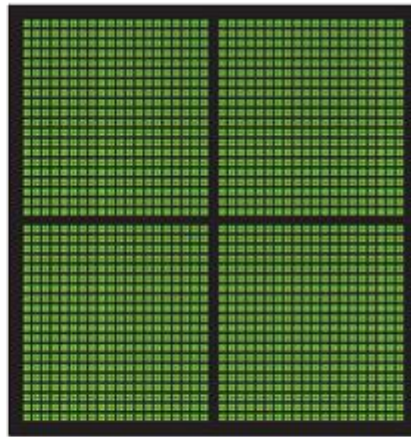


# GSS QUICK START USER GUIDE

best CPU-GPU hybrid solver



CPU  
MULTIPLE CORES



GPU  
THOUSANDS OF CORES

version 2.4

YingShi Chen

4/18/2014

GSS QUICK START USER GUIDE .....	1
best CPU-GPU hybrid solver .....	1
§1 Introduction .....	3
1.1 Some Key Features of GSS .....	3
1.2 GPU Computing in GSS .....	3
1.3 Package Directory .....	4
§2 CONVENTION.....	4
2.1 Naming Conventions.....	4
2.2 Storage Format .....	5
2.3 Library Files .....	5
§3 Matrix Type .....	6
§4 Parameters .....	6
§5 C Routines and Demo .....	7
5.1 32-Bit C Routines.....	7
GSS_init_i?       init and check.....	7
GSS_symbol_i?     do symbolic factorization .....	7
GSS_numeric_i?    do numerical factorization .....	7
GSS_solve_i?      do forward/backward substitution .....	8
GSS_clear_i?      release memory .....	8
5.2 64-Bit C Routines.....	8
GSS_init_l?       init and check.....	8
GSS_numeric_l?    do numerical factorization .....	9
GSS_solve_l?      do forward/backward substitution .....	9
GSS_clear_l?      release memory .....	9
5.3 C Demo .....	10
§6 FORTRAN Demo.....	12
§7 Experimental Results .....	15
§8 Users.....	17
History.....	18

## § 1 Introduction

GSS (GRUS SPARSE SOLVER) is a novel sparse solver that can solve million unknowns in minutes even on PC. The latest version is at <http://www.grusoft.com/product/>. The high performance and generality of GSS has been verified by many commercial users and large testing sets.

### 1.1 Some Key Features of GSS

- High Performance
- **CPU-GPU hybrid computing**  
GSS is the first sparse solver that supports NVidia CUDA technology.  
Novel algorithm to run CPU and GPU simultaneously.  
For large matrices that need long time computing, GSS is about [2-3 times faster than PARDISO](#) and other CPU based solvers.
- Robust  
Handle matrices with high condition numbers or strange patterns.  
Some ill-conditioned matrices can only be solved by GSS.
- Good scalability.
- Support MATLAB.
- [32 parameters](#) with default value.

### 1.2 GPU Computing in GSS

GSS GPU computing is based on NVidia's CUDA technology. GSS 2.4 needs CUDA toolkit 5.5. For detail of download and install CUDA toolkit, see <https://developer.nvidia.com/cuda-zone>. The graphics cards should have **compute capability 3.0** or higher. NVidia provides a full list of CUDA GPUs' capability at <https://developer.nvidia.com/cuda-gpus>.

GSS GPU module is linked with cublas library. GSS 2.4 needs cublas\*\_55.dll. For user's convenience, cublas\*\_55.dll is included in the current installing package. But cublas\*.dll is not a part of GSS package and it will be removed in the future package. So it would be best to install CUDA in your computers.

GSS Hybrid computing support two kinds of [matrix type](#): Structurally Symmetric Matrices(10) and Positive Definite Symmetric matrices(11).

For large matrices and long-time computing, hybrid computing is much faster than CPU version. But Synchronization, scheduling, data transfer and between CPU and GPU need extra time. And many factors make GPU computing much harder than parallelization in threads. So for small matrices, the extra cost may need more time! GSS provide [one parameter](#) to switch between CPU and Hybrid computing. Users can change this parameter to find whether GPU computing is good for your matrices.

We have found much room for improvement. We will continue improve the efficiency in later versions.

GPU computing may fail due to many reasons: graphics card doesn't support CUDA,

insufficient GPU memory, low compute capability...

### 1.3 Package Directory

After installation, the directory contains the following:



Figure 1 directory files

- 1 **bin**                    32 bit library file  
     All files in this directory should be copied to 32-bit exe file's directory.
  - cublas32\_55.dll - It's a CUDA dll and not a part of GSS package!  
     Only for user's convenience, May be removed in the future package.
- 2 **bin\_64**                64 bit library file  
     All files in this directory should be copied to 64-bit exe file's directory.
  - cublas64\_55.dll - It's a CUDA dll and not a part of GSS package!  
     Only for user's convenience, May be removed in the future package.
- 3 **doc**                    Help documentation.
- 4 **include**                C Head file
- 5 **Samples**                C and Fortran demos.

For more details, please contact  
 Phone:    (+86) 013501997193  
 MAIL:    [gsp@grusoft.com](mailto:gsp@grusoft.com)  
 WeChat:    grusoft  
 QQ:        304718494

## § 2 CONVENTION

### 2.1 Naming Conventions

GSS function names have the following structure:

<FUNCTION>\_<SYSTEM><DATA\_TYPE>

The < FUNCTION > is as follows:

Function name	meaning
GSS_init	init and check
GSS_symbol	symbolic factorization

GSS_numeric	numerical factorization
GSS_solve	forward/backward substitution
GSS_clear	release memory

The < SYSTEM > is a character as follows:

	meaning
i	32 Bit
l	64 Bit

The < DATA\_TYPE > is a character that indicates the data type:

	meaning	C	FORTRAN
s	real, single precision	float	REAL
d	real, double precision	double	DOUBLE PRECISION
c	complex, single precision	User defined*	COMPLEX
z	complex, double precision	User defined *	DOUBLE COMPLEX

\* There is no complex data type in ANSI C

## 2.2 Storage Format

The default storage format is [compressed column storage](#) . GSS use 5 parameters as follows:

*int nRow, nCol*

The numbers of row and column.

*int \*ptr, int \*ind, double \*val*

(ptr, ind, val) , where *ind* stores the row indices of each nonzero, *ptr* stores the first element position of each column and *val* vector stores the values of the nonzero elements of the matrix.

For **symmetric** matrices, only **lower triangle** (include diagonals) are stored.

## 2.3 Library Files

File	Contents
GSS_Si.dll; GSS_Si.lib	32 Bit ,REAL
GSS_Di.dll; GSS_Di.lib	32 Bit ,DOUBLE PRECISION
GSS_Ci.dll; GSS_Ci.lib	32 Bit ,COMPLEX
GSS_Zi.dll; GSS_Zi.lib	32 Bit ,DOUBLE COMPLEX
GSS_Sl.dll; GSS_Sl.lib	64 Bit ,REAL
GSS_Dl.dll; GSS_Dl.lib	64 Bit ,DOUBLE PRECISION
GSS_Cl.dll; GSS_Cl.lib	64 Bit ,COMPLEX
GSS_Zl.dll; GSS_Zl.lib	64 Bit ,DOUBLE COMPLEX

### § 3 Matrix Type

There are 5 types supported by GSS.

- 0: General Matrices
- 10: Structurally Symmetric Matrices  
Support CPU/GPU hybrid computing
- 11: Positive Definite Symmetric (Hermitian) matrices  
Support CPU/GPU hybrid computing
- 12: Indefinite Symmetric matrices
- 13: Complex Symmetric matrices

### § 4 Parameters

There are 32 parameters to control GSS. Users should pass these parameters by a double precision array of length 32. All the default value is zero. Some parameters are as follows:

**setting[5] (setting(6) in Fortran)**

Debugging information output.

- 0 No debugging information output.
- 65791 Print debugging information.

**setting[20] (setting(21) in Fortran)**

Switch of GPU-CPU hybrid computing.

- 0 Only CPU computing
- 5 Hybrid GPU-CPU computing. Only for two [matrix type](#): Structurally Symmetric Matrices(10) and Positive Definite Symmetric matrices(11). [More information](#).

**setting[23] (setting(24) in Fortran)**

Set maximum number of iterative-refine step.

- 0 GSS will automatically do iterative-refine on the matrix property.
- 1-10 maximum number of iterative-refine steps.

**setting[24] (setting(25) in Fortran)**

Number of threads.

- 0 GSS will find the most appropriate number automatically (Strongly recommended!)

## § 5 C Routines and Demo

### 5.1 32-Bit C Routines

For the detail of GSS interface, please see “grus\_sparse\_solver.h”.

#### GSS\_init\_i? init and check

```
int GSS_init_id( int nRow, int nCol, int* ptr, int* ind, double *val, int type, double *setting )
int GSS_init_is( int nRow, int nCol, int* ptr, int* ind, float *val, int type, double *setting )
int GSS_init_iz( int nRow, int nCol, int* ptr, int* ind, _double_COMPLEX *val, int type, double *setting )
int GSS_init_ic( int nRow, int nCol, int* ptr, int* ind, _float_COMPLEX *val, int type, double *setting )
```

#### Input

nRow, nCol, ptr, ind, val     [compressed column storage](#) of matrices.  
 type                             [matrix type](#).  
 setting[32]                     [control parameters](#).

#### Return

returns 0x0 if init successfully, otherwise return error code.

#### GSS\_symbol\_i? do symbolic factorization

```
void* GSS_symbol_id( int nRow, int nCol, int* ptr, int* ind, double *val )
void* GSS_symbol_is( int nRow, int nCol, int* ptr, int* ind, float *val )
void* GSS_symbol_iz( int nRow, int nCol, int* ptr, int* ind, _double_COMPLEX *val )
void* GSS_symbol_ic( int nRow, int nCol, int* ptr, int* ind, _float_COMPLEX *val )
```

#### Input

nRow, nCol, ptr, ind, val     [compressed column storage](#) of matrices.

#### Return

returns 0x0 if failed otherwise return the pointer of solver.

#### GSS\_numeric\_i? do numerical factorization

```
int GSS_numeric_id( int nRow, int nCol, int* ptr, int* ind, double *val, void *hSolver )
int GSS_numeric_is( int nRow, int nCol, int* ptr, int* ind, float *val, void *hSolver )
int GSS_numeric_iz( int nRow, int nCol, int* ptr, int* ind, _double_COMPLEX *val, void *hSolver )
int GSS_numeric_ic( int nRow, int nCol, int* ptr, int* ind, _float_COMPLEX *val, void *hSolver )
```

#### Input

nRow, nCol, ptr, ind, val     [compressed column storage](#) of matrices.  
 hSolver                         the pointer of solver.

#### Return

returns 0x0 in success, otherwise return error code.

### GSS\_solve\_i?

### do forward/backward substitution

```
int GSS_solve_id( void *hSolver, int nRow, int nCol, int *ptr, int *ind, double *val, double *rhs )
int GSS_solve_is( void *hSolver,int nRow,int nCol,int *ptr,int *ind,float *val,double *rhs )
int GSS_solve_iz( void *hSolver,int nRow,int nCol,int *ptr,int *ind,_double_COMPLEX
*val,_double_COMPLEX *rhs )
int GSS_solve_ic( void *hSolver,int nRow,int nCol,int *ptr,int *ind,_float_COMPLEX
*val,_float_COMPLEX *rhs )
```

#### Input

nRow, nCol, ptr, ind, val     [compressed column storage](#) of matrices.  
 rhs     the right hand side  
 hSolver     the pointer of solver.

#### Output

rhs     contains the solution.

#### Return

returns 0x0 in success, otherwise return error code.

### GSS\_clear\_i?

### release memory

```
int GSS_clear_id ( void* hSolver )
int GSS_clear_is( void* hSolver )
int GSS_clear_iz( void* hSolver )
int GSS_clear_ic( void* hSolver )
```

#### Input

hSolver     the pointer of solver.

#### Return

returns 0x0 in success, otherwise return error code.

## 5.2 64-Bit C Routines

For the detail of GSS interface, please see “grus\_sparse\_solver.h”.

### GSS\_init\_I?

### init and check

```
int GSS_init_Id( int nRow, int nCol, int* ptr, int* ind, double *val, int type, double *setting )
int GSS_init_Is( int nRow,int nCol,int* ptr,int* ind,float *val,int type,double *setting )
int GSS_init_Iz( int nRow,int nCol,int* ptr,int* ind,_double_COMPLEX *val,int type,double *setting )
int GSS_init_Ic( int nRow,int nCol,int* ptr,int* ind,_float_COMPLEX *val,int type,double *setting )
```

#### Input

nRow, nCol, ptr, ind, val     [compressed column storage](#) of matrices.  
 type     [matrix type](#).  
 setting[32]     [control parameters](#).

#### Return

returns 0x0 if init successfully, otherwise return error code.



Function **GSS\_symbol\_l?** do symbolic factorization

```
void* GSS_symbol_ld( int nRow, int nCol, int* ptr, int* ind, double *val )
void* GSS_symbol_ls( int nRow,int nCol,int* ptr,int* ind,float *val )
void* GSS_symbol_lz( int nRow,int nCol,int* ptr,int* ind,_double_COMPLEX *val )
void* GSS_symbol_lc( int nRow,int nCol,int* ptr,int* ind,_float_COMPLEX *val )
```

Input

nRow, nCol, ptr, ind, val [compressed column storage](#) of matrices.

Return

returns 0x0 if failed otherwise return the pointer of solver.

**GSS\_numeric\_l?** do numerical factorization

```
int GSS_numeric_ld( int nRow, int nCol, int* ptr, int* ind, double *val, void *hSolver )
int GSS_numeric_ls( int nRow,int nCol,int* ptr,int* ind,float *val,void *hSolver )
int GSS_numeric_lz( int nRow,int nCol,int* ptr,int* ind,_double_COMPLEX *val,void *hSolver )
int GSS_numeric_lc( int nRow,int nCol,int* ptr,int* ind,_float_COMPLEX *val,void *hSolver )
```

Input

nRow, nCol, ptr, ind, val [compressed column storage](#) of matrices.

hSolver the pointer of solver.

Return

returns 0x0 in success, otherwise return error code.

**GSS\_solve\_l?** do forward/backward substitution

```
int GSS_solve_ld( void *hSolver, int nRow, int nCol, int *ptr, int *ind, double *val, double *rhs )
int GSS_solve_ls( void *hSolver,int nRow,int nCol,int *ptr,int *ind,float *val,double *rhs )
int GSS_solve_lz( void *hSolver,int nRow,int nCol,int *ptr,int *ind,_double_COMPLEX
*val,_double_COMPLEX *rhs )
int GSS_solve_lc( void *hSolver,int nRow,int nCol,int *ptr,int *ind,_float_COMPLEX
*val,_float_COMPLEX *rhs )
```

Input

nRow, nCol, ptr, ind, val [compressed column storage](#) of matrices.

rhs the right hand side

hSolver the pointer of solver.

Output

rhs contains the solution.

Return

returns 0x0 in success, otherwise return error code.

**GSS\_clear\_l?** release memory

```
int GSS_clear_ld ( void* hSolver )
int GSS_clear_ls( void* hSolver )
int GSS_clear_lz( void* hSolver )
```

```
int GSS_clear_lc( void* hSolver )
```

**Input**

hSolver                                    the pointer of solver.

**Return**

returns 0x0 in success, otherwise return error code.

### 5.3 C Demo

The following code shows how to call GSS to solve double precision matrices.

There are more samples in the *Samples* directory.

**Note:**

1 In C, the first index of a array is 0.

2 (*nRow*, *nCol*, *ptr*, *ind*, *val*) are the matrix in [CCS](#) format. The last parameter *type* is the [matrix type](#).

```
void GSS_demo_( int nRow, int nCol, int *ptr, int *ind, double *val, int type ) {
    int l=0, i, j, r, nRet;
    double t_symbolic, t_solve, t_numeric;
    double *x, *rhs, err=0.0, rhs_norm=0.0, setting[32];
    void *hSolver=NULL;
    clock_t start;

    t_symbolic=0.0;          t_numeric=0.0;          t_solve=0.0;

    /***** symbolic factorization *****/
    start = clock();
    for( i = 0; i < 32; i++ )    setting[i]=0.0;
    nRet = GSS_init_id( nRow, nCol, ptr, ind, val, type, setting );
    if( nRet != GRUS_OK ) {
        printf( "\tERROR at init GSS solver. ERROR CODE:%d\r\n", nRet );
        return;
    }
    hSolver = GSS_symbol_id( nRow, nCol, ptr, ind, val );
    t_symbolic = (clock()-start)/CLOCKS_PER_SEC;
    printf( "\tSymbol Analysis.....\tTime=%g\r\n", t_symbolic );
    if( hSolver == NULL ) {
        printf( "\tERROR at SYMBOLIC ANALYSIS. \r\n" );
        goto EXIT;
    }
    /***** numerical factorization *****/
    start = clock( );
    nRet = GSS_numeric_id( nRow, nCol, ptr, ind, val, hSolver );
    if( nRet != GRUS_OK ) {
```



## § 6 FORTRAN Demo

The following code shows how to call GSS to solve double precision matrices.

For the detail of GSS\_6\_INTERFACE module( GSS Fortran Interface), please see gss\_spd\_demo.f90 and general\_demo.f90 in the *Samples* directory.

**Note:**

1 In FORTRAN, the first index of a array is 1.

2 (*dim, ptr, ind, val*) are the matrix in [CCS](#) format. The last parameter *m\_type* is the [matrix type](#).

```

subroutine GSS_demo_( dim, ptr, ind, val, rhs, m_type )
    use GSS_6_INTERFACE                implicit none
    integer dim, nnz, ptr(*), ind(*), loop, ret
    double precision val(*), rhs(*), start, setting(32)

    double precision t_symbol, t_numeric, t_solve, x(:), err, rhs_norm
    allocatable::: x
    integer hGSS
    integer i, j, r, strategy, nIterRefine, info, m_type
!   clock_t start;

    loop=3
    t_numeric=0.0;          t_solve=0.0
    nnz = ptr(dim+1)-1
    write( *, * ), "dim=", dim, "nnz=", nnz, "m_type=", m_type

    strategy=0;          info=0
    hGSS = 0
! C=>FORTRAN
    do i = 1, dim+1
        ptr(i)=ptr(i)-1
    end do
    do i = 1, nnz
        ind(i)=ind(i)-1
    enddo
    setting=0.0
    ret = GSS_init_id( dim, dim, ptr, ind, val, m_type, setting )
    if( ret/=0 ) then
        write(*,*) "          ERROR at init GSS solver. ERROR CODE", ret
        goto 100;
    endif
    start=SECNDS(0.0)
    hGSS = GSS_symbol_id( dim, dim, ptr, ind, val )
    t_symbol = SECNDS(start)
    
```

```

write( *,* ), "symbol time=", t_symbol
if( hGSS==0 ) then
    write(*,*) "      ERROR at symbol."
    goto 100;
endif
start=SECNDS(0.0)
do i = 1, loop
    ret = GSS_numeric_id( dim, dim, ptr, ind, val, hGSS )
    if( ret /= 0 ) then
        write(*,*) "      ERROR at numeric. ERROR CODE", ret
        hGSS=0;          !must set hGSS to zero
        goto 100
    endif
end do
t_numeric = SECNDS(start)/loop
write( *,* ), "numeric time=", t_numeric

allocate( x(dim) )
start = SECNDS(0.0)
do i = 1, loop
    x(1:dim)=rhs(1:dim)
    call GSS_solve_id( hGSS, dim, dim, ptr, ind, val, x )
end do
t_solve = SECNDS(start)/loop
write( *,* ), "solve time=", t_solve
call GSS_clear_id( hGSS )
100 continue

!C=>FORTRAN
do i = 1, dim+1
    ptr(i)=ptr(i)+1
enddo
do i = 1, nnz
    ind(i)=ind(i)+1
enddo
!/******||Ax-b|******/
if( hGSS/=0 ) then
    rhs_norm=0.0;      err=0.0
    do i = 1, dim
        rhs_norm = rhs_norm+(rhs(i)*rhs(i))
    enddo
    rhs_norm = sqrt( rhs_norm );
    do i = 1, dim
        do j = ptr(i), ptr(i+1)-1

```

---

```
        r = ind(j)
        rhs(r) = rhs(r)-val(j)*x(i)
        if( (m_type==11 .or. m_type==12) .and. r/=i ) then
            rhs(i) = rhs(i)-val(j)*x(r)
        endif
    enddo
end do
do i = 1, dim
    err = err+(rhs(i)*rhs(i))
end do
err = sqrt(err)
write( *,* ), "Residual |Ax-b|=", err, "|b|=", rhs_norm
deallocate( x )
endif
end subroutine
```

## § 7 Experimental Results

The test matrices are all from the [UF sparse matrix collection](#), which need long time in numerical factorization.

Table 1 lists the time of numerical factorization between GSS and PARDISO. PARDISO's version is from Intel Composer XE 2013 SP1. GSS 2.4 use CPU-GPU hybrid computing. The testing CPU is INTEL Core i7-4770(3.4GHz) with 24G memory. The graphics card is ASUS GTX780 (with compute capability 3.5). NVIDIA CUDA Toolkit is 5.5. The operating system is Windows 7 64. Both solvers use default parameters.

For large matrices need long time computing, GSS 2.4 is Nearly 3 times faster than PARDISO. For matrices need short time computing, PARDISO is faster than GSS. One reason is that complex synchronization between CPU/GPU do need some extra time.

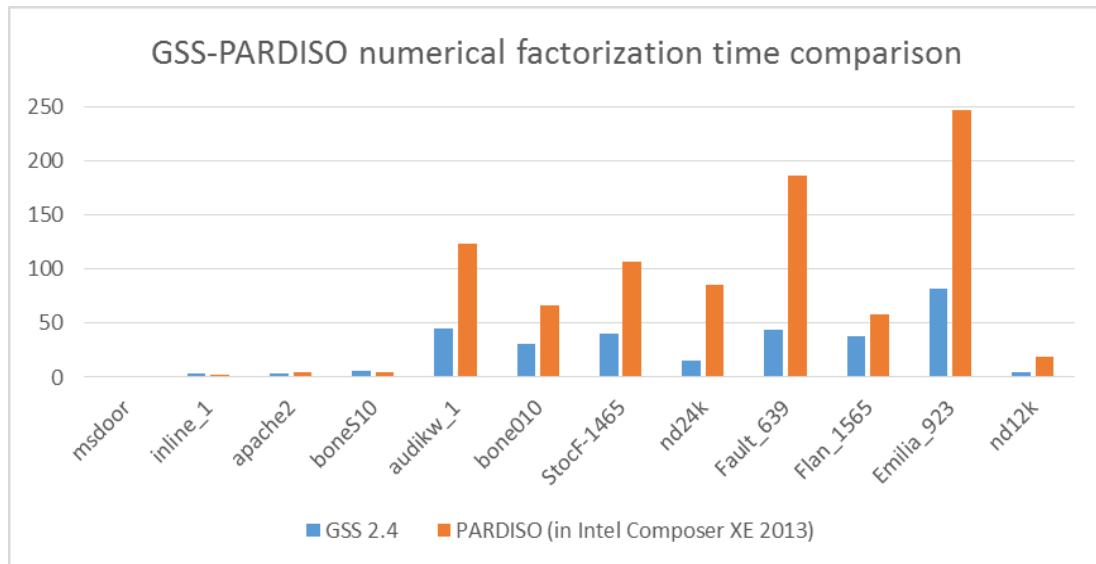


Figure 2 numerical factorization comparison

Table 1 numerical factorization time between GSS and PARDISO

Matrix	Description	Pattern	dimension	Non-zero	GSS	PARDISO
msdoor	Parasol matrices: medium size door		415863	20240935	1.061	0.424
inline_1	structural problem: stiffness matrix		503712	36816342	3.385	2.605
apache2	structural problem: SPD matrix (finite difference 3D) from APACHE small		715176	4817870	3.518	4.386
boneS10	model reduction problem 3D trabecular bone		914898	55468422	5.585	4.636
audikw_1	structural problem: symmetric rb matrix		943695	77651847	45.534	122.832
bone010	model reduction problem: 3D trabecular bone		986703	71666325	30.591	66.892
StocF-1465	computational fluid dynamics problem: flow in porous medium with stochastic permeabilities		1465137	21005389	40.3	106.417
nd24k	2D/3D problem: ND problem set		72000	28715634	15.880	85.519
Fault_639	structural problem: contact mechanics for model of a faulted gas reservoir		638802	28614564	43.633	186.405
Flan_1565	Finite element simulations: gas reservoir and structural problems. 3D model of a steel flange, hexahedral finite elements		1564794	117406044	38.266	58.004
Emilia_923	Finite element simulations: gas reservoir and structural problems. geomechanical model for CO2 sequestration		923136	41005206	82.181	247.17
nd12k	ND problem set. 3D mesh problems.		36000	14220946	5.210	18.954
sum					315.114	904.244



## § 8 Users

GSS has been verified by many commercial users. Some uses are as follows:

Table 2 some commercial users

user	detail	Why they choose GSS
crosslight	Industry leader in TCAD simulation	Hybrid GPU/CPU version, more than 2 times faster than PARDISO, MUMPS and other sparse solvers.
soilvision	The most technically advanced suite of 1D/2D/3D geotechnical software	Much faster than their own sparse solver.
FEM consulting	The leading research teams in the area of the Finite Element Method since 1967	GSS is faster than PARDISO and provide many custom module.
GSCAD	Leading building software in China	GSS provide a user-specific module to deal with ill-conditioned matrix.
ICAROS	A global turnkey geospatial mapping service provider and state of the art photogrammetric technologies developer.	GSS is faster than PARDISO. Also provide some technical help.
EPRI	China Electric Power Research Institute	3-4 times faster than KLU

高校，研究所  
 中国电力科学研究院  
 香港大学  
 中国石油大学  
 电子科技大学  
 三峡大学



2011年成为Intel软件卓越精英合作伙伴

## History

4/16/2014 GSS 2.4 released.

- 1 New GPU-CPU hybrid computing module.
- 2 Improved numerical factorization. About 50% faster than than GSS 2.3.

4/18/2012 GSS 2.3 released.

1. Improved numerical factorization. Faster than GSS 2.2.
2. Need less memory than GSS 2.2.
3. Fix some bugs.

9/19/2009 GSS 2.2 released.

1. Improved numerical factorization for SPD matrices.
2. Improved CPU/GPU hybrid computing. The best speed-up for 500,000 unknowns is 7.

7/31/2008 GSS 2.1 released.

1. Support Nvidia CUDA.
2. Improved out-core module.
3. Improved memory module of LDLT.

12/25/2007 GSS 2.0 released.

1. Add new balance module.
2. Add LU-partial-updating module.
3. Improved out-of-core, in-core and hybrid-core module.
4. Add hybrid multifrontal/Frontal module.
5. Improved iterative refine module and get better estimation of condition number.

11/25/2005 GSS 1.2 released.

1. Parallel version released.
2. Support INTEL Hyper-Threading.
3. Improved numerical factorization for symmetrical matrices.
4. Improved static pivoting.
5. Add iterative refine module.

9/12/2005 GSS1.1 released

1. Add QUOTIENT GRAPH model for symbolic factorization.
2. Improved reorder module of diagonals.
3. Improved Numerical factorization for unsymmetrical matrices.
4. Add scaling module.
5. More experimental results.

7/20/2005 GSS1.0 released.