

# Accelerating implicit integration in multi-body dynamics using GPU computing

Jihyun Jung<sup>1</sup> · Daesung Bae<sup>1</sup>

Received: 15 November 2016 / Accepted: 14 July 2017 / Published online: 7 August 2017  
© Springer Science+Business Media B.V. 2017

**Abstract** A new direct linear equation solver is proposed for GPUs. The proposed solver is applied to mechanical system analysis. In contrast to the DFS post-order traversal which is widely used for conventional implementation of supernodal and multifrontal methods, the BFS reverse-level order traversal has been adopted to obtain more parallelism and a more adaptive control of data size. The proposed implementation allows solving large problems efficiently on many kinds of GPUs. Separators are divided into smaller blocks to further improve the parallel efficiency. Numerical experiments show that the proposed method takes smaller factorization time than CHOLMOD in general and has better operational availability than SPQR. Mechanical dynamic analysis has been carried out to show the efficiency of the proposed method. The computing time, memory usage, and solution accuracy are compared with those obtained from DSS included in MKL. The GPU has been accelerated about 2.5–5.9 times during the numerical factorization step and approximately 1.9–4.7 times over the whole analysis process, compared to an experimental CPU device.

**Keywords** Multi-body dynamics · Implicit integration · Nested dissection · Multifrontal method · Supernodal method · GPU

## 1 Introduction

The availability of modern CAE Software has increased thanks to the improvement of Computer System and Graphics Technology. Nowadays, CPUs for personal computers (PC) have approximately 100 Gflop/s (Giga Floating-Point Operation per Second), and high performance workstations achieve many hundreds Gflop/s. This figure was the performance of the fastest supercomputer only 20 years ago [1]. High performance computers have made it possible to solve bigger mechanical systems and their linear equations. However, research has shown that numerical factorization time increases explosively as the size of a linear equation increases.

---

✉ D. Bae  
[dsbae@hanyang.ac.kr](mailto:dsbae@hanyang.ac.kr)

<sup>1</sup> Department of Mechanical Engineering, Hanyang University, Ansan, Gyeonggi-do, 15588, Korea

The easiest way to resolve the increase of computing time is to replace the computer system with more powerful CPUs. However, advances in CPU speeds are slowing down as technological limits throttle higher performance. The CPU clock speed per core cannot be faster than a specific speed related to power consumption and temperature limits, which is called a clock limit. One way to eliminate the limits and improve the performance is to utilize additional computing devices like GPUs. The origins of CPUs and GPUs are different. The purposes of CPUs were to run an Operating System (OS) and to manage all programs on the OS. In contrast, GPUs were to represent real-time screens and game pictures more naturally and colorfully. Meanwhile, there have been a lot of attempts to graft the high computing performance of GPUs onto numerical computations for general purposes [2]. GPUs generally include hundreds or thousands of cores, which are highly suitable for data parallel algorithms. The GPU devices are used to obtain solutions from the matrices since they have a superior floating-point performance to CPUs.

There are two kinds of widely used methods to solve the linear equation  $\mathbf{Ax} = \mathbf{b}$ . One is an iterative method [3]. It consists of matrix–vector and vector–vector operations and could be applied to well-made sparse and dense BLAS routines. In addition, the parallelization for the iterative method is quite effective. These advantages fit well on additional computing devices, thus some researchers have studied a few kinds of iterative linear equation solvers on GPUs [4, 5]. NVIDIA-CUDA provides some sparse routines to support an iterative method by using cuSparse and cuBLAS Library [6]. ViennaCL was studied to research the solution of large systems of equations by means of iterative methods using optional pre-conditioners on various computing device like GPUs, MICs and CPUs [7]. These GPU-based iterative solvers have already been widely used in various fields like electronics [8], mechanics [9–11] and finance [12]. However, the indirect solvers become very slow when the matrix condition number is very high [13]. Mechanical dynamics usually handles high-stiffness problems, so the condition number of their matrices is extremely high. Therefore, convergence of the iterative methods for the mechanical system dynamics is very slow and it is not easy to find a proper pre-conditioner. These reasons have enforced us to use the direct method in the field of the mechanical system dynamics.

The direct method [14] consists of a finite number of floating-point operations so that it can always obtain an exact solution except for singular system matrices. In this research, a newly proposed multifrontal implementation is presented to maximize utilization of a GPU device. The method originated from carrying out assembly and Gaussian elimination of element matrices at the same time in the area of a finite element method [15, 16]. It has been widely studied and implemented in many large-scale finite element applications on CPUs [17]. The algorithm is relatively complicated compared to the iterative method due to reordering, fill-in and dynamic updatable sparse matrix structures. Also, it needs frequent copies of data between host and GPU memory spaces. For these reasons, studies on the direct method have been less active on GPUs especially. Davis applied CUDA acceleration to CHOLMOD and SPQR algorithms as parts of SuiteSparse linear algebra package [18, 19]. The two algorithms from SuiteSparse, however, are not appropriate to be used for a mechanical dynamic field since the CHOLMOD is a set of routine only for sparse positive definite matrices and the SPQR algorithm has a limitation of a matrix size. A lot of mechanical system matrices failed to carry out a GPU computation using the SPQR algorithm due to the lack of GPU memory size. Therefore, it was necessary to research and implement a linear equation solver using a new direct method for GPUs. Our purpose of this research is to implement a GPU-based direct linear equation solver and to optimize it to handle large mechanical system matrices.

This paper is organized as follows. Section 2 briefly summarizes a detailed numerical method for the equations of motion of constrained mechanical systems. Section 3 explains

traditional DFS-based and proposed BFS-based nested dissection reordering methods. Section 4 presents supernodal and multifrontal methods first, and then introduces a proposed numerical factorization method. Section 5 explains some features of a GPU device and how the proposed implementation can be applied and optimized for a GPU. Sections 6 and 7 present how to determine an optimum maximum block size for the proposed implementation and discuss the results. The mechanical dynamic experiments have been carried out using the proposed method with DSS routine included in MKL. The performance, memory usage and solution accuracy are discussed. Conclusions are drawn in Sect. 8.

## 2 Equation of motion

### 2.1 Constrained mechanical system and integration methods

The constrained mechanical system is often represented as differential-algebraic equation (DAE). A solution of DAE is more difficult than that of ordinary differential equation (ODE). There are two methods to solve DAEs [20].

One method is to carry out an explicit numerical integration and to correct the integration variables so the variables of position, velocity, and acceleration level are satisfied. An advantage of this method is that the system equations are small because the correction is conducted sequentially. However, it also has a disadvantage. The time step for very stiff problems tends to be very small.

The other is an implicit numerical integration method. It can overcome the disadvantage of the explicit method. Kinematic constraints including their derivatives and equations of motion are solved simultaneously. However, a disadvantage of the implicit method is that the size of a system matrix is larger than that of the explicit method. This research investigates the equation solver for the large matrices on many-core GPUs.

### 2.2 Implicit integration for differential-algebraic equations

The equations of motion for a constrained mechanical system are described as

$$\mathbf{v} - \dot{\mathbf{q}} = \mathbf{0}, \tag{1}$$

$$\mathbf{F}(\mathbf{q}, \mathbf{v}, \mathbf{a}, \boldsymbol{\lambda}) = \mathbf{0}, \tag{2}$$

$$\boldsymbol{\Phi}(\mathbf{q}, t) = \mathbf{0}, \tag{3}$$

where  $\mathbf{q}$  is the generalized coordinate vector in Euclidean space  $\mathbf{R}^n$ ,  $\mathbf{v}$  is the generalized velocity vector in  $\mathbf{R}^n$ ,  $\mathbf{a}$  is the generalized acceleration vector in  $\mathbf{R}^n$ ,  $\boldsymbol{\lambda}$  is the Lagrange multiplier vector for constraints in  $\mathbf{R}^m$ ,  $\boldsymbol{\Phi}$  represents the position level constraint vector in  $\mathbf{R}^m$ , and the Jacobian  $\boldsymbol{\Phi}_{\mathbf{q}} \in \mathbf{R}^{m \times n}$  is assumed to have full row-rank. Successive differentiations of Eq. (3) yield velocity and acceleration level constraints,

$$\dot{\boldsymbol{\Phi}}(\mathbf{q}, \mathbf{v}, t) = \boldsymbol{\Phi}_{\mathbf{q}}\mathbf{v} + \boldsymbol{\Phi}_t = \boldsymbol{\Phi}_{\mathbf{q}}\mathbf{v} - \mathbf{v} = \mathbf{0}, \tag{4}$$

$$\ddot{\boldsymbol{\Phi}}(\mathbf{q}, \mathbf{v}, \mathbf{a}, t) = \boldsymbol{\Phi}_{\mathbf{q}}\mathbf{a} + \frac{d}{dt}(\boldsymbol{\Phi}_{\mathbf{q}})\mathbf{v} + \boldsymbol{\Phi}_{tt} = \boldsymbol{\Phi}_{\mathbf{q}}\mathbf{a} - \boldsymbol{\gamma} = \mathbf{0}, \tag{5}$$

where  $\mathbf{v} = -\boldsymbol{\Phi}_t$  and  $\boldsymbol{\gamma} = -(\frac{d}{dt}(\boldsymbol{\Phi}_{\mathbf{q}})\mathbf{v} + \boldsymbol{\Phi}_{tt})$ . Equations (1) to (5) comprise a system of over-determined differential-algebraic equations (ODAE). An algorithm based on backward

differentiation formulas (BDF) to solve ODAE is described as

$$\mathbf{H}(\mathbf{x}) = \begin{bmatrix} \mathbf{F}(\mathbf{x}) \\ \ddot{\boldsymbol{\phi}} \\ \dot{\boldsymbol{\phi}} \\ \boldsymbol{\phi} \\ \mathbf{U}_1^T(h'\mathbf{R}_1) \\ \mathbf{U}_2^T(h'\mathbf{R}_2) \end{bmatrix} = \begin{bmatrix} \mathbf{F}(\mathbf{q}, \mathbf{v}, \mathbf{a}, \boldsymbol{\lambda}) \\ \boldsymbol{\Phi}_{\mathbf{q}}\mathbf{a} - \boldsymbol{\gamma} \\ \boldsymbol{\Phi}_{\mathbf{q}}\mathbf{v} - \mathbf{v} \\ \boldsymbol{\Phi}(\mathbf{q}, t) \\ \mathbf{U}_1^T(h'\mathbf{a} - \mathbf{v} - \boldsymbol{\zeta}_1) \\ \mathbf{U}_2^T(h'\mathbf{v} - \mathbf{q} - \boldsymbol{\zeta}_2) \end{bmatrix} = \mathbf{0} \tag{6}$$

where  $h' = \frac{h}{b_0}$ ,  $\boldsymbol{\zeta}_1 \equiv \frac{1}{b_0} \sum_{i=1}^k \mathbf{b}_i \mathbf{v}_{n-i}$  and  $\boldsymbol{\zeta}_2 \equiv \frac{1}{b_0} \sum_{i=1}^k \mathbf{b}_i \mathbf{q}_{n-i}$  in which  $\mathbf{k}$  is the order of integration, and the  $\mathbf{b}_i$  are BDF coefficients.  $\mathbf{x} = [\boldsymbol{\lambda}^T \mathbf{a}^T \mathbf{v}^T \mathbf{q}^T]^T$  and the columns of  $\mathbf{U}_i \in \mathbf{R}^{n \times (n-m)}$  ( $i = 1, 2$ ) constitute bases for the parameter space of the position and velocity level constraints. The matrices  $\mathbf{U}_i$  are chosen so that  $\begin{bmatrix} \boldsymbol{\phi}_{\mathbf{q}} \\ \mathbf{U}_i^T \end{bmatrix}$  has an inverse. Therefore, the parameter space spanned by the columns of  $\mathbf{U}_i$  and the subspace spanned by the columns of  $\boldsymbol{\phi}_{\mathbf{q}}^T$  constitute the entire space  $\mathbf{R}^n$ .

Equation (6) can be solved since the number of equations and the unknowns are the same. Newton’s numerical method can be applied to acquire the solution  $\mathbf{x}$ ,

$$\mathbf{H}_x^i \Delta \mathbf{x}^i = -\mathbf{H}^i, \tag{7}$$

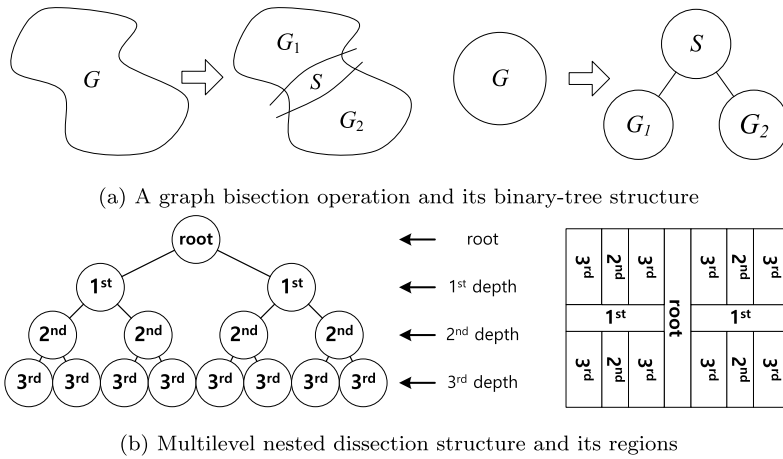
$$\mathbf{x}^{i+1} = \mathbf{x}^i + \Delta \mathbf{x}^i. \tag{8}$$

### 3 Nested dissection

Equation (7) is a typical linear equation  $\mathbf{Ax} = \mathbf{b}$  problem. It is necessary to factorize the matrix  $\mathbf{A}$  to obtain a solution  $\mathbf{x}$ . Four steps are needed to obtain the  $\mathbf{x}$  effectively: ① define a sparse matrix structure, ② reorder the matrix to obtain a permuted vector and symbolically factorize, ③ numerically factorize the matrix, and ④ acquire the solution with a right hand side by for- and backward substitutions. Among the four steps, the third numerical factorization step is usually the most time-consuming part. The reordering traversal in the second step has considerable impact on the performance of the third step. Therefore, the traditional nested dissection algorithm has been reviewed and then a new nested dissection algorithm is proposed.

When there are some connections among nodes composing a mechanical FE model, the data structure of the connections is defined as a graph ( $G$ ). In the graph data structure, a node is called vertex ( $V$ ) and its connection is called edge ( $E$ ) [21]. If there is a set of vertices which can be divided into two sub-graphs with the relatively same size, the set of vertices is called ‘separator’ and the division operation is defined as ‘graph bisection’ as shown in Fig. 1a. The graph bisection operation creates a typical binary-tree structure. The bisected sub-graphs ( $G_1, G_2$ ) are independent. However, each sub-graph depends on its parent separator. When the graph bisection operations are applied recursively, it is defined as ‘nested dissection’ [22]. Figure 1b expresses a tree and a graph region of nested dissection up to three depths as a representative model for this paper.

All sub-graphs including a root are called separator in this research. The highest one is called a root separator and the lowest ones are called leaf separators. The separators in the same depth are independent, while a separator and its higher depth separators have dependency.



**Fig. 1** Graph bisection and nested dissection

### 3.1 Traditional DFS-based nested dissection

The DFS post-order traversal has been traditionally used to obtain a binary tree for a graph. The tree traversal of the DFS post-order is as follows [21].

- (1) Move to a child separator until there is no lower child separator.
- (2) Mark the separator and move to the other children separators of my parent.
- (3) Move to and mark my parent separator if all children separators are visited.
- (4) Repeat processes from (1) to (3) until there are no more unvisited separators.

The multilevel tree of Fig. 2a shows the visited numbers of Fig. 1b vertices, based on the DFS traversal. Figure 2b depicts the region number of 2a, and Fig. 2c illustrates relations of nodes from 2a and regions from 2b as a matrix form.

When visiting a certain separator, all connected descendents of the separator must be already visited in the traversal. For example, the seventh separator in Fig. 2a has to be visited after visiting the first to the sixth separators. This post-ordering improves memory locality during numerical factorization [23]. The algorithm has been representatively implemented at the ‘METIS\_NodeND’ routine of METIS library [24].

### 3.2 Proposed BFS-based nested dissection

A proposed nested dissection in this research assigns the number based on the BFS reverse-level order traversal. The tree traversal of the BFS reverse-level order follows the rules below [21].

- (1) Move to a child separator until there is no more child separator.
- (2) Mark my separator and move to the other sibling separators.
- (3) Move to and mark a parent separator when all sibling separators are visited.
- (4) Repeat processes from (1) to (3) until there is no more unvisited separator.

The multilevel tree of Fig. 3a shows the visited numbers of Fig. 1b vertices, based on the BFS traversal. Figure 3b depicts the region number of 3a, and Fig. 3c illustrates relations of nodes from 3a and regions from 3b as a matrix form.

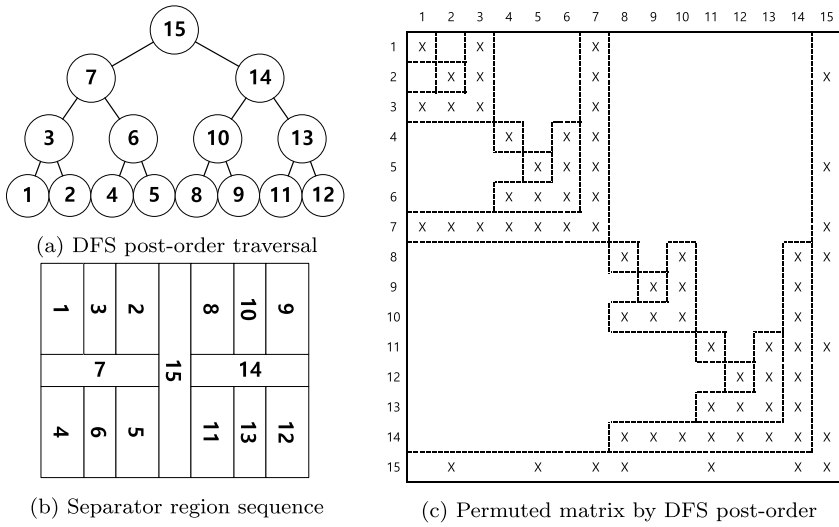


Fig. 2 Results of the DFS post-order traversal

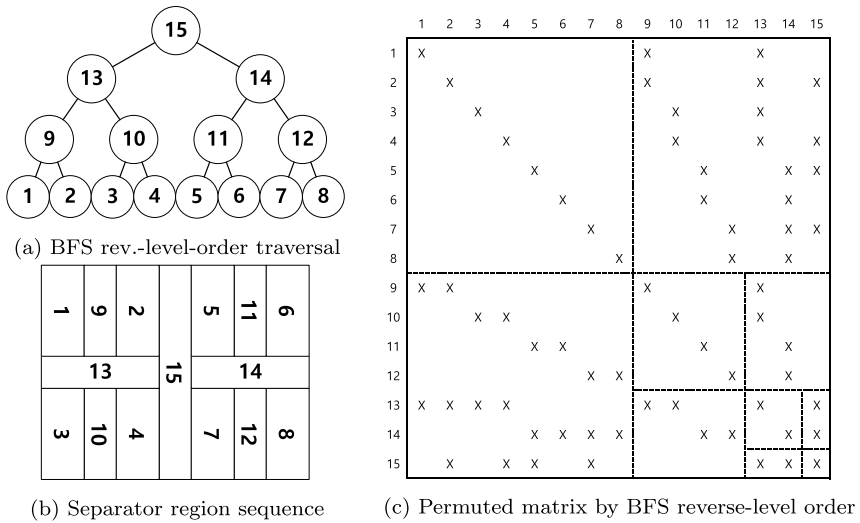
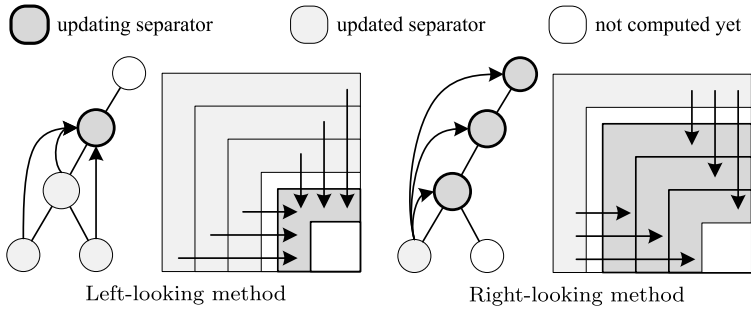


Fig. 3 Results of the BFS reverse-level order

Since a separator is visited after its lower depth separators, it is guaranteed that its sibling independent separators are located nearby. For example, all separators of the third depth in Fig. 3a must be visited before one of the separators from the ninth to the twelfth ones which belong to the second depth.

Although this rule has no effect on the number of non-zeros, compared to the traditional DFS-based nested dissection, it allows better parallelism and flexible control of required data size for an additional computing device. We developed this algorithm by recursively calling ‘METIS\_ComputeVertexSeparator’ routine in the METIS library [24]. The routine



**Fig. 4** Data access pattern for left- and right-looking methods

provides vertex indices of the separator and the two sub-graphs whenever there is a call. Therefore, it is not necessary to identify supernodes from a permuted sparse matrix as a post-process.

## 4 Numerical factorization

This section reviews the conventional supernodal and multifrontal methods and presents the benefits of the proposed method, based on the BFS traversal method.

### 4.1 The supernodal methods

The supernodal method commonly used to refer to left- and right-looking methods. The two methods equally consist of two sorts of operations. One is to factorize the diagonal block of a separator, and then for- and back-substitute the same row blocks of the separator, which is defined as ‘variable factor’ in this research. The other is to update a parent separator from its descendants, which is defined as ‘variable update’ in this research [25].

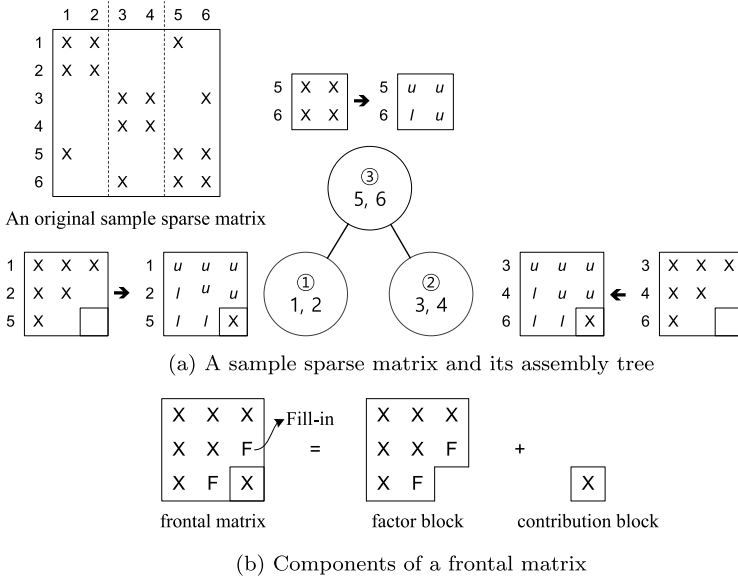
The left-looking methods start by applying variable updates from all descendant separators in the elimination tree to a separator before factorizing the separator. It delays the variable updates as much as possible. While right-looking methods factorize a local separator first, it performs the variable updates for ancestors of the local separator as fast as possible. Figure 4 depicts left- and right-looking as perspectives of a tree and a matrix.

The supernodal methods build a computational structure by performing symbolic factorization before actual numerical factorization. Since fixed memory location for the variable updates are used, the supernodal methods use less memory than the multifrontal methods do.

### 4.2 The multifrontal methods

The computational sequence of the multifrontal method is determined by an assembly tree structure. The actual computation is performed by combining adjacent columns using a supernodal technology. The multifrontal method is based on continuous Gaussian elimination of small dense matrices called a frontal matrix. The small dense matrices in a factorization process act as a vertices of the assembly tree.

In Fig. 5a, the adjacent (1, 2), (3, 4), (5, 6) nodes are combined by a supernodal technology and they are defined as ①, ② and ③ separators, respectively. It also shows the



**Fig. 5** A sample sparse matrix, its assembly tree and three memory spaces

corresponding frontal matrices and their relationships. A frontal matrix can be decomposed into two blocks as shown in Fig. 5b. One is a factor block consisting of eliminating variables, and the other is a contribution block composed of updating variables in the frontal matrix. Once conducting a numerical factorization of children frontal matrices, the updated contribution blocks are merged into their parent frontal matrices [26]. Therefore, the frontal factorization algorithm has a constraint that parent separators can be factorized out only after all children separators are factorized. For example, it is possible for the frontal matrices for ① and ② separators in Fig. 5a to be factorized at the same time, and then the updated contribution blocks can be merged into the parent separator ③. Due to the constraint, it requires additional temporal storage of stack structure to save the contribution blocks of children separators during a frontal numerical factorization [25].

The multifrontal method requires three kinds of memory spaces during a numerical factorization. The first space is used to store factored blocks; the second space is for saving contribution blocks of a stack structure; the third space is needed to operate a current frontal matrix [27]. The first space increases continuously during a factorization process and the third space is reused throughout the process. The second space stores the stacked contribution blocks. Since the stack size depends on the structure of assembly trees, it is not easy to estimate the exact size of the second space [25]. Figures 6a and 6b illustrate an assembly tree along with its frontal matrices of Fig. 2a and the memory transition as proceeding with the factorization steps. The multifrontal factorization steps are as follows,

- (1) Create a space to save a frontal matrix for a current separator.
- (2) Numerically factorize the current frontal matrix (see center of Fig. 6b) after merging contribution blocks from children frontal matrices.
- (3) Save the factored block in the factor saving space (see left side of Fig. 6b), the contribution block in the stacked space (see right side of Fig. 6b) and then release the current frontal matrix space.



- (4) Repeat steps from (1) to (3) until completing the numerical factorization of the root separator.

The memory size of factored blocks gradually increases as the numerical factorization proceeds, whereas the memory size of contribution blocks fluctuates irregularly as shown in Fig. 6c. The fluctuating memory usage often causes a failure of the numerical factorization due to an excess of available memory space [28].

### 4.3 The proposed implementation of the multifrontal method

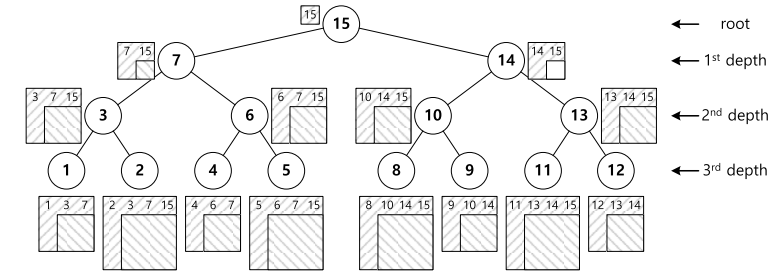
This research implements the multifrontal method with the BFS-based nested dissection. The global frontal operation has been carried out so that it could have more parallel opportunities than multifrontal methods could. The multifrontal method involves one separator with its straight children separators per a frontal matrix, but the proposed method is aimed at all separators in both groups in a multilevel tree.

The variable factor and update operations are treated as one set operation in the conventional implementation of the multifrontal methods. The proposed implementation method divides the set of the variable factor and update operations into two independent operations. Since all factor operations for all separators in the same depth are independent, they can be carried out in parallel, depending on their available memory and processors. Similarly, all update operations for ancestor separators can be done independently; they can be carried out in parallel as well. The independent nature of the proposed implementation method gives more flexible scheduling and parallelism.

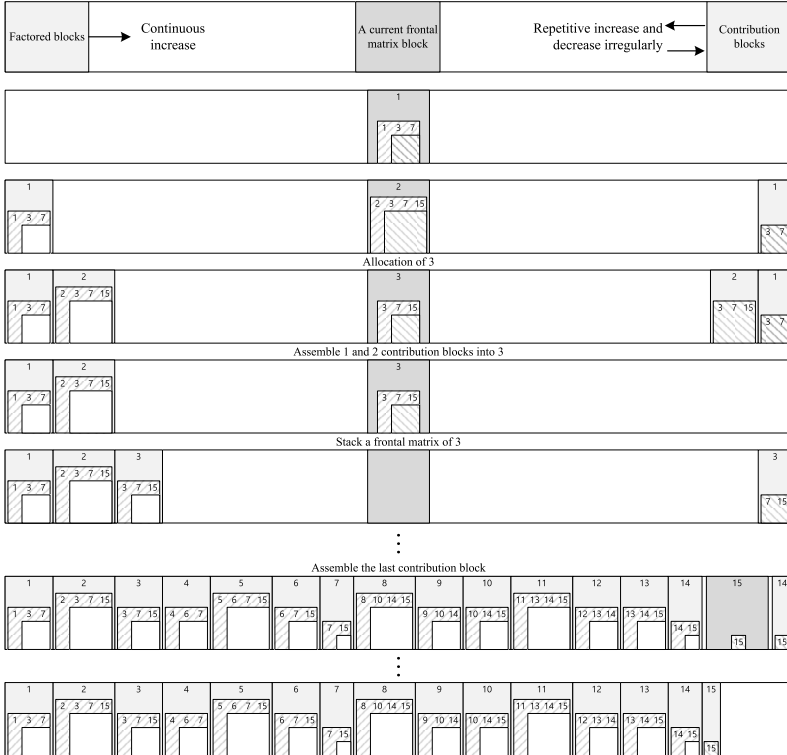
Figure 7 illustrates an operational sequence of the proposed method for the multilevel tree in Fig. 1b. The multilevel-tree separators of the first stage are divided into global factor and contribution groups. If the third depth separators are members of a global factor group, all upper separators are considered as those of a global contribution group. The update operations for the contribution group have to be done after carrying out variable factor operations for the factor group such as multifrontal methods. After the variable factor and update operations of the first stage, the separators of the third depth will not be involved in the rest of operations anymore. Therefore, the separators of the second depth become the next factor group and similar processes will be repeated as the second stage. This process will continue until reaching the root depth.

The frontal matrices of Fig. 8a are the same as those of frontal matrices of Fig. 6a. The only difference is the multifrontal factorization sequence. The form of the assembled blocks in Fig. 8b is almost identical to the factor and contribution blocks in Fig. 5b. The operation sequence of the proposed method is exactly the same as the BFS reverse-level order traversal in Fig. 3a. As a result, if a symbolic factorization is performed before an actual numerical factorization, it is possible to predict the necessary block data size and the locations including fill-in blocks. Thus, the required memory allocation can be done only once and it can be used until the completion of the factorization, as shown in Fig. 8c. This feature is highly similar to that of the supernodal method. The accurate prediction of the required data size makes it possible to discontinuing a numerical factorization due to a lack of host memory size. It could be a problem when the memory for the contribution blocks fluctuates irregularly [28] in the conventional implementation of the multifrontal methods.

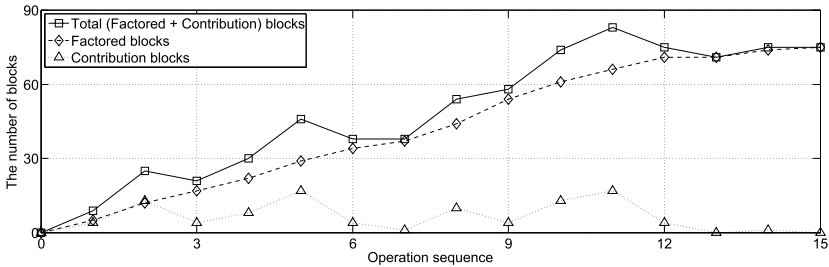
Separators in a global contribution group need descendent separators that have been already factorized in order to carry out variable update operations. Since the connectivity between variable factor and update operations has been divided, every separator of a global contribution group can independently refer to their connecting descendant separators. As a



(a) Assembly tree of Fig. 2a and its frontal matrices

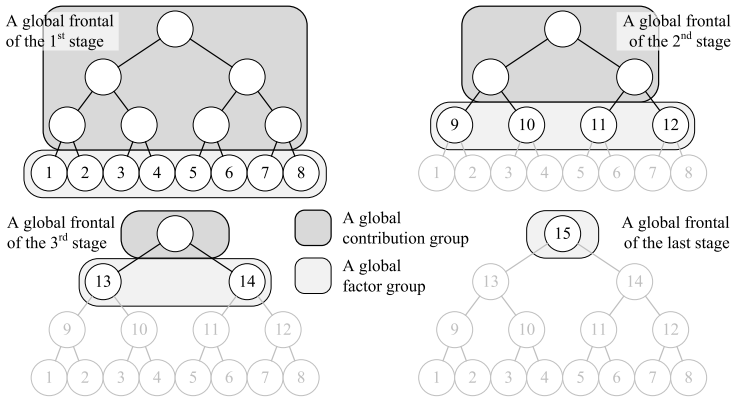


(b) Memory transition for the tree in 6a



(c) Memory usage in operation 6b

Fig. 6 Assembly tree and its memory transition using the original method



**Fig. 7** A proposed numerical factorization process

result, it brings more parallel opportunities by comparison with supernodal or multifrontal methods. Figure 9a shows an independent feature of each variable operation at the first stage.

The independent feature looks as if this method were an either left- or right-looking methods. The proposed method does not belong to the two methods because the separators in each variable operation are independent. Figure 9b compares the proposed method with supernodal methods.

The proposed method also makes it possible to adjust required data size adaptively. GPU devices have different memory sizes. Therefore, it is necessary to adjust an operational region to limit the data size for a GPU device during a numerical factorization. This research proposes to set priorities on sibling and depth directions to adjust the operational regions. Figure 10 describes the concept of deciding priorities on two directions.

The priority of the sibling direction allows controlling the size of a global factor group to be handled at a time, depending on the available memory size. Since the global contribution group is automatically determined by the factor group size, the computational algorithm is highly flexible. The priority of the depth direction controls only the size of a global contribution group. Therefore, the depth priority strategy is less effective in controlling the data size than the sibling direction is. The sibling direction is attempted first, and then the depth direction is followed for fine tuning in the actual implementation. The two priority strategies allow the proposed implementation to be highly adaptive for diverse GPU memory sizes. The same method can be implemented with the DFS-based nested dissection. However, it is not as efficient as the BFS method because the DFS-based method must visit a lot of other separators to identify the same depth, while it is important to seek sibling separators as fast as possible in the proposed strategies. The outstanding prediction and adjustability features of the proposed implementation method make scatter maps inside a GPU device unnecessary [18].

## 5 Implementation for a GPU device

This section reviews the features of a GPU and a CPU. Both the original and the proposed multifrontal methods are considered for their implementations on a GPU device. The advantages of the proposed implementation are presented. In addition, optimization strategies are presented to maximize the performance on a GPU device.

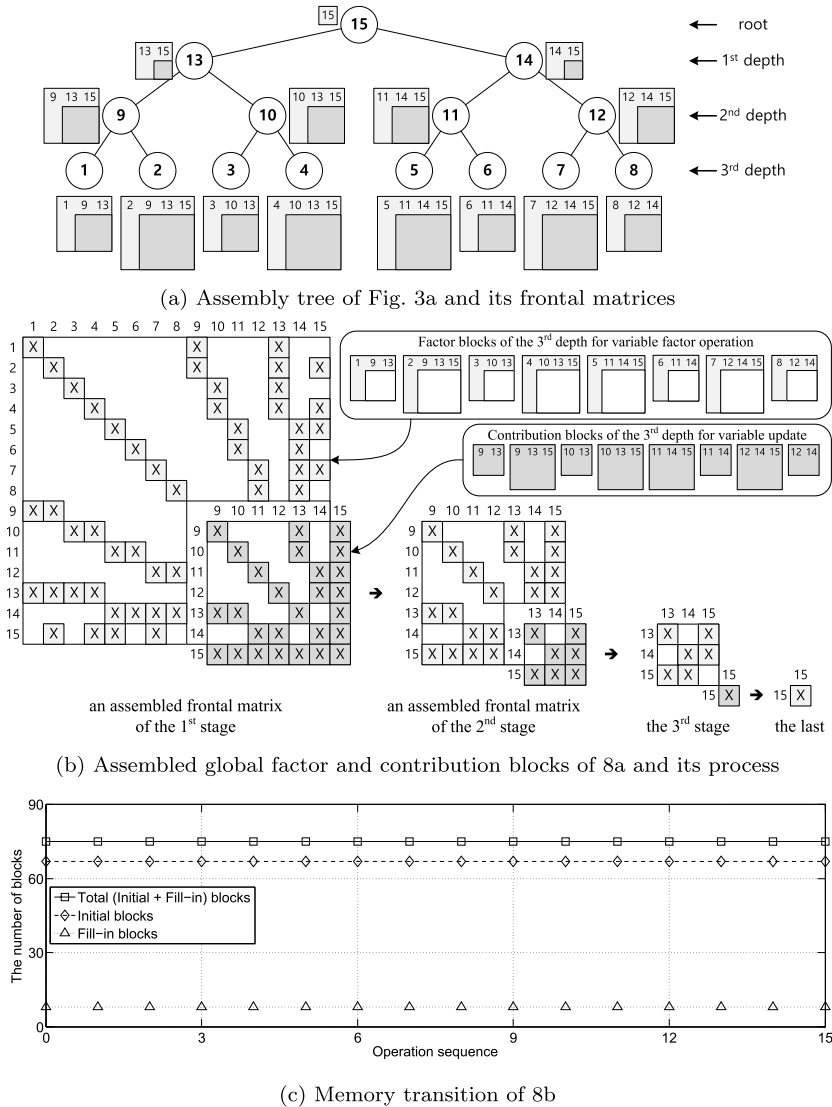
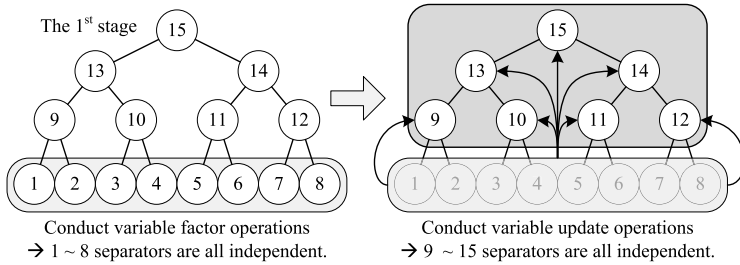


Fig. 8 Multifrontal factorization process using the proposed method

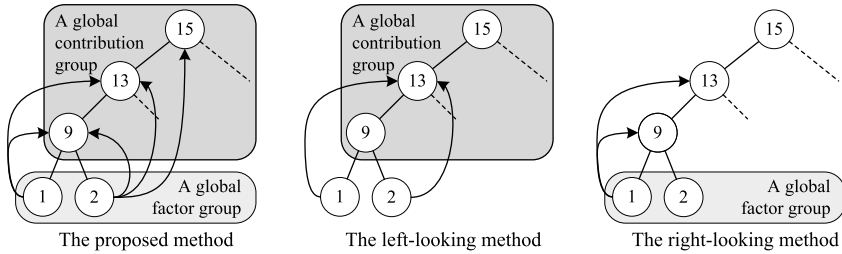
### 5.1 Characteristic of a GPU device

The hardware specifications used for this research are shown in Table 1.

The memory size of a GPU is smaller than that of a host side in most systems. The experimental computer system has 128 GB of memory while the GPU device contains only 6 GB. This difference between the host and the GPU memory size requires frequent data transfers. Moreover, the speed of PCI-Express link (12 GB/s) between CPU and GPU is much slower than that of inside communication in each computing device (CPU: 43.6 GB/s, GPU: 336 GB/s). Thus, the slow link speed has to be considered in maximizing the performance of the GPU.



(a) Numerical factorization on the 1st stage



(b) Comparison of the proposed method and supernodal methods

**Fig. 9** Factorization process on the first stage and comparison with supernodal method

**Table 1** Hardware specifications

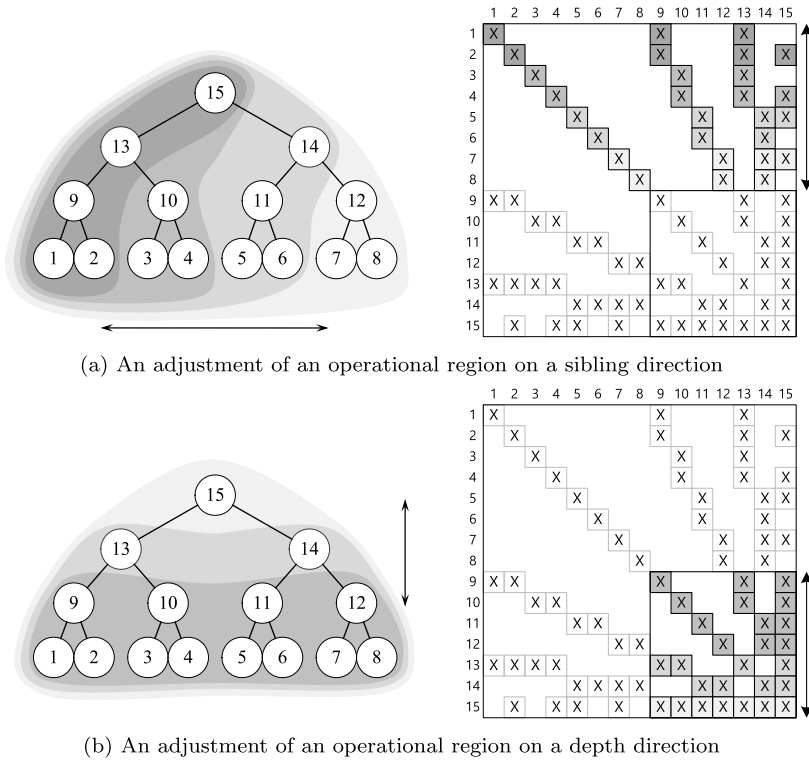
| Item        | Name  |
|-------------|---|
| CPU         | Intel Xeon E5-2609v2 2.5 GHz 4 cores          |
| Memory      | Samsung PC3-12800 16 GB × 8                   |
| Motherboard | ASUS Z9PE-D16                                 |
| GPU         | NVIDIA GeForce GTX TITAN BLACK D5 6 GB HYBRID |

The experimental GPU device, NVIDIA GeForce GTX TITAN BLACK, has Kepler GK110 architecture. A fully enabled Kepler GK110 consists of 15 SMX units. Each SMX unit includes 64 cores for double precision. Because mechanical dynamics solutions require considerably more precise solutions, the double precision data type is used for this research. Therefore, there are total 960 cores for double precision operations in the experimental GPU device [29]. Theoretical performance of a computing device can be estimated by multiplying the number of cores, core clock speed, SIMD and FMA. The theoretical value of the experimental CPU has 80 Gflop/s and the GPU is 1931.52 Gflop/s.

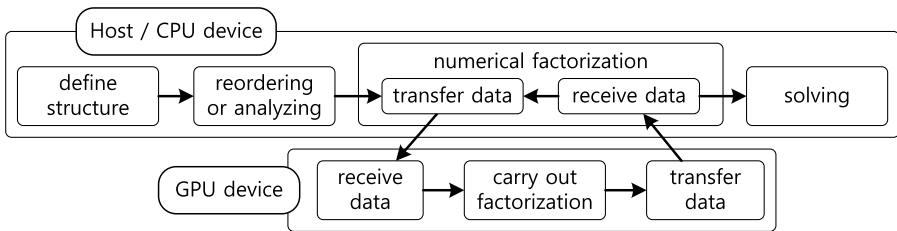
### 5.2 Application of a GPU device

The linear equation solver for a sparse matrix was divided into four steps of defining matrix structure, reordering or analyzing, numerical factorization and solving. The numerical factorization is the most time-consuming step among the 4 steps. A GPU device can be used to assist the numerical factorization step in this research, as shown in Fig. 11.

As a view of memory management, it has been presented that the multifrontal method needs a flexible stack structure during runtime. Since the stack structure is required to have



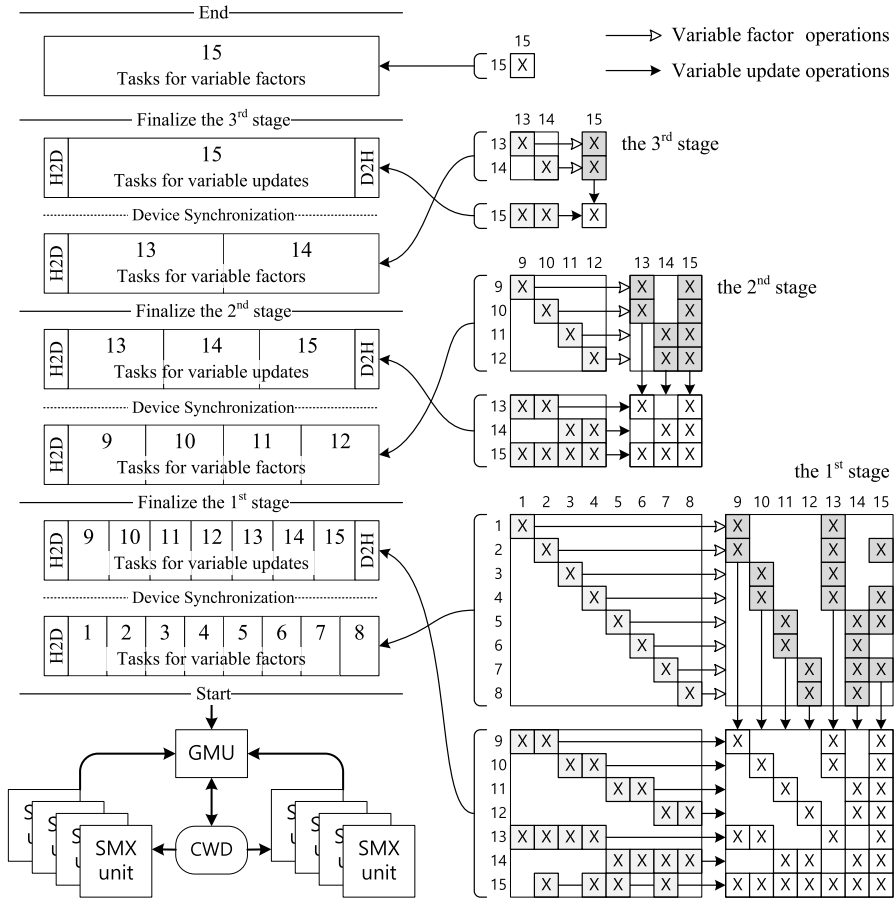
**Fig. 10** Two kinds of priorities to adjust operating regions



**Fig. 11** Roles of CPU and GPU devices for linear equation solver

frequent memory operations for contribution blocks, the parallel efficiency on GPUs of the conventional multifrontal method is poor. In addition, the fluctuating memory usage may cause a GPU computation failure. Because of these drawbacks, the multifrontal method is not suitable for programming on GPUs. SPQR in SuiteSparse introduces a way to conduct an assembly and computation of each frontal matrix in a GPU device. In case of handling a very large frontal matrix, the library splits the trees into sub-trees within the GPU memory size. However, this attempt is sometimes not enough to prevent a failure from exceeding device memory size [19].

As a view of parallel execution, it is necessary to understand principles of a parallel methodology of a GPU device. Once tasks are added into each stream queue, the GMU



**Fig. 12** An actual workflow of the proposed implementation in Fig. 8b

(Grid Management Unit) newly introduced in Kepler GK110 architecture of a GeForce TITAN BLACK manages and prioritizes tasks to be executed. The GMU communicates with a CWD (CUDA Work Distributor) via a bidirectional link and also has a directional connection with SMX units to launch additional tasks via Dynamic Parallelism on the GPU [29].

Figure 12 expresses a workflow of Kepler architecture and the sequence of queuing tasks with respect to variable operations for all depths in Fig. 8b. The proposed method consists of a set of variable factor and update operations. All independent separators associated with the same depth are added to each stream queue and synchronize them until all operations are finalized. Once variable factor operations are completed and synchronized, all ascendant separators are scheduled to conduct variable update operations by referring prior separators already stored in the GPU device. This process will be repeated until reaching a root separator. Since there is no data dependency among the variable factors for the same depth, it is possible to parallelize the variable factor operations for all separators of the same depth. However, the computing time of the variable factors is generally even less than that of the variable updates. Therefore, it is very important to well-parallelize the variable update operations to obtain a good parallel performance.

Update operation for a separator must receive data from their descendant separators which are already factorized at the previous variable factor operation and already has resided inside a GPU device. Besides, the variable update operations are independent of all separators.

Since the GPU has only a limited memory space, two cases must be considered depending on the required data size. The first case is when the required data size is equal to or less than an available GPU memory size. In this case, all data can be transferred from the CPU to the GPU at once at the beginning and the results can be transferred back at the end, which is called 'FULL' version in this research. The second case is when the required data size is bigger than the GPU memory size. The whole data cannot be transferred at once, so that the data must be divided into data smaller than an available GPU memory size. Only a part of all possible separators for variable factors and their associated ascendant separators for variable updates are handled at each time, which is called 'PARTIAL' version in this research.

The version type is decided right after a required data size is estimated from a symbolic factorization. The 'PARTIAL' version uses the two priority strategies in deciding partial operational separators. And then, computing commands are created just once and saved at the host memory space. If there is enough GPU memory space, the proposed implementation needs two operational synchronizations per one depth. Thus, it is required to synchronize the computing commands at least twice the number of deepest multi-tree depth during numerical factorization process. The number of actual synchronizations may increase in accordance with GPU memory size.

### 5.3 Optimization of a GPU device

The 'FULL' and 'PARTIAL' versions are presented in the previous section. This section presents how to implement the 'PARTIAL' version effectively. The 'PARTIAL' version requires frequent data exchanges to synchronize data. Since the variable factor and update operations are dependent of each other, any other numerical operation cannot be conducted while transferring data to GPU device. The PCI-Express link speed is very slow, compared to those of inside communications.

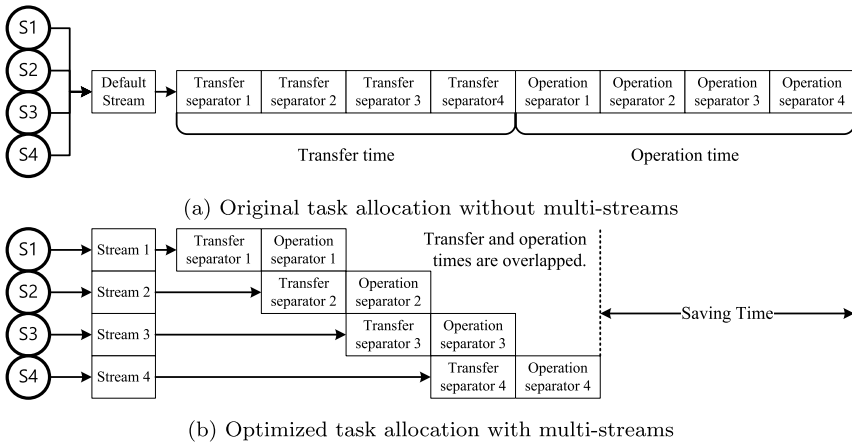
In order to overcome the slow transfer speed in using a GPU, the multi-streams of CUDA are applied. The main feature of the multi-streams is to divide large data to be transferred to some smaller data units. If a set of transfer task of a small data unit and its operation task are added to each stream, a GPU device executes transfer and operation tasks simultaneously from different streams in hardware level [30]. This makes it possible to compute huge data while minimizing time loss caused by slow transfer speed of PCI-Express link and to assign tasks continuously on GPUs with restricted memory space. One stream is equivalent to one separator in this research. Figure 13 explains a principle of multi-streams and its application for the proposed multifrontal method.

Note that it is recommended that the time to transfer data is almost equal to or smaller than the computing time on GPUs. Otherwise, the computing performance may deteriorate due to the waiting time. Therefore, the data size and its complexity of separators are very important to have a good parallel performance on a GPU device. This will be discussed in the next section.

## 6 Division of separators

A multilevel tree obtained by a nested dissection algorithm generally contains separators with different sizes. The root separator is the largest one in numerous cases and the children





**Fig. 13** Partial factorization of the proposed multifrontal method

separators tend to be smaller than their parent. The difference in separator sizes causes an unbalanced overlapping between data transferring and computing time. This approach is similar to ‘tile algorithm’ which is widely used across linear algebra libraries. The tile algorithm shows outstanding computing performance on homogeneous multi-core CPUs [31] and many-core GPUs [32] as well as heterogeneous systems [33, 34]. This section presents the effects of the division of the separators and how to determine an optimum maximum block size.

### 6.1 Experimental models

The flops of variable factors and updates as well as the ratio of them are important in developing an efficient computing algorithm. Table 2a summarizes some BLAS and LAPACK routines used for the numerical factorization step and their approximate flops [35]. Several sparse matrices are selected from University of Florida Sparse Matrix Market to show the effects of the separator division. Though some of original sparse matrices are symmetric and contain only a lower side, an upper side from the original lower side has been filled for this experiment. Table 2b shows substantial flops of the proposed method without division of separators for each model. Note that the variable update operation takes about 80–90% of the total computation.

### 6.2 Effects of division

As a view of memory usage, division of separators acts as restraining the area of the arithmetic operations. Large adjacent dense blocks usually intervene in a variable update operation with original separators. The same operation with divided separators presents that the operational regions decline as well as the smaller blocks are involved in the operations.

Figure 14 shows the effects of the separators.  $A_1$  has two adjacent separators  $A_5$  and  $A_7$  in Fig. 14a. The variable update operation of  $A_1$  has two adjacent separators ( $A_5, A_7, A_{57}$  and  $A_{75}$ ) in blocked matrix. Meanwhile, the variable update of smaller separators from  $^1A_1$  to  $^4A_1$  in Fig. 14b updates only a small part of the same region.

As a view of parallel efficiency, the division of separator hides transfer time of required data from host to devices more efficiently. Figure 15 shows how much efficiency can be

**Table 2** BLAS, LAPACK routines and operation counts of various models

(a) Routine names and their number of operations

| Oper. type      | Package/routine   | Floating-point operation count   |
|-----------------|---|--|
| Variable Factor | LAPACK ( dgetrf ) $\mathbf{A} = \mathbf{LU}$                              | $mn^2 - \frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{5}{6}n$ (where $m$ and $n$ are, respectively, the row and column sizes in $\mathbf{A}$ ) |
|                 | LAPACK ( dgetrs ) $\mathbf{AX} = \mathbf{B}$                              | $nrhs(2n^2 - n)$ (where $n$ is the row size of $\mathbf{A}$ , $nrhs$ is the column size of $\mathbf{B}$ )                                |
| Variable Update | BLAS ( dgemm ) $(\mathbf{A} \times \mathbf{B}) + \mathbf{C} = \mathbf{C}$ | $2mnk$ (where $m$ is the row size of $\mathbf{C}$ , $n$ is the column size of $\mathbf{C}$ and $k$ is the column size of $\mathbf{B}$ )  |

(b) Ratios of variable factor and update Gflops without division of separators

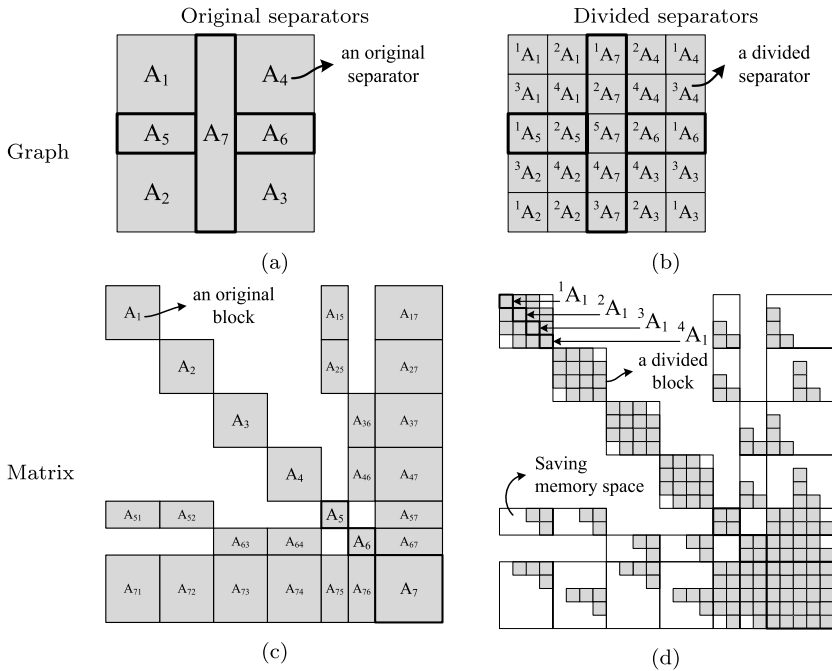
| Model name      | Rows/cols | Non-zeros  | Var. fac. (Gflops/%) | Var. upt. (Gflops/%) |
|-----------------|-----------|------------|----------------------|----------------------|
| Trefethen_20000 | 20,000    | 554,466    | 369/13.37%           | 2,390/86.63%         |
| consph          | 83,334    | 6,010,480  | 115/7.27%            | 1,464/92.73%         |
| torso3          | 259,156   | 4,429,042  | 201/6.18%            | 3,047/93.82%         |
| F1              | 343,791   | 26,837,113 | 191/7.57%            | 2,331/92.43%         |
| audikw_1        | 943,695   | 77,651,847 | 4,767/7.78%          | 56,518/92.22%        |
| Fault_639       | 638,802   | 28,614,564 | 6,629/5.84%          | 106,805/94.16%       |

improved by dividing the separators in parallel processing the numerical factorization. Figure 15a is a case of an original frontal matrix with coarse-grained blocks, and Fig. 15b is a case of a divided frontal matrix into a specified size with fine-grained blocks. The undivided separators may create a small number of non-uniform variable update operations. Meanwhile, the divided separators may create a larger number of uniform separators than coarse-grained. The GPU computation always involves data copy between host and device. The non-uniformed data sizes and their operations cause delays of GPU computing and poor parallel performance. Figure 15c obtained from NVIDIA visual profiler (nvvp) illustrates two computational timelines with respect to Figs. 15a and 15b, respectively. The ocher bars are data copy tasks from host to device and the blue bars describe matrix multiplication operations. The two timelines have the same flops and time-scale under a multi-streams environment.

The bottom line is that since the overlapping time in coarse-grained frontal matrix is not well-fitted, there will be a drop in parallel efficiency. The fine-grained timeline, however, shows that most of computing and copy-task time are overlapped together. The variable update operations take up most of the computing time; the division of separators is highly important in GPU computation.

### 6.3 Optimum maximum block size

The previous section has shown that division of separators improves the parallel efficiency. A block size must be decided to achieve the most efficient parallelism. The factorization time, memory usage and flops have been measured to identify the effects of the maximum block size. Table 4 shows the measured numerical values with numerical factorization type for models in Table 2b when the block size is changed from 64 to 4096. And the same kinds of results from other linear solver routines on CPU and GPU devices are also appended to Table 4. The additional figures are obtained from GPU versions of CHOLMOD, SPQR



**Fig. 14** Fill-ins before and after the division of separators into a few blocks

**Table 3** Software specifications and reordering algorithms

(a) Detailed software specifications

| Item           | Name                             |
|----------------|----------------------------------|
| OS             | CentOS 7.0 x86_64                |
| CUDA Version   | CUDA 7.5                         |
| C/C++ Compiler | Intel C++ Compiler 15.0 Update 6 |
| BLAS Library   | cuBLAS in CUDA 7.5               |
| LAPACK Library | Intel MKL version 11.2 Update 4  |
| DSS Routine    | Intel MKL version 11.2 Update 4  |
| SuiteSparse    | 4.5.3                            |

(b) Reordering algorithms in each linear solver routine

| Routine | CHOLMOD      | SPQR         | DSS   |
|---------|--------------|--------------|-------|
| Reorder | AMD or METIS | SPQR Default | METIS |

included in SuiteSparse and DSS in MKL [36]. The software specifications and reordering algorithms for each linear solver routine are tabulated in Table 3.

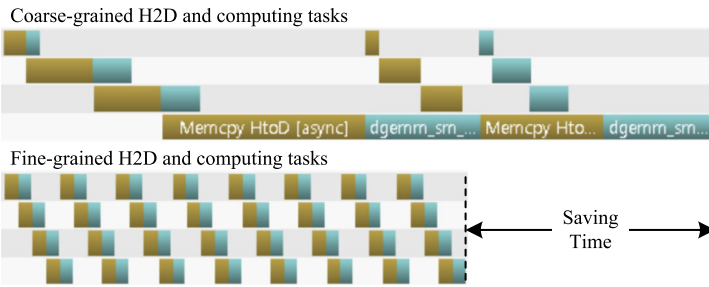
The number of floating-point operations and the amount of peak memory tend to grow up constantly as the maximum block size increases. The computing time, however, decreases to a certain block size and increases again. The size for the minimum computing time has

|          |           |           |
|----------|-----------|-----------|
| $LU_1$   | $A_{12}'$ | $A_{13}'$ |
| $A_{21}$ | $A_2'$    | $A_{23}'$ |
| $A_{31}$ | $A_{32}'$ | $A_3'$    |

(a) a coarse-grained frontal matrix

|              |               |               |               |               |
|--------------|---------------|---------------|---------------|---------------|
| $LU_1$       | $A_{12}'$     | ${}^1A_{13}'$ | ${}^2A_{13}'$ | ${}^3A_{13}'$ |
| $A_{21}$     | $A_2'$        | ${}^1A_{23}'$ | ${}^2A_{23}'$ | ${}^3A_{23}'$ |
| ${}^1A_{31}$ | ${}^1A_{32}'$ | ${}^1A_3'$    | ${}^{12}A_3'$ | ${}^{13}A_3'$ |
| ${}^2A_{31}$ | ${}^2A_{32}'$ | ${}^{21}A_3'$ | ${}^2A_3'$    | ${}^{23}A_3'$ |
| ${}^3A_{31}$ | ${}^3A_{32}'$ | ${}^{31}A_3'$ | ${}^{32}A_3'$ | ${}^3A_3'$    |

(b) a find-grained frontal matrix



(c) Timelines of the coarse- and find-grained computing with H2D copy

**Fig. 15** Improvement of parallel efficiency by division of separators

been found to be in the range of 512–1024. Other matrices have been numerically factorized and have shown a similar behavior. Though the optimum block size of relatively smaller matrices has been found to be 512, the time difference between 512 and 1024 is not big. As a result, the optimum maximum block size is 1024 in this research. However, this block size cannot be applied when a GPU does not have enough memory space to fully store two block rows. In this case, the maximum block size must be reduced so that it does not exceed the GPU memory space.

Although these research results named MFS (MultiFrontal Solver) do not always draw the best performance, most numerical factorization times are similar to or faster than those of CHOLMOD on the same experimental GPU device. Meanwhile, the SPQR sometimes produces wrong consequences of NaN (Not a Number) or fails showing a ‘GPU memory too small’ error message.

## 7 Numerical experiments

Dynamic analysis of three flexible mechanical models has been carried out for 1 second with 100 output step by using the proposed method with a GPU. Figure 16 shows experimental model shapes and nodes to verify analysis results. Each node represents position of the maximum Von Mises stress value among all nodes over the whole analysis process for each model.

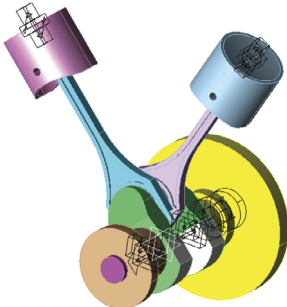
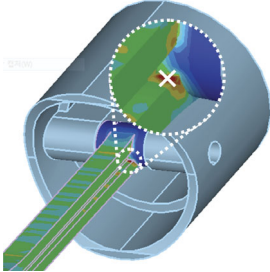
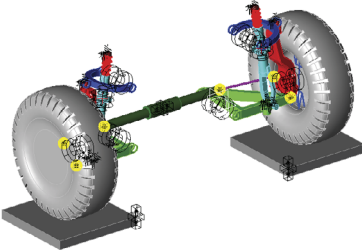
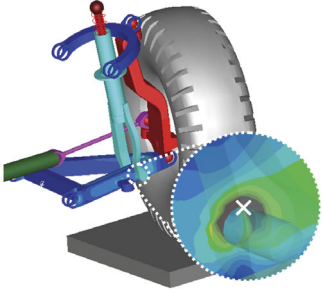
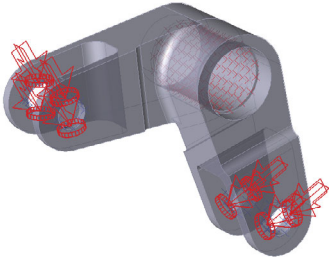
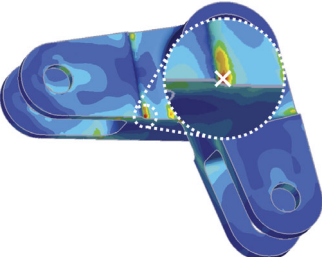
The first ‘Crank piston’ model is a part of engine components. The crank shaft rotates about the ground. Each piston and the shaft are connected to connecting rods that are flexible bodies. Piston translates with respect to the ground.

**Table 4** Changes of time, memory usage and flops of Table 2b models

| Computing device<br>Model (fac. type) | Item        | GPU      |        |        |              |               |        |        |         | CHOL MOD              | SPQR   | CPU<br>DSS |
|---------------------------------------|-------------|----------|--------|--------|--------------|---------------|--------|--------|---------|-----------------------|--------|------------|
|                                       |             | MFS      | 64     | 128    | 256          | 512           | 1024   | 2048   | 4096    |                       |        |            |
| Trefethen_20000 (FULL)                | Time (sec.) | 41.50    | 51.49  | 20.41  | 12.72        | <b>11.89</b>  | 12.19  | 14.53  | 2.93    | 30.79                 | 87.07  |            |
|                                       | Mem. (MB)   | 1,893    | 1,985  | 2,088  | 2,169        | <b>2,223</b>  | 2,241  | 2,250  | 9,958   | 4,657                 | 1,516  |            |
|                                       | Gflops      | 2,201    | 2,368  | 2,493  | 2,574        | <b>2,646</b>  | 2,680  | 2,726  | 2,123   | 12,197                | 1,141  |            |
| Conspfh (FULL)                        | Time (sec.) | 19.21    | 4.46   | 2.13   | <b>2.09</b>  | 2.59          | 3.64   | 4.05   | 3.30    | 25.69                 | 5.81   |            |
|                                       | Mem. (MB)   | 1,675    | 1,858  | 2,094  | <b>2,441</b> | 2,840         | 3,223  | 3,312  | 10,353  | 7,521                 | 1,568  |            |
|                                       | Gflops      | 422      | 505    | 603    | <b>775</b>   | 997           | 1,455  | 1,578  | 942     | 6,960                 | 260    |            |
| tors03 (FULL)                         | Time (sec.) | 25.73    | 6.82   | 4.32   | <b>4.29</b>  | 5.22          | 9.33   | 9.35   | 5.60    | 55.74                 | 10.14  |            |
|                                       | Mem. (MB)   | 3,439    | 3,911  | 4,549  | <b>5,348</b> | 6,112         | 6,807  | 7,324  | 11,819  | 12,441                | 2,308  |            |
|                                       | Gflops      | 550      | 630    | 793    | <b>1,052</b> | 1,403         | 1,821  | 1,853  | 1,419   | 15,745                | 364    |            |
| F1 (PARTIAL)                          | Time (sec.) | 33.29    | 10.27  | 9.57   | <b>8.83</b>  | 9.72          | 10.29  | 10.33  | 16.03   | 21.43 <sup>a</sup>    | 11.31  |            |
|                                       | Mem. (MB)   | 4,463    | 5,126  | 5,947  | <b>6,921</b> | 7,880         | 8,219  | 8,219  | 19,073  | 13,788 <sup>a</sup>   | 5,062  |            |
|                                       | Gflops      | 822      | 1,005  | 1,249  | <b>1,594</b> | 2,048         | 2,261  | 2,261  | 5,269   | 4,944 <sup>a</sup>    | 450    |            |
| audikw_1 (PARTIAL)                    | Time (sec.) | 1,133.39 | 234.52 | 107.94 | 71.93        | <b>69.21</b>  | 85.90  | 113.08 | 153.61  | disabled <sup>b</sup> | 264.58 |            |
|                                       | Mem. (MB)   | 26,806   | 30,095 | 33,850 | 38,345       | <b>45,079</b> | 53,218 | 61,758 | 77,034  | disabled <sup>b</sup> | 26,361 |            |
|                                       | Gflops      | 16,105   | 18,866 | 21,629 | 24,967       | <b>31,484</b> | 42,544 | 56,465 | 119,091 | disabled <sup>b</sup> | 12,397 |            |
| Fault_1 (PARTIAL)                     | Time (sec.) | 1,771.73 | 370.56 | 160.86 | 94.26        | <b>82.87</b>  | 101.83 | 135.64 | 85.79   | disabled <sup>b</sup> | 361.62 |            |
|                                       | Mem. (MB)   | 26,673   | 30,551 | 34,929 | 38,997       | <b>44,665</b> | 52,664 | 61,635 | 41,698  | disabled <sup>b</sup> | 20,638 |            |
|                                       | Gflops      | 25,602   | 30,761 | 36,835 | 41,847       | <b>49,378</b> | 63,828 | 84,999 | 56,527  | disabled <sup>b</sup> | 17,494 |            |

<sup>a</sup>The solution is not a number.

<sup>b</sup>GPU memory too small error.

| Model        | Shape  | Representative node  |
|--------------|--|--|
| Crank piston |   |   |
| Suspension   |   |   |
| Bell crank   |  |  |

**Fig. 16** Three flexible mechanical models for dynamic analysis

The second ‘Suspension’ model is an ordinary double wishbone. Upper and lower control arms rotate with respect to the ground. The upper shocks are fixed to the ground by bushing elements. The upper and lower shocks are connected by a translational joint and spring. The upper and lower control arms are connected by a knuckle which is connected by a revolute joint to the tire. A rack body controls the steering motion of the knuckle. Horizontal motion of the rack and vertical motion of tires are given for the dynamic analysis to observe the knuckle kinematics and compliance characteristics. The lower control arm is modeled as a flexible body.

The last ‘Bell crank’ model has a fixed center hole and there are two upper and lower holes at the end of each arm. Dynamic forces are applied at two hole-faces in the opposite

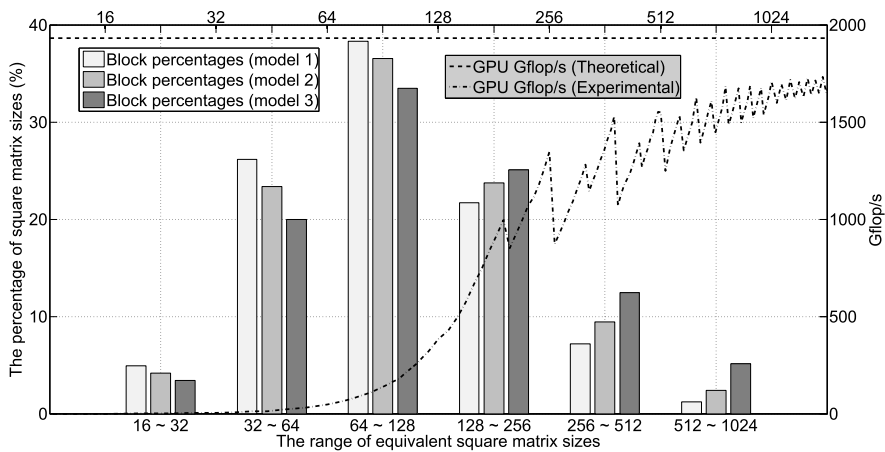
**Table 5** Dynamic analysis time and memory usage results for each model

| Model/dimension | Item           | DSS CPU        |               | MFS GPU        |               | Speed up      |               |
|-----------------|----------------|----------------|---------------|----------------|---------------|---------------|---------------|
|                 |                | Time<br>(sec.) | Rate<br>(%)   | Time<br>(sec.) | Rate<br>(%)   |               |               |
| Crank piston    | Linear solver  | ① Reorder      | 477           | 0.77           | 704           | 2.21          | ×0.68         |
|                 |                | ② Factor.      | <b>50,121</b> | <b>80.70</b>   | <b>19,382</b> | <b>60.80</b>  | × <b>2.59</b> |
|                 |                | ③ Solving      | 4,517         | 7.27           | 4,815         | 15.10         | ×0.94         |
| rows, cols      | ④ Linear sum.  | <b>55,115</b>  | <b>88.74</b>  | <b>24,902</b>  | <b>78.11</b>  | × <b>2.21</b> |               |
| 1,481,232       | ⑤ Remaining    | 6,994          | 11.26         | 6,978          | 21.89         | ×1.00         |               |
| non-zeros       | ⑥ Total (=④+⑤) | <b>62,109</b>  | <b>100.00</b> | <b>31,880</b>  | <b>100.00</b> | × <b>1.95</b> |               |
| 403,713,416     | Peak mem. (MB) | 57,834         |               | 68,960         |               | –             |               |
| Suspension      | Linear solver  | ① Reorder      | 260           | 0.78           | 405           | 2.41          | ×0.64         |
|                 |                | ② Factor.      | <b>28,135</b> | <b>84.30</b>   | <b>10,993</b> | <b>65.44</b>  | × <b>2.56</b> |
|                 |                | ③ Solving      | 1,890         | 5.66           | 2,329         | 13.87         | ×0.81         |
| rows, cols      | ④ Linear sum.  | <b>30,285</b>  | <b>90.75</b>  | <b>13,728</b>  | <b>81.72</b>  | × <b>2.21</b> |               |
| 883,885         | ⑤ Remaining    | 3,087          | 9.25          | 3,070          | 18.28         | ×1.01         |               |
| non-zeros       | ⑥ Total (=④+⑤) | <b>33,373</b>  | <b>100.00</b> | <b>16,798</b>  | <b>100.00</b> | × <b>1.99</b> |               |
| 230,555,929     | Peak mem. (MB) | 34,654         |               | 45,245         |               | –             |               |
| Bell crank      | Linear solver  | ① Reorder      | 157           | 0.37           | 235           | 2.62          | ×0.67         |
|                 |                | ② Factor.      | <b>39,983</b> | <b>93.53</b>   | <b>6,724</b>  | <b>74.86</b>  | × <b>5.95</b> |
|                 |                | ③ Solving      | 1,000         | 2.34           | 414           | 4.61          | ×2.42         |
| rows, cols      | ④ Linear sum.  | <b>41,140</b>  | <b>96.24</b>  | <b>7,373</b>   | <b>82.08</b>  | × <b>5.58</b> |               |
| 778,650         | ⑤ Remaining    | 1,609          | 3.76          | 1,609          | 17.92         | ×.00          |               |
| non-zeros       | ⑥ Total (=④+⑤) | <b>42,749</b>  | <b>100.00</b> | <b>8,982</b>   | <b>100.00</b> | × <b>4.76</b> |               |
| 192,946,566     | Peak mem. (MB) | 36,489         |               | 43,732         |               | –             |               |

direction. Boundary condition is used to fasten the body. Standard elastic steel properties are used for all bodies of the models except tires in the second model.

A variable step size has been used to numerically integrate Eq. (6) and the  $H_x^i$  matrix in Eq. (7) must be generated and solved at every time step. There are numerical errors due to finite-precision arithmetic and condition number of the linear system [25], and a dynamic analysis accumulates the errors over time. Accurate solutions have a decisive effect on iterations of Newton’s numerical method as well as on overall dynamic analysis time. Thus, the iterative refinement option is used to satisfy the accuracy requirement during the solving phase. In order to verify an operational performance of the proposed implementation, DSS is used. Except for the operating system, the same software specifications and reordering algorithm in Table 3b are used again because the dynamic analysis software supports only the Windows operation system. Therefore, CHOLMOD and SPQR routines cannot be involved in this experiment. The number of linear solving steps is the same irrespective of the two routines.

In Table 5, peak memory usage and detailed computing times with their proportions are tabulated. The times are categorized into two parts of the linear equation solver times including reordering, numerical factorization and solving phase and the remaining time for reading input file, converting required data structure and generating system matrix at every time step.



**Fig. 17** A distribution of equivalent square matrix sizes and its Gflop/s

The GPU device is used only in the numerical factorization step, so that the device has outstanding effects on the linear solver time while there is no effect in the other time. Although there are considerable speed-ups in numerical factorization step, the total time improvement ratio cannot surpass each factorization speed-up rate due to the Amdahl law [37]. As a result, the aggregate time of the GPU is accelerated about 1.9 to 4.7 times, a little less than 2.5 to 5.9 times for the factorization improvement, compared to that of the CPU.

The flops of variable update operation occupy most of the total numerical factorization. Since the variable update consists of only matrix multiplication operations, it is important to analyze the matrix multiplication to achieve the best computing performance. Many different sizes of non-square matrices are involved in the variable update. Because it is difficult to estimate the computing time for all kinds of matrices, all matrices must be converted into computationally equivalent square matrices. Figure 17 depicts two kinds of data from one of the sparse matrices generated during dynamic analysis. One depicts both a constant theoretical peak and an experimental Gflop/s of square matrix multiplications in double precision when the matrix size increases. The other shows the percentages of equivalent square matrix sizes whose arithmetic operation numbers are from the Fig. 16 models.

Based on the square matrix size 128 in Fig. 17, the experimental models present different tendencies. Below 128, the first and the second models have higher proportions than the third model; however, the third model shows more percentages above 128, especially 512–1024. The experimental performance for matrix multiplication consistently grows until 2048. Therefore, the outstanding computing performance of the GPU device is highly affected by the ratios of larger equivalent square matrix sizes.

Meanwhile, the peak memory usage of the proposed method using a GPU device is greater than that of MKL DSS on a CPU. The memory usage is affiliated with the block size of separators as shown in Table 4. There is an inverse relationship between block size and factorization time below a certain block size, or 1024. It is possible to reduce the memory usage by reducing the maximum block size, but the action certainly causes increase in computing time for numerical factorization. This trend seems to be the nature of the experimental GPU architecture. Table 6 summarizes the computing time and the memory usage of one sparse matrix among the dynamic analysis according to the maximum block size from 64 to 1024.



**Table 6** A correlation between memory usage and computing time for block size

| Model        | Item        | DSS CPU       | MFS GPU (maximum block size) |        |        |        |               |
|--------------|-------------|---------------|------------------------------|--------|--------|--------|---------------|
|              |             |               | 64                           | 128    | 256    | 512    | 1024          |
| Crank piston | Mem. (MB)   | <b>53,865</b> | 47,337                       | 47,068 | 50,878 | 57,334 | <b>65,059</b> |
|              | Time (sec.) | <b>207.32</b> | 866.44                       | 234.66 | 98.34  | 81.02  | <b>78.43</b>  |
| Suspension   | Mem. (MB)   | <b>32,720</b> | 30,815                       | 31,438 | 33,788 | 37,837 | <b>43,097</b> |
|              | Time (sec.) | <b>136.74</b> | 618.68                       | 166.78 | 70.28  | 53.75  | <b>53.47</b>  |
| Bell crank   | Mem. (MB)   | <b>34,661</b> | 32,965                       | 30,616 | 32,719 | 36,586 | <b>41,886</b> |
|              | Time (sec.) | <b>351.11</b> | 1,091.38                     | 262.63 | 94.62  | 69.78  | <b>56.81</b>  |

**Table 7** The scaled residual values for each solution type

| Value type   | Acc. magnitude | Von Mises stress |
|--------------|----------------|------------------|
| Crank piston | 2.587883E-10   | 2.200535E-05     |
| Suspension   | 3.325773E-07   | 7.547753E-08     |
| Bell crank   | 1.775657E-08   | 1.900251E-11     |

It is also important to compare the solution accuracies of GPU and CPU. Von Mises stress and acceleration magnitude values at the point shown in Fig. 17 are used to check the solution accuracy. The scaled Infinity-norm values in Eq. (9) are compared with evaluation of the accuracy. Table 7 shows that the norm values are almost identical, which verifies the solution accuracy.

$$\text{res}_{\text{scaled}} = \frac{\|\mathbf{x}_{\text{cpu}} - \mathbf{x}_{\text{gpu}}\|_{\infty}}{\|\mathbf{x}_{\text{cpu}}\|_{\infty} + \|\mathbf{x}_{\text{gpu}}\|_{\infty}} \quad (9)$$

## 8 Conclusion

A new linear equation solver for a GPU has been implemented using a BFS-based reordering and multifrontal methods. The proposed implementation is parallelized for an experimental GPU. Since popular direct methods have several drawbacks to apply them to GPUs, a new implementation is needed. In order to get over the drawbacks, a combination of the BFS-based reordering method and multifrontal method is proposed. A global multifrontal operation is carried out from the deepest separators to a root of a multilevel tree. This sequence is exactly the same as the BFS reverse-level order traversal. The BFS-based implementation is more suitable for the GPU device than the DFS-based. It makes a host system easy to set priorities of operational regions to fit the GPU memory size. The operation grouping of variable factor and update gives separators more parallel opportunities. However, another difficulty with the efficient parallel processing comes from the non-uniform size of separators. To resolve the difficulty, large separators are divided into smaller blocks. An experimental approach is used to estimate the optimum maximum block size to be 512 or 1024. Dynamic analysis of three mechanical models has been carried out to demonstrate the effectiveness of the proposed method. The computing time and the memory usage on GPUs are compared with those obtained from DSS routine included in the MKL on CPUs. It performs 1.9–4.7

times faster during the whole computing process. The important factor in deciding the performance improvement on a GPU device is how many percentages of large block matrices in variable update operations are involved. The proposed implementation and the DSS have yielded the same level of accurate solutions. The proposed method will be extended for a multi-GPU system.

## References

1. TOP 500: <http://www.top500.org/>
2. Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C.: GPU computing. *Proc. IEEE* **96**(5), 879–899 (2008). doi:[10.1109/jproc.2008.917757](https://doi.org/10.1109/jproc.2008.917757)
3. Saad, Y.: *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelphia (2003). doi:[10.1137/1.9780898718003](https://doi.org/10.1137/1.9780898718003)
4. Lukash, M., Rupp, K., Selberherr, S.: Sparse approximate inverse preconditioners for iterative solvers on GPUs. In: *Proceedings of the 2012 Symposium on High Performance Computing* (2012)
5. Serban, R., Melanz, D., Li, A., Stanculescu, I., Jayakumar, P., Negrut, D.: GPU-based preconditioned Newton–Krylov solver for flexible multibody dynamics. *Int. J. Numer. Methods Eng.* **102**(9), 1585–1604 (2015). doi:[10.1002/nme.4876](https://doi.org/10.1002/nme.4876)
6. Naumov, M.: Incomplete-LU and Cholesky preconditioned iterative methods using CUSPARSE and CUBLAS. Nvidia white paper (2011)
7. Wong, J., Kuhl, E., Darve, E.: A new sparse matrix vector multiplication graphics processing unit algorithm designed for finite element problems. *Int. J. Numer. Methods Eng.* **102**(12), 1784–1814 (2015). doi:[10.1002/nme.4865](https://doi.org/10.1002/nme.4865)
8. Rodrigues, A.W.D.O., Guyomarch, F., Menach, Y.L., Dekeyser, J.L.: Parallel sparse matrix solver on the GPU applied to simulation of electrical machines. [arXiv:1010.4639](https://arxiv.org/abs/1010.4639) (2010)
9. Negrut, D., Tasora, A., Anitescu, M., Mazhar, H., Heyn, T., Pazouki, A.: Solving large multi-body dynamics problems on the GPU. In: *GPU Gems*, vol. 4, pp. 269–280 (2011). doi:[10.1016/b978-0-12-385963-1.00020-4](https://doi.org/10.1016/b978-0-12-385963-1.00020-4)
10. Mazhar, H., Heyn, T., Negrut, D.: A scalable parallel method for large collision detection problems. *Multibody Syst. Dyn.* **26**(1), 37–55 (2011). doi:[10.1007/s11044-011-9246-y](https://doi.org/10.1007/s11044-011-9246-y)
11. Negrut, D., Tasora, A., Mazhar, H., Heyn, T., Hahn, P.: Leveraging parallel computing in multibody dynamics. *Multibody Syst. Dyn.* **27**(1), 95–117 (2012). doi:[10.1007/s11044-011-9262-y](https://doi.org/10.1007/s11044-011-9262-y)
12. Gaikwad, A., Toke, I.M.: Parallel iterative linear solvers on GPU: a financial engineering case. In: *Euro-micro Conference on Parallel, Distributed and Network-based Processing*, pp. 607–614 (2010). doi:[10.1109/pdp.2010.55](https://doi.org/10.1109/pdp.2010.55)
13. Scott, J.A., Hu, Y.: Experiences of sparse direct symmetric solvers. *ACM Trans. Math. Softw.* **33**(3), 18 (2007). doi:[10.1145/1268769.1268772](https://doi.org/10.1145/1268769.1268772)
14. Davis, T.A.: *Direct Methods for Sparse Linear Systems*. *Fundamentals of Algorithms*, vol. 2. SIAM, Philadelphia (2006). doi:[10.1137/1.9780898718881](https://doi.org/10.1137/1.9780898718881)
15. Irons, B.M.: A frontal solution program for finite element analysis. *Int. J. Numer. Methods Eng.* **2**(1), 5–32 (1970). doi:[10.1002/nme.1620020104](https://doi.org/10.1002/nme.1620020104)
16. Scott, J.A.: A parallel frontal solver for finite element applications. *Int. J. Numer. Methods Eng.* **50**(5), 1131–1144 (2001). doi:[10.1002/1097-0207\(20010220\)50:5<1131::aid-nme68>3.0.co;2-x](https://doi.org/10.1002/1097-0207(20010220)50:5<1131::aid-nme68>3.0.co;2-x)
17. Reid, J.K., Scott, J.A.: An efficient out-of-core multifrontal solver for large-scale unsymmetric element problems. *Int. J. Numer. Methods Eng.* **77**(7), 901–921 (2009). doi:[10.1002/nme.2437](https://doi.org/10.1002/nme.2437)
18. Rennich, S.C., Stolic, D., Davis, T.A.: Accelerating sparse Cholesky factorization on GPUs. In: *Proceedings of the Fourth Workshop on Irregular Applications: Architectures and Algorithms*, pp. 9–16. IEEE Press, Piscataway (2014). doi:[10.1109/IA3.2014.6](https://doi.org/10.1109/IA3.2014.6)
19. Yeralan, S.N., Davis, T.A., Ranka, S.: Algorithm 9xx: sparse QR factorization on the GPU. *ACM Trans. Math. Softw.* (2015)
20. Bae, D.S., Kim, H.W., Yoo, H.H., Suh, M.S.: A decoupling solution method for implicit numerical integration of constrained mechanical systems. *Mech. Struct. Mach.* **27**(2), 129–141 (1999). doi:[10.1080/08905459908915692](https://doi.org/10.1080/08905459908915692)
21. Horowitz, E.: *Fundamentals of Data Structures in C++*. Galgotia Publications, New Delhi (2006)
22. Brainman, I., Toledo, S.: Nested-dissection orderings for sparse LU with partial pivoting. *SIAM J. Matrix Anal. Appl.* **23**(4), 998–1012 (2002). doi:[10.1137/s0895479801385037](https://doi.org/10.1137/s0895479801385037)
23. Davis, T.A., Hager, W.W.: Dynamic supernodes in sparse Cholesky update/downdate and triangular solves. *ACM Trans. Math. Softw.* **35**(4), 27 (2009). doi:[10.1145/1462173.1462176](https://doi.org/10.1145/1462173.1462176)

24. Karypis, G., Kumar, V.: METIS—a software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices, version 5.1.0. University of Minnesota (2013)
25. L'Excellent, J.Y.: Multifrontal methods: parallelism, memory usage and numerical aspects. Ecole Normale Supérieure de Lyon-ENS LYON (2012)
26. Padua, D.: Encyclopedia of Parallel Computing. Springer, Berlin (2011). doi:[10.1007/978-0-387-09766-4](https://doi.org/10.1007/978-0-387-09766-4)
27. Guermouche, A., L'Excellent, J.Y., Utard, G.: On the memory usage of a parallel multifrontal solver. In: Parallel and Distributed Processing Symposium, p. 8 (2003). doi:[10.1109/ipdps.2003.1213187](https://doi.org/10.1109/ipdps.2003.1213187)
28. Guermouche, A., L'Excellent, J.Y., Utard, G.: Analysis and improvements of the memory usage of a multifrontal solver (2003)
29. NVIDIA Kepler GK110 architecture: <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>
30. Jung, J.H., Bae, D.S.: Optimization of operating and assembling mass properties of solid elements on heterogeneous platforms using OpenCL framework. J. Mech. Sci. Technol. **29**(7), 2631–2637 (2015). doi:[10.1007/s12206-015-0508-0](https://doi.org/10.1007/s12206-015-0508-0)
31. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Comput. **35**(1), 38–53 (2009). doi:[10.1016/j.parco.2008.10.002](https://doi.org/10.1016/j.parco.2008.10.002)
32. Wang, L., Wu, W., Xu, Z., Xiao, J., Yang, Y.: BLASX: a high performance level-3 BLAS library for heterogeneous multi-GPU computing. In: Proceedings of the 2016 International Conference on Supercomputing, pp. 20:1–20:11. ACM, New York (2016). doi:[10.1145/2925426.2926256](https://doi.org/10.1145/2925426.2926256)
33. Kurzak, J., Nath, R., Du, P., Dongarra, J.: An implementation of the tile QR factorization for a GPU and multiple CPUs. In: International Workshop on Applied Parallel Computing, pp. 248–257. Springer, Berlin (2010). doi:[10.1007/978-3-642-28145-7\\_25](https://doi.org/10.1007/978-3-642-28145-7_25)
34. Tomov, S., Nath, R., Ltaief, H., Dongarra, J.: Dense linear algebra solvers for multicore with GPU accelerators. In: Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), pp. 1–8 (2010). doi:[10.1109/IPDPSW.2010.5470941](https://doi.org/10.1109/IPDPSW.2010.5470941)
35. Anderson, E., Dongarra, J.J., Ostrouchov, S.: Lapack working note 41: installation guide for lapack. University of Tennessee, Computer Science Department (1992)
36. Intel, Intel Math Kernel Library Reference Manual 11.3, 1575 (2015)
37. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the Spring Joint Computer Conference, April 18–20, 1967, pp. 483–485. ACM, New York (1967). doi:[10.1109/N-SSC.2007.4785615](https://doi.org/10.1109/N-SSC.2007.4785615)