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 handling of data storage are tested for the sake of efficiency. All the test cases have been run three times to filter the fluctuations due to hardware issues.

Introduction

In this report different approaches are tested in order to maximize the performance of Monte Carlo linear solvers algorithms in GPU environment. Both Forward and Adjoint method are analyzed, pointing out the different implementation issues and pinpointing the different viable options to improve timings. Four different numerical test cases are employed with different properties in terms of spectral radius of the iteration matrix, sparsity pattern and degrees of freedom. The study is accomplished both by using a weight cutoff to kill the histories and by using a fixed number of steps for each of those, in order to discriminate the actual effectiveness of the different modifications introduced. An attempt to justify the improvements sometimes detected is provided, however a deeper understanding of the behaviors still has to be carried out.

1 Adjoint method - 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]);

}
```

2. Giving different seeds, and just keeping the sequence number at 0, it is a lot faster but there might be correlation between threads, since there is no guarantee on the separation between each threads. In this case the different seeds can be picked accordingly to different criteria. In this report we explore two approaches:

   - picking the seeds with consecutive indices
   - picking the seeds with indices provided by a generator of integer random numbers

   In case seeds with consecutive indices are chosen the set of instructions employed to generate the seed vector is the following:

```
else if ( d_seed_type==SEED_TYPE::DIFF )
{
    std::cout<<"Different adjacent seeds instantiated"<<std::endl;

    thrust::device_vector<int> seeds( BLOCK_SIZE*num_blocks);
    thrust::sequence(seeds.begin(), seeds.end(), d_rng_seed);
    int* seed_ptr = thrust::raw_pointer_cast(seeds.data());

    initialize_rng<<<num_blocks, BLOCK_SIZE>>>(rng_states, seed_ptr,
        d_num_curand_calls, d_seed_type);
}
```

   Instead in the case of seeds chosen with random indices the set of instructions executed is:

```
else if ( d_seed_type==SEED_TYPE::RAND )
{
    std::cout<<"Different random seeds instantiated from 0 to "<<
    RAND_MAX<<std::endl;

    thrust::device_vector<int> dev_seeds( BLOCK_SIZE*num_blocks);
    thrust::host_vector<int> host_seeds( BLOCK_SIZE*num_blocks );
    std::srand(std::time(0));
    thrust::generate(host_seeds.begin(), host_seeds.end(), std::rand);
    dev_seeds=host_seeds;
    int* seed_ptr = thrust::raw_pointer_cast(dev_seeds.data());

    initialize_rng<<<num_blocks, BLOCK_SIZE>>>(rng_states, seed_ptr,
       d_num_curand_calls, d_seed_type);
}
```

   The macro variable `RAND_MAX` is set to 2147483647. Therefore we have the guarantee that a wide range of integers is spanned for the choice of the seeds.

In case as many seeds as the number of threads are employed, the kernel remains the same independently of the policy adopted for the selection of the seeds.

```
__global__ void initialize_rng(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 shifted Laplacian	10^6	10^4	0.05	99.5	50,016	12.3509
1D shifted Laplacian	10^6	10^7	18.3	81.7	640,563	0.391686
SP_1	25,568	10^7	18.0	82.0	349,096	0.450792

Table 1: Adjoint method - 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 shifted Laplacian	10^6	10^4	0.0	100.0	50,094	12.3914
1D shifted Laplacian	10^6	10^7	0.0	100.0	533,451	0.408134
SP_1	25,568	10^7	0.0	100.0	241,571	0.397887

Table 2: Adjoint method - Timings using different consecutive seeds and same sequence.

Matrix	**Size**	**Nb. Histories**	**% time rng**	**% time kernel**	tot. time (ms)	rel. residual norm
2D Laplacian	900	10^7	0.0	100.0	193,381	0.112068
1D shifted Laplacian	10^6	10^4	0.0	100.0	36,295	12.3404
1D shifted Laplacian	10^6	10^7	0.0	100.0	549,992	0.408296
SP_1	25,568	10^7	0.0	100.0	241,817	0.383815

Table 3: Adjoint method - Timings using different seeds picked randomly and same sequence.

As it can be noticed from the previous tables 1, 2 and 3, the second and third options provides better timings. 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 Adjoint method - 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 Adjoint method - 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. The simulations have been accomplished both with and without 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	191,818
1D shifted Laplacian	10^6	10^4	0.799972	0.639977	35,351
1D shifted Laplacian	10^6	10^7	0.799972	0.639977	517,788
SP_1	25,568	10^7	0.977674	0.999836	349,096

Table 4: Adjoint method - 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	191,818
1D shifted Laplacian	10^6	10^4	0.799972	0.639977	35,351
1D shifted Laplacian	10^6	10^7	0.799972	0.639977	517,788
SP_1	25,568	10^7	0.977674	0.999836	349,096

Table 5: Adjoint method - Timings without LDG instructions and constant history length equal to 10,000.

By comparing the values of tables 4-12 it is pointed out that a benefit in terms of timings is achieved when `ldg` instructions are employed in all the three subroutines at issue. Moreover the improvement is more evident for the 2D laplacian and the SP_1 matrix. The reason of this might

Matrix	**Size**	**Nb. Histories**	$\rho(H)$	$\rho(\hat{H})$	tot. time (ms)
2D Laplacian	900	10^7	0.994869	0.99447	139,882
1D shifted Laplacian	10^6	10^4	0.799972	0.639977	37,610
1D shifted Laplacian	10^6	10^7	0.799972	0.639977	500,092
SP_1	25,568	10^7	0.977674	0.999836	157,545

Table 6: Adjoint method - Timings with LDG instructions at 1) 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	193,336
1D shifted Laplacian	10^6	10^4	0.799972	0.639977	37,759
1D shifted Laplacian	10^6	10^7	0.799972	0.639977	548,909
SP_1	25,568	10^7	0.977674	0.999836	241,490

Table 7: Adjoint method - Timings with LDG instructions at 2) 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	105,010
1D shifted Laplacian	10^6	10^4	0.799972	0.639977	36,340
1D shifted Laplacian	10^6	10^7	0.799972	0.639977	519,408
SP_1	25,568	10^7	0.977674	0.999836	200,559

Table 8: Adjoint method - Timings with LDG instructions at 3) 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	140,347
1D shifted Laplacian	10^6	10^4	0.799972	0.639977	36,344
1D shifted Laplacian	10^6	10^7	0.799972	0.639977	499,007
SP_1	25,568	10^7	0.977674	0.999836	157,458

Table 9: Adjoint method - Timings with LDG instructions at 1), 2) 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	158,468
1D shifted Laplacian	10^6	10^4	0.799972	0.639977	36,141
1D shifted Laplacian	10^6	10^7	0.799972	0.639977	547,777
SP_1	25,568	10^7	0.977674	0.999836	200,325

Table 10: Adjoint method - Timings with LDG instructions at 2), 3) and constant history length equal to 10,000.

be associated with the sparsity pattern. Indeed a higher number of nonzero entries induces the histories to run for longer, enhancing the utility of the texture memory might increase as well. For the 2D laplacian and the SP_1 matrix the employment of `ldg` instructions almost halves the

Matrix	**Size**	**Nb. Histories**	$\rho(H)$	$\rho(\hat{H})$	tot. time (ms)
2D Laplacian	900	10^7	0.994869	0.99447	105,331
1D shifted Laplacian	10^6	10^4	0.799972	0.639977	34,616
1D shifted Laplacian	10^6	10^7	0.799972	0.639977	520,762
SP_1	25,568	10^7	0.977674	0.999836	138,091

Table 11: Adjoint method - Timings with LDG instructions at 1), 2), 3) 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	27,686
1D shifted Laplacian	10^6	10^4	0.799972	0.639977	35,108
1D shifted Laplacian	10^6	10^7	0.799972	0.639977	40,720
SP_1	25,568	10^7	0.977674	0.999836	4,919

Table 12: Adjoint method - Timings without LDG instructions and with weight cutoff.

Matrix	**Size**	**Nb. Histories**	$\rho(H)$	$\rho(\hat{H})$	tot. time (ms)
2D Laplacian	900	10^7	0.994869	0.99447	26,344
1D shifted Laplacian	10^6	10^4	0.799972	0.639977	35,911
1D shifted Laplacian	10^6	10^7	0.799972	0.639977	41,092
SP_1	25,568	10^7	0.977674	0.999836	4,759

Table 13: Adjoint method - Timings with LDG instructions at 1) and with weight cutoff.

Matrix	**Size**	**Nb. Histories**	$\rho(H)$	$\rho(\hat{H})$	tot. time (ms)
2D Laplacian	900	10^7	0.994869	0.99447	27,722
1D shifted Laplacian	10^6	10^4	0.799972	0.639977	36,005
1D shifted Laplacian	10^6	10^7	0.799972	0.639977	40,001
SP_1	25,568	10^7	0.977674	0.999836	5,585

Table 14: Adjoint method - Timings with LDG instructions at 2) and with weight cutoff.

Matrix	**Size**	**Nb. Histories**	$\rho(H)$	$\rho(\hat{H})$	tot. time (ms)
2D Laplacian	900	10^7	0.994869	0.99447	24,814
1D shifted Laplacian	10^6	10^4	0.799972	0.639977	36,604
1D shifted Laplacian	10^6	10^7	0.799972	0.639977	40,305
SP_1	25,568	10^7	0.977674	0.999836	5,065

Table 15: Adjoint method - Timings with LDG instructions at 3) and with weight cutoff.

time for the computation. The time reduction gets weaker for the other test cases. In fact for the 1D laplacian it seems there is no benefit in terms of timings coming from the texture memory.

By looking at the results of the same test cases when a weight cutoff is used, we can see that the utility of `__ldg` instructions is vanished (see Tables 13-18).

Matrix	**Size**	**Nb. Histories**	$\rho(H)$	$\rho(\hat{H})$	tot. time (ms)
2D Laplacian	900	10^7	0.994869	0.99447	26,248
1D shifted Laplacian	10^6	10^4	0.799972	0.639977	35,741
1D shifted Laplacian	10^6	10^7	0.799972	0.639977	40,017
SP_1	25,568	10^7	0.977674	0.999836	4,701

Table 16: Adjoint method - Timings with LDG instructions at 1) and 2) and with weight cutoff.

Matrix	**Size**	**Nb. Histories**	$\rho(H)$	$\rho(\hat{H})$	tot. time (ms)
2D Laplacian	900	10^7	0.994869	0.99447	27,184
1D shifted Laplacian	10^6	10^4	0.799972	0.639977	36,141
1D shifted Laplacian	10^6	10^7	0.799972	0.639977	40,727
SP_1	25,568	10^7	0.977674	0.999836	5,086

Table 17: Adjoint method - Timings with LDG instructions at 2), 3) and with weight cutoff.

Matrix	**Size**	**Nb. Histories**	$\rho(H)$	$\rho(\hat{H})$	tot. time (ms)
2D Laplacian	900	10^7	0.994869	0.99447	24,307
1D shifted Laplacian	10^6	10^4	0.799972	0.639977	36,980
1D shifted Laplacian	10^6	10^7	0.799972	0.639977	40,596
SP_1	25,568	10^7	0.977674	0.999836	4,921

Table 18: Adjoint method - Timings with LDG instructions at 1), 2) and 3) and with weight cutoff.

4 Adjoint method - 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 call to the aforementioned functions. The size of the batch for this grouped random numbers is a parameter that can be tuned to find the optimal configuration, reducing the total 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	192,325	0.119588
1D shifted Laplacian	10^6	10^4	0.0	100.0	36,602	12.3914
1D shifted Laplacian	10^6	10^7	0.0	100.0	517,536	0.408134
SP_1	25,568	10^7	0.0	100.0	241,928	0.397887

Table 19: Adjoint method - Results for a single call of rng for both `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	22,634	0.119588
1D shifted Laplacian	10^6	10^4	0.0	100.0	37,002	12.3914
1D shifted Laplacian	10^6	10^7	0.0	100.0	40,473	0.408134
SP_1	25,568	10^7	0.0	100.0	4,806	0.397887

Table 20: Adjoint method - Results for a single call of rng for both `initializeHistory` and `getNewState` with weight cut-off and use of ldg.

Overall, for a configuration where the length of the random walk is fixed, the second option seems to be slightly more efficient (compare Tables 2 and 20).

The time reduction accomplished by grouping the generation of several random numbers has not produced significant effects though. Therefore from now on `initializeHistory` and `getNewState` will be employed by making an explicit call to the random number generator every time that is necessary to produce a new random number.

5 Adjoint method - Reorganization of data through `struct`

In order to attempt the reduction of GPU timings, a reorganization of the matrices used by the Monte Carlo linear solver is accomplished.

In particular the pursue is to increase the vicinity of data that are going to be consulted in adjacent time steps by the computer. Because of this a C++ `struct` has been introduced:

```
struct device_row_data{
        double H;
        double P;
        double W;
        int inds;
};
```

When a new state has to be taken by a random walk, corresponding values of the transition probability, iteration matrix and weight are picked from the memory storage, providing the motivation for this attempt. The integer `inds` is used to store the index of the state, while the double values `H`, `P` and `W` are employed to store the entries of the iteration matrix, probability and weight respectively. This way, values of H, P, W associated with a particular entry are stored in contiguous cells of the memory.

This allows to reorganize the data necessary for the Monte Carlo linear solver in an array whose entries are `struct` elements. The length of the array corresponds to the number of nonzero entries of the iteration matrix.

In the following tables results associated with the employment of such data structure are presented, both by using a fixed length for the histories and by resorting to a weight cutoff of 10^{-9}.

Matrix	**Size**	**Nb. Histories**	$\rho(H)$	$\rho(\hat{H})$	tot. time (ms)
2D Laplacian	900	10^7	0.994869	0.99447	105,093
1D shifted Laplacian	10^6	10^4	0.799972	0.639977	27,404
1D shifted Laplacian	10^6	10^7	0.799972	0.639977	422,197
SP_1	25,568	10^7	0.977674	0.999836	138,921

Table 21: Adjoint method - Timings with `struct` data used and without weight cutoff. 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	24,669
1D shifted Laplacian	10^6	10^4	0.799972	0.639977	26,907
1D shifted Laplacian	10^6	10^7	0.799972	0.639977	30,773
SP_1	$25,568$	10^7	0.977674	0.999836	5,005

Table 22: Adjoint method - Timings with `struct` data used. LDG instructions at 1), 2) and 3) and with weight cutoff.

6 Forward method - Assignment of tasks to threads

In the Adjoint method a single history contributes in the evaluation of different entries of the solution vector accordingly to the actual state visited. The Forward method instead is defined such that a single history always contributes on the same entry of the solution vector, depending on the state from which it has been initiated.

This typical property of the Forward method opens the way to different techniques to distribute the tasks between threads. In the problems at issue the concept of task coincide with a single history. Therefore the intent is to discover what is the best way to map histories to different threads accordingly to their index.

Regardless of the amount o histories employed for a single entry (when this number is bigger than one), two viable options to execute the assignment are:

- to employ different threads for the estimation of an entry of the solution vector
- create a one-to-one relationship between the entries of the solution vector and the threads initialized during the process.

The first option induces to deal with the issue of different threads trying to modify the value of the same entry (data concurrency). From the point of view of implementation different approaches can be use to cope with it. In the following dissertation we proceed storing the solution vector in global memory and executing an atomization of all those operation that access

and modify values contained in the same memory cell. The second approach, instead, does not entail any data concurrency since each thread always accesses a memory cell which is never consulted by any other thread. However we have a restriction to the number of threads which actually do a job, since it is necessarily equal to the size of the linear system.

The goal is to compare the timings between these two settings, focusing on determining the effects of the serialization occurring in the execution of the kernel for the first option.

In the case when different threads are allowed to work on the same entry, a possible mapping might be the following one:

```
__global__ void run_forward_monte_carlo(...)
{

    ...

    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int entry = tid / entry_histories;

    ...

}
```

As already mentioned previously in the dissertation, this requires a serialization of the access by different threads to the same memory cell. This is accomplished by the following instruction:

```
__device__ void tallyContribution(int state, double wt, double * const x)
{
        // Collision estimator just adds weight
        atomicAdd(x+state,wt);
}
```

The one-to-one relationship between entries and threads instead induces instead the following mapping:

```
__global__ void run_forward_monte_carlo(...)
{

    ...

    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int entry = tid / entry_histories;

    ...

}


\hspace*{-2cm}
\begin{lstlisting}
__global__ void run_forward_monte_carlo(...)
```

```
{

    ...

    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if( tid < N )
    {
       ...
    }

    ...

}
```

In the last case obviously it is necessary to instantiate a number of threads bigger than the actual size of the problem to solve, since in general the size of the blocks is not a divisor of the number of degrees of freedom. However, the higher is the size of the problem, the less this is an issue.

For both the configurations the test cases are run both by using a fixed number of steps equal to 10,000 and by employing a weight cutoff equal to 10^{-9}. No texture memory instructions are employed.

Matrix	**Size**	**Nb. Histories per entry**	$\rho(H)$	$\rho(\hat{H})$	tot. time (ms)
2D Laplacian	900	10^3	0.994869	0.99447	437,444
1D shifted Laplacian	10^6	5	0.799972	0.639977	78,767
SP_1	$25,568$	10^7	0.977674	1.99619	241,573

Table 23: Forward method - mapping of multiple threads to a single entry. Fixed number of steps equal to 10,000.

Matrix	**Size**	**Nb. Histories per entry**	$\rho(H)$	$\rho(\hat{H})$	tot. time (ms)
2D Laplacian	900	10^3	0.994869	0.99447	79,306
1D shifted Laplacian	10^6	5	0.799972	0.639977	74,621
SP_1	$25,568$	200	0.977674	1.99619	241,522

Table 24: Forward method - one-to-one mapping between threads and entries. Fixed number of steps equal to 10,000.

Even if the case of the SP_1 matrix does not converge, we are going to consider the behavior of the code for this test case anyway. Indeed the priority is to focus on the performance of the algorithm with respect to different instantiations of the threads rather than the accuracy of the final numerical result itself.

Comparing values in Tables 23 and 24, we can see that the atomization of the contribution coming from each tally affect considerably the performance of the algorithm for all the test cases analyzed.

Looking at Tables 25 and 26, the employment of the one-to-one map still has a huge impact in terms of performance as concerns that 2D Laplacian and the SP_1 matrix. As regards the 1D shifted Laplacian, instead, the weight cutoff seems to have e dominant effect on the timing,

Matrix	**Size**	**Nb. Histories per entry**	$\rho(H)$	$\rho(\hat{H})$	tot. time (ms)
2D Laplacian	900	10^3	0.994869	0.99447	206,439
1D shifted Laplacian	10^6	5	0.799972	0.639977	39,295
SP_1	25,568	10^7	0.977674	1.99619	33,169

Table 25: Forward method - mapping of multiple threads to a single entry. Weight cutoff equal to 10^{-9}.

Matrix	**Size**	**Nb. Histories per entry**	$\rho(H)$	$\rho(\hat{H})$	tot. time (ms)
2D Laplacian	900	10^3	0.994869	0.99447	11,572
1D shifted Laplacian	10^6	5	0.799972	0.639977	38,484
SP_1	25,568	200	0.977674	1.99619	6,505

Table 26: Forward method - one-to-one mapping between threads and entries. Weight cutoff equal to 10^{-9}.

eliding the utility of the different techniques for the assignment of the tasks to the threads instantiated. A plausible explanation for this phenomenon might be the sparsity pattern of the matrices analyzed. In fact the number of nonzero entries per row is much higher for the first and third problem than for the second. Moreover the spectral radii of H and $\hat{H}$ are much smaller for the shifted 1D Laplacian. Therefore the average number of steps per history is lower for the 1D shifted Laplacian than for the other cases, explaining why this particular problem is transparent to the employment of different mappings for the task distribution.

7 Forward method - Use of texture memory

We now focus on comparing the change of performance of the Forward method depending on whether texture memory instructions are used or not. The collocation of the texture memory instructions is the same as for the Adjoint method.

By default the one-to-one mapping for the execution of the tasks by different threads is employed. Results are shown both by resorting to a fixed length of the histories and also by using an adaptive weight cutoff.

Matrix	**Size**	**Nb. Histories per entry**	$\rho(H)$	$\rho(\hat{H})$	tot. time (ms)
2D Laplacian	900	10^3	0.994869	0.99447	52,077
1D shifted Laplacian	10^6	5	0.799972	0.639977	63,353
SP_1	25,568	200	0.977674	1.99619	72,198

Table 27: Forward method - one-to-one mapping between threads and entries. Fixed number of steps equal to 10,000. LDG instructions employed.

Comparing results in Table 27 with the ones in Table 24, it is detected a significant improvement of the timings thanks to the employment of texture memory. Especially for the SP_1 matrix, supposedly for its sparsity pattern.

By making a comparison between Tables 28 and 26 we discover a similar behavior to the one verified for the Adjoint method. In fact once the weight cutoff is employed, the use of `__ldg` instructions seem not to be effective for the reduction of the computational time.

Matrix	**Size**	**Nb. Histories per entry**	$\rho(H)$	$\rho(\hat{H})$	tot. time (ms)
2D Laplacian	900	10^3	0.994869	0.99447	9,316
1D shifted Laplacian	10^6	5	0.799972	0.639977	40,005
SP_1	25,568	200	0.977674	1.99619	4,269

Table 28: Forward method - one-to-one mapping between threads and entries. Weight cutoff equal to 10^{-9}. LDG instructions employed.

8 Forward method - Reorganization of data through `struct`

In this section we focus on testing the efficiency of the employment of C++ `struct` objects to handle data, on the same lead as for the Adjoint method. The experiments are repeated both for a fixed length of histories equal to 10,000 and for the use of a weight cutoff equal to 10^{-9}, with and without `ldg` instructions as well. The one-to-one mapping between tasks and threads is employed.

Matrix	**Size**	**Nb. Histories per entry**	$\rho(H)$	$\rho(\hat{H})$	tot. time (ms)
2D Laplacian	900	10^3	0.994869	0.99447	80,059
1D shifted Laplacian	10^6	5	0.799972	0.639977	40,005
SP_1	25,568	200	0.977674	1.99619	122,861

Table 29: Forward method - use of struct for data handling. One-to-one mapping between threads and entries. Fixed number of steps equal to 10,000. No LDG instructions.

Matrix	**Size**	**Nb. Histories per entry**	$\rho(H)$	$\rho(\hat{H})$	tot. time (ms)
2D Laplacian	900	10^3	0.994869	0.99447	11,534
1D shifted Laplacian	10^6	5	0.799972	0.639977	29,636
SP_1	25,568	200	0.977674	1.99619	6,247

Table 30: Forward method - use of struct for data handling. One-to-one mapping between threads and entries. Weight cutoff equal to 10^{-9}. No LDG instructions.

Matrix	**Size**	**Nb. Histories per entry**	$\rho(H)$	$\rho(\hat{H})$	tot. time (ms)
2D Laplacian	900	10^3	0.994869	0.99447	50,159
1D shifted Laplacian	10^6	5	0.799972	0.639977	67,993
SP_1	25,568	200	0.977674	1.99619	75,305

Table 31: Forward method - use of struct for data handling. One-to-one mapping between threads and entries. Fixed number of steps equal to 10,000. LDG instructions employed.

Results in tables 29, 30, 31 and 32 show that actually the employment of struct for the data storage does not provide significant improvements for the 2D Laplacian and the SP_1 matrix. However an interesting progress is detected for the shifted 1D Laplacian. These results have a pretty similar trend to the one discovered for the Adjoint method.

Matrix	**Size**	**Nb. Histories per entry**	$\rho(H)$	$\rho(\hat{H})$	tot. time (ms)
2D Laplacian	900	10^3	0.994869	0.99447	8,950
1D shifted Laplacian	10^6	5	0.799972	0.639977	30,314
SP_1	$25,568$	200	0.977674	1.99619	4,040

Table 32: Forward method - use of struct for data handling. One-to-one mapping between threads and entries. Weight cutoff equal to 10^{-9}. LDG instructions employed.