Electrical Power and Energy Systems 111 (2019) 34–43

Contents lists available at ScienceDirect

Electrical Power and Energy Systems

journal homepage: www.elsevier.com/locate/ijepes

GPU-accelerated sparse matrices parallel inversion algorithm for large-scale power systems

Gan Zhou[a,*], Yanjun Feng[a], Rui Bo[b], Tao Zhang[c]

[a] School of Electrical Engineering, Southeast University, Nanjing 210096, China
[b] Department of Electrical and Computer Engineering at Missouri University of Science and Technology, Rolla, MO 65409, USA
[c] State Grid Anshan Electric Power Supply Company, Anshan 114000, China

ARTICLE INFO

Keywords:
Inversion
Forward substitution
Backward substitution
Spares matrix
GPU
Accelerated
Parallelism
Power flow

ABSTRACT

State-of-the-art Graphics Processing Unit (GPU) has superior performances on float-pointing calculation and memory bandwidth, and therefore has great potential in many computationally intensive power system applications, one of which is the inversion of large-scale sparse matrix. It is a fundamental component for many power system analyses which requires to solve massive number of forward and backward substitution (F&B) subtasks and seems to be a good GPU-accelerated candidate application. By means of solving multiple F&B subtasks concurrently and a serial of performance tunings in compliance with GPU's architectures, we successfully develop a batch F&B algorithm on GPUs, which not only extracts the intra-level and intra-level parallelisms inside single F&B subtask but also explores a more regular parallelism among massive F&B subtasks, called inter-task parallelism. Case study on a 9241-dimension case shows that the proposed batch F&B solver consumes 2.92 μs per forward substitution (FS) subtask when the batch size is equal to 3072, achieving 65 times speedup relative to KLU library. And on the basis the complete design process of GPU-based inversion algorithm is proposed. By offloading the tremendous computational burden to GPU, the inversion of 9241-dimension case consumes only 97 ms, which can achieve 8.1 times speedup relative to the 12-core CPU inversion solver based on KLU library. The proposed batch F&B solver is practically very promising in many other power system applications requiring solving massive F&B subtasks, such as probabilistic power flow analysis.

1. Introduction

Inversion of large-scale dense or sparse matrices is required in a few scientific applications, such as statistics and prediction, dynamics analysis, model reduction, and optimal control [1,2]. In power system analysis, most of matrices are very sparse, so the studies focus on inversion of sparse matrices and its applications on state estimation, bad data processing, sensitivity analysis and short-circuit current calculation [3–6]. The most efficient method for calculating inversion of sparse matrices is to solve a set of sparse linear system (SLS) of equations with the same coefficient matrix. More specifically, this method can be divided into two steps: the first step factorizes the target matrix $\boldsymbol{A} = \boldsymbol{LU}$, and the second step performs a set of forward and backward substitution (F&B) subtasks on the factors $\boldsymbol{L}$ and $\boldsymbol{U}$. In some other applications, such as Monte-Carlo simulation (MCS) based probabilistic power flow (PPF), hundreds of thousands of deterministic power flows (DPF) need to be analyzed. If we choose fast decoupled load flow method, the main computational burden of PPF is identical with the matrix inversion problem, ie, solving lots of sparse linear system of equations (SLSE) with the same coefficient matrix. When the number of SLSEs is very large, it will lead to tremendous computation cost and long run-time, which consequently limits the online applications based on matrix inversion and PPF analysis.

As every F&B subtask in matrix inversion can be calculated independently of each other, the coarse-grain parallel computing solution, in which one CPU thread solves one F&B subtask, can be easily implemented on the multi-core CPU platform, a typical Multiple Instruction Multiple Data (MIMD) architecture. Since the F&B task is memory bound, the biggest disadvantage of multi-core CPU solution lies in the saturation of memory bandwidth, which has to be resolved by adding CPUs with separate memory. However, it will dramatically increase capital investment and computing power consumption [7]. Two other fine-grain parallel computing solutions, designed for vector computers, are the sparse vector method and the inverse factors method [8–12]: The former exploits the parallelism existent in the sparse F&B task, which is usually described as a dependence graph with different

* Corresponding author.
E-mail addresses: zhougan2002@seu.edu.cn (G. Zhou), fengyanjun@seu.edu.cn (Y. Feng), rbo@mst.edu (R. Bo), ct_dalian@sina.com.cn (T. Zhang).

https://doi.org/10.1016/j.ijepes.2019.03.074
Received 10 August 2018; Received in revised form 16 March 2019; Accepted 31 March 2019
Available online 05 April 2019

levels. By means of inverting the factor matrix, the latter replaces the F &B task with strong data dependence by matrix-vector product, an embarrassingly parallel problem.

State-of-the-art Graphics Processing Unit (GPU), with Single Instruction Multiple Threads (SIMT) architecture, has superior performances on float-pointing calculation and memory bandwidth. For example, in comparison to the same generation CPUs such as 6-core Intel Xeon E5-2620 2 GHz, GPUs such as NVIDIA K40 have 60 times the floating-point calculation capability and 4.9 times the memory bandwidth. Therefore, GPU offers an alternative and potentially superior solution for sparse matrix inversion problem.

When referring to existing researches to accelerate sparse matrix inversion problem, two different technical frameworks can be proposed. Solution 1 uses GPU to accelerate each F&B subtask one after another. This sequential solution has two inherent shortcomings: Firstly, limited by the problem scale and excessively sequential process, the parallelism of single F&B task cannot saturate the numerous floating-point cores on GPUs. Secondly, the random memory access, resulted from sparse F&B computation process, will lead to massive uncoalesced memory access and therefore the high bandwidth of GPU cannot be fully utilized [13–15]. In order to explore the parallelism to the full extent, Refs. [16,17] use the dependence graph and domino scheme to extract the intra-level and inter-level parallelism existing in single F&B task. However, this kind of GPU-based solution is still slower than the CPU-based solution for most power system matrices. In contrast, Solution 2 packages a large number of independent subtasks to formulate a larger-scale batch task, which has been successfully applied in metaheuristic-based optimization problem [18,19], static security analysis [20] and MCS-based PPF analysis [21]. Obviously, Solution 2 is superior to Solution 1 due to achieving the additional parallelism among multiple subtasks. A natural question arises that whether the existing GPU-based batch algorithm can exploit GPU potential to the full extent. The answer is no because little to no insights have been offered regarding how to make the parallelism more regular in compliance with GPU's SIMT architecture, which is crucial for the performance improvement [22,23]. In this regard, we proposed a novel parallel inversion algorithm on GPUs, which not only extracts various degrees of parallelism inside batch F&B task but also be carefully tuned for making the parallelism more regular. We make the following contributions in this paper.

Firstly, in compliance with GPU's software and hardware architectures, we propose four universal design and tuning criteria for a well-performing GPU-based algorithm, which involves exploring parallelism, fulfilling coalesced memory access, avoiding thread divergence and reducing unnecessary data interactions between CPU and GPU.

Secondly, we propose a novel GPU-accelerated batch F&B algorithm which can be used in various power system applications needing to solve lots of F&B subtasks. In addition to utilizing the conventional intra-level parallelism, we explore an extra inter-level parallelism which can partly tackle the issue that the intra-level parallelism becomes insufficient and meanwhile the computing load per thread becomes heavy with the execution of F&B. However, the problems of thread divergence and uncoalesced memory access remain to be resolved. Therefore, by means of solving multiple F&B subtasks concurrently and a serial of performance tunings, we successfully explore a very regular parallelism among massive number of F&B subtasks, called inter-task parallelism. And on this basis we propose a novel batch F&B algorithm with higher level of parallelism, better memory access efficiency and perfect thread convergence. Case study on a 9241-dimension case shows that the proposed batch solver only consumes 2.92 μs per forward substitution (FS) when the batch size is equal to 3072, which achieves about 65 times speedup relative to KLU library, one of the fastest CPU library for solving SLS in power systems [24].

Lastly, a novel GPU-accelerated inversion algorithm, including overall framework design and detailed performance tunings, is presented. By means of offloading the tremendous computational burden to GPU, it can inverse large-scale SLS in an extremely fast manner. The inversion of a 9241-dimension case consumes only 97 ms, which can achieve 8.1 times speedup when compared with the 12-core CPU parallel computing solution based on KLU library.

The paper is organized as follows. Section 2 presents the mathematical principles of sparse matrix inversion. Section 3 defines four general design criteria for a well-performing GPU algorithm. Section 4, guided by the design criteria, presents the proposed GPU-accelerated batch F&B solver and its tuning strategies in detail. And then, based on the batch F&B solver, Section 5 proposes a novel GPU-based inversion algorithm. In order to show the enormous effect of proposed GPU-accelerated inversion algorithm, totally five computation solutions are tested and compared in Section 6. Finally Section 7 concludes the paper.

2. Algorithm principles of sparse matrix inversion

It is one of the most common method to calculate the inversion of large-scale sparse matrix by solving a set of SLSEs. Assuming $\boldsymbol{Z}$ is the inversion of matrix $\boldsymbol{A}$, there exists

$$\boldsymbol{A}_{n\times n}\boldsymbol{Z}_{n\times n} = \boldsymbol{I}_{n\times n} \tag{1}$$

where the subscript n represents the dimension of matrix and $\boldsymbol{I}$ is the identity matrix. After blocking by column, Eq. (1) can be expressed as

$$\boldsymbol{A}[\boldsymbol{z}_1,\cdots,\boldsymbol{z}_n] = [\boldsymbol{e}_1,\cdots,\boldsymbol{e}_n] \tag{2}$$

where $\boldsymbol{z}_k$ and $\boldsymbol{e}_k$ $(k = 1,\cdots, n)$ represent the column vectors of matrix $\boldsymbol{Z}$ and $\boldsymbol{I}$, respectively. It can be observed that the inversion of matrix $\boldsymbol{A}$ can be calculated by solving a set of SLSEs.

$$\boldsymbol{A}\boldsymbol{z}_k = \boldsymbol{e}_k\,(k = 1,\cdots,n) \tag{3}$$

All SLSEs in (3) have the same coefficient matrix $\boldsymbol{A}$, so the matrix $\boldsymbol{A}$ can be factorized only once.

$$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{U} \tag{4}$$

where the lower triangular matrix $\boldsymbol{L}$ and the upper triangular matrix $\boldsymbol{U}$ are the factorization results of $\boldsymbol{A}$, and then a set of F&B subtasks need to be performed.

$$\boldsymbol{L}\boldsymbol{y}_k = \boldsymbol{e}_k \tag{5}$$

$$\boldsymbol{U}\boldsymbol{z}_k = \boldsymbol{y}_k \tag{6}$$

It is a multidiscipline challenge to solve a SLSE, including graph theory to minimize fill-ins and LU factorization considering numerical stability [25]. Generally, the practical inversion computing can be divided into four steps as follows.

Step 1: Reordering matrix $\boldsymbol{A}$ to minimize fill-ins. The matrix inversion suffers from fill-ins. The more fill-ins are, the more extra floating-point calculations have to be performed. Therefore, it is very important to choose a proper reordering algorithm, such as approximate minimum degree (AMD) permutation and column approximate minimum degree (COLAMD) permutation, to minimize fill-ins. The reordering operation can be expressed as

$$\boldsymbol{A}' = \boldsymbol{Q}\boldsymbol{A}\boldsymbol{Q}^T \tag{7}$$

where $\boldsymbol{Q}$ is a permutation matrix corresponding to the reordering operation, and $\boldsymbol{A}'$ is the result matrix.

Step 2: Performing LU factorization with pivoting only once. The numerical stability of LU factorization should be ensured through the column partial pivoting

$$\boldsymbol{B} = \boldsymbol{P}\boldsymbol{A}' = \boldsymbol{P}\boldsymbol{Q}\boldsymbol{A}\boldsymbol{Q}^T = \boldsymbol{L}_B\boldsymbol{U}_B \tag{8}$$

where $\boldsymbol{P}$ is a permutation matrix corresponding to the column partial pivoting during the LU factorization of $\boldsymbol{A}'$, the lower triangular matrix $\boldsymbol{L}_B$ and the upper triangular matrix $\boldsymbol{U}_B$ are the factorization results of intermediate matrix $\boldsymbol{B}$.

Step 3: Calculating the inverse matrix $\boldsymbol{B}^{-1}$ of intermediate matrix $\boldsymbol{B}$.

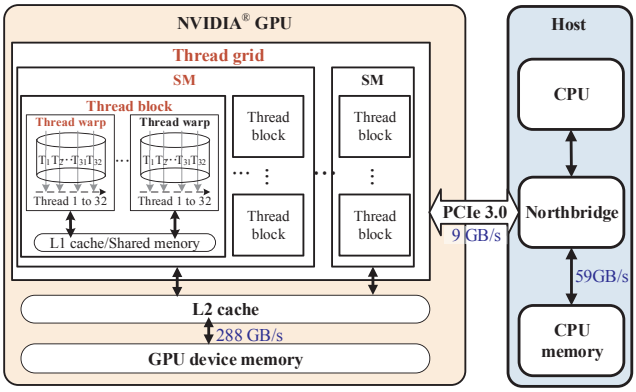

Fig. 1. The architecture of CPU and GPU hybrid programming.

Applying (5) and (6), the inverse matrix $\boldsymbol{B}^{-1}$ can be calculated by solving n times of F&B, which has the highest computational complexity in the entire inversion algorithm.

Step 4: Calculating the inverse matrix $\boldsymbol{A}^{-1}$ of target matrix $\boldsymbol{A}$. As $\boldsymbol{B} = \boldsymbol{PQAQ}^{\mathrm{T}}$, the inverse matrix $\boldsymbol{A}^{-1}$ can be derived from $\boldsymbol{B}^{-1}$ through some permutation operations.

$$\boldsymbol{Z} = \boldsymbol{A}^{-1} = \boldsymbol{Q}^{T}\boldsymbol{B}^{-1}\boldsymbol{PQ} \tag{9}$$

3. Design criteria of GPU-based algorithm

In 2007, NVIDIA presented the concept of compute unified device architecture (CUDA), which treats GPU as a universal co-processor of CPU and can access the numerous computing resources on GPU by C-language programming. However, it is never hardware transparent to design a GPU-based algorithm. On the contrary, the elaborate tunings must be performed in compliance with GPU's SIMT architecture.

As shown in Fig. 1, in SIMT parallel mode, the GPU kernel runs on a thread grid which consists of a large number of thread blocks. Moreover, every 32 threads in the same block are bundled into a thread warp that must work concurrently. All thread blocks are executed in parallel and communicate with each other through global memory. In contrast, all threads in the same thread block can intercommunicate via shared memory or barrier. When a GPU kernel is started, the thread blocks will be enumerated and distributed to the streaming multiprocessors (SM), which means that all threads belonging to the same block will run on a single SM. When a thread block ends or suspends, the scheduling engine will decide whether to start a new block and which block to start [26,27]. In compliance with above SIMT architecture, four general design criteria for GPU-based algorithms are proposed.

Criterion 1: exploring parallelism fully to saturate the numerous computing cores on GPU. GPUs specializing in high performance computing (HPC) own superior float-pointing computing resources. For example, a NVIDIA® K40 GPU contains 15 SMs and each SM consists of 192 single-precision computing cores, 64 double-precision cores and 32 special function units for fast approximate transcendental operations. Correspondingly, its theoretic peak performances of single-precision and double-precision have reached 4.29 Tflops and 1.43 Tflops, respectively. Therefore, a well-designed algorithm must explore the inherent parallelism to the full extent for saturating the numerous computing cores.

Criterion 2: fulfilling coalesced memory access to improve memory efficiency. There are a variety of memory units on GPU, such as register file, shared memory, L1 cache, L2 cache and DRAM, also called device memory. When optimizing the memory-bound algorithm on GPU, such as the sparse F&B algorithm, one of the biggest difficulties lies in how to fulfill the coalesced access to DRAM. More details can be explained as follows. Assuming that every thread in a thread warp is requesting a 4-bytes-long data, if these data are stored at the contiguous address, these 32 requests will be coalesced, automatically by hardware, into a 128-byte-long memory request. On the contrary, if these data are stored at discrete address, at most 32 memory requests will be launched, which seriously degrades the memory-access efficiency. Therefore, the coalesced memory access is crucial.

Criterion 3: avoiding thread divergence of thread warp. If 32 threads belonging to the same thread warp diverge, all different logic branches will be executed in series, which will increase the actual execution time dramatically and should be avoided.

Criterion 4: reducing data communication between CPU and GPU. Compared with the device memory bandwidths of GPU and CPU (about 288 GB/s and 59 GB/s, respectively), the communication bandwidth between the two is much slower. For example, the bandwidth of PCIe 3.0 is only 9 GB/s. Therefore, the data interaction should be minimized.

4. GPU-Based forward substitution algorithm

As mentioned in Section 2, totally n times of F&B need to be performed in matrix inversion, which contributes the primary target of GPU acceleration. In this section, a novel GPU-accelerated batch forward substitution (FS) algorithm is proposed and carefully tuned. The same framework can be applied to backward substitution (BS) algorithm directly.

4.1. Intra-level parallelism

Algorithm 1 (*GPU-based FS algorithm*).

```
1: for 1 to l do in serial//l is the level number.
2:   While there are unfinished rows in current level do in parallel
       //GPU thread i for calculating variable y(i).
3:     p = 0;//p is a temporary variable.
4:     for j = 1: i − 1 where L(i, j) is nonzero do
5:       p + = y(j) * L(i, j);
6:     end for
7:     y(i) = (e(i) − p)/L(i, i);
8:   end while
9: end for
```

As shown in Algorithm 1, in order to accelerate the forward substitution $\boldsymbol{Ly} = \boldsymbol{e}$, multiple GPU threads are used to solve the executable rows concurrently. This kind of parallelism is determined by the sparse structure of $\boldsymbol{L}$ and can be described by a dependence graph with different levels. All different levels must be processed in series but all rows belonging to the same level can be processed in parallel, called intra-level parallelism in this paper. As shown in Fig. 2, the 8-order matrix has been partitioned into four levels. Since there are no off-diagonal elements in row 1, 2, 3 and 5, these four rows are partitioned into level 1 and the corresponding variables y_k (k = 1, 2, 3 and 5) can be solved directly through $y_k = e_k/L_{kk}$. When they are finished, row 4 and 6 in level 2 can be started, and so on. As there is only one row in level 3 and

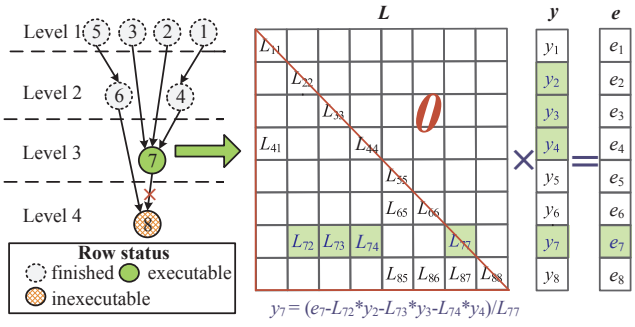

Fig. 2. The dependence graph of an 8-order matrix.

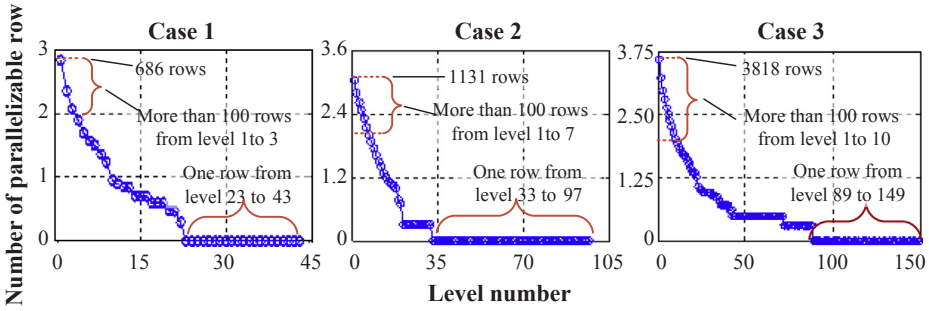

Fig. 3. The intra-level parallelism varying with level number. The number of parallelizable rows is represented as base-10 logarithm.

4, no such intra-level parallelism exists when solving y_7 and y_8. Meanwhile, the computation workload per row increases rapidly. For example, only one division needs to be performed when solving $y_k = e_k/L_{kk}$ (k = 1, 2, 3 and 5) in level 1. In contrast, it needs three floating-point multiplication and addition (FMAD), one addition and one division when solving $y_7 = (e_7 - L_{72} * y_2 - L_{73} * y_3 - L_{74} * y_4)/L_{77}$ in level 3.

In order to analyze the problems of parallelism and computing load more clearly, three large-scale cases from MATPOWER are used: Case 1 is the 1354-bus European high voltage case, Case 2 is the 3375-bus Polish winter peak case and Case 3 is the 9241-bus 110 kV + European high voltage case. The sparse degree of power system matrices is very high: Case 1 is 0.260%, Case 2 is 0.101% and Case 3 is 0.044%. As shown in Figs. 3 and 4, all three cases show the same change trends. Taking Case 3 as an example, firstly, the number of parallelizable rows decreases rapidly with level number. More specifically, there are 3818 rows in the first level and over 100 rows until level 10, but only 1 rows in the last 60 levels. Meanwhile, the computation load per row becomes heavy rapidly. It increases approximately linearly to 70 FMADs until level 100 and then fluctuates around 80 FMADs.

According to the general design criteria proposed in Section 3, Algorithm 1 is difficult to achieve reasonably good performance and the main bottlenecks lie in: Firstly, with the execution of forward substitution, the number of parallelizable rows becomes smaller but the computation workload per row becomes heavier, which violates *Criterion 1*. Secondly, in Algorithm 1, every thread is designed to solve one row. As every row has its unique non-zero structure, which will result in the number of elimination loops (see line 4 to line 6 of Algorithm 1) is quite different, 32 threads in the same thread warp are suffering from serious divergence problem, which violates *Criterion 3*. Thirdly, due to the random memory access of sparse forward substitution, the thread warp is very difficult to fulfill coalesced access to DRAM, which violates *Criterion 2*.

4.2. Inter-level parallelism

The biggest bottleneck of Algorithm 1 lies in that the parallelism reduces rapidly and meanwhile the workload per thread becomes heavy. As shown in Fig. 5, there exists a kind of parallelism across

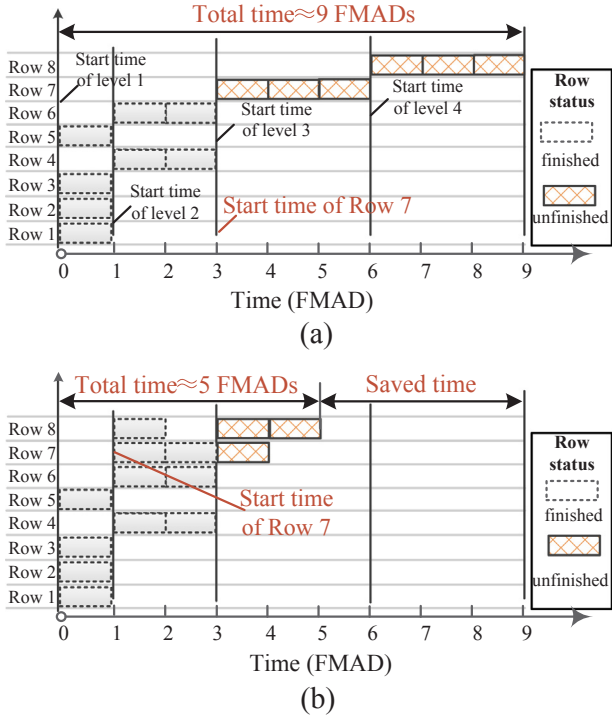

Fig. 5. The illustration of inter-level parallelism: (a) Computing time without inter-level parallelism and (b) Speedup with inter-level parallelism.

different levels, which can alleviate this problem to some extent, called inter-level parallelism in this paper. Taking row 7 as an example, it has three parents in the dependence graph: row 2 and row 3 in level 1, row 4 in level 2. In Algorithm 1, row 7 will wait until level 2 is finished (at the moment all its parents are done) and so its start time ≈ max{level 1} + max{level 2} ≈ 3 FMADs. In fact, there is no need to wait so long, a part of calculation can be started in advance as long as any parent has been done. As shown in Algorithm 2, the flag $\boldsymbol{F}(j)$ is used to mark whether the row j is done and the FMAD operation $p += \boldsymbol{y}(j) * \boldsymbol{L}(7, j)$ in line 5 will be triggered once $\boldsymbol{F}(j)$ is set. For example, when the flag $\boldsymbol{F}(2)$

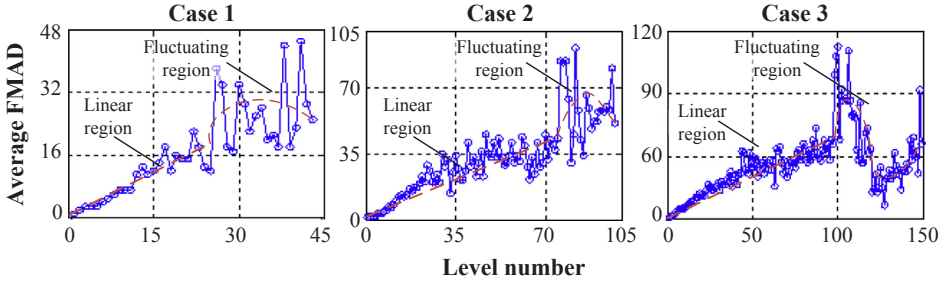

Fig. 4. The average FMAD per row varying with level number.

is set, $p+=\boldsymbol{y}(2)*\boldsymbol{L}(7,2)$ will be done at once, and so on. When row 7 is done, the flag $\boldsymbol{F}(7)$ will be set in line 8 and its children will be triggered. Due to this extra inter-level parallelism, the total computing time of 8-order case has reduced from 9 FMADs to 5 FMADs, which achieves 2.25 times speedup.

In summary, Algorithm 2 can tackle partly the issues that the parallelism becomes insufficient and the computing load per thread becomes heavy with the execution of forward substitution. However, the problems of thread divergence and uncoalesced memory access are still not resolved.

Algorithm 2 (*GPU FS algorithm with inter-level parallelism*).

```
1: While there are unfinished rows do in parallel
     //GPU thread i for calculating y(i).
2:   p = 0;
3:   for j = 1: i − 1 where L(i, j) is nonzero do
4:     Wait for F(j) to be true;//Wait until parent row j is done.
5:     p += y(j) * L(i, j);
6:   end for
7:   y(i) = (e(i) − p)/L(i, i);
8:   Set F(i) to true;//Trigger the children of row i.
9: end while
```

4.3. Inter-task parallelism

A natural question arises that whether Algorithm 2 has exploited GPU potential to the full extent in matrix inversion? The answer is no because no insights have been offered about exploring the parallelism among massive FS subtasks. In this regard, Algorithm 3 shifts the focus on how to extract the parallelism existing in multiple FS subtasks, called Inter-task parallelism. Its design considerations include:

(1) *Unifying sparsity pattern*. The prerequisite of the proposed batch FS algorithm is that all SLSs must have uniform sparsity pattern. This condition has been satisfied naturally because all FS subtasks share the same lower triangular matrix $\boldsymbol{L}$.

(2) *Achieving extra inter-FS-task parallelism*. As each FS subtask $\boldsymbol{L}\boldsymbol{y}_k = \boldsymbol{e}_k$ ($k = 1, \cdots, n$) can be done independently, n times the parallelism can be achieved from these n FS subtasks. As shown in Fig. 6, single FS subtask, such as FS subtask 1, can solve 4 rows in parallel when in level 1. In contrast, the batch FS task can solve $n \times 4$ rows simultaneously. As long as the batch size n is large enough, the parallelism of batch FS solver will keep relatively high and never less than the batch size n even in the last few levels. Obviously, the batch FS algorithm satisfies *Criterion 1* very well.

(3) *Fulfilling coalesced memory access*. As the sparse FS is a typical memory-bound algorithm, one of the biggest challenges for its performance tuning lies in how to fulfill coalesced access to GPU device memory. As shown in Fig. 6, two tactics are used: Firstly, the rows with the same number belonging to different FS subtasks are assigned into the same thread block. For example, all 7th rows are allocated to the 7th thread block in which the t-th thread is responsible for the t-th subtask; Secondly, the data of rows with the same number are saved at the contiguous address. For example, $\boldsymbol{L}_1(7,2), \boldsymbol{L}_2(7,2), \cdots, \boldsymbol{L}_n(7,2)$ are stored contiguously. Thus, all thread warps can fulfill perfectly coalesced memory access, which satisfies *Criterion 2*.

(4) *Avoiding thread divergence*. Due to the identical sparsity pattern and aforementioned tuning strategies on thread and memory allocation, 32 threads in one warp have the same numbers of elimination loops (see line 4 to 7 of Algorithm 3) and execute the identical logic branches. Therefore, the thread warp never diverge, which satisfies *Criterion 3* very well.

So far, by means of solving multiple F&B subtasks concurrently and a serial of performance tunings, we successfully explore a more regular parallelism with better memory access efficiency and perfect thread convergence. The same framework can be applied to backward substitution algorithm directly.

Algorithm 3 (*GPU-based Batch FS algorithm*).

```
1: While there are unfinished rows do in parallel
     //Thread block i for row i of the batch FS task.
2:   i ← block ID and t ← thread ID in this block;
3:   p(t) = 0;
4:   for j = 1: i − 1 where L_t(i, j) is nonzero do
5:     Wait for F(j) to be true;
6:     p(t) += y_t(j) * L_t(i, j);
7:   end for
8:   y_t(i) = (e_t(i)−p(t))/L_t(i, i);
9:   Set F(i) to true;
10: end while
```

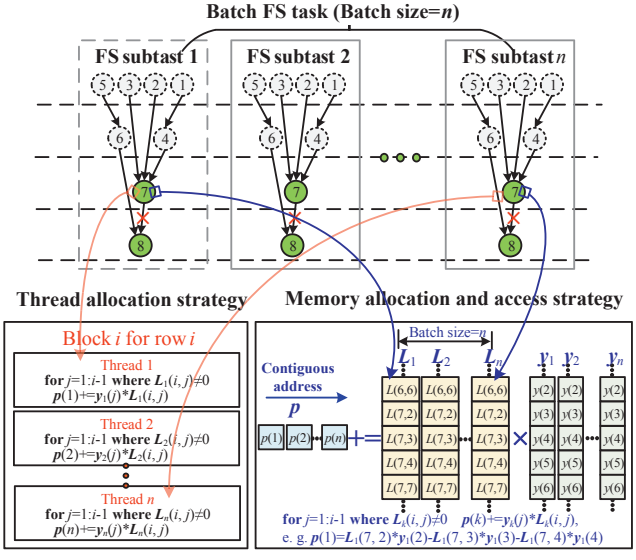

Fig. 6. Inter-task parallelism and how to fulfill coalesced memory access.

4.4. Case study of batch FS algorithm

The test platform is a server equipped with a Tesla K40 GPU and two 6-core Intel Xeon E5-2620 2 GHz CPU. The operating system is CentOS 6.7 and the CUDA version is 7.5. Algorithm 3 is implemented in double-precision floating-point format. Test results are gathered by NVIDIA Visual Profiler.

As shown in Tables 1 and 2, three test cases show the same characteristics: the average computing time per FS reduces rapidly with batch size n increasing. For example, when the batch size of Case 3 varies from 1 to 32, the computing time per FS reduces by 96.8%, from 639 μs to 20.6 μs. When $n = 512$, it will further reduce to 4.66 μs. There is indeed a saturation point where the performance does not further improve. As shown in Fig. 7, Case 3 reaches its saturation point 2.95 μs when $n = 3072$, which can achieve 65 times speedup relative to the calculation time of KLU library (192 μs per FS). At the same time, the device memory bandwidth has increased from 6.9 GB/s to 178 GB/s when the batch size varies from 1 to 3072, which has achieved 80% of available peak bandwidth of K40. This saturation problem can be easily resolved by adding more GPU cards in the same server to obtain extra memory bandwidth as every GPU has its separate device memory.

Table 1
Average computation time Per FS varying with batch size.

Case no.	Average time per FS varying with batch size n (μs)							
	$n = 1$	$n = 32$	$n = 128$	$n = 512$	$n = 1024$	$n = 2048$	$n = 3072$	$n = 8192$
1	141	4.41	1.23	0.60	0.51	No test [a]	No test	No test
2	251	7.94	2.39	1.75	1.45	1.19	1.10	No test
3	639	20.6	6.44	4.66	3.75	3.11	2.95	2.92

[a] As Case 1 is a 1354-dimension matrix, there is no need to test performance when the batch size is greater than 1354. Similarly, there is no test for Case 2 when the batch size is greater than 3375.

5. GPU-accelerated inversion algorithm

Based on Algorithm 3, which can solve the batch F&B problem in an extremely fast manner, a novel GPU-accelerated inversion algorithm for large-scale sparse matrix is proposed in this section. Firstly, an overall framework is presented, and then the detailed case studies are performed.

5.1. Overall framework

Using CPU-GPU heterogeneous computing environment to construct collaborative parallel computing can effectively improve the performance of matrix computing [28,29]. The framework of the proposed GPU-accelerated inversion algorithm is given in Fig. 8. CPU is responsible for controlling the overall flow, preparing input data and performing some serial computations, such as LU factorization. In contrast, GPU is dedicated to the intensive and parallelizable floating-point computations, such as the batch F&B and dense matrix permutation. Main design considerations include:

Step 1: performing LU factorization with reordering and generating dependence graph on CPU. As the sparse LU factorization and dependence graph generation are excessively sequential and the computing scale is very small (The sparse matrix of power system is usually less than 100,000 order, which belongs to a small-scale computing problem for HPC-purposed GPU), it is difficult to create enough parallelism to saturate the numerous cores on GPU. So, first of all, the cuSolver library, one of most high-performing and commercially available CPU-based library for solving SLS in power systems, is used to perform single LU factorization with pivoting. And then, we use an in-house CPU code to generate dependence graph. As mentioned in Section 2, after reordering and numerical factorization, we can obtain $\boldsymbol{L_B U_B = B = PA' = PQAQ}^{\mathrm{T}}$ where the permutation matrix $\boldsymbol{Q}$ and $\boldsymbol{P}$ represent the AMD reordering and the column partial pivoting, respectively, $\boldsymbol{L_B}$ and $\boldsymbol{U_B}$ are the factorization results. It's important to note that, if the computing scale is large enough, some new GPU-based algorithms can be used to shorten the calculating time of matrix decomposition [30,31].

Step 2: Transferring input data to GPU. In this step, only four sparse matrices $\boldsymbol{L_B}$, $\boldsymbol{U_B}$, $\boldsymbol{P}$ and $\boldsymbol{Q}$ need to be transferred to GPU. Among them, $\boldsymbol{L_B}$ and $\boldsymbol{U_B}$ are spare triangular matrices, and in fact $\boldsymbol{P}$ and $\boldsymbol{Q}$ are expressed as two dense permutation vectors.

Step 3: Inversing intermediate matrix $\boldsymbol{B}$ by the batch F&B algorithm on GPU. In the whole process of matrix inversion, this step has the highest computation complexity and therefore constitutes the primary GPU-accelerated target. As mentioned in Section 4, by means of extracting various parallelisms among batch F&B task and a serial of performance tunings, Algorithm 3 can solve the batch F&B problem in an extremely fast manner. Therefore, we use Algorithm 3 to inverse the intermediate matrix $\boldsymbol{B}$, which consequently reduces the whole computing time dramatically.

Step 4: Calculating the inversion of target matrix $\boldsymbol{A}$ on GPU. As shown in (9), the target $\boldsymbol{A}^{-1}$ can be derived from $\boldsymbol{B}^{-1}$ by a serial of dense permutation operations $\boldsymbol{A}^{-1} = \boldsymbol{Q}^{\mathrm{T}}\boldsymbol{B}^{-1}\boldsymbol{PQ}$. In this step, two in-house GPU kernels are used: Kernel 1 realizes row exchanges of dense matrix $\boldsymbol{T} = \boldsymbol{Q}^{\mathrm{T}}\boldsymbol{B}^{-1}$, and Kernel 2 further realizes column exchanges $\boldsymbol{A}^{-1} = \boldsymbol{TPQ} = \boldsymbol{Q}^{\mathrm{T}}\boldsymbol{B}^{-1}\boldsymbol{PQ}$ where $\boldsymbol{T}$ is a temporary dense matrix. More specifically, one thread block is responsible for exchanging one row/column and every thread is responsible for one elements in this row/column. As every row/column exchange can be done independently of each other, the dense matrix permutation is naturally parallel and can achieve high parallelism as long as the matrix dimension is sufficiently big.

Step 5: Transferring analysis results back to CPU. According to *Criterion 4*, the data transmission between CPU and GPU should be minimized as much as possible, so the dense inversion matrix should never be directly transferred back to CPU. Instead, only a part of concerned matrix elements or the final analysis results, depending on different application, will be transferred. For example, in sensitivity analysis application, only the concerned sensitivity parameters will be chosen and transferred back to CPU.

It's important to note that, for a mainstream HPC-purposed GPU such as NVIDIA Tesla K40, a one-million-order sparse matrix is considered as a small-to-medium-scale computation problem. In contrast, the matrix of large-scale power system is typically less than 0.1 million order and so its inversion operation is only a small-scale calculation problem. That means, the GPU memory capacity is not a bottleneck. However, when the proposed algorithm is applied to larger-scale sparse matrix in other fields, whose size approaches the upper limit of GPU memory capacity, the compressed storage format [32–34] and the block algorithm [35–37] of sparse matrix will have a great influence on the computing performance, and so should be carefully tuned.

5.2. Case study of inversion algorithm

Test results on different cases are listed in Table 3. Taking Case 3 as an example, Step 1 consumed 15 ms, in which LU factorization and dependence graph generation consume 13.07 ms and 1.93 ms, respectively. In the Step 2, the total size of $\boldsymbol{L_B}$, $\boldsymbol{U_B}$, $\boldsymbol{P}$ and $\boldsymbol{Q}$ are about 0.81 MB and it takes 0.32 ms to transfer these data. The Step 3, based on batch F

Table 2
Device memory bandwidth varying with batch size.

Case no.	Memory bandwidth varying with batch size n (GB/s)							
	$n = 1$	$n = 32$	$n = 128$	$n = 512$	$n = 1024$	$n = 2048$	$n = 3072$	$n = 8192$
1	2.08	7.8	27.36	73.6	86.8	No test	No test	No test
2	3.37	11.4	45.0	70.8	91.7	118	119	No test
3	6.90	53.3	96.7	149	166	173	178	179

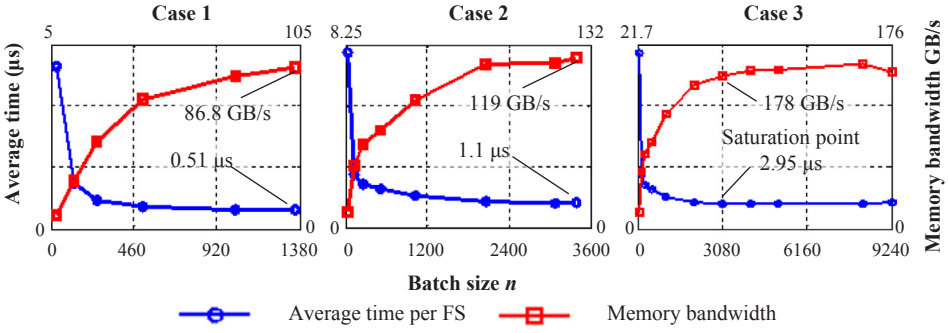

Fig. 7. Average computation time per FS and memory bandwidth varying with batch size.

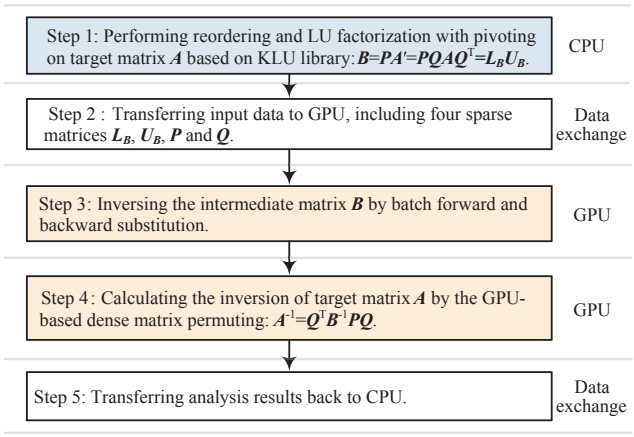

Fig. 8. Proposed GPU-accelerated inversion algorithm for large-scale sparse matrix.

Table 3
Computation time of matrix inversion on different case.

Step no.	Description of algorithm step	Computation time (ms)		
		Case 1	Case 2	Case 3
1	LU factorization with reordering and generating dependence graph	1.60	4.20	15.0
2	Transferring input data to GPU	0.12	0.18	0.32
3	Inversing intermediate inversion by batch F&B algorithm.	1.91	8.67	49.9
4	Calculating target inversion by dense matrix permuting.	0.54	4.14	31.4
5	Transferring result back to CPU[a]	1.40	8.65	65.0
Total		5.57	25.9	162

[a] Depending on different applications, if only a few concerned matrix elements is transferred, this time can be neglected.

&B algorithm, consumes 49.9 ms to calculate $\boldsymbol{B}^{-1}$. The Step 4 takes 31.4 ms to generate the target inversion $\boldsymbol{A}^{-1}$ by twice dense matrix permuting: Kernel 1 consumes 7.9 ms for row exchanges and Kernel 2 consumes 23.5 ms for column exchanges, respectively. It can be observed that Kernel 2 is about three times slower than Kernel 1 although the two have the identical degree of parallelism. The main reason is that the thread warp in Kernel 2 cannot fulfill coalesced memory access because it visits the 32 continuous elements in one column, which are not saved contiguously in GPU device memory. In Step 5, if the complete inversion matrix is transferred back to CPU, there is 0.636 GB data and it will consume 65 ms, with the actual measured bandwidth of about 9 GB/s. As a result, Case 3 can be finished within 97 ms (neglect transmission time of results) or 162 ms (transfer the complete inversion).

As shown in Fig. 9, the proportion of computing time spent on

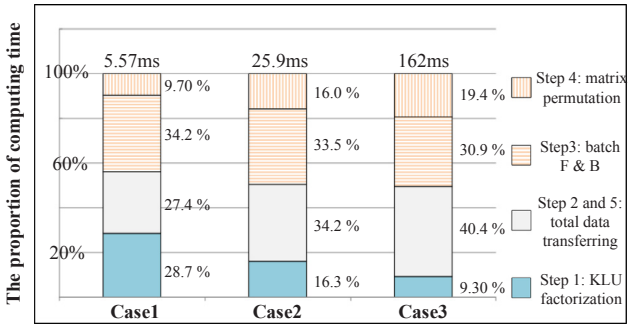

Fig. 9. The proportion of computing time spent on different step.

different step has the following features: With the increase of case dimension, the computing-load proportion of single LU factorization in Step 1 to the total inversion becomes lower and therefore its percentage continues to fall, from the 28.7% of Case 1 down to 9.30% of Case 3; In contrast, the percentage of batch F&B changes little, varying from 34.2% to 30.9%; Since the data size, involved in Step 4 (matrix permutation) and Step 5 (transferring dense inverse matrix back to CPU), is proportional to the square of matrix dimension n, the time consumption of both will increase rapidly with n increasing. On the contrast, Step 2 (transferring sparse original matrix into GPU) consumes very little time and can be ignored in the total data transmission. Consequently, the time proportion of matrix permuting increases from 9.7% to 19.4%, and the proportion of data transmission increases from 27.4% to 40.4%, which constitutes the two biggest bottlenecks of the whole inversion algorithm. Therefore, unless it is necessary, it should avoid transferring the complete inversion back to CPU.

In order to verify the correctness of GPU-accelerated inversion algorithm, the calculation result is compared with MATLAB. The difference of matrix element between the proposed algorithm and MATLAB is less than the order of magnitude of 10^{-13}, which is accurate enough for engineering applications.

In addition, the expensive Kernel 2 for dense column permutation can be removed by some mathematical transformations. Assuming $\boldsymbol{C} = \boldsymbol{Q}^{\mathrm{T}}\boldsymbol{P}^{\mathrm{T}}\boldsymbol{B}$, there exists

$$\boldsymbol{B}\boldsymbol{C}^{-1} = \boldsymbol{P}\boldsymbol{Q}\boldsymbol{I}. \tag{10}$$

The right-side matrix $\boldsymbol{PQI}$ is quite simple and its calculation time can be neglected (less than 0.02 ms in Case 3) because we only need to decide where the unique '1' in each row is.

The batch F&B algorithm can be directly applied to solve $\boldsymbol{C}^{-1}$ and then we can get the target inversion matrix $\boldsymbol{A}^{-1}$ by once row permutation $\boldsymbol{Q}^{\mathrm{T}}$.

$$\boldsymbol{A}^{-1} = \boldsymbol{Q}^{T}\boldsymbol{C}^{-1} = \boldsymbol{Q}^{T}(\boldsymbol{Q}^{T}\boldsymbol{P}^{T}\boldsymbol{B})^{-1} = \boldsymbol{Q}^{T}\boldsymbol{B}^{-1}\boldsymbol{P}\boldsymbol{Q} \tag{11}$$

Thus, the time consumed on Kernel 2 can be removed from the total time. For example, Case 3 can save about 23.5 ms, about 24.2% of total time (24.2% = 23.5/97).

Table 4
Performance comparison of different solutions on Case 3.

Solution no.	Solution description	Analysis time (ms)	Speedup
1	CPU solution based on single-threaded KLU	3566	Benchmark
2	CPU solution based on multi-threaded KLU	786	4.54× (vs. Solution 1)
3	CPU solution based on multi-threaded PARDISO	1386	2.57× (vs. Solution 1)
4	GPU solution based on accelerating single F&B subtask one after another	12,204	0.29× (vs. Solution 1)
5	GPU solution based on cuSolverSP library	2232 ms	1.6× (vs. Solution 1)
6	GPU solution based on the batch F&B solver	97	36.8× (vs. Solution 1) 8.1× (vs. Solution 2) 14.3× (vs. Solution 3) 126× (vs. Solution 4) 23× (vs. Solution 5)

6. Performance comparison and analysis

In order to show the enormous effect of proposed GPU-accelerated inversion algorithm, totally five computation solutions are tested and compared with each other on Case 3. At last, the performance saturation problem is discussed.

6.1. Performance comparison

As shown in Table 4, Solution 1 adopts the single-core CPU computing architecture, in which all F&B subtasks are solved in series using KLU library [24]. To make a fair performance comparison, two multi-core-CPU-accelerated solvers, based on KLU library and PARDISO library [38], respectively, are used in Solution 2 and Solution 3. Solution 4 adopts the GPU-based solver for single sparse triangular linear system [16,17]. Solution 5 adopts cuSolverSP library provided by CUDA to accelerate single sparse triangular linear system [39]. Solution 6 adopts the proposed GPU-based batch F&B solver. Taking Case 3 as an example, the performances of aforementioned five solutions are tested and Solution 1 is chosen as the benchmark. More analyses and comparisons are given as follows.

(1) *Solution 1: solving inversion based on single-threaded KLU library.* KLU is one of the fastest single-threaded library for solving SLS in power systems. It consumes about 15 ms and 0.384 ms on LU factorization and single F&B subtask, respectively, and therefore the computation time of Solution 1 is about 3566 ms (3566 ms ≈ 15 ms + 9041 * 0.384 ms).
(2) *Solution 2: solving inversion based on multi-threaded KLU library.* This solution includes two steps: the first step executes single-threaded LU factorization once, and the second step solves a set of F&B subtasks with the same factor matrix in parallel. Based on the multi-threaded technologies, such as OpenMP, we can start multiple function instances, running on different physical CPU cores, to solve these F&B subtasks simultaneously. For example, if there are N_c physical CPU cores, N_c threads will be created and work together to accomplish the computing task. As shown in Fig. 10, the computing time of Case 3 will decrease rapidly with the number of CPU cores/threads increasing, and start to saturate at 8 CPU cores, at which the computation time is 786 ms and has achieved 4.54 times speedup relative to Solution 1.
(3) *Solution 3: solving inversion based on multi-threaded PARDISO library.* PARDISO is a parallel sparse direct solver and it provides the function interface $\boldsymbol{AX} = \boldsymbol{B}$ where $\boldsymbol{A}$ and $\boldsymbol{X}$, $\boldsymbol{B}$ are n by n and n by $nrhs$ matrices. Therefore, we can use this interface to solve inversion by setting $\boldsymbol{B} = \boldsymbol{I}$. Based on the 12-threaded PARDISIO solver, Solution 3 consumes about 1386 ms and achieves 2.57 times speedup versus Solution 1.
(4) *Solution 4: solving inversion based on GPU-based solver for single sparse triangular linear system.* Solution 4 uses GPU to accelerate each F&B subtask one after another. As mentioned in Section 4, this solution suffers from the small problem scale, excessively sequential computation process and massive uncoalesced memory access, and so it will consume about 1.319 ms on single F&B subtask, about 3.4 times slower than KLU (3.4 ≈ 1.319 ms/0.384 ms). Consequently, Solution 4 consumes about 12204 ms, about 3.42 times slower than Solution 1.
(5) *Solution 5: solving inversion based on cuSolverSP library provided by CUDA.* Solution 5 uses cuSolverSP library provided by CUDA to accelerate each F&B subtask one after another. It consumes 0.24 ms on single F&B subtask, which is about 1.6 times faster than the single-threaded KLU (1.6 ≈ 0.384 ms/0.24 ms). Consequently, Solution 5 consumes 2232 ms to compute the inverse matrix, about 1.6 times faster than Solution 1.
(6) *Solution 6: solving inversion based on GPU-based batch F&B algorithm.* Solution 6 consumes about 97 ms, which neglects transmission time of results, and achieves 36.8 times speedup versus Solution 1. Even in comparison to aforementioned CPU-accelerated parallel solution 2 and solution 3, it can still achieve about 8.1 times and 14.3 times speedup, respectively.

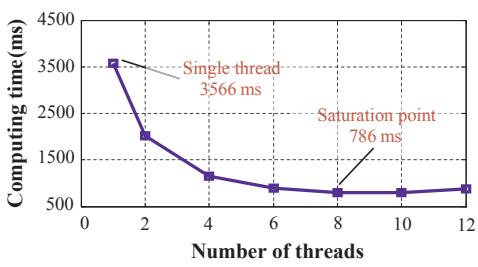

Fig. 10. The acceleration performance of multi-threaded inversion algorithm, at most 12 physical CPU cores are used.

6.2. Discussion on performance saturation problem

It is worth mentioning that a big disadvantage for CPU-based parallel inversion solution is the performance saturation issue. As shown in Fig. 10, the performance improvement of Solution 2 will stop at 8 CPU cores, which means that adding additional CPU cores does not further improve the computational performance and even results in slight performance decline. The reasons are as below.

According to the Amdahl's law

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}} \tag{12}$$

where S_{latency} is the theoretical speedup of the whole task, s is the speedup of the part of the task that benefits from improved system resources, p is the proportion of execution time that the part benefiting from improved resources originally occupied [40].

In Solution 1, the proportion

$$p = \frac{t_{F\&B}}{t_{LU} + t_{F\&B}} = \frac{3551}{15 + 3551} = 0.996 \tag{13}$$

is very high, where $t_{LU} = 15$ ms is the execution time of LU factorization and $t_{F\&B} = 3551$ ms is the execution time of parallelizable F&B subtasks. It can be observed that the biggest bottleneck of Solution 2 lies in that the speedup of solving the F&B subtasks in parallel is too small. The main reason is the memory bandwidth of this computing node has been exhausted. To overcome that, more computing nodes will have to be added along with separate memory installed in order to get extra memory bandwidth. However, it will dramatically increase capital investment, floor space requirement and computing power consumption. In contrast, the saturation problem of GPU-based algorithm can be easily resolved by adding more GPU cards in the same server to obtain extra memory bandwidth as every GPU has its separate device memory.

In addition, the Amdahl's law can be used to assess maximal speedup potential of proposed GPU-based algorithm. The parallelizable proportion

$$p = \frac{t_{parallelizable}}{t_{\text{total}}} = \frac{t_{\text{step 3}} + t_{\text{step 4}}}{t_{\text{total}}} = \frac{81.3}{97} = 0.838 \tag{14}$$

where $t_{\text{parallelizable}}$ and t_{total} represent the time consumption of parallelizable parts and the total time consumption, respectively. Step 3 and Step 4 are parallelizable parts, so $t_{\text{parallelizable}} = t_{\text{Step 3}} + t_{\text{Step 4}}$. Now, we can calculate the maximal speedup potential when $s = \infty$. That means, we can achieve 6.17 times extra speedup at most.

$$S_{\max}(s) = \frac{1}{(1-p) + \frac{p}{s}} = \frac{1}{1 - 0.838} = 6.17 \tag{15}$$

7. Conclusion

This paper firstly presents four general design criteria for a well-performing GPU algorithm. Next, by means of solving multiple F&B subtasks concurrently and a serial of performance tunings in compliance with GPU's SIMT architecture, we successfully develop a GPU-based batch F&B algorithm with higher parallelism, better memory access efficiency and perfect thread convergence. Lastly, the GPU-based inversion algorithm is designed.

Case studies on Case 3 shows that, when the batch size is equal to 3072, the proposed batch F&B solver on NVIDIA Tesla K40 reaches its performance saturation point 2.95 μs per FS and achieves 65 times speedup compared with KLU. At the same time, the memory bandwidth increases up to 178 GB/s, about 80% of available peak bandwidth of K40. By means of offloading the massive F&B subtasks to GPU, the proposed inversion algorithm can inverse large-scale SLSs in an extremely fast manner. For example, Case 3 takes only 97 ms and achieves 36.8/8.1 times speedup compared with the single-core/12-core CPU solution based on KLU library, respectively.

The proposed GPU-accelerated F&B solver is practically very promising, especially in an online environment, and can be widely used in many other power system applications, such as probabilistic power flow analysis based on PQ decouple method.

Acknowledgements

The authors gratefully thank Lung-Sheng Chien (software engineer at NVIDIA, Santa Clara, CA 95050, USA) for his valuable suggestions in performance tuning strategies for GPU algorithm. This study was supported by the National Natural Science Foundation of China (Grant No. 51877038) and the Science and Technology Foundation of State Grid Corporation of China: High-Performance Computing Technology for Analysis and Service on Entire Network of STATE GRID Corporation of China (Grant No. DZB17201800023).

Appendix A. Supplementary material

Supplementary data to this article can be found online at https://doi.org/10.1016/j.ijepes.2019.03.074.

References

[1] Pablo Ezzatti, QuintanaOrtí Enrique S, Remon A. High performance matrix inversion on a multi-core platform with several GPUs. 19th international euromicro conference on parallel, distributed and network-based processing. 2011.
[2] Sun Z, Wang H. Arbitrary larger size matrix inversion on GPU. International conference on electrical, control and automation engineering. 2013.
[3] Balestrino A, Cannata G. Inversion of matrices and matrix functions as a nonlinear discrete system: stability and sensitivity analysis. IEE Proc-Control Theory Appl 2001;148(1):43–8.
[4] Broussolle F. State estimation in power systems: detecting bad data through the sparse inverse matrix method. IEEE Trans Power App Syst 1978;PAS-97(3):678–82.
[5] Morelato A, Amano M, Kokai Y. Combing direct and inverse factors for solving spare network equation in parallel. IEEE Trans Power Syst 1994;9(4):1942–8.
[6] Shultz RD, Muslu M, Smith RD. A new method in calculating line sensitivities for power system equivalencing. IEEE Trans Power Syst 1994;9(3):1465–70.
[7] Green RC, Wang L, Alam M. Applications and trends of high performance computing for electric power systems: focusing on smart grid. IEEE Trans Smart Grid 2013;4(2):922–31.
[8] Huang HS, Lu CN. Efficient storage scheme and algorithms for W-matrix vector multiplication on vector computers. IEEE Trans Power Syst 1994;9(2):1083–91.
[9] Aykanat C, Guven N. Algorithms for efficient vectorization of repeated sparse power system network computations. IEEE Trans Power Syst 1995;10(1):448–56.
[10] Vuong GT, Chahine R, Granelli GP, Montagna M. Dependency-based algorithms for vector processing of sparse matrix forward/backward substitutions. IEEE Trans Power Syst 1996;11(1):198–205.
[11] Montagna M, Granelli GP, Vuong GT, Chahine R. Levelwise algorithms for vector processing of sparse power system matrices. IEEE Trans Power Syst 1996;11(1):239–45.
[12] Basso AR, Minussi CR, Padilha A. Fast-forward/fast-backward substitutions on vector computers. IEEE Trans Power Syst 1999;14(4):1369–74.
[13] Chen D, Li Y, Jiang H, Xu D. A parallel power flow algorithm for large-scale grid based on stratified path trees and its implementation on GPU. Autom Electr Power Syst 2014;38(22):63–9.
[14] Li X, Li F. GPU-based power flow analysis with Chebyshev preconditioner and conjugate gradient method. Electr Power Syst Res 2014;116(11):87–93.
[15] Chen X, Ren L, Wang Y, Yang H. GPU-accelerated sparse LU factorization for circuit simulation with performance modeling. IEEE Trans Parallel Distrib Syst 2015;26(3):786–95.
[16] Naumov M. On the parallel solution of sparse triangular linear systems. Online available: < http://on-demand.gputechconf.com/gtc/2012/presentations/S0149-Parallel-Solution-of-Sparse-Triangular-Linear-Systems.pdf > .
[17] Chien LS. How to avoid global synchronization by domino scheme. Online available: < http://on-demand.gputechconf.com/gtc/2014/presentations/S4188-avoid-global-synchronization-domino-scheme.pdf > .
[18] Roberge Vincent, Tarbouchi Mohammed, Okou Francis. Parallel power flow on graphics processing units for concurrent evaluation of many networks. IEEE Trans Smart Grid 2017;8(4):1639–48. https://doi.org/10.1109/TSG.2015.2496298.
[19] Belic E, Lukac N, Dezelak K, Zalik B, Stumberger G. GPU-based online optimization of low voltage distribution network operation. IEEE Trans Smart Grid 2017;8(3):1460–8.
[20] Zhou G, Feng Y, Bo R, Chien L, et al. GPU-accelerated batch-ACPF solution for N-1 static security analysis. IEEE Trans Smart Grid 2017;8(3):1406–16.
[21] Abdelaziz Morad. GPU-OpenCL accelerated probabilistic power flow analysis using Monte-Carlo simulation. Electr Power Syst Res 2017;147:70–2.
[22] Cook S. CUDA programming: a developer's guide to parallel computing with GPUs. Morgan Kaufmann Publishers Inc.; 2012.
[23] NVIDIA Corporation. NVIDIA CUDA C programming guide. Online available: < http://docs.nvidia.com/cuda/cuda-c-programming-guide/ > .
[24] Davis TA, Natarajan EP. Algorithm 907: KLU, a direct sparse solver for circuit simulation problems. ACM Trans Math Softw 2010;37(7):1–17.
[25] Davis TA. Direct methods for sparse linear systems. Soc Ind Appl Math 2006;33(3):420–60.
[26] Lindholm E, Nickolls J, Oberman S, Monyrym J. NVIDIA Tesla: a unified graphics and computing architecture. IEEE Micro 2008;28(2):39–55.
[27] NVIDIA Corporation. NVIDIA's next generation CUDA compute architecture: Kepler GK110. Online available: < http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf > .
[28] Yang W, Li K, Li K. A hybrid computing method of SpMV on CPU-GPU heterogeneous computing systems. J Parallel Distrib Comput 2017;104:49–60.
[29] Zhou G, Zhang Xu, Lang Y, et al. A novel GPU-accelerated strategy for contingency screening of static security analysis. Int J Electr Power Energy Syst 2016;83(Dec):33–9.
[30] Chen X, Ren L, Wang Y, Yang H. GPU-accelerated sparse LU factorization for circuit simulation with performance modeling. IEEE Trans Parallel Distrib Syst 2015;26(3):786–95.
[31] Li H, Li K, Jiyao A, et al. MSGD: a novel matrix factorization approach for large-scale collaborative filtering recommender systems on GPUs. IEEE Trans Parallel Distrib Syst 2018;29(7):1530–44.
[32] Li K, Yang W, Li K. Performance analysis and optimization for SpMV on GPU using probabilistic modeling. IEEE Trans Parallel Distrib Syst 2015;26(1):196–205.
[33] Guo P, Wang L, Chen P. A performance modeling and optimization analysis tool for sparse matrix-vector multiplication on GPUs. IEEE Trans Parallel Distrib Syst

2014;25(5):1112–23.
[34] Yang W, Li K, Liu Y, et al. Optimization of quasi-diagonal matrix-vector multiplication on GPU. Int J High Perform Comput Appl 2014;28(2):183–95.
[35] Yang W, Li K, Mo Z, et al. Performance optimization using partitioned SpMV on GPUs and multicore CPUs. IEEE Trans Comput 2015;64(9):2623–36.
[36] Yang W, Li K, Li K. A parallel computing method using blocked format with optimal partitioning for SpMV on GPU. J Comput Syst Sci 2018;92:152–70.
[37] Li K, Yang W, Li K. A hybrid parallel solving algorithm on GPU for quasi-tridiagonal system of linear equations. IEEE Trans Parallel Distrib Syst 2016;27(10):2795–808.
[38] Schenk O, Gärtner K. Solving unsymmetric sparse systems of linear equations with PARDISO. J Future Gener Comput Syst 2004;20(3):475–87.
[39] NVIDIA Corporation. CUDA toolkit documentation: CuSolver Library. Online available: < https://docs.nvidia.com/cuda/cusolver/index.html#cusolver-function-reference > .
[40] Hill Mark D, Marty Michael R. Amdahl's law in the multicore era. Computer 2008;41(7):33–8.