GPU-based Multifrontal Optimizing Method in Sparse Cholesky Factorization

Ran Zheng, Wei Wang, Hai Jin, Song Wu, Yong Chen
Services Computing Technology and System Lab
Cluster and Grid Computing Lab
School of Computer Science and Technology
Huazhong University of Science and Technology
Wuhan, 430074, China
Email:hjin@hust.edu.cn

Han Jiang
Department of Power System
China Electric Power Research Institute
Beijing, 100085, China

***Abstract*—In many scientific computing applications, sparse Cholesky factorization is used to solve large sparse linear equations in distributed environment. GPU computing is a new way to solve the problem. However, sparse Cholesky factorization on GPU is hardly to achieve excellent performance due to the structure irregularity of matrix and the low GPU resource utilization. A hybrid CPU-GPU implementation of sparse Cholesky factorization is proposed based on multifrontal method. A large sparse coefficient matrix is decomposed into a series of small dense matrices (frontal matrices) in the method, and then multiple GEMM (*General Matrix-matrix Multiplication*) operations are computed. GEMMs are the main operations in sparse Cholesky factorization, but they are hardly to perform better in parallel on GPU. In order to improve the performance, the scheme of multiple task queues is adopted when performing multiple GEMMs parallelized with multifrontal method; all GEMM tasks are scheduled dynamically on GPU and CPU based on computation scales for load balance and computing-time reduction. Experimental results show that the approach can outperform the implementations of BLAS and cuBLAS, achieving up to $3.15\times$ and $1.98\times$ speedup, respectively.**

Keywords*—*Multifrontal method; Multiple task queues scheme; Task allocation; GPU Acceleration

I. Introduction

In varieties of scientific computing and engineering applications, such as optimization, computational fluid dynamics problem, it is the all-important component to solve large systems of linear equations with the form of $Ax = b$. Over the past decades, many researchers had devoted themselves to this problem with the solutions of direct or iterative methods [1]. In a large number of applications, A is the large and sparse *Symmetric Positive Definite* (SPD) matrix. Direct method, e.g. Cholesky factorization, is widely used to solve sparse SPD linear equations for its high performance, accuracy, and robustness. It factors A into the product LL^T, where L is a lower triangular matrix. Many researchers have sought to exploit novel solutions in distributed memory and shared memory systems to reduce the overall computing time of sparse direct solvers [2][3]. With the increase of matrix scale, CPU cannot process such computation, so we explore the method of accelerating the factorization of large sparse SPD matrix by GPUs.

There exist many algorithms to implement sparse Cholesky factorization. Among them, the multifrontal method [4] shows tremendous attractiveness, and it factors a large sparse matrix into a series of small dense matrices. The factorization of dense matrix involves *general matrix-matrix multiplication* (GEMM) operation associated with a large number of floating point arithmetics, which can be performed by invoking the efficient BLAS (*Basic Linear Algebra Subroutines*) [5] or cuBLAS [6] routines, and cuBLAS is an implementation of standard BLAS on GPU.

The challenge is that sparse direct solvers involve irregular memory access, complex parallelization, strong-dependence on A's structure. With the emergence of GPU, it has gained widespread popularity due to enormous computing power. In recent years, there are several literatures to accelerate sparse direct solvers on GPUs [7][8][9]. The approaches in these literatures involve off-loading large dense operations to GPU to accelerate the performance, and the other operations are processed on host CPU. However, no highly parallel hybrid CPU-GPU implementation of sparse Cholesky factorization exists.

In this study, a parallel hybrid CPU-GPU algorithm of sparse Cholesky factorization is presented. An existing shared-memory sparse direct solver, *TAUCS* [10], is extended by adding GPU acceleration during Cholesky factorization. Although multiple frontal matrices, which are not in an ancestor-descendant relationship, can be factorized in parallel in *TAUCS*, it is hard to implement the parallel scheme on GPU through current GPU programming paradigm. In order to achieve higher performance, two strategies are proposed: the first is to adopt multiple task queues scheme to perform GEMM operations in parallel, which can guarantee the kernel computation from multiple frontal matrices be overlapped; the second is to set a calculating quantity threshold to decide the platform of GEMM operation, i.e. CPU or GPU. Specifically, if the calculated quantity is larger than the threshold, then the operation is offloaded to GPU, otherwise, it will be processed on CPU.

The rest of the paper is organized as follows. Section II describes related work of sparse linear solvers on GPU. Section III gives a background of multifrontal method and *TAUCS*, and then exploits the parallelism of *TAUCS* based on GPU. The design of multiple task queues scheme is proposed in

Section IV. A farther strategy is adopted to reduce the overall computation time in Section V. Section VI shows experimental results and Section VII summarizes the conclusions.

II. Related Work

In recent years, sparse linear solvers on GPU have attracted many researchers' attentions.

George et al. presented an adaptive hybrid approach for accelerating multifrontal sparse Cholesky factorization based on a coupled CPU-GPU [11]. They put forward four policies to distribute the workload between CPU and GPU. They obtained significant improvements over WSMP [12] implementation.

Sao et al. proposed a hybrid CPU-GPU implementation of a distributed memory right-looking unsymmetric sparse direct solver [8]. It aggregated the collections of small BLAS operations into larger ones, hid long-latency operations and exploited parallelism between CPU and GPU. Experiments verified that their method outperformed SUPERLU_DIST [13].

Lucas et al. utilized GPUs to accelerate the factorization of large sparse indefinite matrices, which demonstrated that GPU could dramatically accelerate the factorization relative to one host CPU [14]. Vuduc et al. exploited multithreaded CPU and GPU implementations of sparse Cholesky factorization, which confirmed that the structure of input matrix determined the performance [15]. Yu et al. proposed three strategies for accelerating the factoring of unsymmetric multifrontal method based on a hybrid CPU-GPU system, which achieved remarkable result for computation-expensive problems [16]. Yeralan et al. presented a sparse multifrontal QR factorization algorithm on GPU, and experiments exhibited that their method was up to ten times faster than a highly optimized method on a multicore CPU [17]. Li et al. integrated multifrontal method and cuBLAS to accelerate sparse direct solver, which greatly improved the performance [7]. Ren et al. proposed a GPU-based sparse LU solver [18], which optimized work partitioning, and memory access pattern. Their experimental results showed $1.49\times$ speedup over multicore CPU.

However, most of the above methods rely on GPU to accelerate the operation of the large compute-intensive dense matrices, and process the other matrices on CPU. Since multiple frontal matrices can not be factorized in parallel on GPU, the computation resources of GPU cannot be utilized sufficiently. Our approach can take the utilization of GPU's computational resource into consideration, and propose multiple task queues scheme. Meanwhile, threshold setting is adopted to improve the performance of GEMM operation.

III. System Analysis and Design

In this section, the background information of multifrontal sparse Cholesky factorization and *TAUCS* are introduced briefly, and then the parallelism scheme of multifrontal algorithm on GPU is exploited.

A. Analysis of Multifrontal Method and TAUCS

Multifrontal method is widely used in solving sparse linear equations. The factorization procedure of sparse Cholesky factorization based on multifrontal method can be summarized in the following main steps:

- *Forming matrix*: row data is transformed into original matrix A.
- *Preprocessing*: The original matrix A is transformed into a new matrix B by column pre-ordering to reduce potential fill-ins (new non-zero in L but not in A).
- *Symbolic factorization*: B is decomposed into a series of small dense matrices. The elimination tree and the number of non-zero in matrix L are gotten in this phase.
- *Numerical factorization*: Frontal matrix F^n and update matrix U^n are processed. For each node in elimination tree, firstly, frontal matrix is formed; then, pivot rows and columns in the current frontal matrix are factorized; lastly, update matrix is updated. The main operation in numerical factorization involves three arithmetic operations: Cholesky factorization, trianglular matrix multiplication and symmetric rank-k update.
- *Forward and Backward*: After A is factorized into the product LL^T, x can be solved by a forward and backward triangular solve.

As known to all, numerical factorization is the most compute-intensive component in multifrontal method. The experiments in [11] show that the processing of frontal matrices and update matrices consume about 80% of the overall computation time for large matrices. Therefore, we mainly focus on the numerical factorization phase and take some effective measures to accelerate the computation of frontal matrices and update matrices in this paper.

TAUCS [10] is a C language library for sparse linear solvers, in which multithreaded multifrontal supernodal sparse Cholesky factorization has been implemented. The parallel component of sparse multifrontal Cholesky factorization factorizes multiple frontal matrices at the same time. This parallel algorithm has been implemented with *Cilk*[19], which is a language for multithreaded parallel programming. The programmer can concentrate on structuring the program to expose parallelism, and the *Cilk* runtime system takes communication protocols, load balancing, etc into account. *TAUCS* is an open source library and has been implemented on CPU, so we can modify it and carry it out on GPU to gain more efficient performance.

B. Parallel Design of TAUCS base on Hybrid CPU-GPU

The main computation component of multifrontal algorithm is the computation of F^n and U^n, which involves vast dense linear operations. The multiple frontal matrices are factorized in parallel in *TAUCS* by multiple CPU threads. BLAS is introduced to process the above dense operations. As GPUs have been developed to be massively parallel computing devices with excellent computation power and memory bandwidth, we attempt to explore GPU to accelerate multifrontal algorithm.

According to the design philosophy of *TAUCS*, the computation focuses on F^n and U^n, and the processing procedure of F^n and U^n can be divided into four parts in *TAUCS*:

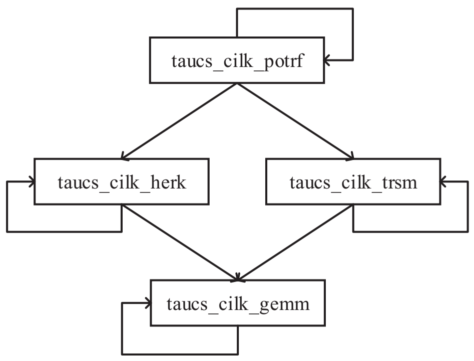


Fig. 1. The function call relationship in factoring one frontal matrix

- *taucs_cilk_potrf*: a Cholesky factorization (*POTRF*) on the frontal $k \times k$ block, where k is the size (the number of consecutive rows and columns with the same nonzero pattern in L) of supernode n;
- *taucs_cilk_trsm*: solve a triangular linear system with multiple right-hand-sides (*TRSM*) on matrices of size $m \times k$ and $k \times k$, where m is the number of non-pivot elements of the last pivot column;
- *taucs_cilk_herk*: a symmetric rank-k update (*SYRK*) on a matrix of size $m \times m$, where the matrix is symmetric and stored in lower or upper mode;
- *taucs_cilk_gemm*: a GEMM operation of the form $C = AB$ with dimensions A ($m_1 \times n_1$), B ($n_1 \times k_1$) and C ($m_1 \times k_1$), respectively. A, B and C are all full matrices.

In the above analysis, the factorization of frontal matrix is made up of functions *taucs_cilk_potrf*, *taucs_cilk_herk*, *taucs_cilk_trsm* and *taucs_cilk_gemm*. There are some dependency relationship among the above four functions. From the Fig. 1, we can observe that *taucs_cilk_gemm* is the foundation of the other three functions, i.e. both of *taucs_cilk_herk* and *taucs_cilk_trsm* directly call it and *taucs_cilk_potrf* calls it indirectly, since *taucs_cilk_potrf* needs to call *taucs_cilk_herk* and *taucs_cilk_trsm*. In addition, all of the four functions can be called by themselves.

The dependency relationship among the above four functions indicates that the large matrix processed by each function is divided into a series of submatrices and then can be processed in parallel. Therefore, the performance of multicore CPU can be sufficiently utilized by BLAS. However, due to the limited cores of CPU, the resource factorizing multiple frontal matrices in parallel is not sufficient. While, excellent performance can easily be achieved by these matrix operations on GPU due to the outstanding computational power of GPU and its specialized structure.

As a result of the vast computational power, GEMM operation can be suitably implemented on GPU. The size of a node is limited, and Cholesky factorization on a $k \times k$ matrix cannot sufficiently utilize the computing resource of GPU, therefore, *POTRF* should be implemented on CPU. Both of *TRSM* and *HERK* involve the operations of triangular matrix, and the computation time by CPU outperforms GPU because of the overhead of data transfer on GPU. Based on this analysis, we draw the system design of numerical factorization based on CPU and GPU, as shown in Fig. 2. The factorization of single frontal matrix generates several GEMM operations. So, when multiple frontal matrices are factorized in parallel, there will exist a large number of GEMM operations to process. Therefore, how to perform plenty of GEMM operations more efficiently?

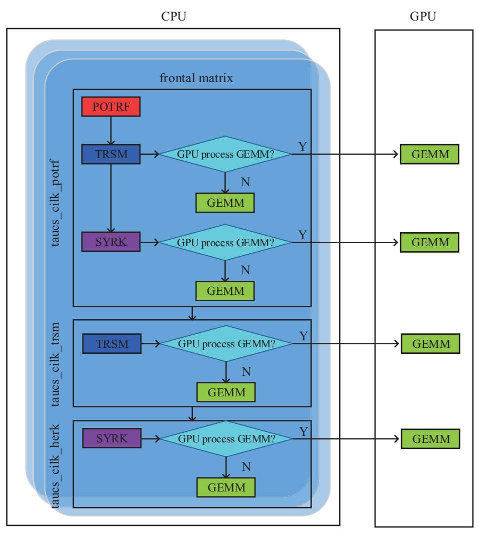


Fig. 2. System design of numerical factorization

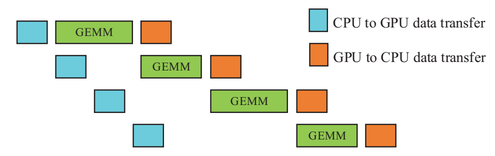
We attempt to explore the parallelism of multiple GEMM operations on GPU with the following steps:

- A *multiple task queues scheme* (MTQS) is proposed to solve GEMM operations in parallel. It is difficult to perform GEMM operations in parallel on GPU by current GPU programming paradigm. Although cuBLAS achieves excellent performance to accelerate single GEMM operation and can be executed asynchronously, only data transfer and kernel launching can be executed in parallel, as shown in Fig. 3(a). Therefore, this method can not satisfactorily address the problem of insufficient GPU resource utilization. On the contrary, multiple task queues scheme can execute multiple GEMM operations in parallel, as shown in Fig. 3(b), which makes GPU computation resource utilization more adequate.
- In order to achieve high performance, the GEMM operations involving large computation scale should be processed on GPU because the overhead of data transfer can be overlapped. On the contrary, the other GEMM operations should be processed on CPU. In order to make the factorization of frontal matrix more efficient, a threshold is set to decide whether GPU should be used or not.

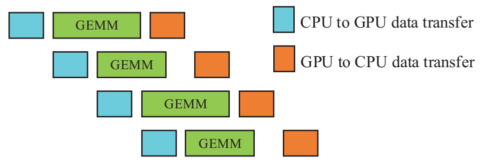
IV. Design of Multiple Task Queues Scheme

A. Task Queue Scheme

Chen et al. developed a task queue based load balancing scheme for GPU systems [20]. In order to execute multiple



(a) overlapping GEMM operations with current GPU programming paradigm



(b) overlapping GEMM operations with MTQS

Fig. 3. Execution mode of GEMM operations on current GPU programming paradigm and MTQS

tasks, they launch a persistent kernel with B TBs (*thread block*), where B is the maximum number of concurrently active TBs that a specific device can support. The host process inserts task to task queue. The kernel takes out tasks from task queue and executes them according to task information. The document [20] adopted two different CUDA streams to implement the task queue scheme. One stream is used for kernel execution, and the other stream is used for performing queue operations.

However, there are some deficiencies in the task queue scheme. First, only one stream is adopted to perform queue operations, when this stream is waitting for the result from device, the other tasks can not be processed until data transfer from device to host is finished, causing the serious performance issue; second, in order to determine the non-empty task queue, each TB has to check all the task queues, which will cause additional overhead on empty task queues, in addition, some measures must be taken to avoid multiple TBs accessing the same task queue; last, if the host needs to run in multithreaded environment, this scheme will not work correctly.

Although the above task queue scheme has some disadvantages, it can also be introduced to optimize multifrontal method on GPU. In order to make task queue scheme more efficient, some measures are taken to overcome these disadvantages, and the optimizing scheme is called multiple task queues scheme.

B. Multiple Task Queues Scheme

1) System Design

According to the characteristic of *TAUCS*, there are multiple threads running in host concurrently. In order to execute multiple GEMM operations in parallel, n threads are created on host and each thread processes one GEMM operation, where n is the number of host processors. When one GEMM operation is ready, the corresponding thread will choose one empty queue to place matrices. When data transfer from host to device is finished, the host thread will inform the device kernel to perform GEMM operation, and it goes into the waiting state. Then the device kernel fetches matrices from the queue and executes them by groups of threads, which is called *task execution unit* (TEU). When the computation on the kernel is finished, the TEU will inform the host thread to get the result

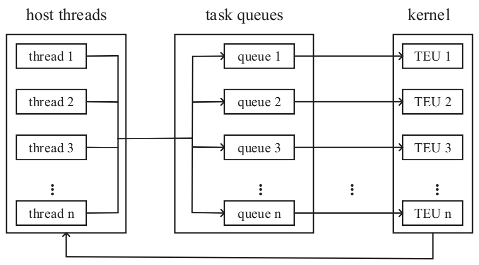


Fig. 4. Multiple task queues scheme

and it enters into the waiting state. Then the host thread copies matrix from device back to host, and processes other GEMM operation. After all GEMM operations are executed, the host thread will enqueue signalling task to terminate the kernel. The multiple task queues scheme is illustrated in Fig. 4.

2) Difficulties

If we want to implement the scheme of multiple task queues, the following issues must be solved.

The first issue is how to avoid multiple host threads accessing the same queue. Before host thread inserts a task, it needs to be confirmed which queue is empty. As *Cilk* is used for task scheduling, each queue can not be assigned to a specific host thread. Therefore, in order to determine the non-empty queue, each host thread has to check all queues. If multiple host threads access the same queue, the content of the queue is uncertain, so the ultimate result is uncertain. Lock can guarantee that multiple host threads choose different queues, but it is very costly because of the utilization of blocking synchronization mechanism.

The second issue is how to reduce the overhead for searching the non-empty queue on GPU. If each TEU searches the non-empty queue by checking all queues, it has to spend extra time on empty queues, causing the performance issue. At the same time, multiple TEUs may operate the same queue resulting in data race. Although atomic functions can ensure the correctness on operating the same queue, in order to confirm the non-empty queue, all the queues must be checked.

Atomic functions can ensure the correctness on accessing the same queue by multiple host threads. If one host thread needs to make sure which is the non-empty queue, the function *__sync_bool_compare_and_swap* is called to check whether the queue is empty. When one non-empty queue is chosen, host thread will transfer data from host to device. Once the computation on GPU has completed, the function *__sync_fetch_and_sub* is called to reset the queue to be null. According to the design philosophy of atomic functions, they will not be interrupted until one command has completed. Therefore, although multiple host threads try to access the same queue, only one can actually enter the queue. This solves the first issue.

In order to decrease the overhead for seeking the non-empty queue, the best choice is to avoid empty queue as much as possible. Based on the mentioned above, the optimal strategy is that each TEU has a specified queue. First, the host thread inserts the task to empty queue, and then it informs TEU to take out the task from the queue and waits for the computation result from TEU, finally, the TEU gets the task, processes it and returns the result to host thread. This solves the second issue.

V. GEMM Distribution between CPU and GPU

There are vast GEMM operations in *TAUCS*, some involve large computation cost and some involve small computation cost. As the excellent computation power, the GEMM operation that involves large scale matrix can be performed better on GPU. Small scale matrix can not sufficiently utilize GPU's computation resources and there is data transfer overhead between CPU and GPU, therefore, the GEMM operation that involves small scale matrix should be performed on CPU. Based on the above analysis, a threshold should be set to determine whether a GEMM operation should be performed on GPU or not.

Let T_{cpu} denote the time on CPU for GEMM operation, while T_{gpu} denotes the same in the case of GPU. The time for host CPU implementation can be estimated as (1):

$$T_{cpu} = \frac{N_1}{\alpha_{cpu}} \tag{1}$$

The basic GPU implementation includes data copy costs as (2):

$$T_{gpu} = \frac{N_1}{\alpha_{gpu}} + \frac{N_2}{\beta} \tag{2}$$

where $N_1 = m_1 \cdot n_1 \cdot k_1$ is the number of operations for GEMM, and $N_2 = m_1 \cdot k_1 + n_1 \cdot k_1 + 2 \cdot m_1 \cdot n_1$ is the size of data transfer. α_{cpu} and α_{gpu} represent the average computing speeds for GEMM operation on CPU, GPU respectively. β represents the average speed for data transfer on CPU-GPU.

When T_{cpu} is greater than or equal to T_{gpu}, GEMM operation should be performed on GPU, and the relationship among m_1, n_1, k_1 is as (3):

$$T_{cpu} \geq T_{gpu} \tag{3}$$

Then put (1), (2) and N_1, N_2, into (3), and make some conversion to form (4):

$$\frac{1}{m_1} + \frac{1}{n_1} + \frac{2}{k_1} \leq \beta \cdot \left(\frac{1}{\alpha_{cpu}} - \frac{1}{\alpha_{gpu}}\right) \tag{4}$$

According to the mathematical theorem, (4) can be converted into (5):

$$\beta \cdot \left(\frac{1}{\alpha_{cpu}} - \frac{1}{\alpha_{gpu}}\right) \geq \frac{1}{m_1} + \frac{1}{n_1} + \frac{2}{k_1} \geq 3 \cdot \sqrt[3]{\frac{2}{m_1 \cdot n_1 \cdot k_1}} \tag{5}$$

Then make some conversion to form (6):

$$m_1 \cdot n_1 \cdot k_1 \geq \frac{54 \cdot \alpha_{cpu}^3 \cdot \alpha_{gpu}^3}{(\alpha_{gpu} - \alpha_{cpu})^3 \cdot \beta^3} \tag{6}$$

We denote the latter part of (6) as N_t, i.e., the threshold of determing if GPU is used.

Equation (6) provides theoretical foundation for GEMM operation distribution between CPU and GPU. For each GEMM operation, N_1 with N_t are compared to decide whether GPU should be applied.

TABLE I. The description of testing matrices

Name	Dimension	NNZ	Sparseness	Application
minsurfo	40806^2	203622	0.999878	Optimization
2cubes_sphere	101492^2	1647264	0.999840	Electromagnetics
thermomech_dM	204316^2	1423116	0.999966	Thermal
parabolic_fem	525825^2	3674625	0.999987	Computational fluid dynamics
apache2	715176^2	4817870	0.999991	Structural
ecology2	999999^2	4995991	0.999995	2D/3D

VI. Experimental Results

In this section the experimental configuration and testing cases are described, and then performance reports on four testing schemes are presented.

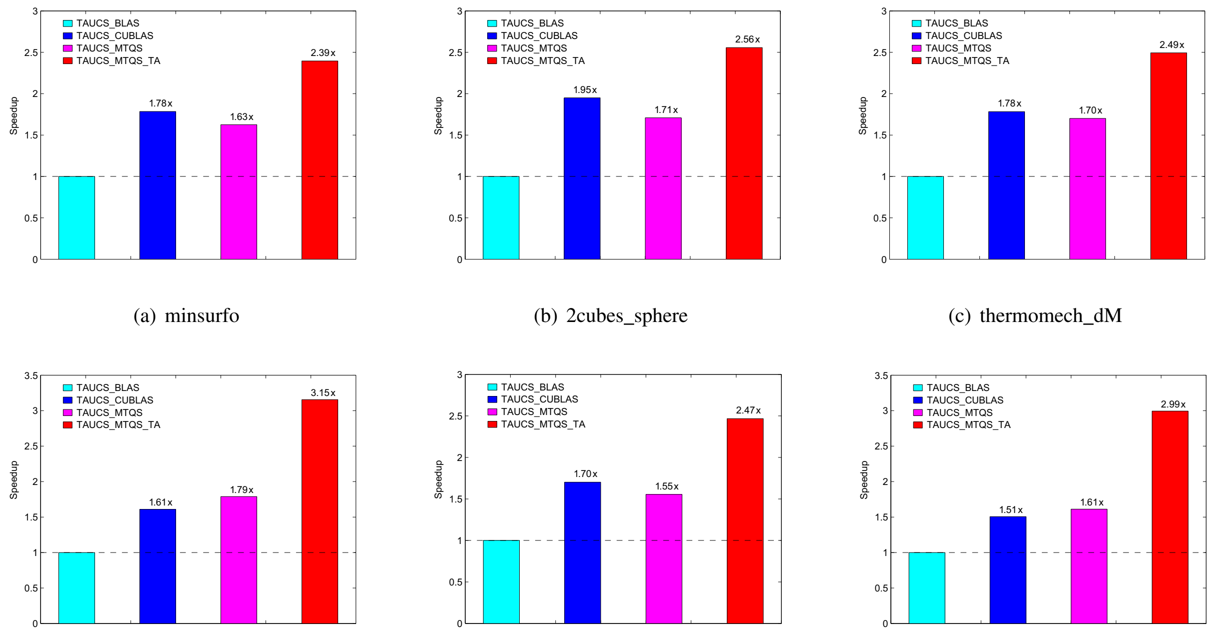
A. Experimental Environment

All experiments are run on GPU of NVIDIA Geforce GTX460, whose CUDA capability is 2.1 and GPU clock rate is 1.55GHz. The testing system is equipped with an Intel Core i7 950 CPU with 8 processors. The operating system is 64-bit Red Hat Enterprise Linux 5.9 with Linux kernel 2.6.18 and NVIDIA CUDA runtime version is 4.0.

The experimental testing matrices shown in Table I are selected from Tim Davis's sparse matrix collection [21]. All of them are derived from real applications, such as optimization, electromagnetics, thermal, computational fluid dynamics, structural and 2D/3D problem. The terms in Table I consist of name, dimension, *numbers of non-zero* (NNZ), sparseness (the proportion of non-zero in overall numbers) and application area.

There are four testing schemes to be performed on each testing case:

- *TAUCS_BLAS*: it is the baseline scheme, based on *TAUCS* Version 2.2, which implements the multithreaded Cholesky factorization on CPU. The computation of vectors and matrices in the course of factorization of frontal matrix are solved by BLAS.
- *TAUCS_CUBLAS*: it is the same as *TAUCS_BLAS*, but GEMM operation with large computation scale is solved by cuBLAS and GEMM operation with small computation scale is solved by BLAS. As the cuBLAS can be executed asynchronously, the overhead of data transmission can be overlapped. When multiple GEMM operations need to be processed, CUDA streams can be used to overlap the overhead in these GEMM operations.
- *TAUCS_MTQS*: it is the proposed MTQS based on *TAUCS* in the paper, and all GEMM operations are performed on GPU. The programmer cannot modify the implementation details of cuBLAS, which may cause the insufficient utilization of GPU resources. Therefore, MTQS is proposed to execute multiple GEMM operations on GPU in parallel.



(a) minsurfo (b) 2cubes_sphere (c) thermomech_dM

(d) parabolic_fem (e) apache2 (f) ecology2

Fig. 5. Overall performance evaluations on four testing schemes

- *TAUCS_MTQS_TA*: it is the same as *TAUCS_MTQS*, but GEMM operations are scheduled dynamically on CPU and GPU based on computation scales.

B. Performance Evaluation

Fig. 5 shows the performance comparison for different testing matrices based on four testing schemes. We plot the speedup as the vertical axis in the figure. The horizontal axis represents the testing scheme. The speedup *Sp* is defined as (7):

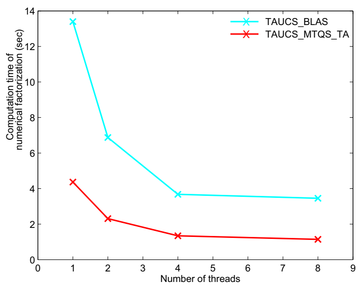
$$S_p = \frac{T_{slow}}{T_{fast}} \tag{7}$$

where T_{slow} represents the computation time of *TAUCS_BLAS* and T_{fast} represents the computation time of *TAUCS_CUBLAS*, *TAUCS_MTQS* or *TAUCS_MTQS_TA*. As *TAUCS_BLAS* has implemented the multithreaded Cholesky factorization, the computation time in Fig. 5 derives from the computational performance of 4 threads. The computing time of T_{slow} is normalized to the baseline.

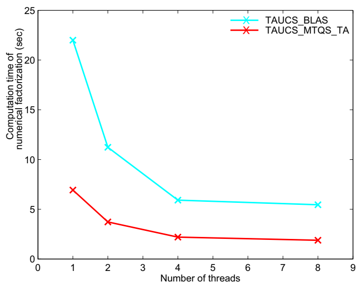
Fig. 5 shows that comparing with *TAUCS_BLAS*, *TAUCS_MTQS_TA* has a significant performance improvement, which achieves up to more than $2\times$ speedup in each testing case, specially, about $3\times$ speedup is achieved in parabolic_fem and ecology2. At the same time, all the experimental results show that the performance of *TAUCS_CUBLAS* outperforms *TAUCS_BLAS*, which has achieved to $1.5 \sim 1.95\times$ speedup. Both of *TAUCS_CUBLAS* and *TAUCS_MTQS_TA* implement the acceleration of GEMM operation on GPU, but as the difference of the implementation mechanism, the performance of *TAUCS_MTQS_TA* is better than *TAUCS_CUBLAS*, achieving up to approximate $2\times$ speedup. All GEMM operations are performed on GPU in *TAUCS_MTQS*, and the overhead of data transfer can not be overlapped for small computation scale GEMM operations, therefore, the performance of *TAUCS_MTQS_TA* is also better than *TAUCS_MTQS*. With the increase of the proportion of the large computation scale GEMM operations, the performance of *TAUCS_MTQS* will be better due to that a majority of GEMM operations are performed on GPU. On the whole, with the increase of matrix dimension, *TAUCS_MTQS_TA* achieves better performance, which can be observed in Fig. 5.

With the development of GPU compute capability, data transfer and kernel execution can be performed in parallel. In the newest version of cuBLAS, it can be executed asynchronously. When multiple GEMM operations need to be performed, the overhead of data transfer can be hidden. Benefit from the powerful computational of GPU, the computing time of GEMM operation on GPU is less than the computation time on CPU, causing the performance of *TAUCS_CUBLAS* outperforms *TAUCS_BLAS*. However, when GEMM operations associated with kernel launching, kernel termination, resource allocation and resource release are performed by cuBLAS, the performance of *TAUCS_CUBLAS* will be degraded, meanwhile, multiple GEMM operations can not be executed on GPU at the same time, consequently, it only achieves $1.5 \sim 1.95\times$ speedup.

In order to make full use of GPU resource and reduce the time of resource allocation and release, *TAUCS_MTQS_TA* makes several TBs as a TEU, therefore, multiple GEMM operations can be performed concurrently on device kernel. Compared with *TAUCS_CUBLAS*, the computation time of multiple GEMM operations on GPU can be overlapped in *TAUCS_MTQS_TA*. As the allocation of storage space consumes much time, it will be allocated space before frontal matrix is factorized. The overhead of *TAUCS_MTQS_TA* consists of data transfer and kernel execution, when multiple GEMM



(a) parabolic_fem



(b) ecology2

Fig. 6. Performance comparison on multithreaded environment

operations are performed concurrently by multiple TBs, data copy overhead can be overlapped as much as possible. For example, if there are 100 GEMM operations need to be processed on GPU, the ratio of data transfer overhead and kernel execution cost is 1 to 2, t_1 represents the overhead of kernel launching, kernel termination, resource allocation and resource release in a GEMM operation and t_2 represents the overhead of data transfer in the same case, so the overall computation time of *TAUCS_MTQS_TA* and *TAUCS_CUBLAS* are (t_1+103t_2) and (100t_1+202t_2), respectively. Obviously, the performance of *TAUCS_MTQS_TA* exceeds *TAUCS_CUBLAS*.

Fig. 6 shows the computation time of numerical factorization based on different number of host threads for different test problems. There are two curves in Fig. 6, corresponding to *TAUCS_BLAS* and *TAUCS_MTQS_TA* respectively. The variation tendency of curve indicates the variation of computation time on different number of host threads. The vertical axis shows the computing time and the horizontal axis represents the number of host threads.

The two curves have the similar variation tendency in each testing case. With the exponential increase of the number of threads, the computing time degrades exponentially. However, the variation tendencies on 4 threads and 8 threads do not meet this rule: the computation times on them have not too much differences. This is caused by lots of extra costs, e.g. for 8 threads configuration, 8 threads are created, but there are no enough GEMM operations needing to be performed concurrently, therefore, the overhead of unnecessary threads communication and synchronization brings extra costs. The computation scales can affect the computation time more easily, so with the increase of matrix dimension, the computing time will also increase.

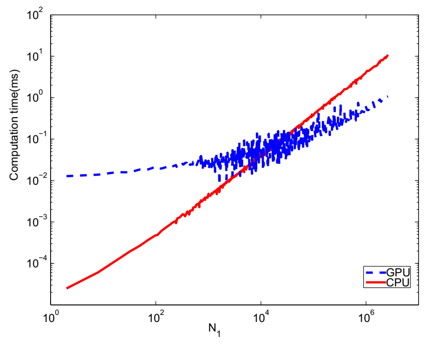


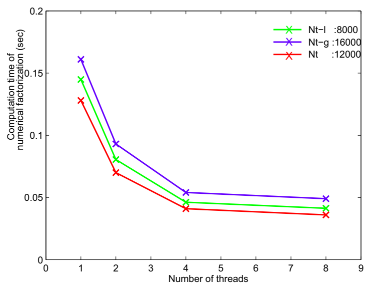
Fig. 7. Processing time of GEMM operation on CPU and GPU

C. GEMM Performance on CPU and GPU

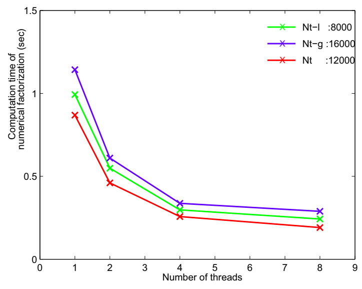
In order to respectively show the computation times of different scales of GEMM operations on CPU and GPU, we process a lot of different scale GEMM operations and then record the computation time of each sample. Finally we give the statistic results in Fig. 7. The horizontal axis represents the variation range of N_1 and the vertical axis represents the computation time on CPU and GPU, respectively.

With the increase of N_1, T_{cpu} increases approximate linearly. T_{gpu} is not only related to N_1, but also related to m_1, k_1, and n_1. For two different scale GEMM operations G_1 and G_2, even though the calculated amount N_{1-1} from G_1 is less than N_{1-2} from G_2, the computation time T_{gpu-1} from G_1 may be greater than T_{gpu-2} from G_2. But generally speaking, with the increase of N_1, T_{gpu} increases. We do some experiments to achieve the average performance for GEMM operation on CPU and GPU, respectively. The experimental result indicates that the values of α_{cpu} and α_{gpu} are 9.53 GFlops and 123.63 GFlops. We also check the average bandwidth β achieved for copying matrix between CPU and GPU, which is observed to be approximately 1.7 GB/s. We get the theoretical value of N_t by putting α_{cpu}, α_{gpu} and β into Equation (6), and it is verified that the value of N_t consist with intersection point of curves in the Fig. 7.

In order to verify whether the theoretical value of N_t is reasonable or not, we take other two values N_{t-l} and N_{t-g} as threshold and perform experiments with *TAUCS_MTQS_TA* on two testing cases, where N_{t-l} is less than N_t and N_{t-g} is greater than N_t. Fig. 8 shows the performance comparison based on N_t, N_{t-l}, N_{t-g}. Obviously, when N_t is set as threshold, the performance of *TAUCS_MTQS_TA* is better than the other two values as thresholds. Because N_{t-l} is less than theoretical value, there are much more small scale GEMM operations processed by GPU. As the overhead of data transfer can not be overlapped for small scale GEMM operations, the performance of *TAUCS_MTQS_TA* based on N_{t-l} will be degraded. Similarly, due to N_{t-g} is greater than theoretical value, there are much more larger scale GEMM operations processed by CPU. As the computational power of CPU is not enough, CPU needs to spend more time performing large scale GEMM operations, the performance of *TAUCS_MTQS_TA* based on N_t is better than N_{t-g}. Therefore, the theoretical value of N_t is resonable.



(a) minsurfo



(b) thermomech_dM

Fig. 8. Computation times on different threshold N_t

VII. Conclusions and Future Work

It is the most compute-intensive component to solve large sparse SPD linear equations in many scientific computing applications. There is no highly efficient implementation of sparse Cholesky factorization on GPU. In order to more efficiently solve large sparse SPD linear equations, a hybrid CPU-GPU algorithm of sparse Cholesky factorization is presented based on multifrontal method. Two strategies are proposed to accelerate it: multiple task queues scheme is adopted to perform multiple GEMM operations in parallel on GPU; a calculating quantity threshold is set to distribute GEMM operations to CPU or GPU based on computation scales. Experimental results show that our approach (*TAUCS_MTQS_TA*) outperforms the implementation of BLAS (*TAUCS_BLAS*) and cuBLAS (*TAUCS_CUBLAS*), achieving up to 3.15× and 1.98× speedup, respectively. In the future, multiple task queues scheme based on multi-GPU will be adopted to decrease the overall computation time of numerical factorization of multifrontal method.

Acknowledgment

This work is supported by the National Natural Science Foundation of China (grant No. 61133008), the National Basic Research Program (973 Program) (grant No. 2013CB2282036) and the independent innovation project of Huazhong University of Science and Technology.

References

[1] M. T. Heath, E. Ng, and B. W. Peyton, "Parallel algorithms for sparse linear systems," *SIAM review*, vol. 33, no. 3, pp. 420–460, 1991.

[2] J. W. Demmel, J. R. Gilbert, and X. S. Li, "An asynchronous parallel supernodal algorithm for sparse gaussian elimination," *SIAM Journal on Matrix Analysis and Applications*, vol. 20, no. 4, pp. 915–952, 1999.

[3] H. Avron, G. Shklarski, and S. Toledo, "Parallel unsymmetric-pattern multifrontal sparse LU with column preordering," *ACM Transactions on Mathematical Software*, vol. 34, no. 2, p. 8, 2008.

[4] J. W. Liu, "The multifrontal method for sparse matrix solution: Theory and practice," *SIAM review*, vol. 34, no. 1, pp. 82–109, 1992.

[5] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Transactions on Mathematical Software*, vol. 16, no. 1, pp. 1–17, 1990.

[6] C. Nvidia, "Cublas library," http://docs.nvidia.com/cuda/cublas/.

[7] X. Li, F. Li, and J. M. Clark, "Exploration of multifrontal method with GPU in power flow computation," in *Proceedings of Power and Energy Society General Meeting*. IEEE, 2013, pp. 1–5.

[8] P. Sao, R. Vuduc, and X. S. Li, "A distributed CPU-GPU sparse direct solver," in *Proceedings of European Conference on Parallel and Distributed Computing*. Springer, 2014, pp. 487–498.

[9] O. Schenk, M. Christen, and H. Burkhart, "Algorithmic performance studies on graphics processing units," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1360–1369, 2008.

[10] S. Toledo, D. Chen, and V. Rotkin, "Taucs: A library of sparse linear solvers," http://www.tau.ac.il/ stoledo/taucs/.

[11] T. George, V. Saxena, A. Gupta, A. Singh, and A. R. Choudhury, "Multifrontal factorization of sparse SPD matrices on GPUs," in *Proceedings of International Parallel & Distributed Processing Symposium*. IEEE, 2011, pp. 372–383.

[12] A. Gupta, "WSMP: Watson sparse matrix package (part-I: direct solution of symmetric sparse systems)," IBM TJ Watson Research Center, Tech. Rep., 2000.

[13] X. S. Li and J. W. Demmel, "SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems," *ACM Transactions on Mathematical Software*, vol. 29, no. 2, pp. 110–140, 2003.

[14] R. F. Lucas, G. Wagenbreth, D. M. Davis, and R. Grimes, "Multifrontal computations on GPUs and their multi-core hosts," in *Proceedings of High Performance Computing for Computational Science–VECPAR*. Springer, 2011, pp. 71–82.

[15] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Guney, and A. Shringarpure, "On the limits of GPU acceleration," in *Proceedings of USENIX Conference on Hot Topics in Parallelism*. USENIX Association, 2010, pp. 13–13.

[16] C. D. Yu, W. Wang, and D. Pierce, "A CPU-GPU hybrid approach for the unsymmetric multifrontal method," *Parallel Computing*, vol. 37, no. 12, pp. 759–770, 2011.

[17] S. N. Yeralan, T. Davis, and S. Ranka, "Sparse QR factorization on GPU architectures," University of Florida, Tech. Rep., 2013.

[18] L. Ren, X. Chen, Y. Wang, C. Zhang, and H. Yang, "Sparse LU factorization for parallel circuit simulation on GPU," in *Proceedings of Design Automation Conference*. ACM, 2012, pp. 1125–1130.

[19] MIT CSAIL Supertech Research Group, "Cilk: A linguistic and runtime technology for algorithmic multithreaded programming," http://supertech.csail.mit.edu/cilk/.

[20] L. Chen, O. Villa, S. Krishnamoorthy, and G. R. Gao, "Dynamic load balancing on single-and multi-GPU systems," in *Proceedings of International Parallel & Distributed Processing Symposium*. IEEE, 2010, pp. 1–12.

[21] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," http://www.cise.ufl.edu/research/sparse/matrices/.