

# Technical Report about implementation in CUDA of Monte Carlo Linear Solvers (MCLS)

Steven Hamilton, Massimiliano Lupo Pasini

## Abstract

In this report timings are presented as concerns Monte Carlo linear Solvers (MCLS) with different configurations. Different test cases are used to get quite a general overview of the behavior. In particular various options to generate random numbers as well as use of texture memory are tested for the sake of efficiency.

## 1 Different random number generation techniques

The generation of random numbers plays an important role in the calculation of the solution to a linear system through a Monte Carlo procedure, since a new random number must be generated to either step ahead in a random walk or to kink off a new one. Since this operation must be repeated very frequently in the code, it is recommendable to decrease its cost as much as possible, not to affect the efficiency of the overall performance. Two viable options to initialize random number generators are:

1. Having the same seed but different sequence number generates a number guaranteed to be  $2^{67}$  away from each other, but the downside is the heavy computations to advance the  $2^{67}$  position

```
__global__ void initialize_rng(curandState *state, int seed, int offset)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    curand_init(seed, tid, offset, &state[tid]);
}
\caption{Use of different seeds with the same sequence.}
```

2. Giving different seeds, and just keep the sequence number at 0, it's a lot faster but there might be correlation between threads, since there is no guarantee on the separation between each threads

```
__global__ void initialize_rng2(curandState *state, int*seed, int offset)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
```

```

    curand_init(seed[tid], 0, offset, &state[tid]);
}

```

Matrix	Size	Nb. Histories	% time rng	% time kernel	tot. time (ms)	rel. residual norm
2D Laplacian	900	$10^7$	36.0	64.0	299,563	0.0931241
1D Laplacian	$10^6$	$10^4$	0.05	99.5	50,016	12.3509
1D Laplacian	$10^6$	$10^7$	18.3	81.7	640,563	0.391686
SP <sub>1</sub>	25,568	$10^8$	18.0	82.0	31,542	0.943897

Table 1: Timings using same seed and different sequences.

Matrix	Size	Nb. Histories	% time rng	% time kernel	tot. time (ms)	rel. residual norm
2D Laplacian	900	$10^7$	0.0	100.0	191,796	0.119588
1D Laplacian	$10^6$	$10^4$	0.0	100.0	50,094	12.3914
1D Laplacian	$10^6$	$10^7$	0.0	100.0	533,451	0.408134
SP <sub>1</sub>	25,568	$10^8$	0.0	100.0	26,367	1.06455

Table 2: Timings using different seeds and same sequence.

As it can be noticed from the previous tables 1 and 2, the second option provides a better timing. However the payoff is an increase of the final relative residual, likely due to the correlation between the sequences generated by different seeds. The quantities "time rng" and "time kernel" are expressed in terms of percentage of the total time.

## 2 Use of texture memory

The LDG instruction (exposed via the `__ldg` intrinsic) is a memory load that uses the texture path. It has the advantage that it does not require the explicit use of textures, since it does not explicitly bind one. Therefore `__ldg()` reads data through the texture path, without requiring a texture itself. It is an overloaded function with the prototype `__ldg(const *T)` where T is one of CUDA's built-in types. The perk of using LDG instruction is that explicit uses of textures causes a certain amount of code clutter and overhead (e.g. for API calls to bind textures). Classical textures also use the texture load path, but in addition can transform both index (e.g. clamping modes) and data returned (e.g. interpolation) in various ways; the necessary control information is provided to the hardware during texture binding. Because the texture cache is non-coherent with respect to writes in the same kernel, use of the texture load path requires that the underlying data is read-only across the entire kernel.

## 3 LDG calls into the code

LDG instructions have been introduced in the following functions of Profugus code:

1. `lower_bound`
2. `initialize_history`
3. `getNewState`

#### 4. tallyContribution

All the instructions employed above provide a runtime decrease. However, some of them are more significant than the others. In particular the most effective calls are the ones located in 1) and 3). This is due to the fact that these functions are the ones called most frequently. The option of using LDG instruction at point 4) actually plays a role just when the expected value estimator is employed.

In the examples represented below the length of the history is set to 10000 steps, with a weight cut off equal to  $10^{-9}$ .

Matrix	Size	Nb. Histories	$\rho(H)$	$\rho(\hat{H})$	tot. time (ms)
2D Laplacian	900	$10^7$	0.994869	0.99447	27,686
1D Laplacian	$10^6$	$10^4$	0.799972	0.639977	35,108
1D Laplacian	$10^6$	$10^7$	0.799972	0.639977	40,720
SP <sub>1</sub>	25,568	$10^8$	0.977674	0.999836	45,562

Table 3: Timings without LDG instructions.

Matrix	Size	Nb. Histories	$\rho(H)$	$\rho(\hat{H})$	tot. time (ms)
2D Laplacian	900	$10^7$	0.994869	0.99447	33,791
1D Laplacian	$10^6$	$10^4$	0.799972	0.639977	35,911
1D Laplacian	$10^6$	$10^7$	0.799972	0.639977	41,092
SP <sub>1</sub>	25,568	$10^8$	0.977674	0.999836	38,306

Table 4: Timings with LDG instructions at 1).

Matrix	Size	Nb. Histories	$\rho(H)$	$\rho(\hat{H})$	tot. time (ms)
2D Laplacian	900	$10^7$	0.994869	0.99447	27,722
1D Laplacian	$10^6$	$10^4$	0.799972	0.639977	35,226
1D Laplacian	$10^6$	$10^7$	0.799972	0.639977	40,001
SP <sub>1</sub>	25,568	$10^8$	0.977674	0.999836	45,500

Table 5: Timings with LDG instructions at 2).

Matrix	Size	Nb. Histories	$\rho(H)$	$\rho(\hat{H})$	tot. time (ms)
2D Laplacian	900	$10^7$	0.994869	0.99447	25,559
1D Laplacian	$10^6$	$10^4$	0.799972	0.639977	35,582
1D Laplacian	$10^6$	$10^7$	0.799972	0.639977	40,305
SP <sub>1</sub>	25,568	$10^8$	0.977674	0.999836	40,377

Table 6: Timings with LDG instructions at 3).

By comparing the values of tables 1-9 it is pointed out that a benefit in terms of timings is achieved overall when the `ldg` instruction is employed in all the subroutines. Moreover the improvements are more evident for the 2D laplacian and the SP<sub>1</sub> matrix. The reason of this might be associated with the spectral radius. Indeed a higher value of the spectral radius induces the

Matrix	Size	Nb. Histories	$\rho(H)$	$\rho(\hat{H})$	tot. time (ms)
2D Laplacian	900	$10^7$	0.994869	0.99447	24,679
1D Laplacian	$10^6$	$10^4$	0.799972	0.639977	35,741
1D Laplacian	$10^6$	$10^7$	0.799972	0.639977	40,017
SP <sub>1</sub>	25,568	$10^8$	0.977674	0.999836	38,321

Table 7: Timings with LDG instructions at 1) and 2).

Matrix	Size	Nb. Histories	$\rho(H)$	$\rho(\hat{H})$	tot. time (ms)
2D Laplacian	900	$10^7$	0.994869	0.99447	25,614
1D Laplacian	$10^6$	$10^4$	0.799972	0.639977	36,141
1D Laplacian	$10^6$	$10^7$	0.799972	0.639977	40,631
SP <sub>1</sub>	25,568	$10^8$	0.977674	0.999836	40,406

Table 8: Timings with LDG instructions at 2) and 3).

Matrix	Size	Nb. Histories	$\rho(H)$	$\rho(\hat{H})$	tot. time (ms)
2D Laplacian	900	$10^7$	0.994869	0.99447	22,607
1D Laplacian	$10^6$	$10^4$	0.799972	0.639977	34,616
1D Laplacian	$10^6$	$10^7$	0.799972	0.639977	40,064
SP <sub>1</sub>	25,568	$10^8$	0.977674	0.999836	37,478

Table 9: Timings with LDG instructions at 1), 2) and 3).

Matrix	Size	Nb. Histories	$\rho(H)$	$\rho(\hat{H})$	tot. time (ms)
2D Laplacian	900	$10^7$	0.994869	0.99447	191,818
1D Laplacian	$10^6$	$10^4$	0.799972	0.639977	35,351
1D Laplacian	$10^6$	$10^7$	0.799972	0.639977	517,788
SP <sub>1</sub>	25,568	$10^6$	0.977674	0.999836	25,285

Table 10: Timings without LDG instructions and constant history length equal to 10,000.

Matrix	Size	Nb. Histories	$\rho(H)$	$\rho(\hat{H})$	tot. time (ms)
2D Laplacian	900	$10^7$	0.994869	0.99447	103,018
1D Laplacian	$10^6$	$10^4$	0.799972	0.639977	35,381
1D Laplacian	$10^6$	$10^7$	0.799972	0.639977	509,643
SP <sub>1</sub>	25,568	$10^6$	0.977674	0.999836	13,248

Table 11: Table 11. Timings with LDG instructions at 1), 2), 3) and 4). Constant history length equal to 10,000.

histories to run for longer to satisfy the prescribed tolerance on the weight cutoff. Since the histories keep running for more steps in these cases, the utility of the texture memory might increase as well. This possible explanation is validated by the results shown in the tables 8 and 9, where the length of the histories is fixed. For the 2D laplacian and the  $SP_1$  matrix the employment of `ldg` instructions almost halves the time for the computation. The time reduction gets weaker for the other test cases.

## 4 Generation of random numbers

Random numbers must be employed for each random walk to:

- determine what is the initial state
- determine the following state in the path accordingly to the current one

Essentially the generation of random numbers is located in the subroutines:

1. `initializeHistory`
2. `getNewState`

These two operations have to be repeated for all the random walks employed in the computations. In order to minimize the time spent in the generation of random numbers, it might be useful to gather the generation of many of these at the same time. Therefore a gathering of the random number generator's calls has been accomplished, partially modifying the subroutines "`initializeHistory`" and "`getNewState`". The relationship one-to-one between the generation of a random number and a call to one of these two subroutines is broken. A group of random numbers is generated in advance before the use of the aforementioned functions. The size of the batch of this grouped random numbers is a tuning parameter that can be set up to find the optimal configuration, reducing the overall time of execution.

The viable options that might be adopted are:

1. to call separately the random number generator once for the `initializeHistory` and then employ the batch for the successive steps
2. to start employing the batch even for the initial step of the history

Matrix	Size	Nb. Histories	% time rng	% time kernel	tot. time (ms)	rel. residual norm
2D Laplacian	900	$10^7$	0.0	100.0	191,796	0.119588
1D Laplacian	$10^6$	$10^4$	0.0	100.0	50,094	12.3914
1D Laplacian	$10^6$	$10^7$	0.0	100.0	533,451	0.408134
SP <sub>1</sub>	25,568	$10^8$	0.0	100.0	26,367	1.06455

Table 12: Results for separate calls of rng for `initializeHistory` and `getNewState`.

Matrix	Size	Nb. Histories	% time rng	% time kernel	tot. time (ms)	rel. residual norm
2D Laplacian	900	$10^7$	0.0	100.0	192,325	0.119588
1D Laplacian	$10^6$	$10^4$	0.0	100.0	36,602	12.3914
1D Laplacian	$10^6$	$10^7$	0.0	100.0	517,536	0.408134
SP <sub>1</sub>	25,568	$10^8$	0.0	100.0	25,483	1.06455

Table 13: Results for a single call of rng for both `initializeHistory` and `getNewState`.

Overall, for a configuration where the length of the random walk is fixed, the second option seems to be slightly more efficient.