Mathematical Problems of Computer Science 35, 46–52, 2011.

Realization of 1D Classic Heisenberg Spin-Glass System on GPU

Hakob G. Abajyan

Institute for Informatics and Automation Problems of NAS of R.A
e-mail habajyan@ipia.sci.am

Abstract

New high performance parallel algorithm for simulation of 1D classic Heisenberg spin-glass system is developed. The realization is done on Graphics Processing Units (GPU). Numerical simulations of the classic Heisenberg spin glass model show that this is a good example of applications that may benefit of the GPU computing capabilities. GPU performance time and memory statistics are presented.

1. Introduction

Let us consider classical ensemble of disordered 1D steric spin-chains (SSC), where it is supposed that interactions between spin-chains are absent (later it will be called an ideal ensemble) and that there are N_s spins in an each chain. Despite some ideality of the model it can be interesting enough and rather convenient for investigation of a number of important and difficult applied problems of physics, chemistry, material science, biology, evolution, organization dynamics, hard-optimization, environmental and social structures, human logic systems, financial mathematics etc (see [1, 2, 3, 4]).

Mathematically mentioned type of ideal ensemble can be generated by 1D Heisenberg spin-glass Hamiltonian without external field [3, 4]:

$$H_0(N_s) = -\sum_{i=0}^{N_s-1} J_{i,i+1} S_i S_{i+1}. \qquad (1)$$

where S_i describes the i-th spin which is a unit length vector and has a random orientation. In the expression (1) $J_{i,i+1}$ characterizes a random interaction constant between i and $i+1$ spins, which can have positive and negative values as well. The distribution of $J_{i,i+1}$ is detailed described in [4].

For further investigations it is useful to rewrite the Hamiltonian (1) in spherical coordinates:

$$H_0(N_s) = -\sum_{i=0}^{N_s-1} J_{i,i+1}[\cos\psi_i \cos\psi_{i+1} \cos(\varphi_i - \varphi_{i+1}) + \sin\psi_i \sin\psi_{i+1}]. \qquad (2)$$

A stationary point of the Hamiltonian is given by the system of trigonometrical equations:

$$\frac{\partial H_0}{\partial \psi_i} = 0. \qquad \frac{\partial H_0}{\partial \varphi_i} = 0. \qquad (3)$$

where $\Theta_i = (\psi_i, \varphi_i)$ are angles of i-th spin in the spherical coordinates system (ψ_i is a polar and φ_i is an azimuthal angles). $\Theta = (\Theta_1, \Theta_2 .... \Theta_{N_s})$ respectively describe the angular part of a spin-chain configuration.

In case all the interaction constants between i-th spin with its nearest-neighboring spins $J_{i-1,i}$, $J_{i,i+1}$ and angle configurations $(\psi_{i-1}, \varphi_{i-1})$, (ψ_i, φ_i) are known, it is possible to explicitly calculate the pair of angles $\Theta_{i+1} = (\psi_{i+1}, \varphi_{i+1})$. Correspondingly, the i-th spin will be in the ground state (in the state of minimum energy) if in the stationary point $\Theta_i^0 = (\psi_i^0, \varphi_i^0)$ the following conditions are satisfied (for more details see [4]):

$$A_{\psi_i,\psi_i}(\Theta_i^0) > 0, \qquad A_{\psi_i,\psi_i}(\Theta_i^0) A_{\varphi_i,\varphi_i}(\Theta_i^0) - A^2_{\psi_i,\varphi_i}(\Theta_i^0) > 0, \tag{4}$$

where $A_{\alpha_i,\alpha_i}(\Theta_i^0) = \partial^2 H_0 / \partial \alpha_i^2$. $A_{\alpha_i,\beta_i}(\Theta_i^0) = A_{\beta_i,\alpha_i}(\Theta_i^0) = \partial^2 H_0 / \partial \alpha_i \partial \beta_i$.

2. GPGPU Technology

In spite of the availability of high performance multi-core systems based on traditional architectures, there is recently a renewed interest in floating point accelerators and co-processors that can be defined as devices that carry out arithmetic operations concurrently with or in place of the CPU. Among the solutions that have received more attention from the high performance computing community there are the NVIDIA Graphics Processing Units (GPU), originally developed for video cards and graphics, since they are able to support very demanding computational tasks (see [7, 8. 9]). As a matter of fact, astonishing results have been reported by using them for a number of applications covering, among others, atomistic simulations, fluid-dynamic solvers and option pricing. Simulations of statistical mechanics systems and the classic Heisenberg spin glass models in particular, are other examples of applications that may benefit of the GPU computing capabilities.

CUDA has a hierarchy of several kinds of memory (see Figure 1):

- *global* memory (DRAM): this is the main memory of the GPU and any location of it is visible by any thread. The bandwidth between the *global* memory and the multiprocessors is more than 100 GB/sec but the latency for the access is also large (approximately 200 cycles).
- *shared* memory: access to data stored in the shared memory has a latency of only 2 clock cycles. However, shared memory variables are local to the threads running within a single multiprocessor and the size of the shared memory is tiny compared to the global memory that is, usually, in the range of GBytes.
- *registers*: on a GPU there are thousands of 32 bits registers. It is worth noting that, for each multiprocessor, there is more space for data in the registers than in the shared memory.
- *cache*: L1 and L2 caches have been included in the Fermi architecture (see [5]). Actually, on each multiprocessor there are 64Kbytes of private L1 cache that can be split, at run time, in a 48 Kbytes shared memory and a 16KB L1 cache or in a 16Kbytes shared memory and a 48 L1 cache.
- *constant* and *texture*: these are special memories used respectively to store constant values and to cache global memory (separate from register and shared memory) offering dedicated interpolation hardware separate from the thread processors.

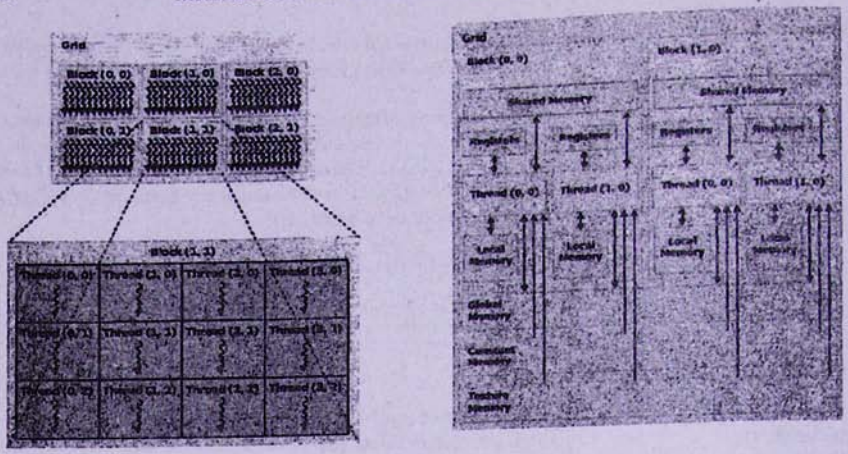

Figure 1: CUDA Memory Hierarchy

Functions running on a GPU with CUDA have some limitations: they can not be recursive; they do not support static variables; they do not support variable number of arguments; function pointers are meaningless. Further information about the features of the NVIDIA GPU and the CUDA programming technology can be found in [5, 7].

3. CUDA Implementation

For the GPU programming, CUDA Software Development Toolkit 3.2 version was employed that offers an extended C compiler and is available for all major platforms (Windows, Linux, Mac OSX). The extensions to the C language supported by the compiler allow starting computational kernels on the GPU, copying data back and forth from the CPU memory to the GPU memory and explicitly managing the different types of memory available on the GPU (with the notable exception of the caches) The programming model is a Single Instruction Multiple Data (SIMD) type. Each multiprocessor is able to perform the same operation on different data 32 times so the basic computing unit (called warp) consists of 32 threads. To ease the mapping of data to threads, the threads identifiers may be multidimensional and, since a very high number of threads run in parallel, CUDA groups threads in blocks and grids.

One of the crucial requirements to achieve a good performance on the NVIDIA GPU is to hide the high latency of the global memory accesses (both read and write) by following a set of rules that depend on the specific level of the architecture. Also important is to avoid running out of registers since registers-spilling, although supported, has a very high cost.

Here is the pseudo-code of CUDA realization :

```
...
#include<stdio.h>
```

```
#include<cuda.h>


    // Kernel that executes on the CUDA device
    __global__ void kernel_test (Node* currentLayer, int currentNodeCount,
    Node* nextLayer, float* randNumbers)
{
    // The main logic that should be done on GPU
}


    // The main routine that executes on the host
int main (int argc , char* argv[])
{
    float *array_host, *array_device; // Pointer to host and device arrays
    int N = atoi(argv[1]); // Number of elements in arrays
    size_t size = N * sizeof(float);
    array_host = (float*)malloc(size); // Allocate array on host
    cudaMalloc((void**) &array_device, size); // Allocate array on device
    // Initialize host array and copy it to the CUDA device
    cudaMemcpy(array_device, array_host, size, cudaMemcpyHostToDevice);
    // Do some preparation work on host
    ...

    // Do calculation on device
    kernel_test<<< gridDims , blockDims >>> (parameter_list);
    // Retrieve result from device and store it in host array
    cudaMemcpy(array_host, array_device, size, cudaMemcpyDeviceToHost);
    // Do final work on host
    ...

    // Cleanup
    free(array_host);
    cudaFree(array_device);
    return 0;
}
```

GPU Cores	240
Clock in MHz	1300
Single Precision	933
Double Precision	78
Floating Point Precision	IEEE 754 single and double
Bus Type	GDDR3
Internal RAM	4 GB
Memory Speed	800 MHz
Memory Interface	512-bit
Internal RAM Speed	102 GB/sec
Compute Capability	1.3
TDP Watts	187.8
Form Factor and Features	2 slot video card
Auxiliary Power Connectors	6-pin and 8-pin
Thermal Solution	Active fan sink
Software Development Tools	C-based CUDA Toolkit

Table 1: NVIDIA Tesla C1060 Card Specifications

In Table 1 and Table 2 the key aspects of NVIDIA GPU (Tesla C1060) which was used for numerical experiments are mentioned.

GPU model	Tesla C1060
Number of Multiprocessors	30
Number of cores	240
Shared memory per block (in bytes)	16384
L1 Cache	N/A
L2 Cache	N/A
Number of registers per block	16384
Max Number of thread per block	512
Error Checking and Correction (ECC)	No

Table 2: Main features of the NVIDIA GPU used for the experiments

```
__global__ void set_vector_to_zero (int* array, int size)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < size) {
    array[i] = 0;
  }
}
```

In Table 3 and Table 4 there are correspondingly presented GPU time and memory statistics. It is generated by "CUDA Compute Visual Profiler" tool (see [6]). In the presented tables there are mentioned the following methods:

- kernel_test - the main kernel method which runs one GPU.
- set_vector_to_zero - helper kernel method which runs one GPU.
- cudaMemcpyHostToDevice - copy data from host to device method.

- cudaMemcpyDeviceToHost - copy data from device to host method.

Method	GPU Time (sec)	GPU Time (%)
kernel_test	331.968	83.71
set_vector_to_zero	2.368	0.59
cudaMemcpyHostToDevice	41.664	10.5
cudaMemcpyDeviceToHost	20.544	5.18

Table3: CUDA Compute Visual Profiler : GPU Time Statistics

Method	Shared Memory Per Block	Registers Per Thread	Host Memory Transfer Type
kernel_test	72	55	-
set_vector_to_zero	28	2	-
cudaMemcpyHostToDevice	-	-	Pageable
cudaMemcpyDeviceToHost	-	-	Pageable

Table 4: CUDA Compute Visual Profiler : Memory Statistics

4. Conclusion

The rapid increase in the performance of graphics hardware, coupled with recent improvements in its programmability, have made graphics hardware a compelling platform for computationally demanding tasks in a wide variety of application domains. The main advantage of GPGPU technology is that GPUs are a tremendously cost-effective way to boost performance. They are used for efficient and cost-effective supercomputing.

Using equations for stationary points of Heisenberg Hamiltonian and conditions of energy minimum of system on nodes of periodic lattice, new high performance parallel algorithm for a simulation of $1D$ classic Heisenberg spin-glass system is developed. GPGPU technology is used and the realization of the algorithm is done on NVIDIA Tesla C1060 computing processor (See Table 1 and Table 2). Since the ideology of GPU technology is SIMD (Single Instruction Multiple Data) and the algorithm is from that class of problems. fully paralleled implementation was obtained. Numerical simulations of the classic Heisenberg spin glass model show that this is a good example of applications that may benefit of the GPU computing capabilities.

Also in the paper GPU time and memory statistics are presented. They are generated by "CUDA Compute Visual Profiler" tool.

Acknowledgment

Special thanks to Prof.. DrSci.. Ashot S. Gevorkyan for introduction to Heisenberg spin glass systems and to Ph.D. Hrachya V. Astsatryan for the access to Tesla C1060 and for cooperation.

References

[1] A. P. Young (ed.). *Spin Glasses and Random Fields*. World Scientific. Singapore. 1998.

[2] R. Fisch and A. B. Harris. "Spin-glass model in continuous dimensionality". *Phys. Rev. Let.*. 47. p. 620. 1981.

[3] A. S. Gevorkyan et al.. "New mathematical conception and computation algorithm for study of quantum 3D disordered spin system under the influence of external field". *Trans. On Comput. Sci., VII, LNCS*, pp. 132-153, Spinger-Verlage. 10.1007/978-3-642-11389-58.

[4] A. S. Gevorkyan, H. G. Abajyan and H. S. Sukiasyan. ArXiv: cond-mat.dis-nn 1010.1623v1.

[5] NVIDIA CUDA, C Programming Guide, version 3.2. 2010.

[6] NVIDIA Compute Visual Profiler User Guide, 2010.

[7] L. Nyland, M. Harris, J. Prins, "Fast N-body simulation with CUDA. in: GPU Gems 3", *Addison-Wesley Professional*, Ch. 31, pp. 677695. 2007.

[8] J. D. Owens. D. Luebke, N. Govindaraju, M. Harris. J. Kruğer, A. Lefohn. T. J. Purcell. "A survey of general-purpose computation on graphics hardware", *Computer Graphics Forum*, vol. 26, Issue 1, pp. 80113, 2007.

[9] D. Yuen. J. Wang, L. Johnsson, C. H. Chi, Y. Shi(Eds.). *GPU solutions to multi-scale problems in science and engineering*, Springer. 1st Edition, 2011.

1D Դասական հեյզենբերգ սպին-ապակի համակարգի իրականացումը գրաֆիկական պրոցեսորների վրա

Հ. Աբաջյան

Ամփոփում

Աշխատանքում բերված է 1D դասական Հեյզենբերգ սպին-ապակի համակարգի մոդելավորման նոր բարձր արտադրողականության զուգահեռ ալգորիթմ: Ալգորիթմն իրականացված է րաֆիկական Պրոցեսորների (Պ) վրա: Թվային սիմուլյացիան ցույց է տալիս, որ դասական Հեյզենբերգ սպին-ապակի մոդելը կիրառության լավ օրինակ է Պ հաշվողական հնարավորությունների օգտագործման տեսանկյունից: Աշխատանքում նաև բերված են արագագործության ժամանակային և հիշողության վիճակագրական տվյալները: