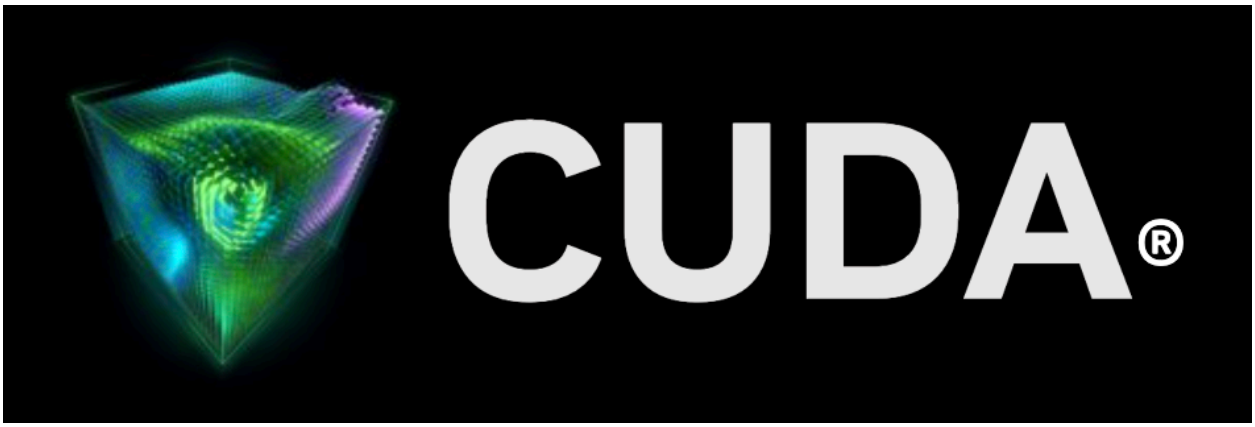


Incomplete-LU and Cholesky Preconditioned Iterative Methods

Release 13.4

NVIDIA Corporation

Oct 12, 2026

Contents

1 Preconditioned Iterative Methods **3**
1.1 Algorithm 1 Conjugate Gradient (CG) 4
1.2 Algorithm 2 Bi-Conjugate Gradient Stabilized (BiCGStab) 7

2 Numerical Experiments **11**

3 Conclusion **19**

4 Acknowledgements **21**

5 References **23**

6 Notices **25**
6.1 Notice 25
6.2 OpenCL 26
6.3 Trademarks 26

Incomplete-LU and Cholesky Preconditioned Iterative Methods Using cuSPARSE and cuBLAS

White paper describing how to use the cuSPARSE and cuBLAS libraries to achieve a 2x speedup over CPU in the incomplete-LU and Cholesky preconditioned iterative methods.

The solution of large sparse linear systems is an important problem in computational mechanics, atmospheric modeling, geophysics, biology, circuit simulation and many other applications in the field of computational science and engineering. In general, these linear systems can be solved using direct or preconditioned iterative methods. Although the direct methods are often more reliable, they usually have large memory requirements and do not scale well on massively parallel computer platforms.

The iterative methods are more amenable to parallelism and therefore can be used to solve larger problems. Currently, the most popular iterative schemes belong to the Krylov subspace family of methods. They include *Bi-Conjugate Gradient Stabilized* (BiCGStab) and *Conjugate Gradient* (CG) iterative methods for nonsymmetric and *symmetric positive definite* (s.p.d.) linear systems, respectively [2], [11]. We describe these methods in more detail in the next section.

In practice, we often use a variety of preconditioning techniques to improve the convergence of the iterative methods. In this white paper we focus on the incomplete-LU and Cholesky preconditioning [11], which is one of the most popular of these preconditioning techniques. It computes an incomplete factorization of the coefficient matrix and requires a solution of lower and upper triangular linear systems in every iteration of the iterative method.

In order to implement the preconditioned BiCGStab and CG we use the sparse matrix-vector multiplication [3], [15] and the sparse triangular solve [8], [16] implemented in the cuSPARSE library. We point out that the underlying implementation of these algorithms takes advantage of the CUDA parallel programming paradigm [5], [9], [13], which allows us to explore the computational resources of the graphical processing unit (GPU). In our numerical experiments the incomplete factorization is performed on the CPU (host) and the resulting lower and upper triangular factors are then transferred to the GPU (device) memory before starting the iterative method. However, the computation of the incomplete factorization could also be accelerated on the GPU.

We point out that the parallelism available in these iterative methods depends highly on the sparsity pattern of the coefficient matrix at hand. In our numerical experiments the incomplete-LU and Cholesky preconditioned iterative methods achieve on average more than 2x speedup using the cuSPARSE and cuBLAS libraries on the GPU over the MKL [17] implementation on the CPU. For example, the speedup for the preconditioned iterative methods with the incomplete-LU and Cholesky factorization with 0 fill-in (ilu0) is shown in Figure 1 for matrices resulting from a variety of applications. It will be described in more detail in the last section.

In the next sections we briefly describe the methods of interest and comment on the role played in them by the parallel sparse matrix-vector multiplication and triangular solve algorithms.

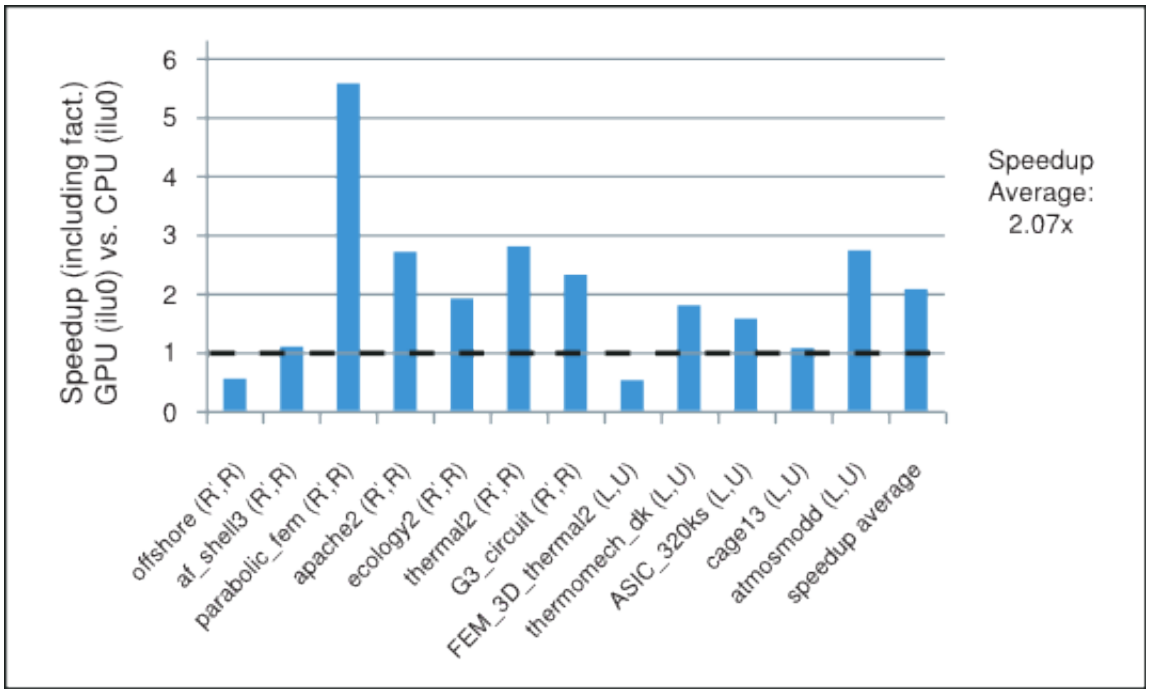


Fig. 1: Speedup of the Incomplete-LU Cholesky (with 0 fill-in) Prec. Iterative Methods

Chapter 1. Preconditioned Iterative Methods

Let us consider the linear system

$A\mathbf{x} = \mathbf{f}$	(1)

where $A \in \mathbb{R}^{n \times n}$ is a nonsingular coefficient matrix and $\mathbf{x}, \mathbf{f} \in \mathbb{R}^n$ are the solution and right-hand-side vectors.

In general, the iterative methods start with an initial guess and perform a series of steps that find more accurate approximations to the solution. There are two types of iterative methods: (i) the stationary iterative methods, such as the splitting-based *Jacobi* and *Gauss-Seidel* (GS), and (ii) the nonstationary iterative methods, such as the *Krylov* subspace family of methods, which includes *CG* and *BiCGStab*. As we mentioned earlier we focus on the latter in this white paper.

The convergence of the iterative methods depends highly on the spectrum of the coefficient matrix and can be significantly improved using preconditioning. The preconditioning modifies the spectrum of the coefficient matrix of the linear system in order to reduce the number of iterative steps required for convergence. It often involves finding a preconditioning matrix M, such that M^{-1} is a good approximation of A^{-1} and the systems with M are relatively easy to solve.

For the s.p.d. matrix A we can let M be its incomplete-Cholesky factorization, so that $A \approx M = \widetilde{R}^T \widetilde{R}$, where $\widetilde{R}$ is an upper triangular matrix. Let us assume that M is nonsingular, then $\widetilde{R}^{-T} A \widetilde{R}^{-1}$ is s.p.d. and instead of solving the linear system (1), we can solve the preconditioned linear system

$\left(\widetilde{R}^{-T} A \widetilde{R}^{-1}\right)\left(\widetilde{R}\mathbf{x}\right) = \widetilde{R}^{-T}\mathbf{f}$	(2)

The pseudocode for the preconditioned CG iterative method is shown in Algorithm 1.

1.1. Algorithm 1 Conjugate Gradient (CG)

1:	Letting initial guess be $\mathbf{x}_0$, compute $\mathbf{r} \leftarrow \mathbf{f} - A\mathbf{x}_0$	
2:	**for** $i \leftarrow 1, 2, \ldots$ until convergence **do**	
3:	Solve $M\mathbf{z} \leftarrow \mathbf{r}$	▷ Sparse lower and upper triangular solves
4:	$\rho_i \leftarrow \mathbf{r}^T\mathbf{z}$	
5:	**if** $i == 1$ **then**	
6:	$\mathbf{p} \leftarrow \mathbf{z}$	
7:	**else**	
8:	$\beta \leftarrow \frac{\rho_i}{\rho_{i-1}}$	
9:	$\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p}$	
10:	**end if**	
11:	Compute $\mathbf{q} \leftarrow A\mathbf{p}$	▷ Sparse matrix-vector multiplication
12:	$\alpha \leftarrow \frac{\rho_i}{\mathbf{p}^T\mathbf{q}}$	
13:	$\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$	
14:	$\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{q}$	
15:	**end for**	

Notice that in every iteration of the incomplete-Cholesky preconditioned CG iterative method we need to perform one sparse matrix-vector multiplication and two triangular solves. The corresponding CG code using the cuSPARSE and cuBLAS libraries in C programming language is shown below.

```
/***** CG Code *****/
/* ASSUMPTIONS:
   1. The cuSPARSE and cuBLAS libraries have been initialized.
   2. The appropriate memory has been allocated and set to zero.
   3. The matrix A (valA, csrRowPtrA, csrColIndA) and the incomplete-
      Cholesky upper triangular factor R (valR, csrRowPtrR, csrColIndR)
      have been computed and are present in the device (GPU) memory. */

//create the info and analyse the lower and upper triangular factors
cusparseCreateSolveAnalysisInfo(&inforRt);
cusparseCreateSolveAnalysisInfo(&inforR);
cusparseDcsrsv_analysis(handle,CUSPARSE_OPERATION_TRANSPOSE,
                        n, descrR, valR, csrRowPtrR, csrColIndR, inforRt);
cusparseDcsrsv_analysis(handle,CUSPARSE_OPERATION_NON_TRANSPOSE,
                        n, descrR, valR, csrRowPtrR, csrColIndR, inforR);

//1: compute initial residual r = f -  A x0 (using initial guess in x)
cusparseDcsrmv(handle, CUSPARSE_OPERATION_NON_TRANSPOSE, n, n, 1.0,
               descrA, valA, csrRowPtrA, csrColIndA, x, 0.0, r);
cublasDscal(n,-1.0, r, 1);
cublasDaxpy(n, 1.0, f, 1, r, 1);
nrmr0 = cublasDnrm2(n, r, 1);
```

(continues on next page)

(continued from previous page)

```
//2: repeat until convergence (based on max. it. and relative residual)
for (i=0; i<maxit; i++){
    //3: Solve M z = r (sparse lower and upper triangular solves)
    cusparseDcsrsv_solve(handle, CUSPARSE_OPERATION_TRANSPOSE,
                         n, 1.0, descrpR, valR, csrRowPtrR, csrColIndR,
                         inforRt, r, t);
    cusparseDcsrsv_solve(handle, CUSPARSE_OPERATION_NON_TRANSPOSE,
                         n, 1.0, descrpR, valR, csrRowPtrR, csrColIndR,
                         inforR, t, z);

    //4: \rho = r^{T} z
    rhop= rho;
    rho = cublasDdot(n, r, 1, z, 1);
    if (i == 0){
        //6: p = z
        cublasDcopy(n, z, 1, p, 1);
    }
    else{
        //8: \beta = rho_{i} / \rho_{i-1}
        beta= rho/rhop;
        //9: p = z + \beta p
        cublasDaxpy(n, beta, p, 1, z, 1);
        cublasDcopy(n, z, 1, p, 1);
    }

    //11: Compute q = A p (sparse matrix-vector multiplication)
    cusparseDcsrmv(handle, CUSPARSE_OPERATION_NON_TRANSPOSE, n, n, 1.0,
                   descrA, valA, csrRowPtrA, csrColIndA, p, 0.0, q);

    //12: \alpha = \rho_{i} / (p^{T} q)
    temp = cublasDdot(n, p, 1, q, 1);
    alpha= rho/temp;
    //13: x = x + \alpha p
    cublasDaxpy(n, alpha, p, 1, x, 1);
    //14: r = r - \alpha q
    cublasDaxpy(n,-alpha, q, 1, r, 1);

    //check for convergence
    nrmr = cublasDnrm2(n, r, 1);
    if (nrmr/nrmr0 < tol){
        break;
    }
}

//destroy the analysis info (for lower and upper triangular factors)
cusparseDestroySolveAnalysisInfo(inforRt);
cusparseDestroySolveAnalysisInfo(inforR);
```

For the nonsymmetric matrix A we can let M be its incomplete-LU factorization, so that $A \not= M = \widetilde{L}\widetilde{U}$, where $\widetilde{L}$ and $\widetilde{U}$ are lower and upper triangular matrices, respectively. Let us assume that M is nonsingular, then $M^{-1}A$ is nonsingular and instead of solving the linear system (1), we can solve the preconditioned linear system

$\left(M^{-1}A\right)\mathbf{x} = M^{-1}\mathbf{f}$	(3)

The pseudocode for the preconditioned BiCGStab iterative method is shown in Algorithm 2.

1.2. Algorithm 2 Bi-Conjugate Gradient Stabilized (BiCGStab)

1:	Letting initial guess be $\mathbf{x}_0$, compute $\mathbf{r} \leftarrow \mathbf{f} - A\mathbf{x}_0$	
2:	Set $\mathbf{p} \leftarrow \mathbf{r}$ and choose $\widetilde{\mathbf{r}}$, for example you can set $\widetilde{\mathbf{r}} \leftarrow \mathbf{r}$	
3:	**for** $i \leftarrow 1, 2, \ldots$ until convergence **do**	
4:	$\rho_i \leftarrow \widetilde{\mathbf{r}}^T \mathbf{r}$	
5:	**if** $\rho_i == 0.0$ **then**	
6:	method failed	
7:	**end if**	
8:	**if** $i > 1$ **then**	
9:	**if** $\omega == 0.0$ **then**	
10:	method failed	
11:	**end if**	
12:	$\beta \leftarrow \frac{\rho_i}{\rho_{i-1}} times \frac{\alpha}{\omega}$	
13:	$\mathbf{p} \leftarrow \mathbf{r} + \beta(\mathbf{p} - \omega\mathbf{v})$	
14:	**end if**	
15:	Solve $M\hat{\mathbf{p}} \leftarrow \mathbf{p}$	▷ Sparse lower and upper triangular solves
16:	Compute $\mathbf{q} \leftarrow A\hat{\mathbf{p}}$	▷ Sparse matrix-vector multiplication
17:	$\alpha \leftarrow \frac{\rho_i}{\widetilde{\mathbf{r}}^T \mathbf{q}}$	
18:	$\mathbf{s} \leftarrow \mathbf{r} - \alpha\mathbf{q}$	
19:	$\mathbf{x} \leftarrow \mathbf{x} + \alpha\hat{\mathbf{p}}$	
20:	**if** $\|s\|_2 \leq tol$ **then**	
21:	method converged	
22:	**end if**	
23:	Solve $M\hat{\mathbf{s}} \leftarrow \mathbf{s}$	▷ Sparse lower and upper triangular solves
24:	Compute $\mathbf{t} \leftarrow A\hat{\mathbf{s}}$	▷ Sparse matrix-vector multiplication
25:	$\omega \leftarrow \frac{\mathbf{t}^T \mathbf{s}}{\mathbf{t}^T \mathbf{t}}$	
26:	$\mathbf{x} \leftarrow \mathbf{x} + \omega\hat{\mathbf{s}}$	
27:	$\mathbf{r} \leftarrow \mathbf{s} - \omega\mathbf{t}$	
28:	**end for**	

Notice that in every iteration of the incomplete-LU preconditioned BiCGStab iterative method we need

to perform two sparse matrix-vector multiplications and four triangular solves. The corresponding BiCGStab code using the cuSPARSE and cuBLAS libraries in C programming language is shown below.

```
/***** BiCGStab Code *****/
/* ASSUMPTIONS:
   1. The cuSPARSE and cuBLAS libraries have been initialized.
   2. The appropriate memory has been allocated and set to zero.
   3. The matrix A (valA, csrRowPtrA, csrColIndA) and the incomplete-
      LU lower L (valL, csrRowPtrL, csrColIndL)  and upper U (valU,
      csrRowPtrU, csrColIndU) triangular factors have been
      computed and are present in the device (GPU) memory. */

//create the info and analyse the lower and upper triangular factors
cusparseCreateSolveAnalysisInfo(&infoL);
cusparseCreateSolveAnalysisInfo(&infoU);
cusparseDcsrsv_analysis(handle, CUSPARSE_OPERATION_NON_TRANSPOSE,
                        n, descrL, valL, csrRowPtrL, csrColIndL, infoL);
cusparseDcsrsv_analysis(handle, CUSPARSE_OPERATION_NON_TRANSPOSE,
                        n, descrU, valU, csrRowPtrU, csrColIndU, infoU);

//1: compute initial residual r = b - A x0 (using initial guess in x)
cusparseDcsrmv(handle, CUSPARSE_OPERATION_NON_TRANSPOSE, n, n, 1.0,
               descrA, valA, csrRowPtrA, csrColIndA, x, 0.0, r);
cublasDscal(n,-1.0, r, 1);
cublasDaxpy(n, 1.0, f, 1, r, 1);
//2: Set p=r and \tilde{r}=r
cublasDcopy(n, r, 1, p, 1);
cublasDcopy(n, r, 1, rw,1);
nrmr0 = cublasDnrm2(n, r, 1);

//3: repeat until convergence (based on max. it. and relative residual)
for (i=0; i<maxit; i++){
    //4: \rho = \tilde{r}^{T} r
    rhop= rho;
    rho = cublasDdot(n, rw, 1, r, 1);
    if (i > 0){
        //12: \beta = (\rho_{i} / \rho_{i-1}) ( \alpha / \omega )
        beta= (rho/rhop)*(alpha/omega);
        //13: p = r + \beta (p - \omega v)
        cublasDaxpy(n,-omega,q, 1, p, 1);
        cublasDscal(n, beta, p, 1);
        cublasDaxpy(n, 1.0,  r, 1, p, 1);
    }
    //15: M \hat{p} = p (sparse lower and upper triangular solves)
    cusparseDcsrsv_solve(handle, CUSPARSE_OPERATION_NON_TRANSPOSE,
                         n, 1.0, descrL, valL, csrRowPtrL, csrColIndL,
                         infoL, p, t);
    cusparseDcsrsv_solve(handle, CUSPARSE_OPERATION_NON_TRANSPOSE,
                         n, 1.0, descrU, valU, csrRowPtrU, csrColIndU,
                         infoU, t, ph);

    //16: q = A \hat{p} (sparse matrix-vector multiplication)
    cusparseDcsrmv(handle, CUSPARSE_OPERATION_NON_TRANSPOSE, n, n, 1.0,
                   descrA, valA, csrRowPtrA, csrColIndA, ph, 0.0, q);

    //17: \alpha = \rho_{i} / (\tilde{r}^{T} q)
    temp = cublasDdot(n, rw, 1, q, 1);
    alpha= rho/temp;
```

(continues on next page)

(continued from previous page)

```
    //18: s = r - \alpha q
    cublasDaxpy(n,-alpha, q, 1, r, 1);
    //19: x = x + \alpha \hat{p}
    cublasDaxpy(n, alpha, ph,1, x, 1);

    //20: check for convergence
    nrmr = cublasDnrm2(n, r, 1);
    if (nrmr/nrmr0 < tol){
        break;
    }

    //23: M \hat{s} = r (sparse lower and upper triangular solves)
    cusparseDcsrsv_solve(handle, CUSPARSE_OPERATION_NON_TRANSPOSE,
                         n, 1.0, descrL, valL, csrRowPtrL, csrColIndL,
                         infoL, r, t);
    cusparseDcsrsv_solve(handle, CUSPARSE_OPERATION_NON_TRANSPOSE,
                         n, 1.0, descrU, valU, csrRowPtrU, csrColIndU,
                         infoU, t, s);

    //24: t = A \hat{s} (sparse matrix-vector multiplication)
    cusparseDcsrmv(handle, CUSPARSE_OPERATION_NON_TRANSPOSE, n, n, 1.0,
                   descrA, valA, csrRowPtrA, csrColIndA, s, 0.0, t);

    //25: \omega = (t^{T} s) / (t^{T} t)
    temp = cublasDdot(n, t, 1, r, 1);
    temp2= cublasDdot(n, t, 1, t, 1);
    omega= temp/temp2;
    //26: x = x + \omega \hat{s}
    cublasDaxpy(n, omega, s, 1, x, 1);
    //27: r = s - \omega t
    cublasDaxpy(n,-omega, t, 1, r, 1);

    //check for convergence
    nrmr = cublasDnrm2(n, r, 1);
    if (nrmr/nrmr0 < tol){
        break;
    }
}

//destroy the analysis info (for lower and upper triangular factors)
cusparseDestroySolveAnalysisInfo(infoL);
cusparseDestroySolveAnalysisInfo(infoU);
```

As shown in Figure 2 the majority of time in each iteration of the incomplete-LU and Cholesky preconditioned iterative methods is spent in the sparse matrix-vector multiplication and triangular solve. The sparse matrix-vector multiplication has already been extensively studied in the following references [3], [15]. The sparse triangular solve is not as well known, so we briefly point out the strategy used to explore parallelism in it and refer the reader to the NVIDIA technical report [8] for further details.

To understand the main ideas behind the sparse triangular solve, notice that although the forward and back substitution is an inherently sequential algorithm for dense triangular systems, the dependencies on the previously obtained elements of the solution do not necessarily exist for the sparse triangular systems. We pursue the strategy that takes advantage of the lack of these dependencies and split the solution process into two phases as mentioned in [1], [4], [6], [7], [8], [10], [12], [14].

The *analysis* phase builds the data dependency graph that groups independent rows into levels based on the matrix sparsity pattern. The *solve* phase iterates across the constructed levels one-by-one and

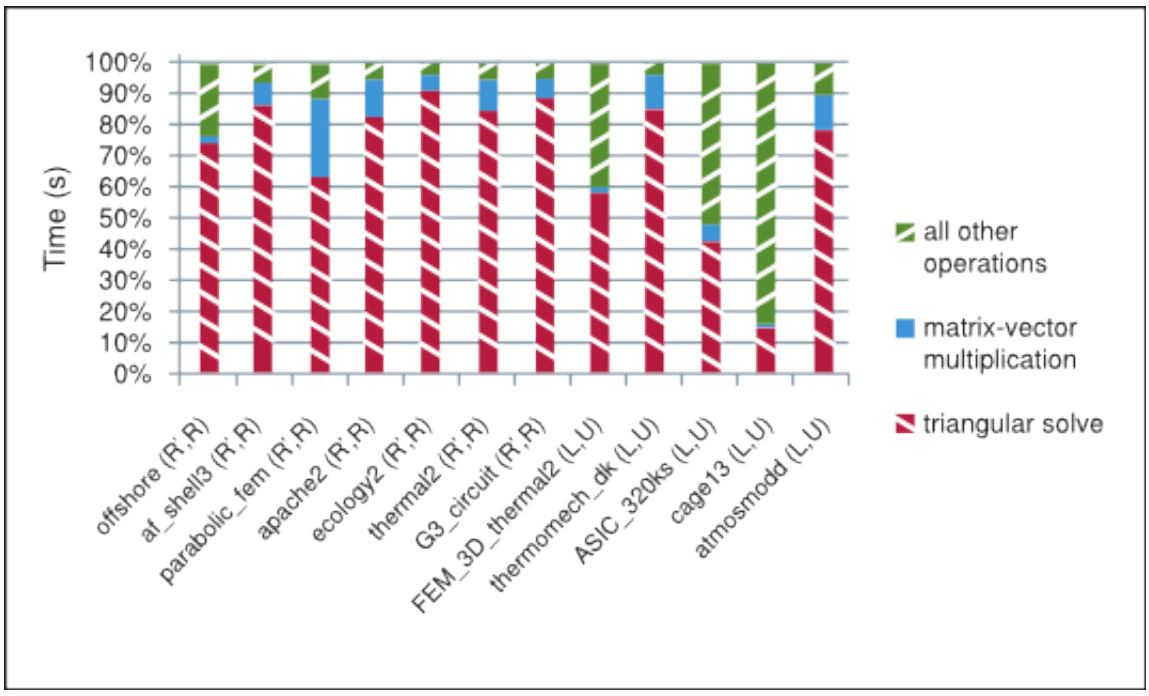


Fig. 1: The Splitting of Total Time Taken on the GPU by the Preconditioned Iterative Method

computes all elements of the solution corresponding to the rows at a single level in parallel. Notice that by construction the rows within each level are independent of each other, but are dependent on at least one row from the previous level.

The *analysis* phase needs to be performed only once and is usually significantly slower than the *solve* phase, which can be performed multiple times. This arrangement is ideally suited for the incomplete-LU and Cholesky preconditioned iterative methods.

Chapter 2. Numerical Experiments

In this section we study the performance of the incomplete-LU and Cholesky preconditioned *BiCGStab* and CG iterative methods. We use twelve matrices selected from The University of Florida Sparse Matrix Collection [18] in our numerical experiments. The seven s.p.d. and five nonsymmetric matrices with the respective number of rows (m), columns (n=m) and non-zero elements (nnz) are grouped and shown according to their increasing order in Table 1.

Table 1: Table 1. Symmetric Positive Definite (s.p.d.) and Non-symmetric Test Matrices

#	Matrix	m,n	nnz	s.p.d.	Application
1	offshore	259,789	4,242,673	yes	Geophysics
2	af_shell3	504,855	17,562,051	yes	Mechanics
3	parabolic_fem	525,825	3,674,625	yes	General
4	apache2	715,176	4,817,870	yes	Mechanics
5	ecology2	999,999	4,995,991	yes	Biology
6	thermal2	1,228,045	8,580,313	yes	Thermal Simulation
7	G3_circuit	1,585,478	7,660,826	yes	Circuit Simulation
8	FEM_3D_thermal2	147,900	3,489,300	no	Mechanics
9	thermomech_dK	204,316	2,846,228	no	Mechanics
10	ASIC_320ks	321,671	1,316,08511	no	Circuit Simulation
11	cage13	445,315	7,479,343	no	Biology
12	atmosmodd	1,270,432	8,814,880	no	Atmospheric Model

In the following experiments we use the hardware system with NVIDIA C2050 (ECC on) GPU and Intel Core i7 CPU 950 @ 3.07GHz, using the 64-bit Linux operating system Ubuntu 10.04 LTS, cuSPARSE library 4.0 and MKL 10.2.3.029. The MKL_NUM_THREADS and MKL_DYNAMIC environment variables are left unset to allow MKL to use the optimal number of threads.

We compute the incomplete-LU and Cholesky factorizations using the MKL routines `csrilu0` and `csrilut` with 0 and threshold fill-in, respectively. In the `csrilut` routine we allow three different levels of fill-in denoted by (5,10^{-3}), (10,10^{-5}) and (20,10^{-7}). In general, the (k, tol) fill-in is based on $nnz/n+k$ maximum allowed number of elements per row and the dropping of elements with magnitude $|l_{ij}|, |u_{ij}| < tol \times \|\mathbf{a}_i^T\|_2$, where l_{ij}, u_{ij} and $\mathbf{a}_i^T$ are the elements of the lower L, upper U triangular factors and the *i*-th row of the coefficient matrix A, respectively.

We compare the implementation of the BiCGStab and CG iterative methods using the cuSPARSE and

cuBLAS libraries on the GPU and MKL on the CPU. In our experiments we let the initial guess be zero, the right-hand-side $\mathbf{f} = A\mathbf{e}$ where $\mathbf{e}^T = (1, \ldots, 1)^T$, and the stopping criteria be the maximum number of iterations 2000 or relative residual $\|\mathbf{r}_i\|_2 / \|\mathbf{r}_0\|_2 < 10^{-7}$, where $\mathbf{r}_i = \mathbf{f} - A\mathbf{x}_i$ is the residual at *i*-th iteration.

Table 2: Table 2. `csrilu0` Preconditioned CG and BiCGStab Methods

	ilu0		**CPU**			**GPU**			**Speedup**
#	fact. time(s)	copy time(s)	solve time(s)	$\frac{\|\mathbf{r}_i\|_2}{\|\mathbf{r}_0\|_2}$	# it.	solve time(s)	$\frac{\|\mathbf{r}_i\|_2}{\|\mathbf{r}_0\|_2}$	# it.	vs. ilu0
1	0.38	0.02	0.72	8.83E-08	25	1.52	8.83E-08	25	0.57
2	1.62	0.04	38.5	1.00E-07	569	33.9	9.69E-08	571	1.13
3	0.13	0.01	39.2	9.84E-08	1044	6.91	9.84E-08	1044	5.59
4	0.12	0.01	35.0	9.97E-08	713	12.8	9.97E-08	713	2.72
5	0.09	0.01	107	9.98E-08	1746	55.3	9.98E-08	1746	1.92
6	0.40	0.02	155	9.96E-08	1656	54.4	9.79E-08	1656	2.83
7	0.16	0.02	20.2	8.70E-08	183	8.61	8.22E-08	183	2.32
8	0.32	0.02	0.13	5.25E-08	4	0.52	5.25E-08	4	0.53
9	0.20	0.01	72.7	1.96E-04	2000	40.0	2.08E-04	2000	1.80
10	0.11	0.01	0.27	6.33E-08	6	0.12	6.33E-08	6	1.59
11	0.70	0.03	0.28	2.52E-08	2.5	0.15	2.52E-08	2.5	1.10
12	0.25	0.04	12.5	7.33E-08	76.5	4.30	9.69E-08	74.5	2.79

Table 3: Table 3. `csrilut`(5,10^{-3}) Preconditioned CG and BiCGStab Methods

	ilut(5,10^{-3})		**CPU**			**GPU**			**Speedup**	
#	fact. time(s)	copy time(s)	solve time(s)	$\frac{\|\mathbf{r}_i\|_2}{\|\mathbf{r}_0\|_2}$	# it.	solve time(s)	$\frac{\|\mathbf{r}_i\|_2}{\|\mathbf{r}_0\|_2}$	# it.	vs. ilut (5,10^{-3})	vs. ilu0
1	0.14	0.01	1.17	9.70E-08	32	1.82	9.70E-08	32	0.67	0.69
2	0.51	0.03	49.1	9.89E-08	748	33.6	9.89E-08	748	1.45	1.39
3	1.47	0.02	11.7	9.72E-08	216	6.93	9.72E-08	216	1.56	1.86
4	0.17	0.01	67.9	9.96E-08	1495	26.5	9.96E-08	1495	2.56	5.27
5	0.55	0.04	59.5	9.22E-08	653	71.6	9.22E-08	653	0.83	1.08
6	3.59	0.05	47.0	9.50E-08	401	90.1	9.64E-08	401	0.54	0.92
7	1.24	0.05	23.1	8.08E-08	153	24.8	8.08E-08	153	0.93	2.77
8	0.82	0.03	0.12	3.97E-08	2	1.12	3.97E-08	2	0.48	1.10
9	0.10	0.01	54.3	5.68E-04	2000	24.5	1.58E-04	2000	2.21	1.34
10	0.12	0.01	0.16	4.89E-08	4	0.08	6.45E-08	4	1.37	1.15
11	4.99	0.07	0.36	1.40E-08	2.5	0.37	1.40E-08	2.5	0.99	6.05
12	0.32	0.03	39.2	7.05E-08	278.5	10.6	8.82E-08	270.5	3.60	8.60

The results of the numerical experiments are shown in Table 2 through Table 5, where we state the speedup obtained by the iterative method on the GPU over CPU (speedup), number of iterations required for convergence (# it.), achieved relative residual ($\frac{\|\mathbf{r}_i\|_2}{\|\mathbf{r}_0\|_2}$) and time in seconds taken by the factorization (fact.), iterative solution of the linear system (solve), and `cudaMemcpy` of the lower and upper triangular factors to the GPU (copy). We include the time taken to compute the incomplete-LU and Cholesky factorization as well as to transfer the triangular factors from the CPU to the GPU memory in the computed speedup.

Table 4: Table 4. `csrilut`(10,10^{-5}) Preconditioned CG and BiCGStab Methods

	ilut(10,10^{-5})		**CPU**			**GPU**			**Speedup**	
#	fact. time(s)	copy time(s)	solve time(s)	$\frac{\|\mathbf{r}_i\|_2}{\|\mathbf{r}_0\|_2}$	# it.	solve time(s)	$\frac{\|\mathbf{r}_i\|_2}{\|\mathbf{r}_0\|_2}$	# it.	vs. ilut (10,10^{-5})	vs. ilu0
1	0.15	0.01	1.06	8.79E-08	34	1.96	8.79E-08	34	0.57	0.63
2	0.52	0.03	60.0	9.86E-08	748	38.7	9.86E-08	748	1.54	1.70
3	3.89	0.03	9.02	9.79E-08	147	5.42	9.78E-08	147	1.38	1.83
4	1.09	0.03	34.5	9.83E-08	454	38.2	9.83E-08	454	0.91	2.76
5	3.25	0.06	26.3	9.71E-08	272	55.2	9.71E-08	272	0.51	0.53
6	11.0	0.07	44.7	9.42E-08	263	84.0	9.44E-08	263	0.59	1.02
7	5.95	0.09	8.84	8.53E-08	43	17.0	8.53E-08	43	0.64	1.68
8	2.94	0.04	0.09	2.10E-08	1.5	1.75	2.10E-08	1.5	0.64	3.54
9	0.11	0.01	53.2	4.24E-03	2000	24.4	4.92E-03	2000	2.18	1.31
10	0.12	0.01	0.16	4.89E-11	4	0.08	6.45E-11	4	1.36	1.18
11	2.89	0.09	0.44	6.10E-09	2.5	0.48	6.10E-09	2.5	1.00	33.2
12	0.36	0.03	36.6	7.05E-08	278.5	10.6	8.82E-08	270.5	3.35	8.04

Table 5: Table 5. `csrilut`(20,10^{-7}) Preconditioned CG and BiCGStab Methods

	ilut(20,10^{-7})		**CPU**			**GPU**			**Speedup**	
#	fact. time(s)	copy time(s)	solve time(s)	$\frac{\|\mathbf{r}_i\|_2}{\|\mathbf{r}_0\|_2}$	# it.	solve time(s)	$\frac{\|\mathbf{r}_i\|_2}{\|\mathbf{r}_0\|_2}$	# it.	vs. ilut (20,10^{-7})	vs. ilu0
1	0.82	0.02	47.6	9.90E-08	1297	159	9.86E-08	1292	0.30	25.2
2	9.21	0.11	32.1	8.69E-08	193	84.6	8.67E-08	193	0.44	1.16
3	10.04	0.04	6.26	9.64E-08	90	4.75	9.64E-08	90	1.10	2.36
4	8.12	0.10	15.7	9.02E-08	148	22.5	9.02E-08	148	0.78	1.84
5	8.60	0.10	21.2	9.52E-08	158	53.6	9.52E-08	158	0.48	0.54
6	35.2	0.11	29.2	9.88E-08	162	80.5	9.88E-08	162	0.56	1.18
7	23.1	0.14	3.79	7.50E-08	14	12.1	7.50E-08	14	0.76	3.06
8	5.23	0.05	0.14	1.19E-09	1.5	2.37	1.19E-09	1.5	0.70	6.28
9	0.12	0.01	55.1	3.91E-03	2000	24.4	2.27E-03	2000	2.25	1.36
10	0.14	0.01	0.14	9.35E-08	3.5	0.07	7.19E-08	3.5	1.28	1.18
11	218	0.12	0.43	9.80E-08	2	0.66	9.80E-08	2	1.00	247.
12	15.0	0.21	12.2	3.45E-08	31	4.95	3.45E-08	31	1.35	5.93

The summary of performance of BiCGStab and CG iterative methods preconditioned with different incomplete factorizations on the GPU is shown in Figure 3, where "*" indicates that the method did not converge to the required tolerance. Notice that in general in our numerical experiments the performance for the incomplete factorizations decreases as the threshold parameters are relaxed and the factorization becomes more dense, thus inhibiting parallelism due to data dependencies between rows in the sparse triangular solve. For this reason, the best performance on the GPU is obtained for the incomplete-LU and Cholesky factorization with 0 fill-in, which will be our point of reference.

Although the incomplete factorizations with a more relaxed threshold are often closer to the exact factorization and thus result in fewer iterative steps, they are also much more expensive to compute. Moreover, notice that even though the number of iterative steps decreases, each step is more computationally expensive. As a result of these tradeoffs the total time, the sum of the time taken by the factorization and the iterative solve, for the iterative method does not necessarily decrease with a more relaxed threshold in our numerical experiments.

The speedup based on the total time taken by the preconditioned iterative method on the GPU with

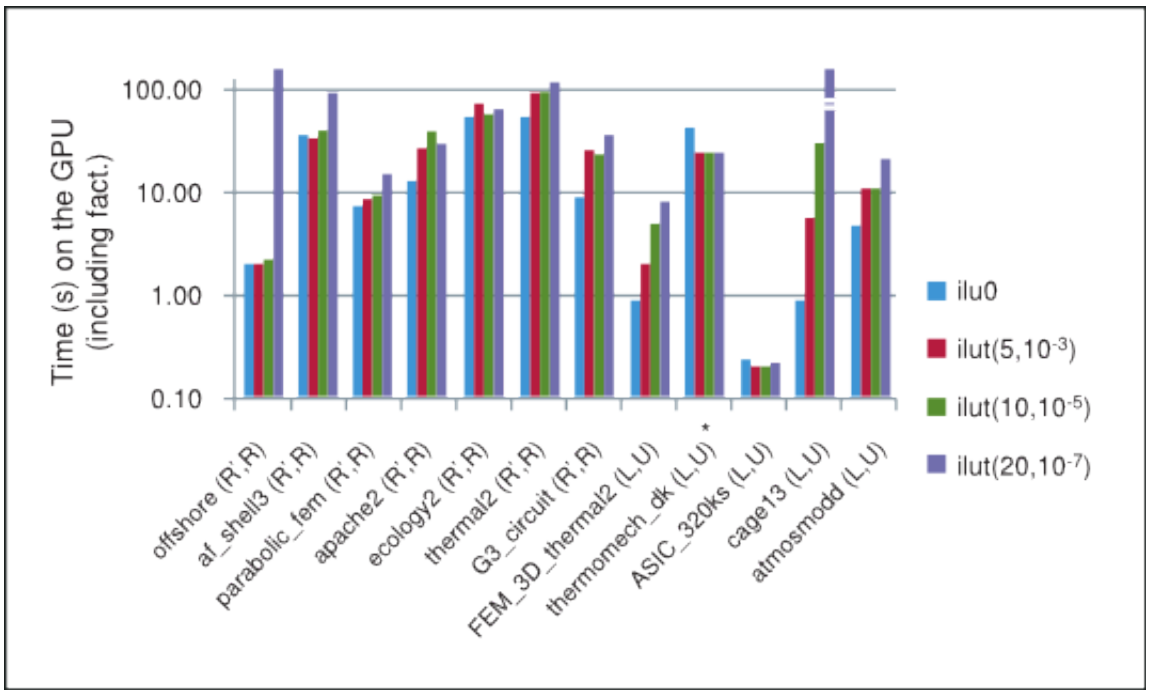


Fig. 1: Performance of BiCGStab and CG with Incomplete-LU Cholesky Preconditioning

`csrilu0` preconditioner and CPU with all four preconditioners is shown in Figure 4. Notice that for majority of matrices in our numerical experiments the implementation of the iterative method using the cuSPARSE and cuBLAS libraries does indeed outperform the MKL.

Finally, the average of the obtained speedups is shown in Figure 5, where we have excluded the runs with cage13 matrix for `ilut`(10,10^{-5}) and runs with offshore and cage13 matrices for `ilut`(20,10^{-7}) incomplete factorizations because of their disproportional speedup. However, the speedup including these runs is shown in parenthesis on the same plot. Consequently, we can conclude that the incomplete-LU and Cholesky preconditioned BiCGStab and CG iterative methods obtain on average more than 2x speedup on the GPU over their CPU implementation.

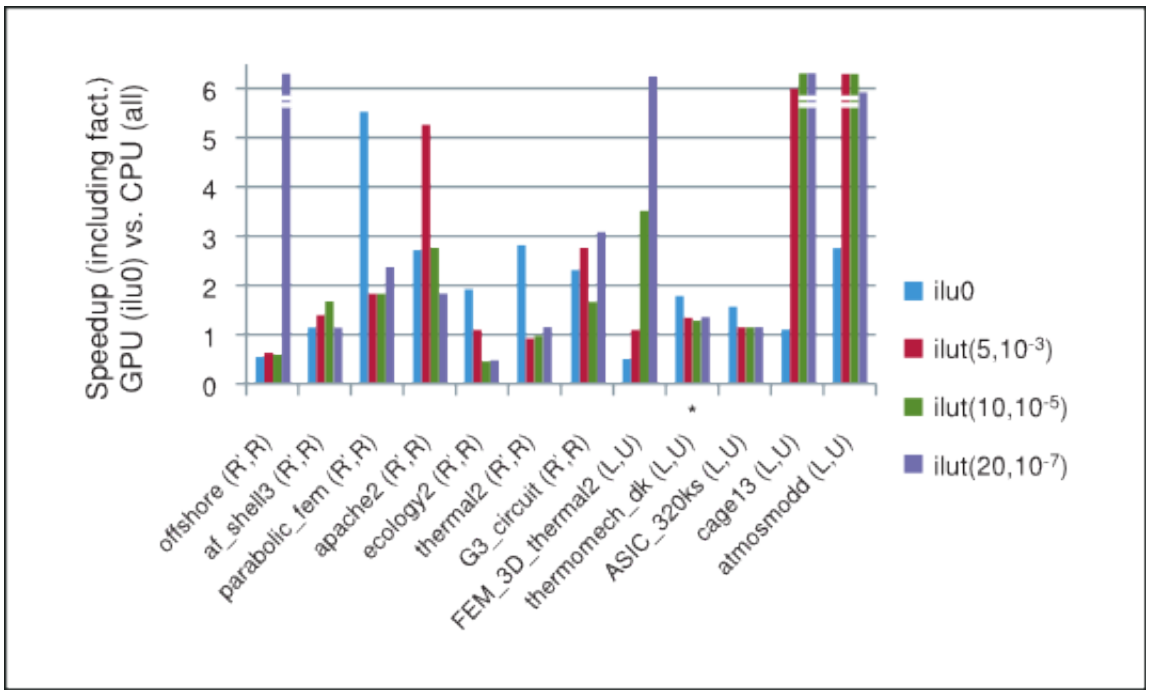


Fig. 2: Speedup of prec. BiCGStab and CG on GPU (with `csrilu0`) vs. CPU (with all)

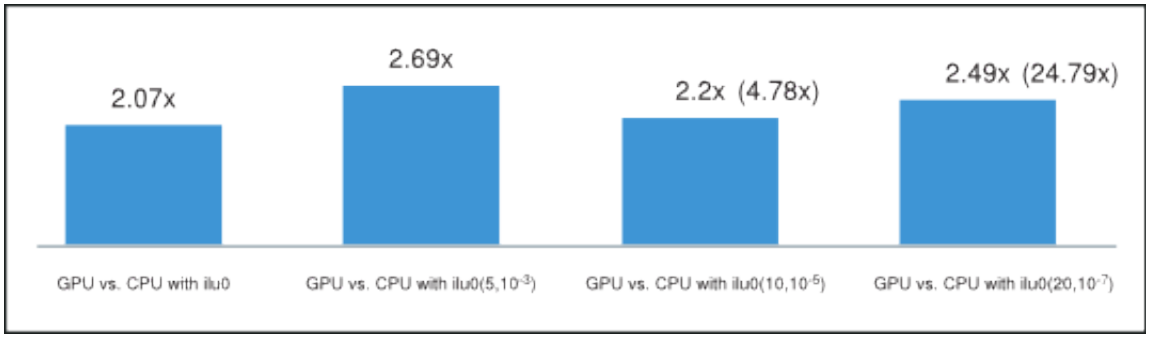


Fig. 3: Average Speedup of BiCGStab and CG on GPU (with `csrilu0`) and CPU (with all)

Chapter 3. Conclusion

The performance of the iterative methods depends highly on the sparsity pattern of the coefficient matrix at hand. In our numerical experiments the incomplete-LU and Cholesky preconditioned BiCGStab and CG iterative methods implemented on the GPU using the cuSPARSE and cuBLAS libraries achieved an average of 2x speedup over their MKL implementation.

The sparse matrix-vector multiplication and triangular solve, which is split into a slower *analysis* phase that needs to be performed only once and a faster *solve* phase that can be performed multiple times, were the essential building blocks of these iterative methods. In fact the obtained speedup was usually mostly influenced by the time taken by the *solve* phase of the algorithm.

Finally, we point out that the use of multiple-right-hand-sides would increase the available parallelism and can result in a significant relative performance improvement in the preconditioned iterative methods. Also, the development of incomplete-LU and Cholesky factorizations using CUDA parallel programming paradigm can further improve the obtained speedup.

Chapter 4. Acknowledgements

This white paper was authored by Maxim Naumov for NVIDIA Corporation.

Permission to make digital or hard copies of all or part of this work for any use is granted without fee provided that copies bear this notice and the full citation on the first page.

Chapter 5. References

[1] E. Anderson and Y. Saad Solving Sparse Triangular Linear Systems on Parallel Computers, Int. J. High Speed Comput., pp. 73-95, 1989.

[2] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. van der Vorst, Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, SIAM, Philadelphia, PA, 1994.

[3] N. Bell and M. Garland, Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors, Proc. Conf. HPC Networking, Storage and Analysis (SC09), ACM, pp. 1-11, 2009.

[4] A. Greenbaum, Solving Sparse Triangular Linear Systems using Fortran with Parallel Extensions on the NYU Ultracomputer Prototype, Report 99, NYU Ultracomputer Note, New York University, NY, April, 1986.

[5] D. B. Kirk and W. W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Elsevier, 2010.

[6] J. Mayer, Parallel Algorithms for Solving Linear Systems with Sparse Triangular Matrices, Computing, pp. 291-312 (86), 2009.

[7] R. Mirchandaney, J. H. Saltz and D. Baxter, Run-Time Parallelization and Scheduling of Loops, IEEE Transactions on Computers, pp. (40), 1991.

[8] M. Naumov, Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU, NVIDIA Technical Report, NVR-2011-001, 2011.

[9] J. Nickolls, I. Buck, M. Garland and K. Skadron, Scalable Parallel Programming with CUDA, Queue, pp. 40-53 (6-2), 2008.

[10] E. Rothberg and A. Gupta, Parallel ICCG on a Hierarchical Memory Multiprocessor - Addressing the Triangular Solve Bottleneck, Parallel Comput., pp. 719-741 (18), 1992.

[11] Y. Saad, Iterative Methods for Sparse Linear Systems, SIAM, Philadelphia, PA, 2nd Ed., 2003.

[12] J. H. Saltz, Aggregation Methods for Solving Sparse Triangular Systems on Multiprocessors, SIAM J. Sci. Statist. Comput., pp. 123-144 (11), 1990.

[13] J. Sanders and E. Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley, 2010.

[14] M. Wolf, M. Heroux and E. Boman, Factors Impacting Performance of Multithreaded Sparse Triangular Solve, 9th Int. Meet. HPC Comput. Sci. (VECPAR), 2010.

[15] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick and J. Demmel, Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms, Parallel Comput., pp. 178-194 (35-3), 2009.

[16] NVIDIA cuSPARSE and cuBLAS Libraries,http://www.nvidia.com/object/cuda_develop.html

[17] Intel Math Kernel Library,http://software.intel.com/en-us/articles/intel-mkl

[18] The University of Florida Sparse Matrix Collection, http://www.cise.ufl.edu/research/sparse/matrices/.

Chapter 6. Notices

6.1. Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation (“NVIDIA”) makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer (“Terms of Sale”). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer’s own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer’s sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer’s product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or

services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

6.2. OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

6.3. Trademarks

NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

©2011-2026, NVIDIA Corporation & affiliates. All rights reserved