# Chapter 2
# Enhancing the Performance of the DCA When Forming and Solving the Equations of Motion for Multibody Systems

**Jeremy J. Laflin, Kurt S. Anderson and Mike Hans**

**Abstract** This chapter provides an initial investigation into using the Graphics Processing Unit (GPU) (or similar hardware) to execute the Divide-and-Conquer Algorithm (DCA), which forms and solves the equations-of-motion for articulated multibody systems. The computational time required to form and solve the equations-of-motion of a simple $n$-length pendulum using the GPU is compared with a standard serial CPU implementation, a rudimentary parallelization on the CPU using OpenMP, and some combinations of the CPU and the GPU. The hybrid version uses the GPU for a select number of levels in the recursive sweeps and uses an OpenMP parallelization on a multi-core CPU for the remaining levels of recursion. The results demonstrate a significant performance increase when the GPU is used despite recursive algorithms being ill-suited to hardware designed for Single Instruction Multi-Data (SIMD). This is largely due to the tree-type structure of recursive processes, with half of the required operations being contained in the first level of recursion for a binary tree.

## 2.1 Introduction

Since computational performance is critically important for simulations to be used as an effective tool to study and design dynamic systems, the computing performance gains offered by GPUs should not be ignored. The GPU has been used to increase the computational performance of many tasks necessary to simulate multibody systems [6, 10, 11, 15, 16]. Since the GPU is designed to execute a very large number of simultaneous tasks (nominally SIMD), recursive algorithms in general, such as the DCA, are not well suited to be executed on GPU-type architecture. This

J.J. Laflin (✉) · K.S. Anderson
Rensselaer Polytechnic Institute, Troy, NY, USA
e-mail: laflij2@rpi.edu

K.S. Anderson
e-mail: anderk5@rpi.edu

M. Hans
Jet Propulsion Laboratory, Pasadena, CA, USA
e-mail: Michael.A.Hans.Jr@jpl.nasa.gov

© Springer International Publishing Switzerland 2016

J.M. Font-Llagunes (ed.), *Multibody Dynamics*, Computational Methods in Applied Sciences 42, DOI 10.1007/978-3-319-30614-8_2

is because each level of recursion is dependent on the previous level. Therefore, all tasks associated with the algorithm cannot be executed independently. The primary issue is the amount of data transfer that must occur when moving from one level of recursion to the next. However, the GPU can be leveraged to increase computational performance when using the DCA to form and solve the equations-of-motion for articulated multibody systems with a large number of degrees-of-freedom due to the inherent tree structure of DCA.

Computational performance of dynamic simulations is highly dependent on the nature of the underlying formulation and the number of generalized coordinates used to characterize the system. Therefore, algorithms that scale in a more desirable (lower order) fashion with the number of degrees-of-freedom are generally preferred when dealing with large ($N > 10$) systems. However, the utility of using simulations as a scientific tool is directly related to actual compute time. The DCA, and other top performing methods, have demonstrated the desirable property of the required compute time scaling linearly with ($O(n)$) with the number of degrees-of-freedom ($n$) and sublinearly ($O(\log n)$) when implemented in parallel. However for the DCA, total compute time could be further reduced using parallel hardware, such as the GPU, by exploiting the large number of independent operations involved in the first few levels of recursion.

### *2.1.1 The DCA for Forming and Solving Equations of Motion for Multibody Systems*

The DCA was first introduced by Featherstone for both open-loop [5] and closed-loop [4] topologies and was notable for the level of coarse-grain parallelism it could achieve for unbranched systems. Additionally, there have been a number of modifications to the original method [1–3, 7–9, 13, 14, 17, 18]. The basic method is reproduced herein using the notation of Mukherjee and Anderson [12].

The DCA consists of two recursive processes: assembly and disassembly. These recursive processes take place using a hierarchical tree structure. The tree structure is defined by the kinematic joints connecting the bodies of the system. Typically the inboard and outboard joint of the body coincides with a reference point called a handle. The inverse inertial properties of two adjoining parent bodies are combined to represent a fictitious assembly (child body), see Fig. 2.1. This is possible because the kinematics of the joint are known, which allows the constraint forces acting at the connecting joint to be excluded from the equations-of-motion of the outboard handles of the parent bodies (the handles of the child body). The relative motion between the parent bodies is captured by an equation that describes the amount of motion happening in the directions of motion that are allowed by the joint, which are known. This process is repeated until there is only one body, the root body (see Fig. 2.2). At this point the boundary conditions are known and the equations-of-motion can be solved, which do not contain any of the constraint forces acting at the non-terminal joints.

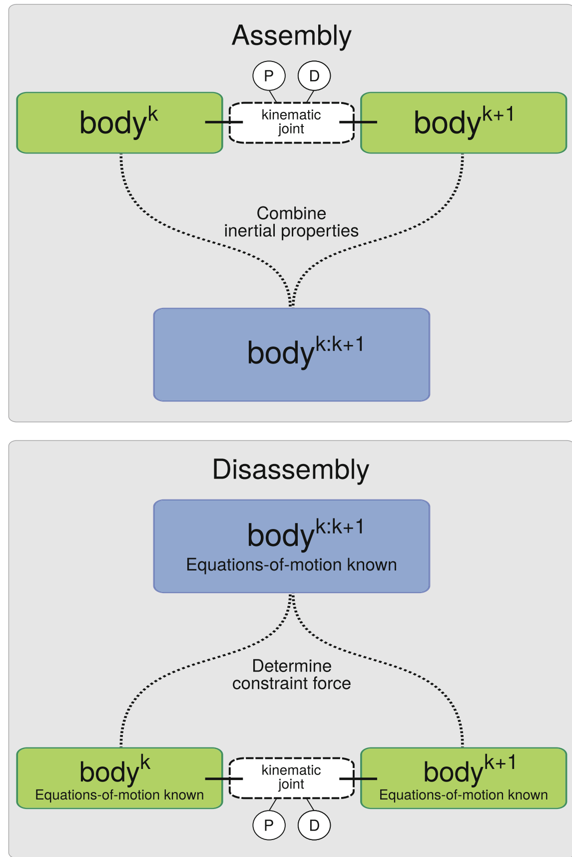


**Fig. 2.1** DCA kernel operations: These kernels both require that the motion allowed by the kinematic joint is mapped into a matrix containing the unit vectors of allowed motion (P), and a matrix containing the unit vectors of restricted motion (D)

In general, a body of the system may possess any number of handles, though the basic aspects of the method are most easily conveyed by discussing a chain system. For such systems, each body possesses two handles that connect the body to its inboard and outboard neighbors. The basic method involves writing the spatial equations-of-motion corresponding to these two reference points (handles) $H_i^k$ $i = 1, 2$ on each body, as

$$A_1^k = \zeta_{11}^k F_{1c}^k + \zeta_{12}^k F_{2c}^k + \zeta_{13}^k, \tag{2.1}$$
$$A_2^k = \zeta_{21}^k F_{1c}^k + \zeta_{22}^k F_{2c}^k + \zeta_{23}^k, \tag{2.2}$$

and

$$A_1^{k+1} = \zeta_{11}^{k+1} F_{1c}^{k+1} + \zeta_{12}^{k+1} F_{2c}^{k+1} + \zeta_{13}^{k+1}, \tag{2.3}$$
$$A_2^{k+1} = \zeta_{21}^{k+1} F_{1c}^{k+1} + \zeta_{22}^{k+1} F_{2c}^{k+1} + \zeta_{23}^{k+1}. \tag{2.4}$$

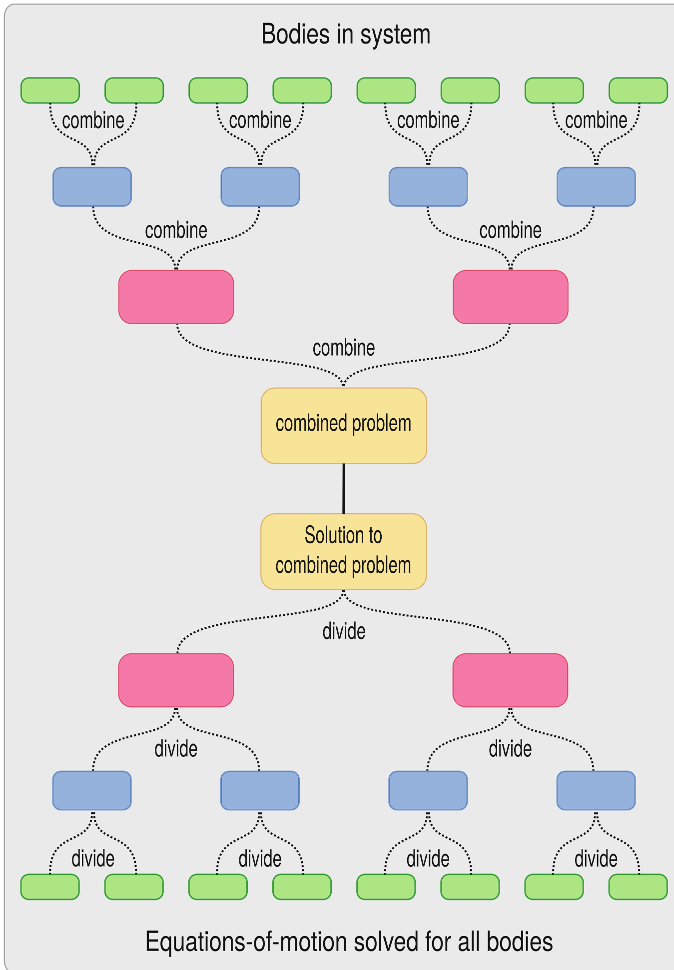


**Fig. 2.2** Divide-and-Conquer Algorithm

These bodies, $Body^k$ and $Body^{k+1}$, connected by a kinematic joint $j$ and therefore are subject to the kinematic constraint

$$P^j \dot{u} = A_1^{k+1} - A_2^k - \dot{P}^j u. \tag{2.5}$$

Here, the $P$ matrix is a property of the kinematic joint whose columns define the unit vectors that are aligned with the directions of motion permitted by the kinematic joint. Similarly, the $D$ matrix is that whose columns define the directions of motion

that are restricted by the joint. $A_i^k$, and $F_i^k$, are the spatial acceleration of, and force on, handle $i$ respectively and are defined as

$$A_i^k = \begin{bmatrix} \boldsymbol{\alpha}^k \\ \boldsymbol{a}_i^k \end{bmatrix}, \tag{2.6}$$

and

$$F_{ic}^k = \begin{bmatrix} \boldsymbol{\tau}_i^k \\ \boldsymbol{f}_i^k \end{bmatrix}. \tag{2.7}$$

The rotational acceleration of $Body^k$ is $\boldsymbol{\alpha}^k$, $\boldsymbol{a}_i^k$ is the translational acceleration of the reference point $H_i^k$, and $\boldsymbol{\tau}_i^k$ and $\boldsymbol{f}_i^k$ are the constraint torques and forces acting at $H_i^k$, respectively. The $\zeta_{ij}^k$ $(i, j = 1, 2)$ terms are the spatial matrix representations of the inverse inertial properties at the handles, while $\zeta_{i3}^k$ $(i = 1, 2)$ contains applied forces acting on the body and other velocity dependent terms. The resulting set of equations, Eqs. (2.1)–(2.4), can be reduced by exploiting the fact that the constraint forces are equal and opposite, i.e., $F_{2c}^k = -F_{1c}^{k+1}$, and that the kinematics of the connecting joint are specified. Specifically, Eq. (2.5) describes the relative acceleration between connecting bodies using the generalized acceleration $\dot{u}$ along known directions defined by the connecting joint partial velocity (mode of motion) $P^j$. The equations-of-motion for the assembled fictitious pseudo-body $Body^{k:k+1}$, at $H_1^k$ and $H_2^{k+1}$ can be expressed as

$$A_1^k = \zeta_{11}^{k:k+1} F_{1c}^k + \zeta_{12}^{k:k+1} F_{2c}^{k+1} + \zeta_{13}^{k:k+1} \tag{2.8}$$

and

$$A_2^{k+1} = \zeta_{21}^{k:k+1} F_{1c}^k + \zeta_{22}^{k:k+1} F_{2c}^{k+1} + \zeta_{23}^{k:k+1} \tag{2.9}$$

by algebraically eliminating the constraint forces at the connecting joint. The resulting Eqs. (2.8) and (2.9) are of the same form as the equations-of-motion for the handles of any generic body.

In the above equations, $\zeta_{ij}^{k:k+1}$ represents the inertial quantities of the fictitious pseudo-body resulting from the assembly of $Body^k$ and $Body^{k+1}$. For the derivation of the inverse inertial terms and the details of the assembly process, the reader is referred to the work of Featherstone [4] or Mukherjee and Anderson [12]. This assembly process is then repeated recursively, until only a single assembled pseudo-body remains (root body), as shown in Fig. 2.2. This is possible because the form of the equations-of-motion for the handles of an assembled body is indistinguishable from the form of the equations-of-motion for the handles of a generic body. The

assembly process yields the equations-of-motion associated with the two boundary handles

$$A_1^1 = \zeta_{11}^{1:n} F_{1c}^{1:n} + \zeta_{12}^{1:n} F_{2c}^{1:n} + \zeta_{13}^{1:n} \tag{2.10}$$

and

$$A_2^n = \zeta_{21}^{1:n} F_{1c}^{1:n} + \zeta_{22}^{1:n} F_{2c}^{1:n} + \zeta_{23}^{1:n}, \tag{2.11}$$

which are written in terms of only the spatial inertial quantities of all bodies in the system and the constraint forces acting at the two handles of the root body (boundary handles).

The spatial accelerations of, and constraint forces acting at $H_1^1$ and $H_2^n$ can now be determined using the known boundary conditions. After determining these quantities, the disassembly process begins, in which all unknown spatial accelerations of the handles and constraint forces acting at all connecting joints are determined. This recursive process determines the constraint forces acting at a joint in terms of the constraint forces acting at the handles of the assembly, and the inertial properties of the assembled body, as

$$F_{1c}^{k+1} = W\zeta_{21}^{k} F_{1c}^{k} - W\zeta_{12}^{k+1} F_{2c}^{k+1} + Y. \tag{2.12}$$

The terms $W$ and $Y$ are terms containing inertial properties from the assembly of the two bodies, see Featherstone [4], or Mukherjee and Anderson [12] for derivation of these terms. Once this constraint force acting at a joint is determined, the spatial accelerations of the handles that are connected by this joint can be determined using Eqs. (2.8) and (2.9). This allows the computation of the generalized acceleration ($\dot{u}$) at the joint using Eq. (2.5).

### *2.1.2 Potential Challenges Executing the DCA on the GPU*

Because this algorithm is by nature a recursive one, at first it seems ill-suited to run on the GPU architecture. This is because, by definition recursive algorithms can not be completely parallelized and require a minimum number of sequential operations. These sequential processes become problematic due to the potentially high cost of data transfer from the CPU memory to the GPU device memory. The lower connection speed is due to the physical arrangement of the CPU and the GPU and the type of connections used, which is continually improving.

The recursion becomes problematic, for the DCA, because the number of floating-point operations that need to be performed is low at the levels of recursion approaching the root body. Therefore, there is potentially a large amount of computational overhead in comparison to the number of floating-point operations that must be performed. This is contrary to the purpose for which GPUs were designed, which is

to produce a large amount of throughput. For this reason, it may seem that other algorithms which may scale more poorly with the number of bodies in the system, but can be more completely parallelized may achieve a lower compute time than the DCA or other recursive algorithms.

Although data transfer time is a significant source of overhead, there are others that are problematic for recursive algorithms, for example, kernel launch times and synchronization. Kernel launch time is particularly problematic for the DCA because the DCA consists of two operations, assembly and disassembly, which are each separate kernels and must be launched recursively. The recursive nature of the assembly and disassembly processes also necessitates that the kernels are synchronized at each level of recursion adding even more computational overhead.

### *2.1.3 Using the GPU Effectively With the DCA*

Despite the apparent drawbacks of executing the DCA on the GPU, there are ways the inexpensive and powerful computing resources offered by the GPU can be utilized with the DCA. For large systems, there are a large number of independent assembly and disassembly operations that occur in the levels near the leaf level of the recursive tree. For a system of 2048 bodies the approximate operations counts for each level of recursion are given in Table 2.1. By examining the number of kernel operations (assembly and disassembly) necessary at each level of recursion, it is estimated that 50 % of the operations per kernel sweep are performed in the first level and 75 % are performed in the first two levels of recursion. By performing only three of the twelve levels of recursion on the GPU, 94 % of all the operations are parallelized. However, performing only one level's kernel operations on the GPU results in 50 % of the operations per solve can be parallelized with the penalty of associated with only one kernel launch and no synchronization penalty. Furthermore, 75 % of the operations can be parallelized incurring the data transfer and synchronization penalties of only one level of recursion in addition to the leaf level operations. As the system's size increases, using the GPU to perform the operations in these levels could allow much larger problems to be investigated by effectively halving or quartering the original problem seen by any CPU parallelization strategy.

By using the GPU in only the levels that have a relatively high number of parallel operations, the computational overhead compared to the number of floating-point operations is kept low. The CPU can then be used only for the remaining levels in

**Table 2.1** Estimation of operations per level of recursion of the DCA

| Level | Leaf | Leaf+1 | Leaf+2 | Leaf+3 |
|---|---|---|---|---|
| Kernel operations | 1024 | 1536 | 1792 | 1920 |
| % Operations/Solve | 50 | 75 | 88 | 94 |

which there is a large number of sequential computations in comparison to a relatively low number of floating-point operations per level. In this way the GPU can be used to provide a large amount of throughput outweighing the overall overhead required.

Additionally, there are many ways that the data communication cost can be effectively hidden and the impact on overall simulation time can be minimized. For just one example of this consider a simulation of a bio-molecular system where coarse-graining has resulted in an articulated multibody system. In such an example, the data corresponding to the leaf level bodies can be transferred while computing the electrostatic forces.

Since the impact of the data communication cost associated with approximately 50 % of the operations can be mitigated, the penalty of using the GPU for two levels of recursion can be reduced to the cost of the data transfer between levels one and two ($\sim 1/4$ of the overall data), the synchronization between the two levels, and the second launch of the associated kernels.

Using the specifications of the known hardware it is possible to estimate whether or not using the GPU is beneficial. The flow-chart, shown in Fig. 2.3, demonstrates how this is performed for the DCA written as a recursive function. At each level of assembly an estimation is made whether the GPU or CPU is more efficient. This is repeated until all levels of assembly are completed, then the same estimation is made concerning the disassembly and each level of recursion is disassembled.

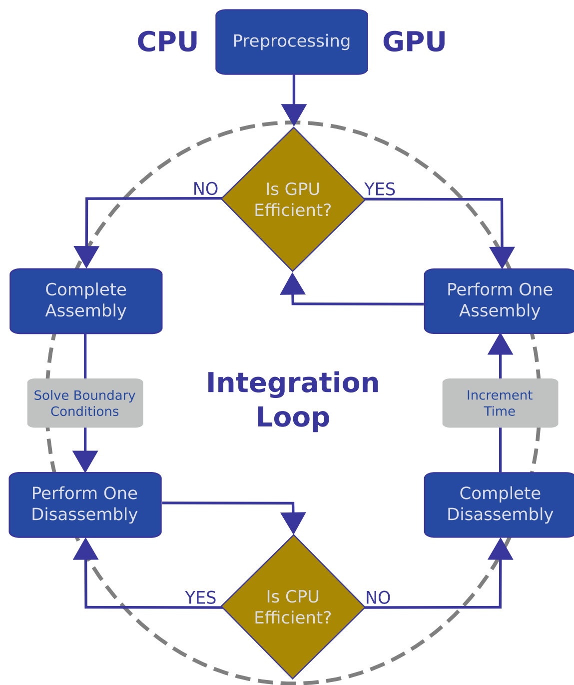


**Fig. 2.3** Using DCA on the GPU: This figure describes how the DCA, written as a recursive function, can be effectively implemented in a way that uses the GPU only when advantageous

## 2.2 Implementation and Numerical Example

A simple numerical example was performed to investigate the computational advantages of executing the DCA on GPU architecture. The equations of motion are formed and solved using the DCA in a variety of parallel arrangements. The DCA is implemented in serial and parallel on multi-core CPU using OpenMP to parallelize the assembly and disassembly operations happening at each level. Other lower level CPU parallelization implementations are possible and with more effort may yield faster results. However, as a platform to investigate the scalability of the algorithm on the GPU as compared to a parallel CPU implementation, OpenMP is sufficient. The CPU was a Intel® Xeon® W3565 3.20 GHz having four cores.

The number of levels of recursive assembly, and subsequent disassembly was selected a priori and arbitrarily. However, with some simple functions to measure the data transfer time and kernel launch time the optimal number of levels for a particular hardware combination could be approximately determined. This was not performed because the information is specific to the hardware arrangement and is of little interest to other users. The levels of the DCA that are performed using the GPU were executed on a NVIDIA® Corporation GF108GL Quadro® 600. Therefore, the NVIDIA® CUDA® platform was used for the parallel implementation of the assembly and disassembly kernels.

### *2.2.1 Test Case*

A multi-link pendulum, as shown in Fig. 2.4, was chosen as the test system to investigate how the computational time scales with the number of bodies using various approaches to GPU-parallelization. Although this example is $2D$, all matrices and vectors are "full-sized" as they would be in the $3D$ case so that the data communication cost is nearly the same. The number of bodies was varied up to 2048. The number of levels of recursion that were executed on the GPU was varied from zero to all levels. For 2048 bodies this corresponds to eleven levels of recursive operations. The pendulum was released from rest at $q_i = 0$ for all $i$, and the simulation duration was a single time-step.

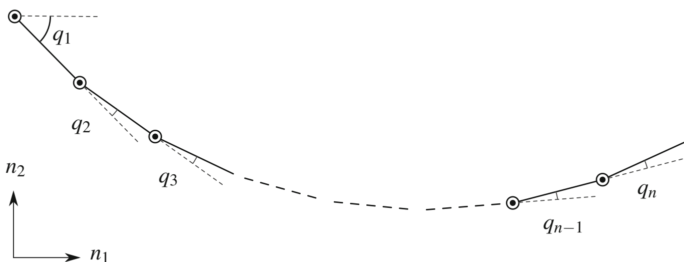


**Fig. 2.4** Variable length pendulum test problem

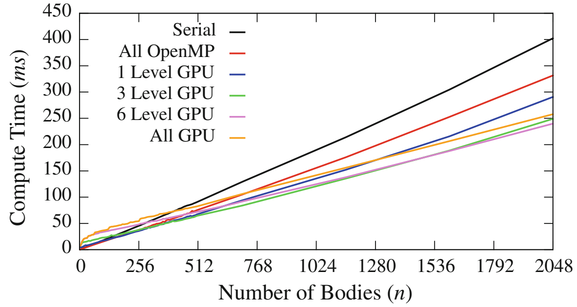


**Fig. 2.5** Full time-step: Serial run-time takes approximately 400 ms while the best hybrid solution takes approximately 240 ms. The OpenMP solution takes approximately 325 ms

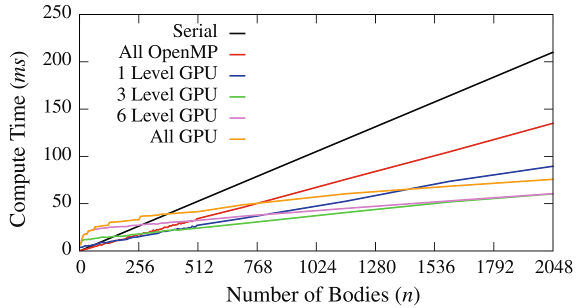


**Fig. 2.6** DCA solve time: Serial run-time takes approximately 210 ms while the best hybrid solution takes approximately 60 ms. The OpenMP solution takes approximately 140 ms

So that the effects of executing the DCA on GPU-type hardware can be better understood, the time to complete three different portions of the simulation are measured. These are the total time to complete one time-step of the simulation, the solve time of DCA, and the data transfer time. The total time to perform one time-step including all data transfer and operations needed to update the inverse inertial quantities for the next time step is shown in Fig. 2.5. The compute-time required to complete the assembly and disassembly sweeps, in which the equations of motion are formed and solved including determination of constraint forces acting at the joints, is plotted versus the number of bodies in the system in Fig. 2.6. Lastly, the time to transfer the necessary data to the device and then back to the host is shown in Fig. 2.7.

### *2.2.2 Results*

Figure 2.5 demonstrates a decrease in computational time as compared to OpenMP when executing the DCA on GPU architecture, even for a moderately large number of bodies. Additionally, this figure also shows that parallelizing the assembly and disassembly operations of the leaf level, provides the largest reduction in compute time for this arrangement of 2048 bodies.

As the number of bodies increases, executing more levels of recursion provides still further reductions in computational time, although at a decreasing rate. It can

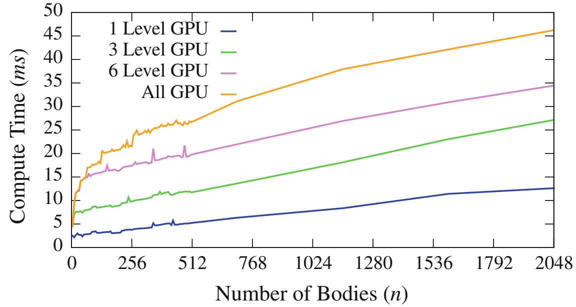


**Fig. 2.7** Data transfer time: Data transfer time takes approximately 46 ms for all levels of recursion, which compares to 60 ms for solve time and 240 ms for the full time-step

also be seen from Fig. 2.5 that executing all levels of recursion outperforms the OpenMP solution near 512 bodies, which would mostly likely be reduced with a more sophisticated NVIDIA® CUDA® implementation of the DCA assembly and disassembly kernels. Conversely, this number may be increased with a more advanced CPU parallelization implementation. Finally, it can be seen that the speed-up gained from parallelizing all levels on the GPU is approximately double to that of the OpenMP parallelization for 2048 bodies.

By comparing the DCA solve-time, shown in Fig. 2.6, with the total compute-time of one time step, it can be seen that the DCA is approximately half of the total time associated with each time-step. This could most likely be lowered with a more thoughtful implementation of overhead operations required. Therefore, significant reductions of the solve-time directly and meaningfully impact the overall simulation time, which will only be more notable if other overhead is reduced. The solve time for parallelizing all levels of recursion on the GPU is approximately half of the solve-time compared to the OpenMP parallelization.

Additionally, the time required to transfer the data to and from the device, shown in Fig. 2.7, is a significant portion of the solve time. When performing all levels of recursion on the GPU, it can be seen that the solve-time is nearly entirely composed of data transfer time. It can also be inferred from this figure that most of data transfer cost is associated with initiating the transfer since the increase in compute-time related to more bodies (data) is much less than the increase due to more data transfers.

## 2.3 Conclusions

Despite the obvious drawbacks to executing a recursive algorithm on GPU architecture, a significant reduction in compute-time was observed when doing so with the DCA. For this implementation, the solve-time is reduced in half and the total compute-time is reduced by approximately a quarter when comparing the OpenMP and the best hybrid solution. This reduction in compute-time is due, for the most part, to the large amount of independent operations that occur in the first levels of recursion. Although a hybrid implementation using parallel processes on both the

CPU and the GPU provide the lowest compute times for moderately sized systems, the penalty for using the GPU exclusively becomes less important relatively quickly. Therefore for very large systems, such as those encountered in biomolecular simulations where the number of degrees-of-freedom ($n$) easily exceeds $10^5$, a hybrid approach may not be worthwhile.

Although the total time reduction to simulate a time-step is not as large as the solve-time reduction, it is still significant. The 25 % reduction in total-time demonstrates that implementing the DCA on GPU-type hardware is a viable option to achieve significant reduction in compute-time at low-cost. It also suggests that the desirable solve-time scaling properties of the DCA (and other recursive methods) may be preserved while significantly reducing the actual solve time using relatively inexpensive resources.

Furthermore, with the new features available with state-of-the-art compute-capability, such as dynamic parallelism, the penalty associated with using the GPU can be dramatically reduced, if not eliminated. This is because with this new technology, threads can launch kernels and therefore the data does not need to travel back to the host between levels of recursion. Additionally, threads can be launched at varying degrees of resolution. This not only allows the obvious benefit of not having to communicate back to the host device, but also allows the parallelism to be performed in a more straightforward manner. For example, consider a topology of many branches including various closed loops and other complex structures. Each branch can be given to a thread which recursively launches other threads for each nested structure until the final level of threads launches the ideal number of kernels to parallelize the decomposed topology. This technology changes the way in which recursive algorithms are thought to perform on the GPU.

## References

1. Bhalerao KD, Anderson KS, Trinkle JC (2009) A recursive hybrid time-stepping scheme for intermittent contact in multi-rigid-body dynamics. J Comput Nonlinear Dyn 4(4):041010
2. Bhalerao KD, Poursina M, Anderson KS (2009) An efficient direct differentiation approach for sensitivity analysis of flexible multibody systems. Multibody Syst Dyn 23(2):121–140
3. Bhalerao KD, Crean C, Anderson KS (2011) Hybrid complementarity formulations for robotics applications. ZAMM-J Appl Math Mech/Zeitschrift für Angewandte Mathematik und Mechanik 91(5):386–399
4. Featherstone R (1999) A Divide-and-Conquer Articulated-body algorithm for parallel O(log(n)) calculation of rigid-body dynamics. Part 1:l. Int J Robot Res 18(9):867–875
5. Featherstone R (1999) A Divide-and-Conquer Articulated-body algorithm for parallel O(log(n)) calculation of rigid-body dynamics. Part 2: trees, loops, and accuracy. Int J Robot Res 18(9):876–892
6. Khude N, Stanciulescu I, Melanz D, Negrut D (2013) Efficient parallel simulation of large flexible body systems with multiple contacts. J Comput Nonlinear Dyn 8(4):041003
7. Malczyk P, Frczek J (2012) A divide and conquer algorithm for constrained multibody system dynamics based on augmented Lagrangian method with projections-based error correction. Nonlinear Dyn 70(1):871–889

8. Malczyk P, Mukherjee RM (2013) Parallel algorithm for modeling multi-rigid body system dynamics with nonholonomic constraints. In: Proceedings of the ASME 2013 international design engineering technical conferences, pp 1–9
9. Malczyk P, Frczek J, Cuadrado J (2010) Parallel index-3 formulation for real-time multibody dynamics simulations. In: Proceedings of the 1st joint international conference on multibody system dynamics. Lappeenranta, Finland
10. Mazhar H, Heyn T, Negrut D (2011) A scalable parallel method for large collision detection problems. Multibody Syst Dyn 26(1):37–55
11. Melanz D, Khude N, Jayakumar P, Negrut D (2013) A matrix-free NewtonKrylov parallel implicit implementation of the absolute nodal coordinate formulation. J Comput Nonlinear Dyn 9(1):011006
12. Mukherjee RM, Anderson KS (2006) Orthogonal complement based Divide-and-Conquer algorithm for constrained multibody systems. Nonlinear Dyn 48(1–2):199–215
13. Mukherjee RM, Anderson KS (2007) A logarithmic complexity Divide-and-Conquer algorithm for multi-flexible articulated body dynamics. J Comput Nonlinear Dyn 2(1):10
14. Mukherjee RM, Bhalerao KD, Anderson KS (2007) A divide-and-conquer direct differentiation approach for multibody system sensitivity analysis. Struct Multidiscip Optim 35(5):413–429
15. Negrut D, Tasora A, Mazhar H, Heyn T, Hahn P (2012) Leveraging parallel computing in multibody dynamics. Multibody Syst Dyn 27(1):95–117
16. Negrut D, Serban R, Mazhar H, Heyn T (2014) Parallel Computing in multibody system dynamics: why, when, and how. J Comput Nonlinear Dyn 9(4):041007
17. Poursina M, Anderson KS (2013) An extended Divide-and-Conquer algorithm for a generalized class of multibody constraints. Multibody Syst Dyn 29(3):235–254
18. Poursina M, Anderson KS (2013) Canonical ensemble simulation of biopolymers using a coarse-grained articulated generalized divide-and-conquer scheme. Comput Phys Commun 184(3):652–660. doi: 10.1016/j.cpc.2012.10.029, http://www.sciencedirect.com/science/article/pii/S0010465512003724

http://www.springer.com/978-3-319-30612-4

Multibody Dynamics
Computational Methods and Applications
Font-Llagunes, J.M. (Ed.)
2016, VIII, 321 p. 162 illus., Hardcover
ISBN: 978-3-319-30612-4