N-Body Simulation Using GP-GPU: Evaluating Host/Device Memory Transference Overhead

Sergio M. Martin[1], Fernando G. Tinetti[2,3], Nicanor B. Casas[1],
Graciela E. De Luca[1], Daniel A. Giulianelli[1]

[1]Universidad Nacional de La Matanza
Florencio Varela 1903 - San Justo, Argentina
[2]III-LIDI, Facultad de Informática, UNLP
Calle 50 y 120, 1900, La Plata, Argentina
[3]Comisión de Inv. Científicas de la Prov. de Bs. As.
fernando@info.unlp.edu.ar, {smartin, ncasas, gdeluca, dgiulian}@ing.unlam.edu.ar

Abstract. N-Body simulation algorithms are amongst the most commonly used within the field of scientific computing. Especially in computational astrophysics, they are used to simulate gravitational scenarios for solar systems or galactic collisions. Parallel versions of such N-Body algorithms have been extensively designed and optimized for multicore and distributed computing schemes. However, N-Body algorithms are still a novelty in the field of GP-GPU computing. Although several N-body algorithms have been proved to harness the potential of a modern GPU processor, there are additional complexities that this architecture presents that could be analyzed for possible optimizations. In this article, we introduce the problem of host to device (GPU) – and vice versa – data transferring overhead and analyze a way to estimate its impact in the performance of simulations.

Keywords: N-Body Simulation, GPU Optimization, Data Transference Overhead.

1 Introduction

The N-body problem is largely known within the physics, engineering, and mathematical research faculty. It is commonly used to calculate – as precisely as possible – the future position, velocity, momentum, charge, potential, or any other aspect of a massive/charged body in regard to other bodies that interact with it within a time interval. Although some efforts have been made [1], many theorists have unsuccessfully tried for centuries to find a purely mathematical solution that could resolve any application of this problem in a series of steps linearly related to the amount (n) of bodies. Therefore, currently, the only way to approximate to a real solution is to use a differential method with tiny time slices (differentials) using the power of modern computers. However, this approach presents some downsides.

First, the usage of finite (as opposed to infinitesimal) time differentials is detrimental to the precision of the result. All positions and momentums are taken

from the starting moment of the differential and are kept as constants during the calculation. Since the simulated forces remain constant during such differential, the results obtained suffer from a subtle degradation after each iteration. In consequence, the larger time differential is used, the more error is produced [2].

On the other hand, if we use smaller time differentials for the simulation, more iterations will have to be calculated until the end time is reached. As a result, simulations will require more computing time.

It is therefore important to keep in mind that the length of the selected time differential ultimately defines the precision admissible for the result expected, and the time that a computer will take to complete the simulation. Using high-precision libraries to augment the precision will also redound in increased computing time [3].

A way to calculate the amount iterations (n) to be simulated is to evaluate the inverse relation between the entire simulation time interval (Δ_t), and the time differential (∂t) as shown in Eq. (1).

$$n = \frac{\Delta_t}{\partial t} \tag{1}$$

Yet another reason why time differential is an important factor to be taken into account is that it defines the amount of data being transferred between the host memory – traditionally known as RAM – and the device – graphics processing unit – through the PCI-Express bus. The time taken for the simulation will increase if more resources/time should be spent on unnecessary data transmission rather than just processing [4].

In traditional CPU-based schemes, this kind of data transference overhead is negligible since all data is present and up-to-date within the host memory after each iteration is calculated. In those cases, it is possible to use/access to all the positions of all n bodies and use them in real-time – for instance, for saving them into an output file, or rendering them into the screen. However, when GPU devices are used for these algorithms it is required to define explicit data transferences from the results obtained within the device memory back to the host in order to enable them for any use. Such overhead is detrimental to the overall performance and any efforts made to reduce it can yield significant optimizations [5]. Estimating such overhead is the object of our analysis in this article.

2 CUDA Implementation of N-BODY

The CUDA programming model as an extension of the C language provides an excellent scheme to parallelize scalable N-Body algorithms for GP-GPU architectures [6]. In this model, in opposition to the conventional multicore CPU model, the programmer is encouraged to create as many threads as needed depending on the amount of data elements in the problem. By doing this, it is possible, in the generality of cases, to yield the maximum performance from the many simple yet extremely

parallelizable GPU cores. Of course, existing algorithms similar to the one used in this article are not exceptions [7] [8].

In the particular implementation of our N-body algorithm for CUDA, we created one thread per body in the simulation that will be in charge to execute the same function – called *CUDA kernel* – within the GPU processor. This kernel is programmed to execute the following steps:

- Load a single body's initial values from the device global memory. Each thread will load a different body based on its thread ID.
- For each other body in the simulation:
  - Load the body's values from the device global or shared memory.
  - Calculate the force that all other bodies impose to the loaded body.
- Save the new values for acceleration in the body data back into the device global memory.

Although threads perform better when no synchronization or communication functions are executed within the kernel, the CUDA architecture allows the programmer to specify *blocks* where a certain number of threads – depending on the infrastructure capability – can work coordinately. Based on this possibility, several memory access optimizations can be done in order to reduce the memory latency overhead.

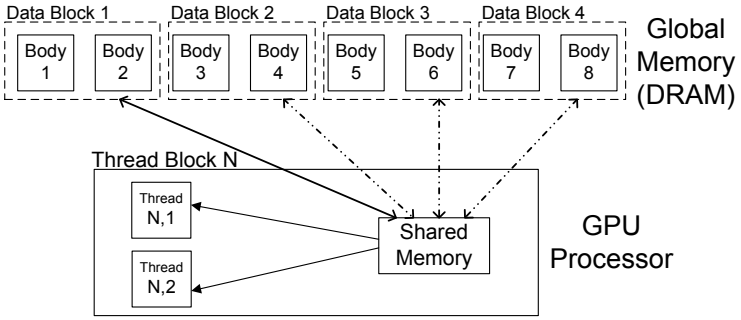

Fig. 1: Shared Memory utilization for the N-Body CUDA kernel

The most successful optimization that we implemented was the usage of intra-block shared memory. Since constantly accessing global memory (low latency) forces executing threads to stall until data is effectively loaded, the overall performance is greatly reduced. For that reason, this architecture provides the programmer with intermediate memory banks – such as shared and register memory – which reside within the processor and could be used to reduce the amount of accesses to global memory.

Fig. 1 shows an example of such optimization where threads are grouped into one-dimensional blocks of size two[1]. In the same fashion, bodies' data present in global memory were divided in data blocks that are loaded, one by one, into the block shared memory. By doing this, all threads read data from the global memory only once and

[1] This size was arbitrarily defined for simplicity reasons in this article while, in fact, blocks of 512 threads were actually used in our experiments.

access it several times within the shared memory, thus reducing the total memory latency overhead.

The pseudo-code shown next represents the N-body kernel to be executed by every thread using the CUDA terminology:

```
void nbodyKernel(vec)
{
  thread_body = vec[TID + BID * BSIZE]

  For each i in  BCOUNT Do
     Shared_data[TID] = vec[TID + BSIZE * i]
     For each j in BSIZE Do
       Compute(thread_body, i*BSIZE + j)
       Update acceleration of thread_body
     End for
  End For

  vec[TID + BID*BSIZE] = thread_body
 }
```

Where:

- **thread_body** is the private memory for the body data pertaining to each thread.
- **vec** is the collection of bodies' data stored in the global memory of the device.
- **Shared_data** is a vector of size **BSIZE** where a complete block of data is stored and used as shared memory by a particular block of threads.
- **TID** is the thread identifier within the block.
- **BID** is the block identifier.
- **BSIZE** is the size of each block.
- **BCOUNT** is the amount of blocks created.

A variety of other optimizations has been applied to the algorithm used in our experiments. Some of them have been already described in our previous work regarding the usage of multi-core clusters described in [9] and [10], and were used as the base for the CUDA version of our algorithm. However, more GPU-specific optimizations such as memory coalescing, shared memory usage, loop unrolling, interleaved thread-body processing were applied. Most of these optimizations are defined as good practices for any CUDA algorithm [11] [12]. Consequently, we assume for our experiments that the algorithm cannot be optimized any further.

3 Memory Transference Overhead

There are many types of research that requires scientists to run N-Body simulations in physics or engineering topics. In some, only the final result – for example, final position of the bodies involved – is needed; in others, it is more important to know the path that those bodies took during the simulation. Depending on each case – or a

combination thereof –, scientists could choose to have the intermediate results stored in a device, transmitted through a network or displayed on a screen. In other cases, they would discard part or the entire journey in order to reduce memory transference overhead.

As mentioned for CPU based algorithms, all information is present in the host memory to be used at all times. Even if it is not used, stored, or sent through a network during the simulation, no extra time is required for memory transmission. However, in the case of GP-GPU algorithms, copying the data back to the host is necessary if some action is to be performed with them.

It is important to mention that, even if no intermediate data is needed for the simulation purposes, it is still necessary to guarantee results with acceptable precision by calculating the necessary amount of iterations of rather small time differentials until the total simulation time is reached. This forces every simulation to be performed with a certain number of iterations, even if only the final result is needed.

In this research, we sought to measure the impact of data transmission on the overall performance of the algorithms, letting aside other possible overheads introduced by its usage. By measuring this, we were be able to determine how much performance can be gained by only obtaining the final results of a N-Body simulation, in comparison with transmitting the intermediate results at each iteration. This allowed us to define the minimum and maximum performance gain possible regarding data transmission between the host (CPU) and device (GPU), having all other possible combinations (for instance, transmitting one result every two iterations) in between those two results.

We have verified through experimentation that these relations do not vary when the iteration count[2] is changed. Using a rather high amount of iterations, deviation becomes insignificant. For iterations counts close to 1, however, execution interference from the operating system introduces a more noticeable deviation.

In order to measure how much overhead is introduced by transmitting data at each iteration in relation to doing so only at the beginning and the end of the simulation, we ran the same set of tests to compare two algorithms. Algorithm *Nbody1* transmits – yet it does not use – intermediate results after each iteration, and *Nbody2* calculates all iterations without interruptions. The architecture used for our tests is shown in Table 1, and the results obtained are shown in Table 2.

Table 1: GPU Architecture used.

GPU Device	GeForce GTX 550Ti
CUDA Cores	192
Capability	CUDA 2.1
DRAM	1 GB GDDR5

Table 2: GFlop/s obtained for both versions

n	*Nbody1*	*Nbody2*
4096	271	307
8192	315	333
16384	343	353
32768	357	362

The first detail to notice from the results is that the difference between the GFlop/s obtained from both versions – the amount of overhead introduced by data transmission – reduces as *n* (amount of bodies being simulated) increases. This can be

[2]We used 65536 simulation iterations in all our experiments.

explained by the fact that the algorithm complexity is quadratic – becomes 4 times bigger, when we double the data – while data transmission increases only linearly regarding the problem size – transmission time will only double. In other words, as the threads take more time to execute the kernel, the overhead of data transmission becomes less significant. This relationship can be seen in the results presented in Fig. 2.

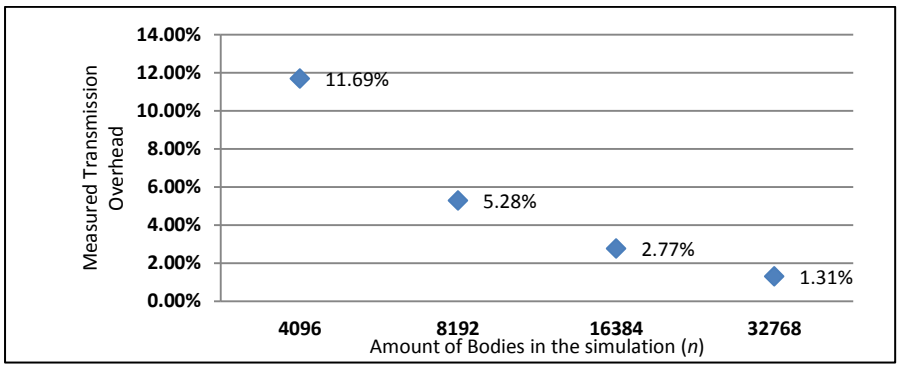

Fig.2: Measured transmission overhead ratio.

4 Transmission/execution ratio evaluation

Since we have empirically obtained values of ratio between the transmission overhead size (expressed in Flop) for several cases of *n*, we deemed necessary to look for a relationship that could allow us to evaluate this ratio for any given *n*. Moreover, expressing this relationship in terms of bytes and Flops could allow calculating an estimate of transmission overhead for other types of algorithms, and not only for N-body problems. The first step in order to obtain such relationship is to find how data transmission requirement increases given a discrete increase in one body. We used profiling tools [13] and techniques [14] that obtain precise information about memory usage directly from the hardware counters. Fig. 3 shows the host/device transference volume for a single iteration with *n*=4096 elements and Fig. 4 shows the host/device transference volume for the same N count.

	Source	Destination	Size (bytes)
1	Host Unpinned	Device	917504
2	Device	Host Unpinned	917504

Fig.3: Transmitted data for one simulation iteration.

	Function Name	Achieved FLOPS [1]: Single FLOP Count
1	nbodyKernel	20,400,472,064.00

Fig.4: Single-Precision Flop Count for the N-Body kernel per iteration.

We need now a way to tell how much data transmission increases when adding another body to the simulation. This can be obtained by multiplying the calculated data transmission per iteration by count of the iterations being simulated and dividing it by the *n* used. The resulting expression will determine *T(n)* – total transmission requirements – as a function of *n*. Eq. (2) shows n for the performed test:

$$T(n) = \frac{917504\ bytes}{4096}\ n = \ 224n\ bytes \tag{2}$$

The second step is to determine the size of the problem – expressed in MFlop – increases, given a similar increase in one element. To obtain this value, the same profiling tool allowed us to know how many floating operations were performed during the execution of the N-body kernel. As a result we can consider *F(n)* – total amount of Flop – as a function of *n*, using the obtained single-precision Flop count per body/body compute as in Eq. (3):

$$F(n) = \ 1216\ n(n-1)\ FLOP \tag{3}$$

Having *T(n)* and *F(n)* as functions of *n*, it is possible to establish the relationship between the bytes of data being transmitted and the amount of Flops for each additional element of an algorithm with quadratic complexity. As a result, we can obtain a data overhead ratio (*dor*) as in Eq. (4):

$$dor(n) = \frac{N(n)}{F(n)} = \frac{224\ n\ bytes}{1216\ \ n(n-1)\ FLOP} \cong \frac{0{,}185}{(n-1)} \left[\frac{byte}{FLOP}\right] \tag{4}$$

The data overhead ratio (*dor*) obtained indicates, for this algorithm, how many bytes will be transmitted per floating point operation to be executed, given *n* elements. The *dor* value for every integer between 4096 and 32768 resemble the same inverse relation that our experimental measures shown in Fig. 2.

What is most important about this relation is that it is architecture-independent. This means that, no matter which GPU device model we use, the execution of this kernel will have the same ratio between data transmission and Flop processing. Thus, we only have to link it with the actual cost of transmission of this specific architecture to get its fraction of the performance overhead.

This proportion can be easily calculated since we know that the optimal performance of the GPU device doesn't vary, and it is only being reduced by the data transmission overhead. Thus, we can assume that the increase in the problem size – measured in GFlop – is the *r* relation for the performance drop observed in Table 2. For N = 4096, Eq. (5) reflects this increase:

$$r(4096) = \frac{307 - 271}{307} = 0.117\ \frac{Bytes\ (transfered)}{FLOP\ (processed)} \tag{5}$$

Therefore, if this relation is observed for *n* = 4096, there has to be a constant *k* that allows to represent perfectly the percentage of performance drop due to data transmission as seen from our measurements for this specific architecture. Calculating it from the *r*(4096) ratio value, we obtained the result shown in Eq. (6):

$$r(n) = k * \ dor(n) = \frac{480}{(n-1)} \tag{6}$$

Having the relation *r* as a function of *n* will allow us to obtain the data overhead ratio for any *n* positive integer without having to perform any additional tests. As can be seen in Fig. 5, this inference matches perfectly with those measured in experiments and shown in Fig. 2.

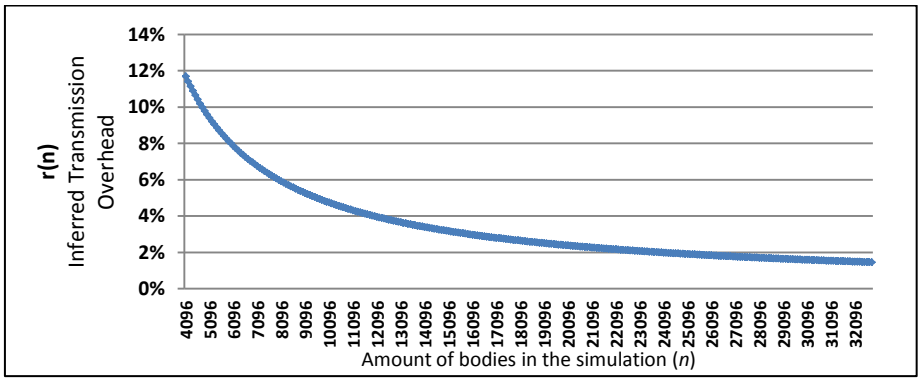

Fig.5: Inferred transmission overhead ratio.

It is important to note that the proportion *k* obtained is the particular value of the GPU device – and underlying architecture – we used. Therefore, for each other architecture used to execute our kernel, a new value for *k* should be provided that reflects the estimated performance drop.

On the other hand, since the *dor(n)* ratio will not vary between different architectures, it should be calculated only once per algorithm. Then, just combining it with the appropriate *k* proportion to obtain the *r(n)* of that specific algorithm-architecture scenario.

The most valuable aspect of having such pre-calculated proportions is that a table containing different *k* values for the available architectures, and *dor(n)* values for the available algorithms, we could predict the performance drop for data transmission for combinations of algorithms and architectures that were not tested in actual experiments.

4.1 Interleaved transference per iteration ratios

We have surmised through our tests that the performance overhead of transmitting the simulation's intermediate results at each iteration for different values of *n* can be estimated. However, it could also be helpful to calculate the overhead if just a certain portion of intermediate results should be gathered. In such case, we would have data transmissions every *m* number of iterations.

As we could appreciate in the previous section, the *dor(n)* ratio for this algorithm was calculated for a 1/1 proportion of transmissions per iteration. However, if we wanted to change that proportion to 1/2, (which means: transmitting every two iterations) its value would proportionately drop to a half. Thus, we can extend our definition of *r* to take into account the amount of iterations per transmission as in Eq. (7):

$$r(n,m) = \frac{k * dor(n)}{m} = \frac{480}{(n-1)m} \quad (7)$$

In order to test the accuracy of the estimations made with the *r(n,m)* equation, we verified its estimations with a series of tests using variations for the values of *n* and *m*. We confirmed that every result approached the estimations with negligible deviations. Therefore, such equation could effectively determine the impact of data transmission in a wide variety of cases. In Fig. 6, we show different curves as functions of *n*, using different values for *m*.

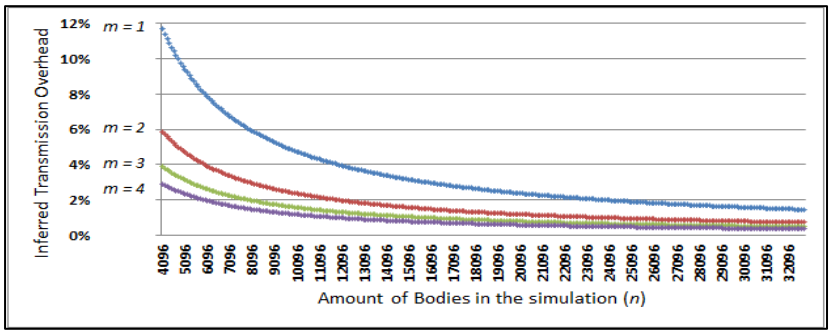

Fig.6: Inferred transmission overhead ratio for different values of *m*.

The extent at which scientists will be willing to sacrifice intermediate data to be discarded by this approach should be considered for each case. However, having estimations for all combinations of *n* and *m* we can provide valuable clues for establishing the best option in each case.

5 Conclusions and Further Work

Balancing and fine-tuning the two factors that define the numerical precision of a simulation (total time interval and differential) can be a very complicated task. Since they define the amount of iterations being calculated, they will also define how much real time will be spent on the actual calculations. Certainly, for scientists only interested in a final result, estimating the negligible data transference overhead is of a little interest. However, for simulations that need to store intermediate data, time spent on device/host transference would become an important issue.

Providing scientists with a way to estimate how much processing time will be added in data overhead – given the amount of iterations and the interleave transfers – could allow them to estimate the best option for their time/architecture availability without having to try all the possible combinations, which could demand more effort than the performing the simulation itself.

In that sense, we have defined and tested a method to estimate the impact of data transmission vs. processing time in GPU-based simulations and N-Body algorithms. It could be evaluated for other types of GP-GPU algorithms since we were able to narrow it down to a bytes/Flop relationship. We estimate that it would only require to

calculate a data overhead relation – a constant for the algorithm –, and a data transmission cost – a constant for the device, as a metric for size in Flops. However, more testing on a diversity of algorithms and architectures should be performed in order to verify whether this relationship could be extrapolated.

The next step on this research will be focused in evaluating how other device performance counters could best allow us to estimate the costs of transmitting data, and how it could be optimized. Additionally, it will be necessary to determine how to estimate the transference overhead N-body algorithms ran in multiple device architectures or GPU clusters. Those cases hold much larger penalties for data transferences, and thus offer more challenges for data overhead estimation.

References

1. F. Diacu, "The solution of the n-body problem",*The Mathematical Intelligencer*, 18(3), 1996, 66-70.
2. P. E. Zadunaisky, "A method for the estimation of errors propagated in the numerical solution of a system of ordinary differential equations" *Proceedings from Symposium on The Theory of Orbits in the Solar System and in Stellar Systems*. August, 1964. Thessaloniki, Greece.
3. T. Nakayama, D. Takahashi,"Implementation of Multiple-Precision Floating-Point Arithmetic Library for GPU Computing". *Parallel and Distributed Computing and Systems*. December, 2011. Dallas, United States.
4. C. Gregg, K. Hazelwood, "Where is the data? Why you cannot debate CPU vs. GPU performance without the answer".*IEEE International Symposium on Performance Analysis of Systems and Software.* April, 2011. Texas, United States.
5. D.Mudigere,"Data access optimized applications on the GPU using NVIDIA CUDA".Master's thesis, *Technische Universität München.*October, 2009. Munich, Germany.
6. J.Nickolls, I. Buck, M. Garland, K. Skadron, "Scalable parallel programming with CUDA". *Queue - GPU Computing*, 6(2), 2008, 40-53.
7. J. Siegel, J. Ributzka, Li Xiaoming, "CUDA memory optimizations for large data-structures in the Gravit simulator". *International Conference on Parallel Processing Workshops*. September, 2009. Vienna, Austria.
8. R. G. Belleman, J. Bedorf, S. P. Zwart, "High Performance Direct Gravitational N-body Simulations on Graphics Processing Units". *New Astronomy*, 13(2), 2008, 103-112.
9. F. G. Tinetti, S. M. Martin "Sequential optimization and shared and distributed memory parallelization in clusters: N-Body/Particle Simulation." *Proceedings of Parallel and Distributed Computing and Systems*.November, 2012. Las Vegas, United States.
10. F. G. Tinetti, S. M. Martin, F. E. Frati, M. Méndez. "Optimization and parallelization experiences using hardware performance counters". *International Supercomputing Conference Mexico.* March, 2013. Colima, Mexico.
11. NVIDIA CUDA™ Programming Guide Version 5.0, NVIDIA Corporation, 2012.
12. NVIDIA CUDA™ Best Practices Guide Version 5.0, NVIDIA Corporation. 2012.
13. NVIDIA Nsight™ Visual Studio Edition 3.0 User Guide. NVIDIA Corporation. 2013.
14. G.L.M. Teodoro, R.S. Oliveira, D.O.G. Neto, R.A.C. Ferreira, "Profiling General Purpose GPU Applications". *21st International Symposium on Computer Architecture and High Performance Computing.* October, 2009. Sao Paulo, Brazil.