Second, GASPRNG can populate buffers of pseudorandom numbers in GPU global memory for application threads executing on GPUs, referred to as the *global GASPRNG interface*. This case corresponds to existing GPU applications that require high-quality, high-performance pseudorandom number generators, with an emphasis on managing buffers on the GPU. Finally, GASPRNG can support tight integration with GPU programs in which the threads can create their own local, independent pseudorandom number generators, an approach referred to as the *device GASPRNG interface*.

The paper is organized as follows: Section 2 describes the framework design, Section 3 describes the GPU kernel implementation for the six SPRNG algorithms, Section 4 is a brief interface description, and Section 5 includes experimental results and analysis. Sections 6 and 7 list the installation and testing procedures. Section 8 provides conclusions for the work.

## 2. GASPRNG framework

GASPRNG has three interfaces: *host GASPRNG interface* (*HOST\_GASPRNG*), *global GASPRNG interface* (*GLOBAL\_GASPRNG*), and *device GASPRNG interface* (*DEVICE\_GASPRNG*). *HOST\_GASPRNG* provides random numbers to CPU routines; *GLOBAL\_GASPRNG* manages a random number pool in GPU global memory; *DEVICE\_GASPRNG* creates one generator for each GPU thread, and integrates the generator kernel into its caller. The purpose and implementation of each interface aims at providing an efficient random number source for different application requirements.

### 2.1. *HOST\_GASPRNG*

This interface uses the GPU to produce pseudorandom numbers and place them in main memory to support applications executing on the host. With *HOST\_GASPRNG*, there are three main steps: (1) generating pseudorandom number on the GPU; (2) transferring

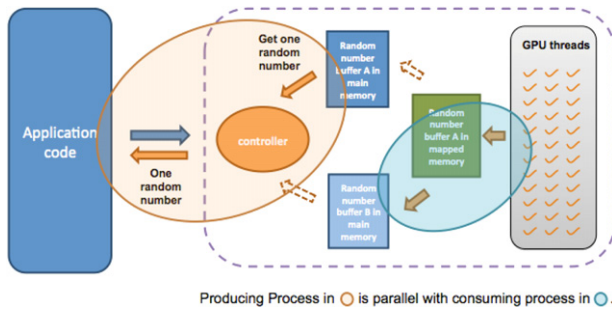


Fig. 1. Framework for HOST\_GASPRNG interface.

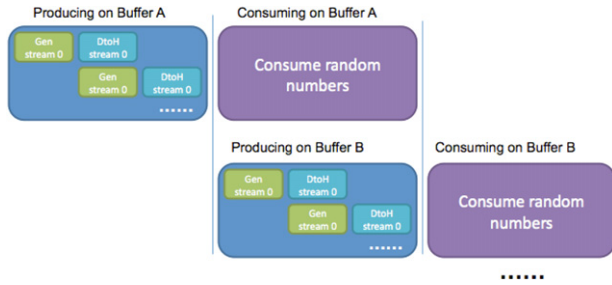


Fig. 2. Two-level pipeline to shorten the runtime; producing and consuming happen in parallel.

random numbers across the PCIe bus; and (3) reading random numbers from main memory. How to improve the efficiency of this process is the central step to make GASPRNG outperform SPRNG.

HOST\_GASPRNG has two work modes: NUMBER\_MODE and BUFFER\_MODE. For both of them, the library sets up a two-level pipeline to shorten the runtime. Fig. 1 is the framework of the design. The left side is the application that gets random numbers through host interface functions. The outermost layer of HOST\_GASPRNG is a controller; it connects with two page-locked host memory buffers to implement double-buffering. This double-buffering strategy aims at improving the parallelism of the random number production and consumption. When the application is consuming numbers in one buffer, the other buffer accepts new numbers from the GPU. When the current consumer buffer is empty, the buffers roles are exchanged, with the empty buffer filled by the GPU while the CPU consumer retrieves random numbers from the full buffer.

The generation is pipelined with help of CUDA's asynchronous streams. The generator kernel executes in parallel with random number transfer across the PCIe bus. The whole task is divided among several CUDA streams, and each of them fills a part of the global memory buffer. The two-level pipeline is shown in Fig. 2. In future architecture, if the PCIe bus is not a necessary part, the inner level pipeline could be removed.

When no CPU routine requires random numbers, or both host buffers are full, the GPU generators are suspended until the controller triggers the next producing process. In the meantime, the GPU could be used for other purposes. GASPRNG generators use shared memory to hold context data. In order to pause and resume random number production, context data must be maintained safely. GASPRNG contains routines for this purpose. At the end of kernel execution, the context data is copied to off-chip storage; and when the next kernel execution starts, the context is restored.

The BUFFER\_MODE gives the address and the size of the host buffer to provide more flexibility. For example, when multiple CPU threads exist and each needs a small amount of random numbers, instead of starting multiple producing processes, these threads could share the same buffer. When the current buffer is empty, one thread notifies the controller and triggers random

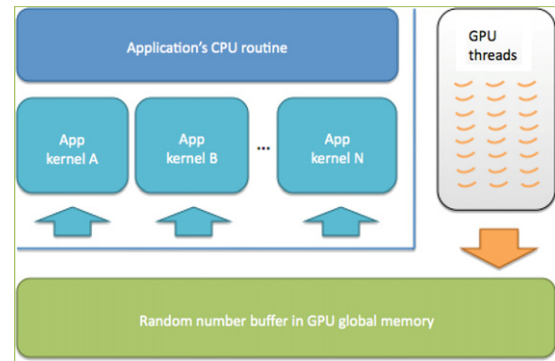


Fig. 3. Application model of GLOBAL\_GASPRNG.

number generation. An example of this behavior is included in `$(GASPRNG_PATH)/check/omp_host_buff_gasprng`.

## 2.2. GLOBAL\_GASPRNG

This interface provides a random number buffer in GPU global memory, so GPU kernels can share the buffer and consume the random numbers. The intent is for GPU codes to use GASPRNG to generate random numbers and place them in a buffer; other kernels will then consume them as needed. Fig. 3 shows the relation between GASPRNG and an application. The generator kernel is the producer and multiple application kernels can be consumers. As in HOST\_GASPRNG, context save/restore routines are used for suspending and resuming generator kernel execution. Between every two iterations of generator kernel execution, other kernels could occupy GPU resources. In contrast to CURAND and other commonly used GPU random number generators, GASPRNG takes over the management of random number buffers, simplifying the integration task for application developers.

Sometimes applications have multiple GPU kernels or launch the same kernel multiple times. When each kernel execution needs only a small amount of random numbers, since some generator types have complex state data and initialization process, it is a waste to maintain one generator for each GPU thread. By using GLOBAL\_GASPRNG and setting the producer–consumer relation in Fig. 3, applications can use these generators more efficiently. Quantum Monte Carlo simulation (QMC) is an example presented in Section 5.5.

## 2.3. DEVICE\_GASPRNG

As with the GLOBAL\_GASPRNG interface, the DEVICE\_GASPRNG interface is intended for GPU codes that will consume random numbers while executing on the GPU. In contrast to using a buffer in global memory, the intent for the DEVICE\_GASPRNG interface is to support kernels that are tightly coupled with GASPRNG; the threads of an application will generate numbers on demand while executing within the *same* CUDA kernel. SPRNG supports MPI programs; each process has a generator and the process rank is used to determine the seed. In the last few years, many Monte Carlo simulations were ported to CPU/GPU hybrid platforms. One typical behavior is to obtain a generator for each GPU thread. Currently most GPU random number tools provide this interface. GASPRNG also defines the DEVICE\_GASPRNG interface for this purpose.

One common concern of GPU programming is the limitation of on-chip memory resources. Other GPU random number generators make state data as small as possible to solve this problem. To provide high quality random number streams, generator algorithms in SPRNG maintain relatively complex state structures. Since GPU kernel execution normally starts thousands of threads

simultaneously, the on-chip storage is not enough for hosting the state data of all generators. Section 3 presents the main schemes to solve this problem for generator kernels in GASPRNG.

The main part of this interface is defined as a set of `__device__` functions; they are called by application `__global__` functions. On the one hand, since random numbers are produced and consumed inside one kernel execution, they could reside in on-chip storage and avoid extra global memory operation. On the other hand, as a consequence of the code integration, the execution of generator and application kernels are inter-dependent because they share on-chip resources and there is a constraint that both GPU routines must run with the same block/thread number. To help with application development, the package includes a calculator for the resource requirements of GASPRNG.

Both the `GLOBAL_GASPRNG` and `DEVICE_GASPRNG` interfaces provide random numbers for GPU kernels. However, `GLOBAL_GASPRNG` is more flexible than `DEVICE_GASPRNG`. On the other hand, `DEVICE_GASPRNG` fits well for GPU kernels that consume massive quantities of random numbers; we include Pi simulation as a basic example.

### 3. SPRNG generator equations and GPU kernel design

SPRNG includes six types of generators: Modified Lagged Fibonacci Generator (LFG), Linear Congruential Generator (LCG), 64-bit LCG (LCG64), Multiplicative Lagged Fibonacci Generator (MLFG), Combined Multiple Recursive Generator (CMRG), and Prime Modulus Linear Congruential Generator (PMLCG). In GASPRNG, each of these generator types is implemented as an efficient GPU kernel. Each GPU thread works as one generator. This section introduces the implementation and optimization of these kernels.

#### 3.1. LFG

The LFG (Lagged Fibonacci Generator) is a widely used generator; it has excellent random number statistics and is the default generator for SPRNG. The formula of LFG is:

$$Z(n) = Z(n - k) + Z(n - l)(\text{Mod } 2^{64}) \quad (1)$$

where  $k$  and  $l$  are the lags. It uses previously generated pseudorandom numbers with indices  $(n - k)$  and  $(n - l)$ . SPRNG support 11 different pair of lags; the period of the generator is  $2^{31(2^l - 1)}$ , and the number of concurrent streams is  $2^{31(l-1)-1}$  [3]. LFG uses a large data structure to save state data of one generator. This structure includes several arrays. The sizes of these arrays are determined by lag values. For the lag values  $\{1279, 861, 1, 233\}$ , the length of the state array is 1279. On a GPU, hundreds of generators can share one SM. If each generator has so much state data, the GPU on-chip storage is not large enough for this purpose. If the whole array of state data is stored in global memory, it causes unnecessary waste.

To solve this problem, the state data structure for one generator is divided into two parts according to the way it is used. For generators in one GPU thread block, the following data members are reorganized to be integrated into one data structure that serves for one thread block:

- The data are determined by a generator initialization step and remain constant during random number generation; its size is not large and it is shared among all generators. In such a case, shared memory is used as a constant buffer during random number generating period. One example is the base pointers of array members stored in global memory. These pointers are used for generating each random number.

- The data is frequently used and the size is small. An example is pointer walk through an array, it changes when generating each random number. In such cases the shared memory is used as a cache.

For array members and other large size data members of original state structure, global memory is used to hold one state data for each generator. In order to improve memory operation efficiency, a memory address mapping is set up to realize coalescing memory visiting. Although this mapping involves more operations, the performance improvement outweighs the extra work. Finally, the kernel code is optimized so that the number of global memory access is reduced as much as possible.

#### 3.2. LCG

The 48 bit LCG is based on:

$$Z(n) = a \times Z(n - 1) + p(\text{Mod } 2^{48}) \quad (2)$$

where  $a$  is a multiplier and  $p$  is a prime number. The period of this generator is  $2^{48}$ . The number of parallel streams is of the order of  $2^{19}$  [3]. Since the lag here is only 1, the space needed for saving the generator state is much less than that for the LFG. The new generator state structure resides in shared memory as described in Section 3.1, GASPRNG defines one state data structure in shared memory for all generators of one GPU thread block: it includes arrays of seeds, prime numbers, and some necessary constants. Compared with the LFG, the LCG speed is much faster because no global memory operations are needed for visiting the generator state data.

#### 3.3. LCG64

The LCG64 is based on the 48 bit LCG, but with a larger modulo value as below:

$$Z(n) = a \times Z(n - 1) + p(\text{Mod } 2^{64}). \quad (3)$$

The period of this generator is  $2^{64}$  and the number of parallel streams is over  $10^8$  [3]. The method of organizing state data in the GPU memory hierarchy is similar to the LCG.

#### 3.4. CMRG

The CMRG is based on:

$$\begin{aligned} X(n) &= a \times X(n - 1) + p(\text{Mod } 2^{64}) \\ Y(n) &= 107374182 \times Y(n - 1) + 104480 \\ &\quad \times Y(n - 5)(\text{mod } 2147483647) \end{aligned}$$

$$Z(n) = X(n) + Y(n) \times 2^{32}(\text{Mod } 2^{64}). \quad (4)$$

The period of this generator is around  $2^{219}$ ; and the number of distinct streams is over  $10^8$  [3]. The method of organizing state data in the GPU memory hierarchy is similar to the LFG.

#### 3.5. MLFG

The MLFG is based on:

$$Z(n) = Z(n - k) \times Z(n - l)(\text{Mod } 2^{64}) \quad (5)$$

where  $l$  is the lag. The period of this generator is  $2^{61}(2^l - 1)$ , and the distinct streams number is  $2^{63(l-1)-1}$  [3]. For the default lag value, this means around  $2^{1008}$  streams. The MLFG kernel implementation is similar to the LFG. The state has large arrays in the generator state structure; they are reorganized in the same way as with the LFG and stored using both global memory and shared memory.



### 3.6. PMLCG

The PMLCG is based on:

$$Z(n) = a \times Z(n-1) \pmod{2^{61} - 1}. \quad (6)$$

The period of this generator is  $2^{61} - 2$ ; and the number of streams is around  $2^{58}$  [3]. The method of organizing state data in the GPU memory hierarchy is similar to the LCG. In contrast to the other five generator types, the PMLCG requires two extra arrays to initialize the generators that are populated by using the GMP library. Currently, the CPU version of GMP is used. In practice, this can reduce the scalability of the PMLCG generator due to slow initialization.

The context data discussed in Section 2 are stored in shared memory. This data must be stored in global memory when the GPU is reclaimed for other purposes, and they can be restored to shared memory before continuing random number generation.

## 4. Programming interface

### 4.1. HOST\_GASPRNG

The HOST\_GASPRNG interface is similar to SPRNG. There are three main phases: initializing random number streams, generating random numbers, and releasing the streams. The initialization function takes configuration parameters and creates a group of generators on the GPU, returning the handle of one random number stream that is generated by the generator group (for function parameter details and examples, please refer to the user manual). Other functions take this handle as a parameter to operate on the stream. Beside this handle variable, all state information is hidden from caller routines. The following is the calling sequence for NUMBER\_MODE:

```
handle = init_gasprng (generator_type, streamIdx, rn_type, seed, param);
```

```
for i = 0 to rn_num do
    rn = dgasprng (handle); //igasprng returns integer, and fgasprng
    returns float
end for
```

```
destroy_gasprng (handle);
```

As mentioned above, CPU routines may require a buffer of random numbers; the buffer address is returned by `gasprng_buff()`. The pointer type returned is `void *`, before using it, a type conversion is necessary. The following is the typical usage for the BUFFER\_MODE:

```
handle = init_gasprng_buff (generator_type, streamIdx, rn_type, seed, param, buff_size)
```

```
ptr = (Type *)gasprng_buff (handle);
```

```
//use random numbers in the buffer pointed by ptr
```

```
destroy_gasprng_buff (handle);
```

### 4.2. GLOBAL\_GASPRNG

The steps of using this interface are similar to the previous ones: initialize random streams, obtain random number buffer information, and destroy random number streams. The buffer size is required to initialize the random number stream; the type of the pointer returned by `gasprng_global()` is `void *` as well. (For function parameter details and examples, please refer to the user manual.)

```
handle = init_gasprng_global (... , buff_size...);
```

```
rn_buff = (Type *)gasprng_global (handle);
```

```
//use random number in rn_buff
```

```
destroy_gasprng_global (handle);
```

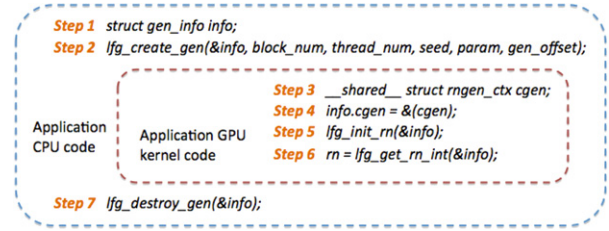


Fig. 4. Seven steps to use DEVICE\_GASPRNG for an example LFG generator.

### 4.3. DEVICE\_GASPRNG

Fig. 4 shows the 7 steps to use the DEVICE\_GASPRNG interface. Steps 1, 2 and 7 happen in CPU host code, and steps 3, 4, 5, and 6 occur in GPU kernel code. The first step defines an empty `struct_info` object, which is used to store the information necessary for creating random number generators. Step 2 fills the empty object according to the given parameters. Step 3 defines a context object, which resides in shared memory. Step 4 saves a pointer to the context object in the `struct_info`. Step 5 initializes the generator for the current GPU thread. Step 6 gets random numbers in sequence. Finally, step 7 deletes the generators and releases the related resources.

## 5. Experiments and results

The following platforms are used to evaluate the performance of GASPRNG:

Single machine with GPU:

- (1) Intel i7, 920, 2.67 GHz, 11 GB; NVIDIA Fermi GTX480; GNU/Linux, CUDA 4.0
- (2) Intel Xeon, X5570, 2.93 GHz, 23 GB; NVIDIA Tesla C1060, GNU/Linux, CUDA 4.0

CPU/GPU hybrid cluster:

- (3) Keeneland System [27]. Each node: 2 x 6 Intel Xeon X5660, 2.8 GHz; 23 GB; NVIDIA M2070; GNU/Linux, CUDA 4.0.

The performance test covers: (1) six generator types; (2) three level interfaces; and (3) three random number data types. The results are compared with SPRNG 2.0 [2,3] and CURAND from the CUDA 4.0 toolkit (CURAND's default uniform generator XORWOW is used.). The efficiency of the two-level pipeline is analyzed; actual timing results are compared. Finally, two real applications using GASPRNG are presented.

### 5.1. DEVICE\_GASPRNG

Tables 1–3 show the performance of DEVICE\_GASPRNG, the result is compared with SPRNG performance on one core. The performance of SPRNG is obtained through a timing tool in the SPRNG 2.0 package. Compared with SPRNG 2.0, GASPRNG provides  $5x \sim 97x$  speedup for integer,  $11x \sim 283x$  for single precision, and  $11x \sim 271x$  for double precision. Lee implements HASPRNG, which is a FPGA implementation of SPRNG integer generators [28]. Compared with its performance data on the Xilinx XC2VP50 FPGA [28], GASPRNG shows  $11x \sim 235x$  speedup. Fig. 5 shows the performance compared with CURAND.

### 5.2. GLOBAL\_GASPRNG performance

Fig. 6 shows the performance of GLOBAL\_GASPRNG. The result is compared with SPRNG on one core. Compared with DEVICE\_GASPRNG, the performance loss ranges from 30% to 80% due to two reasons: (1) each kernel has one more global

**Table 1**

Performance of DEVICE\_GASPRNG on integer random number generation (Millions of Random Numbers Generated per Second – MRS) and speedup compared to Xeon x5570.

	LFG		LCG		LCG64		CMRG		MLFG		PMLCG	
	MRS	Speedup	MRS	Speedup	MRS	Speedup	MRS	Speedup	MRS	Speedup	MRS	Speedup
Fermi GTX480	2231	17.6	22 694	63.6	37 713	98.0	4592	20.2	8898	34.8	8257	59.4
Tesla M2070	1585	12.5	18 714	52.4	30 344	78.8	3743	16.5	7313	28.6	6789	48.8
Tesla C1060	1614	12.7	6 866	19.2	9 336	24.3	1157	5.1	3521	13.8	2357	17.0
Xeon x5570	127	–	357	–	385	–	227	–	256	–	139	–

**Table 2**

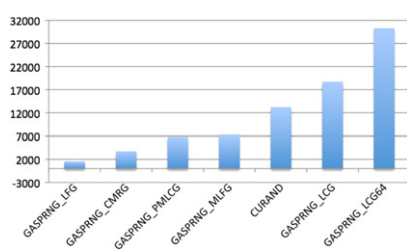
Performance of DEVICE\_GASPRNG on single precision random number generation (Millions of Random Numbers Generated per Second – MRS) and Speedup compared to Xeon.

	LFG		LCG		LCG64		CMRG		MLFG		PMLCG	
	MRS	Speedup	MRS	Speedup	MRS	Speedup	MRS	Speedup	MRS	Speedup	MRS	Speedup
Fermi GTX480	2232	17.4	22 874	89.4	37 720	283.6	4593	46.9	8898	39.2	8257	67.1
Tesla M2070	1585	12.4	18 716	73.1	30 844	231.9	3743	38.2	7313	32.2	6789	55.2
Tesla C1060	1615	12.6	6 866	26.8	9 336	70.2	1157	11.8	3521	15.5	2357	19.2
Xeon x5570	128	–	256	–	133	–	98	–	227	–	123	–

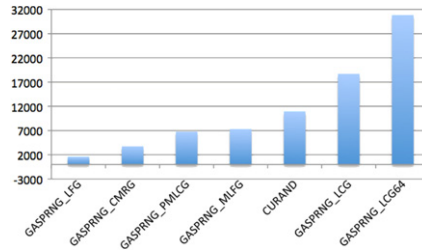
**Table 3**

Performance of DEVICE\_GASPRNG on double precision random number generation (Millions of Random Numbers Generated per Second – MRS) and Speedup compared to Xeon.

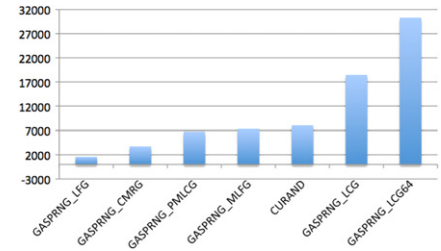
	LFG		LCG		LCG64		CMRG		MLFG		PMLCG	
	MRS	Speedup	MRS	Speedup	MRS	Speedup	MRS	Speedup	MRS	Speedup	MRS	Speedup
Fermi GTX480	2232	19.9	22 834	68.6	37 716	271.3	4596	45.5	8898	34.8	8258	50.8
Tesla M2070	1585	14.2	18 468	55.5	30 332	218.2	3743	37.1	7313	28.6	6789	51.1
Tesla C1060	1614	14.4	6 865	20.6	9 336	67.2	1157	11.5	3521	13.8	2357	17.7
Xeon x5570	112	–	333	–	139	–	101	–	256	–	133	–



(a) Integer.



(b) Single precision.



(c) Double precision.

**Fig. 5.** Comparing DEVICE\_GASPRNG performance with CURAND (Millions of Random Numbers Generated per Second – MRS).

memory write operation; (2) interface and memory management causes extra overhead. Compared with SPRNG, GLOBAL\_GASPRNG provides 4x ~ 43x speedup for integer, 9x ~ 116x for single precision, and 7x ~ 72x for double precision. As discussed above, this interface provides flexible support for applications with diverse random number use requirements.

### 5.3. HOST\_GASPRNG performance

Fig. 7 shows the performance of HOST\_GASPRNG in NUMBER\_MODE. The caller routine acquires random numbers one by one through interface functions. This figure also includes the performance of SPRNG running on one core. The result shows that GASPRNG outperforms SPRNG: the speedup is 1.2x ~ 3.7x for integer, 1.8x ~ 5x for single precision, and 1.04x ~ 4.6x for double precision.

Fig. 8 shows the performance of the BUFFER\_MODE. Compared with NUMBER\_MODE, it avoids the overhead of iterative memory reading. The speedup over SPRNG is 3.5x ~ 10x for integer, 5x ~ 12x for single precision, and 2x ~ 7x for double precision. This figure also includes the performance of CURAND. For most cases, GASPRNG presents comparable performance with CURAND.

### 5.4. HOST\_GASPRNG pipeline analysis

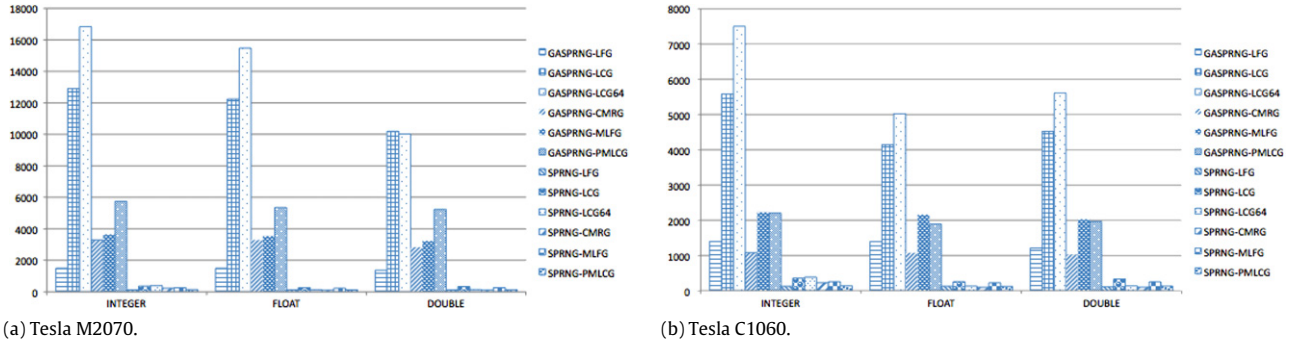
To test the efficiency of the two-level pipeline in HOST\_GASPRNG, we measure the dominant parts in Fig. 2 and estimate the execution runtime. The runtime model is:

$$\text{Produce}_t = \text{MAX}(\text{Kernel}_t, \text{DtoH}_t) + \text{MIN}(\text{Kernel}_t, \text{DtoH}_t) / \text{Stream\_num}$$

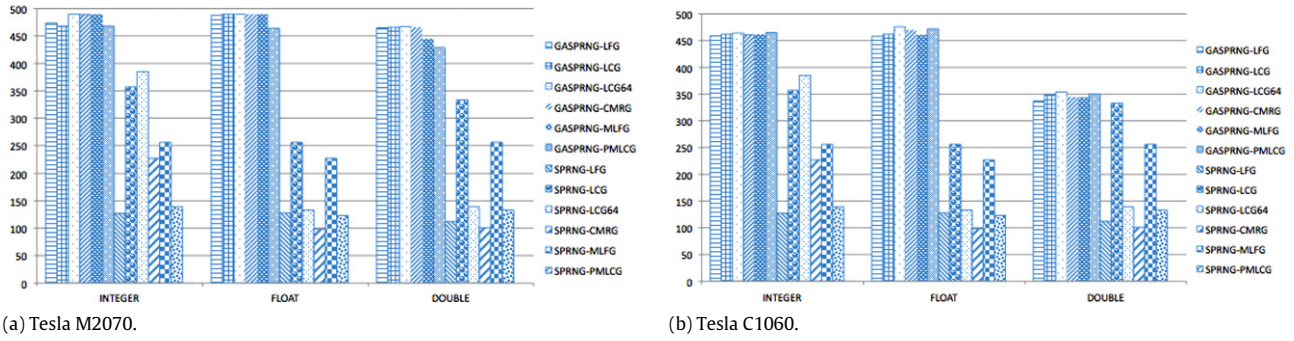
$$\text{Execution}_t = \text{MAX}(\text{Produce}_t, \text{Consume}_t) \times \text{Buff\_number} + \text{MIN}(\text{Produce}_t, \text{Consume}_t)$$

$$\text{MRNs} = \text{rn\_num} / \text{Execution}_t$$

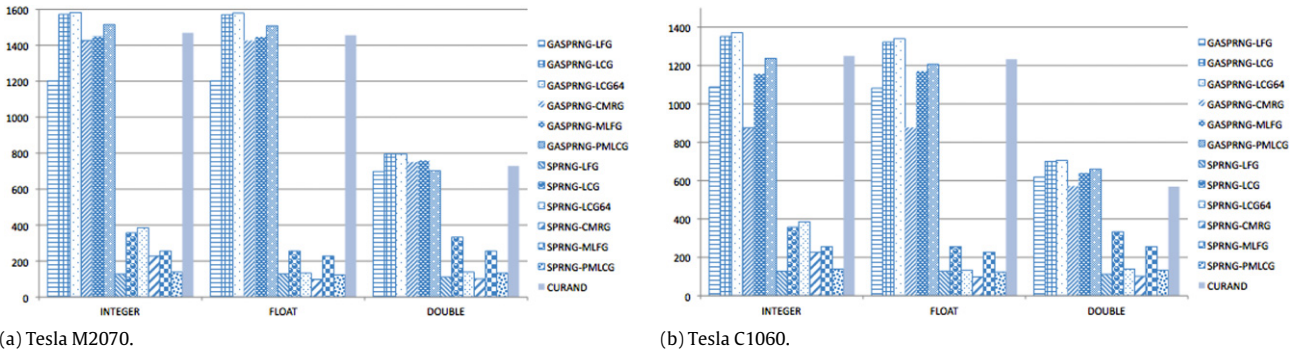
$\text{Produce}_t$  is the time for filling one of the two host buffers. This task is divided and assigned to several streams, the  $\text{Kernel}_t$  and  $\text{DtoH}_t$  are the kernel execution time and DtoH data transfer time for each stream. We use the runtime of GLOBAL\_GASPRNG to simulate the  $\text{Kernel}_t$ , this interface generates random numbers and writes them to a global memory buffer. The  $\text{DtoH}_t$  is calculated using the PCIe bus bandwidth, which is around 6 GB/s.  $\text{Consume}_t$  is the time of consuming random numbers in host memory. When measuring this value, the interactions between controller and internal GPU generator engine are disabled; the controller assumes each random number is ready and reads the same host buffer iteratively.



**Fig. 6.** Performance of GLOBAL\_GASPRNG interface (Millions of Random Numbers Generated per Second – MRS).



**Fig. 7.** Performance of HOST\_GASPRNG interface in NUMBER\_MODE (Millions of Random Numbers Generated per Second – MRS).



**Fig. 8.** Performance of HOST\_GASPRNG interface in BUFFER\_MODE (Millions of Random Numbers Generated per Second – MRS).

This test generates  $57600000 \times 10$  integer (or double precision) random numbers; the size of the host buffers is large enough to avoid caching. Four CUDA streams are used for testing with one node of the Keeneland system [27].

Fig. 9 compares the estimated result and actual timing result in MRS (Millions of Random Numbers Generated Per Second). The accuracy is above 95% for both double precision numbers and integer numbers (the results of single precision and integer are similar). Fig. 10 compares the timing of three dominant parts used for runtime estimation. It shows that *Consume\_t* is the bottleneck. This is caused by overhead of the controller; it manages a hidden data structure and maintains the valid location information.

## 5.5. Hybrid cases

### 5.5.1. Pi simulation with MPI+OMP+CUDA

This is a simple Monte Carlo integration. On a hybrid platform, the whole task is distributed across multi-core and GPU resources. One random number generator period is divided and shared among all CPU/GPU threads; each of them finishes one part of the

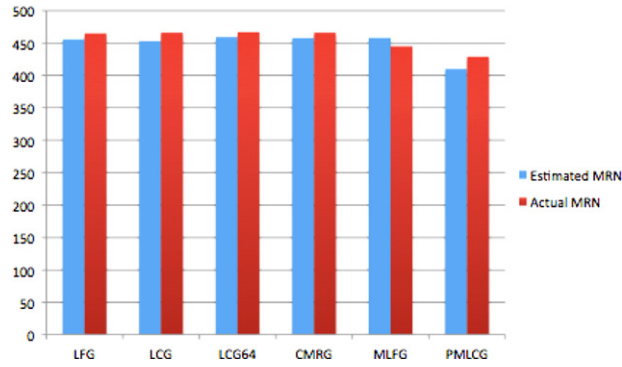
integration. The final result is calculated by a two-level reduction operation. This example does not include the scheduling between multi-core and GPU, users can further make use of a static/dynamic scheduler to improve its performance. The source code of this example can be found in `/gasprng1.0/example/Pi/hybrid/`.

Another three Pi examples running on multiple nodes can be found in `/gasprng1.0/example/Pi/MPI`. They use HOST\_GASPRNG, GLOBAL\_GASPRNG, and DEVICE\_GASPRNG, respectively.

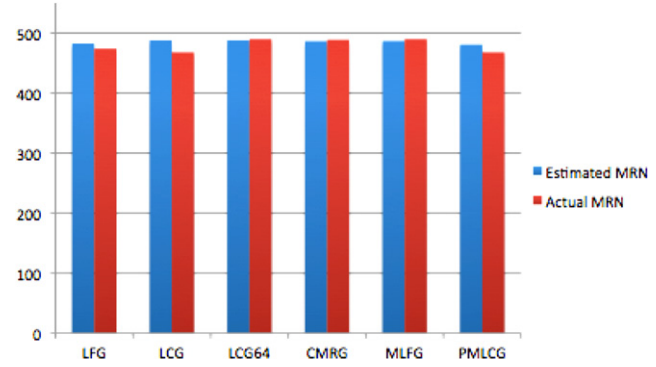
### 5.5.2. QMC simulation

The second example is a Quantum Monte Carlo simulation (QMC) [29,30]. In chemistry and physics, this method is commonly used to find state properties. In the simulation, multiple walkers are defined and each of them evaluates a single point in the function space. The main nested loop is shown in Fig. 11. In the inner loop, the coordinates of particles are randomly perturbed, and the state of current state is computed. This result is evaluated to determine whether current situation should be accepted or not. This procedure continues until the system reaches equilibrium.



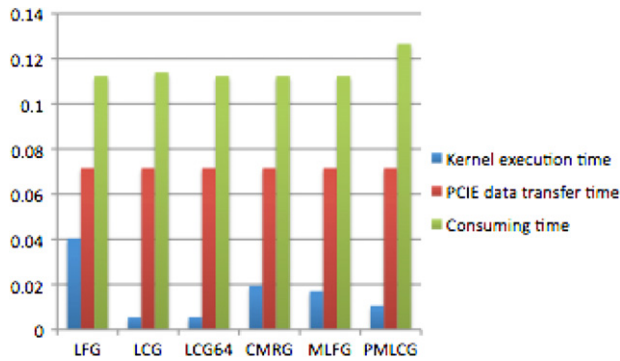


(a) Double precision.



(b) Integer.

**Fig. 9.** Estimated MRS and Actual MRS for HOST\_GASPRNG LFG generator (Millions of Random Numbers Generated per Second — MRS).



**Fig. 10.** *Kernel\_t/DtoH\_t/Consume\_t*: generating/transferring/consuming 57600000 double precision numbers.

Both steps of perturbing particles and making accept/reject decisions need random numbers. The first one is data parallel; it could be implemented as a GPU kernel. The second one is short and simple, so it remains on the CPU. Thus, in this code, we need both CPU random number generators and GPU random number generators.

The GPU kernels must be launched multiple times; in each kernel execution, only three random numbers are needed to perturb one particle. As introduced, the GLOBAL\_GASPRNG interface fits well for this type of application. When making accept/reject decisions, SPRNG is used as the CPU generator.

## 6. Installation instructions

We briefly describe here the process for installing GASPRNG. For additional information, the reader is encouraged to check the GASPRNG user guide. To install GASPRNG, the following software and hardware are required.

- Software: GMP, SPRNG, CUDA toolkits 4.0
- Hardware: Nvidia GPU above sm\_13 (either Tesla or Fermi GPUs).

(Use `$(CUDA_SDK_PATH)/C/bin/linux/release/deviceQuery` to check the number of installed device)

The main install steps are as following:

- Download GASPRNG from the website (<http://TBD>).
- Download and install *gmp-5.0.2.tar.gz*, and *sprng2.0b.tar.gz*
- Uncompress the GASPRNG package: `tar zxvf gasprng1.0.tar.gz`
- `./configure`:
  - Option “`–prefix=PATH`”: use it to configure the install path.
  - Option “`–enable-sm13`”: if the GPU device’s capability is *sm\_13*, set “`–enable-sm13=yes`”. By default *sm\_20* is set as GPU device capability level.
  - Option “`–with-cuda=PATH`”: if CUDA toolkit is not installed under default path of */usr/local/cuda*, use this option to set the CUDA path.
  - Option “`–with-sprng=PATH`”: if SPRNG is not installed under default path of */usr/local*, use this path to set the installing location of SPRNG.
  - Option “`–with-gmp=PATH`”: if GMP is not installed under default path of */usr/local*, use this path to set the installing location of GMP.
- Use “`make`” command to build the library.
- User “`make install`” to install the *libgasprng.a* and header files.
- User “`make check`” to build examples and binaries for correctness/binary check.
- Execute `./bin/check_corr.sh` for a basic correctness test.

## 7. Test run description

1. To verify the correctness of GASPRNG implementation, please run the script:

```

for i = 1 to walker_number do
  Initialize configuration
  for j = 1 to iteration_number do
    1. Perturb particle positions: RNs generated by GLOBAL_GASPRNG
    2. Calculate state properties
    3. Determine to accept/reject current configuration: RN generated by SPRNG
  End for
End for

```

**Fig. 11.** Basic QMC algorithm.



- `$(GASPRNG_PATH)/bin/check_corr.sh` (By default, this script checks three level interfaces. User can use option “`-help`” to list the options accepted by this script. Basically, user can use these options to choose which level interface to check)

2. To measure the performance of three level interfaces please run the script:

- `$(GASPRNG_PATH)/bin/check_perf.sh`

3. To run MPI correctness test:

- Set the paths of MPI when executing. `/configure`:
  - Option “`-with-mpi`”: if MPI is not installed under default path of `/usr/local`, use this option to set the MPI library path.
- `cd $(GASPRNG_PATH)`
- `make mpi`
- Make sure you have at least 2 processors available.
- `$(GASPRNG_PATH)/bin/check_corr_mpi.sh` (By default, this script checks three level interfaces. User can use option “`-help`” to list the options accepted by this script. Basically, user can use these options to choose which level interface to check.)

## 8. Conclusions and future work

GASPRNG is a GPU accelerated implementation of the SPRNG library. It includes CUDA kernels for six generator types and three types of interfaces to support various usage models, so programmers can choose the most efficient way to use generators in GASPRNG. For each generator, GASPRNG outperforms SPRNG running on one CPU core. With its double-buffering scheme and two-level pipeline structure, the GASPRNG host interface provides good performance with a SPRNG-like interface. Additional GASPNG interfaces support codes that consume random numbers in GPU kernels. The future work includes supporting additional commonly used probability distributions into GASPRNG (e.g., normal, exponential, Rayleigh). After that, an OpenCL version will be developed and architecture-related performance portability will be considered. Currently, GASPRNG users maintain the seeds across all CPU/GPU processes/threads. In the future, a seed management module will be added, makes it easier to share one random number period across different devices.

## Acknowledgments

The authors would like to thank Junkyu Lee, David Jenkins, Rick Weber, R.J. Hinde, and Robert Harrison for suggestions. We thank the University of Tennessee/Oak Ridge National Lab's Joint Institute for Computational Science Application Acceleration Center of

Excellence for access to the Keeneland system for testing [27]. This work was partially supported by NSF grants DGE-0801540 and CHE-0625598.

## References

- [1] P. L'Ecuyer, R. Simard, *ACM Transactions on Mathematical Software* 33 (2007).
- [2] M. Mascagni, D. Ceperley, A. Srinivasan, *ACM Transactions on Mathematical Software* 26 (2000) 436.
- [3] Scalable Parallel Random Number Generators Library SPRNG User Guide. <http://sprng.fsu.edu/Version2.0/generators.html>.
- [4] General Purpose Computation on Graphics Hardware. <http://gpgpu.org>.
- [5] Wen-mei Hwu (Ed.), *GPU Computing Gems*, Morgan-Kaufman Publishers, 2011.
- [6] NVIDIA, <http://www.nvidia.com>.
- [7] D.B. Kirk, W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan-Kaufman Publishers, 2010.
- [8] Khronos Group, <http://www.khronos.org/opencl/>.
- [9] Khronos Group, *The OpenCL Specification*, Version 1.1, June 2011.
- [10] B. Block, P. Virnau, T. Preis, *Computer Physics Communications* 181 (2010).
- [11] J.S. Meredith, G. Alvarez, T.A. Maier, T.C. Schulthess, J.S. Vetter, *Parallel Computing* 35 (2009).
- [12] C. Gong, J. Liu, L. Chi, Q. Hu, L. Deng, Z. Gong, *Accelerating Pseudo-Random Number Generator for MCNP on GPU*, in: *ASIP Conference Proceedings*, 2010, p. 1281.
- [13] Parallel Mersenne Twister, Nvidia CUDA SDK. <http://developer.nvidia.com/cuda-toolkit-40>.
- [14] CUDA CURAND Library, Nvidia CUDA Toolkits. <http://developer.nvidia.com/cuda-toolkit-40>.
- [15] L. Howes, D. Thomas, *GPU Gems3*, Addison-Wesley, 2007.
- [16] W.B. Langdon, *2008 IEEE World Congress on Computational Intelligence*, 2008, pp. 459.
- [17] V. Demchik, *Computer Physics Communications* 182 (2011).
- [18] X. Tian, K. Benkrid, *Proceedings of NASA/ESA Conference on Adaptive Hardware and Systems*, 2009.
- [19] W.B. Langdon, *Proceedings of Genetic and Evolutionary Computation Conference*, Montreal, Canada, 2009, pp. 2511.
- [20] F. Zafar, M. Olano, A. Curtis, *Proceedings of the Conference on High Performance Graphics*, 2010.
- [21] N. Nandapalan, R.P. Brent, L.M. Murray, A. Rendell, <http://arxiv.org/abs/1108.0486>.
- [22] C.L. Phillipsa, J.A. Andersonb, S.C. Glotzer, *Journal of Computational Physics* 230 (2011) 7191.
- [23] J.J.M. Chan, B. Sharma, J. Lv, G. Thomas, R. Thulasiram, P. Thulasiraman, *IEEE 13th International Conference on High Performance Computing and Communications*, HPCC, 2011, pp. 161.
- [24] J. Passerat-Palmbach, C. Mazel, D.R.C. Hill, *Principles of Advanced and Distributed Simulation*, PADS, 2011, p. 1.
- [25] S. Hissoiny, B. Ozell, *Radiation Therapy Physics* 38 (2011).
- [26] John K. Salmon, Mark A. Moraes, Ron O. Dror, David E. Shaw, *High Performance Computing, Networking, Storage and Analysis (SC)* (2011).
- [27] J.S. Vetter, R. Glassbrook, J. Dongarra, K. Schwan, B. Loftis, S. McNally, J. Meredith, J. Rogers, P. Roth, K. Spafford, S. Yalamanchili, *IEEE Computing in Science and Engineering* 13 (2011).
- [28] J. Lee, Y. Bi, G.D. Peterson, R.J. Hinde, R.J. Harrison, *Computer Physics Communications* 180 (12) (2009) 2574.
- [29] Rick Weber, Akila Gothandaraman, Robert J. Hinde, Gregory D. Peterson, *IEEE Transactions on Parallel and Distributed Systems* 22 (2011) 58.
- [30] Akila Gothandaraman\*, Gregory D. Peterson, G. Lee Warren, Robert J. Hinde, Robert J. Harrison, *Computer Physics Communications* 180 (2009) 2563.