High Performance GPU-based Power Grid Analysis

Manuel Reis
Instituto Superior Técnico
Lisbon, Portugal
manuel.b.reis@tecnico.ulisboa.pt

***Abstract*—The work developed under this dissertation belongs to the fields of simulation and optimization, in this case applied to the verification of circuits with a large complexity. Namely, the power delivery network of integrated circuits — *power grid* — is studied.**

The problem faced consists of identifying if the power grid is designed to ensure it powers correctly all the devices in the circuit, which, by commuting, drain the required currents to keep their voltage levels within appropriate values. This is particularly hard given extension of the power grid and the enormous amount of devices connected to it commuting at high speeds.

The most computationally demanding step in the verification of the electric voltages on the nodes of a *power grid* is the solution of linear systems of equations. These systems present recurrent properties such as sparsity or symmetry, which can be explored to enhance the efficiency of the process. For that purpose, two types of methods were studied: direct methods and iterative methods. In both cases using techniques to reduce their complexity and the GPU (Graphics Processing Unit) to accelerate the computational process.

It was verified in both types that the existence of data dependencies limits the performance and the optimal usage of the GPU resources. It was confirmed that the use of direct methods, is limited by their higher memory demand when compared to the iterative. On the other hand, better performances can be obtained more easily.

The results were obtained using real *power grid* descriptions, with the respective solutions. An artificial set of matrices was generated in the attempt to evaluate the dependency of the performance with the problem regularity.

***Index Terms*—Power Grid, power delivery network, GPU, CUDA, Cholesky decomposition, Reverse Cuthill-McKee, fill-in, direct methods, Conjugate Gradient method, iterative preconditioned methods**

I. INTRODUCTION

Modern electronic suffered an enormous evolution, being able to shrink the components, allowing the chips to contain more processing power, increasing their performance, and reducing their power consumption. However, by doing so, some problems arise, the chip's complexity becomes very high, and the power delivery system design becomes a critical step to ensure the (lower) voltages don't leave a security interval.

This power delivery system is called a power grid, and it is composed of a network of conductors that must be subjected to detailed behavior analysis, to improve its robustness, and guarantee the (strict) requirements for proper functioning.

A. Problem definition

The power grid serves the purpose of driving the power and ground voltages from pad locations to all devices in the design. Its performance and reliability may be compromised when facing voltage drops due to non-ideal resistive wires since it immediately compromises the intent of delivery the correct power signal to the components. Designing a power grid becomes a challenge when pursuing excellent voltage regulation at the points where power is drained, especially with the vast power demand across the chip. Having a robust power grid is essential to have a reliable, functional system; therefore, evaluating its robustness and integrity has become a significant concern.

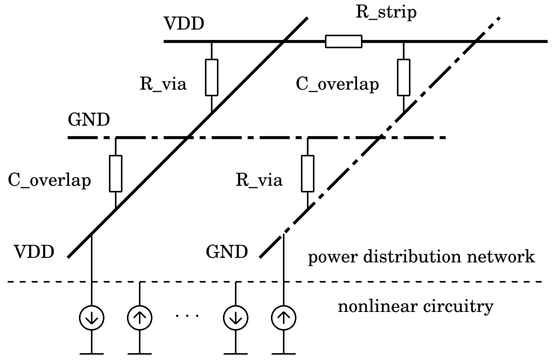


Fig. 1: The power grid is modeled as a RC mesh.

To solidify the importance of the analysis problem and the growing complexity of a power grid (with millions of nodes) an example from [1] is presented:

In a high-performance processor, it is common to dedicate 10% of the wiring to the power delivery. This happens because the power dissipation in a multi-processor can reach $100W$, working at $1V$ power supply. In the same conditions, a small resistance of $100m\Omega$ causes a voltage drop of 10% of the voltage supply. One can see a significant impact on the fulfillment of the objective.

Usually, an integrated circuit has several metal layers. visualizing a chip with eight layers, each with 10^3 wires, if each connection between metal layers (via) is considered, each wire will model 10^3 resistive nodes. Within a metal level, there will be 10^6 nodes, resulting in a power grid with 8 million nodes.

Different components within an integrated circuit draw power from the power grid, resulting in a time-dependent behavior. To perform power delivery analysis is therefore usual to consider two situations: steady-state or DC conditions and time-domain transient analysis. It will be shown that by focusing on steady-state (DC), the time domain computionally demanding problem can be solved in a similar way. To solve this analysis problem, given its size growth with the increasing complexity of the grid, computational requirements are high, both in terms of memory and processing power, although one can benefit from the structure of the problem to customize the processing stage.

Given the requirements, the use of GPU is considered in this context because these devices are commonly available and are massive parallel processors, giving the possibility of achieving better performance than CPUs on types of problems that present a data parallelism nature.

II. Background and Related Work

The power grid analysis is divided into two parts, which is very common when there are components with time-dependent behavior present in the circuit, in this case, capacitors. In both cases, the goal is to know the voltage in any node of the grid, to check if there are values outside the interval of safety.

A. Steady state

When the voltages and the currents are stable in every node, the components with time-dependent behavior are not active (capacitors act like open circuits). In these conditions, the goal to compute the DC voltage in every node (v) becomes simple, with the first-order terms removed from the equation.

It is only necessary to consider the resistive components, given by the matrix G that represents all the conductances between nodes in a power grid, and the stable current loads in each node (b), which is also known. The solution is obtained by solving the system:

$$G \cdot v = b \tag{1}$$

B. Time domain

When a disturbance takes place (the currents loads change, for instance), the analysis becomes time-dependent, a transient phase occurs, and the capacitances of the nodes (C) can no longer be ignored. Therefore, the relation between voltage and current is given as a function of time by the following differential equation:

$$G \cdot v(t) + C \cdot \dot{v}(t) = b(t) \tag{2}$$

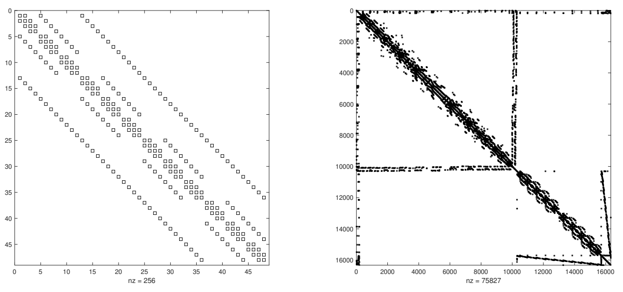
Using discretization methods with a constant time step h, for instance implicit Euler method [2], one can define $\dot{v}(t+h) = \frac{v(t+h)-v(t)}{h}$. The Equation (2) can be rewritten as Equation (3), after the simplification.

$$\left(G + \frac{C}{h}\right) \cdot v(t+h) = \frac{C \cdot v(t)}{h} + b_{t+h} \tag{3}$$

For a given t (with the prior solution, $v(t-h)$), all the elements on the right-hand side are known. G and C are known as well, they don't change under these circumstances (constant h). If both matrices are factorized, the solution process becomes cheaper; this is a strong reason to consider matrix decomposition methods.

C. Sparse Matrix Formats

Matrices G and C represent a respective relation between every pair of nodes in the power grid. Given the mesh-like format of the physical grid, these relations will only be existent between neighbor nodes. This means that the sparsity is very high, and its density decreases as it grows, for the same reason symmetry is observed. If power grid would be a perfect mesh the matrices obtain would look like Figure 2a. However, the extraction process of the grid induces the order of the nodes, some of them are merged when handling shorts, and the fact the grid is not regular (lacking several connections) lead to a more chaotic scenario in terms of patterns that can be seen in Figure 2b.



(a) (b)

Fig. 2: The first matrix is Laplacian and obtained from a regular 4x3x4 grid. By its turn, the second matrix is obtained by processing a real power grid description file (*ibmpg1* from IBM benchmarks [1]).

Despite matrices look totally different, both have approximately the same number of non-zero per row elements. To store this types of matrices two formats were considered, Band storage and ELLPACK.

On the **Band storage**, a matrix with nonzeros only within *Lb* diagonals bellow and *Ub* above the main diagonal is stored in a matrix of size $(Lb + Ub + 1) \times N$ (*data*). The columns are stored in the respective columns of *data*. To exemplify, Algorithm 1 and Figure 3 can be helpful, in the case when $Lb = 2$ and $Ub = 1$ causing $M = (Lb + Ub + 1) = 4$.

Algorithm 1 Band storage indexing.

```
procedure BANDINDEXING($i, j, data, M, Lb, Ub$)
    if $\max(1, j - Ub) \le i \le \min(M - 1, j + Lb)$ then
        return $data[Ub + 1 + i - j][j]$
    return $0$
```

$$A = \begin{bmatrix} a_{11} & a_{12} & * & * \\ * & a_{22} & a_{23} & * \\ a_{31} & * & a_{33} & a_{34} \\ * & a_{42} & * & a_{44} \end{bmatrix} \quad \text{data} = \begin{bmatrix} * & a_{12} & a_{23} & a_{34} \\ a_{11} & a_{22} & a_{33} & a_{44} \\ * & * & * & * \\ a_{31} & a_{42} & * & * \end{bmatrix}$$

Fig. 3: Banded Storage matrix representation.

ELLPACK becomes more efficient when the average number of nonzeros per row does not differ significantly from its maximum. The $N \times N$ matrix is represented by two matrices of dimensions $N \times M$, where M is the maximum number of elements per rows observed. The rows are indexed implicitly, but the columns of a given row are compressed over the respective row of the two arrays (*columns* and *values*). To clarify, the Algorithm 2 combined with Figure 4 show how a_{ij} maps into *columns* and *values*.

Algorithm 2 ELLPACK indexing.

```
1: procedure ELLINDEXING($i, j, columns, values, M$)
2:     for $k = 0,$ to $M - 1$ do
3:         if $column[i][k] == j$ then
4:             return $value[i][k]$
5:     return 0
```

$$A = \begin{bmatrix} a_{11} & a_{12} & * & * \\ * & a_{22} & a_{23} & * \\ a_{31} & * & a_{33} & a_{34} \\ * & a_{42} & * & a_{44} \end{bmatrix}$$

$$\text{columns} = \begin{bmatrix} 0 & 1 & * \\ 1 & 2 & * \\ 0 & 2 & 3 \\ 1 & 3 & * \end{bmatrix} \qquad \text{values} = \begin{bmatrix} a_{11} & a_{12} & * \\ a_{22} & a_{23} & * \\ a_{31} & a_{33} & a_{34} \\ a_{42} & a_{44} & * \end{bmatrix}$$

Fig. 4: ELLPACK sparse matrix representation.

D. Direct Methods

Direct methods solve a system of equations in a finite number of operations. This group of methods is characterized for determining an exact solution of the system (when the error associated with rounding operations is absent).

To reduce the complexity when solving multiple systems with the same matrix, factorization will be addressed [3]. Matrix decomposition or factorization consists in defining a matrix as a product of others. The computational cost of the factorization algorithms is of the same order as most direct solvers, asymptotic complexities round $\mathcal{O}(N^3)$, where N is the size of the matrix. After the matrix is factorized, the cost to solve the equivalent system is lower, $\mathcal{O}(N^2)$. In the equivalent system, instead of using the original matrix, it is replaced by the product of the factors: $Ax = b \iff F_1F_2x = b$.

There are several factorization techniques [4], [5]. Cholesky decomposition can only be used over positive-definite matrices (symmetric). It determines the lower triangular matrix L (Equations (4) and (5)), that multiplied by its conjugate transposed, gives A ($A = LL^T$).

$$L_{j,j} = \sqrt{A_{j,j} - \sum_{k=1}^{j-1} L_{j,k}^2} \tag{4}$$

$$L_{i,j} = \frac{1}{L_{j,j}}(A_{i,j} - \sum_{k=1}^{j-1} L_{i,k}L_{j,k}) \quad , \quad \text{for} \quad i > j \tag{5}$$

This definition of L has a strong and well-defined dependence between its coefficients, for instance, to compute $L_{i,j}$ one must already have the values of all $L_{i,k}$ and $L_{j,k}$ where $k < j$. There are three common variants for this algorithm, up-looking, right-looking and left-looking, also called *ijk* variants (from many others covered in this survey [6]). Each of them has its advantages and disadvantages.

The right-looking method is known to be more successfully parallelized than the others because it offers more independent operation during the *submatrix update* [7], [8]. The algorithm iterates over the columns of the matrix, only considering, at each time, the values below the main diagonal of the current iteration, called *panel* in [7] and the ones at its right, denominated *trailing submatrix* by the same source.

At each iteration, the *panel* is factorized and not used again in the following. Before moving to the next *panel*, all the coefficients to its right will be updated (*trailing submatrix update*) using the respective factorized entries. This can be seen in Algorithm 3.

It is very important to highlight that during the decomposition of sparse matrices fill-in occurs, making the factors more dense than the respective part of the original matrix. This increases drastically the memory requirements and the computational complexity, if not handled.

Algorithm 3 Right-looking Cholesky.

```
1: procedure RLCHOLESKY($A$)
2:     $L = A$
3:     for $k = 1,$ to N do
4:         $L_{k,k} = \sqrt{L_{k,k}}$
5:         for $i = k + 1,$ to N do
6:             $L_{i,k} = \frac{L_{i,k}}{L_{k,k}}$
7:         for $i = k + 1,$ to N do
8:             for $j = k + 1,$ to N do
9:                 $L_{j,i} = L_{j,i} - L_{j,k} \times L_{i,k}$
```

E. Iterative methods

Iterative methods do successive approximations with the goal of obtaining more accurate solutions to a linear system of equations, and can be also used in this problem. The convergence rate of an iterative method is intimately dependent on the coefficient matrix's spectrum (the set of its eigenvalues [9], [10]). Hence, is common to use another matrix to transform the spectrum of the original matrix into one with a spectrum that helps increasing the convergence rate. These methods are called preconditioned iterative methods, where the second matrix is referred to as preconditioner (P). The preconditioned system to be solved is $P^{-1}Ax = P^{-1}b$, instead of the original.

The Conjugate Gradient is one of the best iterative methods known for solving sparse symmetric positive definite linear systems. At each step, the method computes an approximation of the solution, the associated residual error, and new directions used to perform the following updates to the iterates and residuals. This makes the Conjugate Gradient Method quite attractive computationally [10]. Its preconditioned version pseudocode is presented in Algorithm 4, using $I = P$ is equivalent to not using preconditioning.

Algorithm 4 Preconditioned Conjugate Gradient

```
1: procedure PCG($A$,$b$)
2:     Initialize $x^0$, $r^0 = b - Ax^0$, $k = 1$
3:     while not convergence do
4:         solve $Pz^{k-1} = r^{k-1}$
5:         $\rho_{k-1} = {r^{k-1}}^T z^{k-1}$
6:         if $k == 1$ then
7:             $p^1 = z^0$
8:         else
9:             $\beta_{k-1} = \frac{\rho_{k-1}}{\rho_{k-2}}$
10:            $p^k = z^{k-1} + \beta_{k-1} p^{k-1}$
11:        $q^k = Ap^k$
12:        $\alpha_k = \frac{{r^{(k-1)}}^T r^{(k-1)}}{{p^k}^T q^k}$
13:        $x^k = x^{k-1} + \alpha_k p^k$
14:        $r^k = r^{k-1} - \alpha_k q^k$
15:        $k = k + 1$
```

F. GPU programming

The GPU is designed to manage a large number of threads (compared to a CPU), because their management is done at the hardware layer making it very efficient [11]. It employs a SIMT (same-instruction multiple-thread) architecture, where the threads are free to diverge, however to efficiently use the GPU, threads should execute the same instructions as much as possible. The GPU is composed of several streaming multiprocessors (SMs), each one having eight streaming processors (SPs), their threads can access registers and shared memory with low latency, but also global, and constant memories. The off-chip memory (global and constant) has bandwidth, but if threads randomly access it, the throughput decreases drastically.

In CUDA programming model [12], the host (CPU) program, throws parallel programs (kernels) to be executed by a parallel device (GPU) [13]. It organizes the threads that will execute a kernel in a *grid* of *thread blocks*. This organization can be from one-dimensional up to three-dimensional. Within the same block, threads can cooperate using barrier synchronization and shared memory. The launch of the kernel requires the specification of the grid size (number of *blocks* and *threads per block*). When the host program invokes a kernel, the grid's blocks are enumerated and then distributed to the several multiprocessors available. The threads within a block execute on the same multiprocessor. Multiple blocks can run concurrently on the same multiprocessor. When blocks terminate, new blocks are launched occupying the vacated multiprocessors [12].

G. GPU solvers

It is a challenge to efficiently implement sparse matrix solvers on the GPU [14]. A sparse matrix LU decomposition, GLU, was suggested in [8], [15], it performs symbolic analysis and reorders the matrix before the GPU computations to reduce the fill-in. The algorithm itself is based on a right-looking, but instead of iterating over columns, it traverses the layers of the dependency tree (generated by the symbolic analysis). Within the same layer, each node represents a column that is linearly independent of the others. It uses CSR sparse matrix format. The symbolic analysis also focuses on reducing fill-in, using AMD [16] to find such permutation. As for the parallelization design, since within a level of dependency there may exist more than one column, the computations over each one are parallelized. However, as mentioned before, the right-looking algorithm presupposes a submatrix update that itself can be independently parallelized. Therefore multiple submatrix updates are being performed simultaneously, causing race conditions, handled by thread synchronization and atomic floating-point operations.

CHOLMOD [17] is a set of routines, which includes the factorization of sparse symmetric matrices and solving triangular systems, from many other methods. Its Cholesky decomposition implementation is hybrid in the sense that GPU acceleration is optional, but the results show significant improvement when using it. It is a supernodal left-looking variant, and exploits dense matrix kernels, achieving high performance on modern computers. The fill-in reduction problem is handled by using a combination of the nested dissection and minimum degree orderings [18], [19], but its unavoidable occurrence is harnessed by the supernodal method. A supernode is a collection of similar columns, and by grouping them, the remaining fill-in occurs is a more controlled way, exploiting the dense matrix parallel operations on certain regions of the matrix.

The Conjugate Gradient method can be easily parallelized on the GPU with the resource to a combination of libraries such as cuBLAS [20], cuSPARSE [21], and for the preconditioned version, cuSOLVER [22]. As mentioned in [23], the use of the cuSPARSE library is very optimized, especially for the CSR matrix representation. A CUDA approach was explained [24] where there is an effort to parallelize efficiently the operations performed during an iteration.

III. Architecture

A. GPUBandSolver

The direct method approach developed (GPUBandSolver) is a Cholesky decomposition that uses the Reverse Cuthill-McKee permutation [6], [25]. This ordering method, is applied to a symmetric sparse matrix and reduces its bandwidth. Despite purpose of this matrix permutionation is not minimization of fill-in, by using it, the benchmark matrices become banded matrices and that case, the fill-in appearance is restricted to the entries within the band (proven in [3]). The Reverse Cuthill-McKee implementation used is available in the cuSOLVER library [22], which is based on the SPARSPAK implementation described by George and Liu [26].

After the permutation is applied and the decomposition performed, the pattern obtained can be seen in Figure 5. It

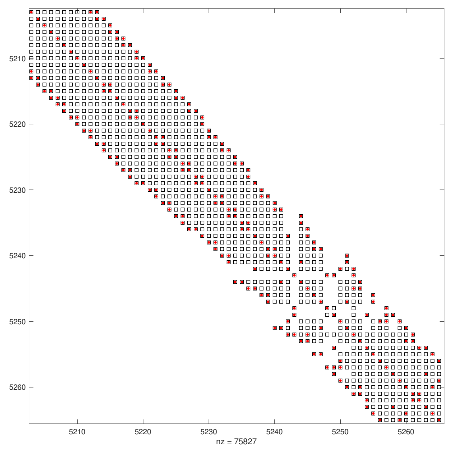


Fig. 5: Fill-in structure in band matrix — in red the original entries, in black the ones that appear during the decomposition.

$$A = \begin{bmatrix} a_{11} & * & a_{13} & * \\ * & a_{22} & * & a_{24} \\ a_{31} & * & a_{33} & * \\ * & a_{42} & * & a_{44} \end{bmatrix}$$

$$\begin{bmatrix} a_{11} & * & a_{31} & a_{22} & * & a_{42} & a_{33} & * & - & a_{44} & - & - \end{bmatrix}$$

Fig. 6: GPU band matrix representation — the hyphens represent the "ghost entries", which have no meaning with respect to the original matrix.

is a dense band structure, and for that reason a Band Storage-like format was used to represent the matrix. Although it is not the most efficient in terms of the ratio between the number of non-zeros and the number of entries allocated in memory, it allows more regular data access (which is harder to obtain in compressed formats, such as CSR). This format can also be seen as an implicit ELLPACK since all the elements within a column have sequential row indices (each column as an implicit row index offset). In this manner, less memory is required since the need to store the column indices disapears.

Figure 6 shows how the symmetric matrix A is converted into the format used on the GPU during the decomposition. It only uses the lower part of the original matrix, grouping the entries in a column-wise way, only using the elements from the main diagonal, down to the S^{th} row (bandwidth). Given the size of the matrix (N) and its bandwidth (S), the memory requirement by this storage type is in the order of $\mathcal{O}(S \times N)$. The indexing of an entry that is within the band is done by: `A[i][j] = data[i+S×j]`.

Recalling the description of the right-looking Cholesky decomposition, reducing the loops to the band allows us to discards unnecessary computation. The execution was divided into three steps, the panel factorization, followed by the respective submatrix — both are performed N times — and a last one that computes the square root of the diagonal. There is

Algorithm 5 Implicit synchronization right-looking Cholesky. Note that all functions called are kernels with their arguments for the respective launching within square brackets.

```
procedure CHOLESKYIMPLICITSYNC($A, S, N, TpB$)
    if explitic version then
        $aB = ((S \times (S-1)/2) + TpB - 1)/TpB$
    else
        $aB = ((S \times S) + TpB - 1)/TpB$
    $size = S \times N$
    $sB = (S + TpB - 1)/TpB$
    $nB = (N + TpB - 1)/TpB$
    if explitic version then
        $computeIndices[aB, TpB](I, S)$
    for $k = 1,$ to N do
        $factorize[sB, TpB](A, S, sizek)$
        if explitic version then
            $updateExplicit[aB, TpB](A, S, I, size, k)$
        else
            $updateImplicit[aB, TpB](A, S, size, k)$
    $sqrt[nB, TpB](A, S, size, N)$
```

a restriction imposed by all variants, a synchronization barrier between each of three prior steps. Restrictions of this nature become stronger when using GPUs since there are limited resources in terms of thread synchronization (only within each *thread block*). However, an implicit and primitive to perform the *barrier*, is launching each step in different kernels.

The column (panel) factorization, comprises S independent operations, the submatrix update $(S^2 - S)/2$, and the square root of the diagonal, N operations. Two slightly different versions of the update were implemented: in the *explicit indexing* version, the number of threads launched is close to the number of submatrix entries ($(S^2 - S)/2$), but comes with a cost of assigning each thread to the respective entry; On the *implitic indexing*, the number of threads required is more than double ($S \times S$), trying to overcome the indexing costs observed in the prior version. The launching process can be seen in Algorithm 5, where the dimensions of each grid are determined dynamically, fixing the number of threads per block (TpB). A represents the matrix.

1) Column factorization: The column factorization step consists of the division of all elements of a column by its diagonal entry. The kernel is simplified in Algorithm 6 and the threads organization is schematized in Figure 7. Note that it performes the redundant computation to avoid sequential code, requiring each thread to compute the square root of the diagonal entry before the division.

Algorithm 6 Column factorization.

```
procedure FACTORIZE($column, S, N, K$)
    $idx = blockIdx.x \times blockDim.x + threadIdx.x$
    $d = k \times S$
    if $idx < S$ and $idx > 0$ then
        $column[d{+}idx] = column[d{+}idx]/\sqrt{column[d]}$
```

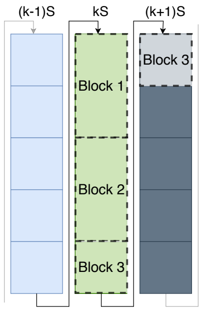


Fig. 7: Thread block organization during column factorization — the region is a previously factorized column, and the green is where modifications will occur.

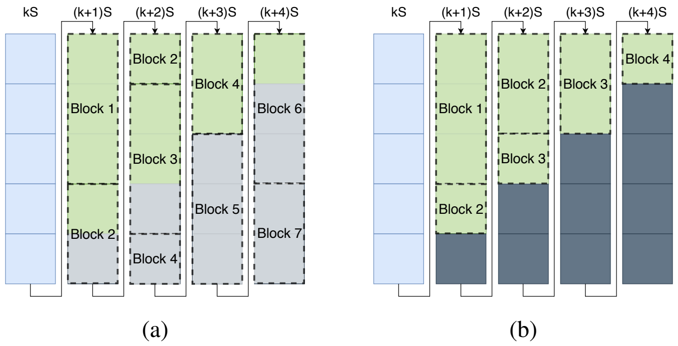


Fig. 8: Thread block organization using implicit (a) and explicit indexing (b).

2) Submatrix update: In the both version of the update kernel, the threads blocks are grouped in a column-wise manner over submatrix, and each updates one entry. On the implicit, all threads access sequential submatrix entries, but on the explicit indexing version, there are gaps on this sequentially. Figure 8a shows how the blocks are distributed among consecutive columns, from both version. Both version require two accesses to a given column (k) that are not sequential.

As the Algorithm 7 shows, in order to modify the i^{th} entry of the $k+j$ column, the thread also needs to access the $(i+j+1)^{th}$ and the $(j+1)^{th}$ element of the k^{th} column. This process implies that all the active threads will disorderly access the same memory region, resulting in higher latency. Note that both version have their own kernel implementation.

3) Diagonal square root: The redundant computations done during the column factorization step do not update the diagonal entries. For that purpose, the threads of this kernel access, and update the first entry of each column (diagonal element) to compute its final value. The accesses have a stride of S, destroying the possibility for coalesced memory access since they are not even sequential accesses. This kernel is not inside any loop and is performed just once, therefore is not a critical point for optimization. Its existence is a significant improvement compared to the alternative of performing a sequential operation on the column factorization kernel.

Algorithm 7 Update submatrix kernel simplification.

```
procedure $updateSubmatrix(A, S, indices, size, k)$
    $id = blockIdx.x \times blockDim.x + threadIdx.x$
    if explicit version then
        $id = indices[id]$
        $idxCondition = (tid \leq (S^2 - S)/2)$
    else
        $idxCondition = (i + j \geq S)$
    $i = id \bmod S$
    $j = id/S + 1$
    $ij = i + (j + k) \times S$
    $jk = j + k \times S$
    $ik = jk + i$
    if $ij \geq size$ or $idxCondition$ then
        return
    $A[ij] = A[ij] - A[ik] \times A[jk]$
```

B. Batched Conjugate Gradient

The iterative method implemented is the Preconditioned Conjugate Gradient, and performs batched iterations. Since on this method no fill-in is observed, the matrix remains sparse and with a low maximum number of elements per column or row (M). The pattern obtained is not modified with resource to any ordering algorithm and the matrix representation used was the ELLPACK, since it automatically structures the rows to have the same number of entries — such regularity is essential for when the GPU is used. This representation requires two auxiliary and smaller matrices to represent the original. The representation of these matrices on the GPU is done by grouping the rows of the originals into a respective array. The same format is used on the lower and upper part of the preconditioner matrix, keeping the main diagonal separately.

The Algorithm 4, contains several types of operations during an iteration that can be grouped in (with the respective asymptotic complexities, where M is the maximum number of nonzeros per row, and N the number of rows):

- Vectors sum and scaling — $\mathcal{O}(N)$;
- Dot product between vectors — $\mathcal{O}(N)$;
- Matrix-Vector multiplication — $\mathcal{O}(M \times N)$;
- System solve (preconditioner matrix) — $\mathcal{O}(M \times N)$;

The strategy used intends to parallelize each type of the operations.

1) Vectors sum and scaling: The operation $z = \alpha \times x + \beta \times y$ is the most general way to cover both vector scaling and sum. The variables z, x, and y are arrays. Both α and β are scalars. If $\beta = 0$ and $z = x$ then a scaling of α is being done on x. Any other combination allows us to sum (or subtract) two vectors and store the result in a third.

The parallelized strategy is the same for both types of operations. Consecutive and respective entries of each array are assigned to each block of threads to allow coalesced memory access to the three arrays involved. This can be visualized in Figure 9. To avoid launching too many blocks, if the dimension of the grid is lower than the size of each array,

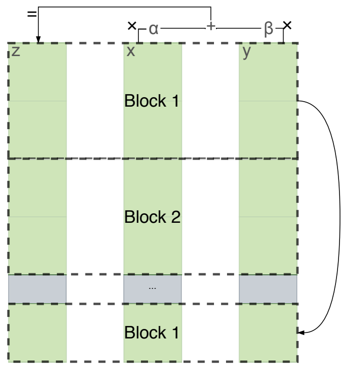


Fig. 9: Sum vector, thread organization strategy.

the blocks keep performing the operations over the remaining the entries.

2) Dot product: The internal product of two arrays can be decomposed into two operations: the product of the respective entries of both, followed by a sum reduction over the result. Instead of performing them separately, the reduction, can be modified such that it accumulates the product of two entries rather than a single one.

The reduction is a common and important data-parallel operation. It consists in applying a operator ($+$, $-$, $\times$, $/$, bit-wise operators) cumulatively over all elements of an array. Although it has the same asymptotic complexity as the vector sum, its parallelization becomes a bit more difficult and not so efficient.

The reduction is performed in the GPU following a hierarchical sequence, from the thread's partial reduction to the warp's partial reduction, up to the block's partial reduction. The idea is to perform the operations in a binomial tree-based approach. The final result, instead of the usual scalar, is an array with the size of the number of thread blocks used (typically very small compared with the size of the array). This happens due to the absence of shared memory between blocks, which is essential to allow this operation to as efficient as possible. The implementation used was taken from [27], where several versions are presented and compared.

Most of them are simple derivates from another by making successive optimizations concerning sequential memory access, avoiding divergent execution, and hiding latency

3) Matrix-Vector multiplication: The matrix-vector multiplication is the application of N internal products between each row of the matrix and the vector. The approach used, distributes groups of consecutive rows to each block of threads. Within the block, each thread accesses its row, and performs the respective dot product, accessing the vector as well. In this manner, each dot product is performed sequentially by the respective thread. Each row has only M non-zero element, therefore each thread will have to perform $M-1$ additions and M multiplications.

4) System solve: The approach used to solve the precontioner's system depends on the kind of preconditioner used. When using the diagonal of the matrix as the preconditioner, the parallelization strategy follows the same reasoning as the vector scaling. However when using the

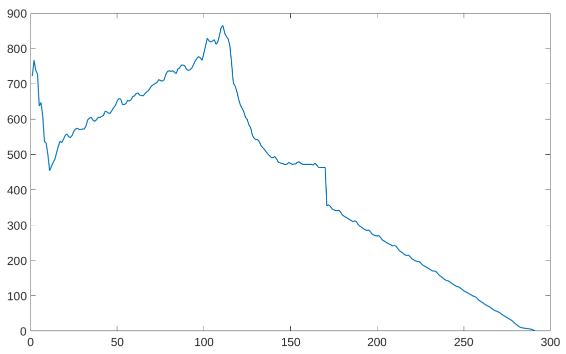


Fig. 10: The vertical axis shows the number of rows with the same depth, and the horizontal axis represent the respective value.

incomplete Cholesky decomposition preconditioner [28] with no fill-in, the solution process consists of the forward elimination followed by a backward substitution, since the matrices obtain by this process are triangular. To maximize the parallelism the order of the operations must be changed. This is called *level scheduling* and reorders the matrix (wavefront ordering), rearranging the dependencies. In this manner, both methods can determine in parallel the groups of unknowns. The implemented algorithm to find the wavefront ordering was based on the description from [28]. The methods iterate through the levels (groups of rows) obtain from the analysis and compute the respective unknowns in parallel. Usually the number of levels is much lower than the size of the matrix. Figure 10 shows how many unknowns can be determined in each level for the *ibmpg2* benchmark.

The Conjugate gradient method was implemented by assembling all the operations mentioned before following the structure of Algorithm 4. The CPU manages the various kernel launches, only checking the convergence of the method when a given batch of iterations was computed. This batch can have an adjustable size (even 1). In this manner, the only data transfer required between host and device, after a batch, is the residual error, allowing the host to decide if a new batch will be launched and if so, specifying its size.

The reason why this approach was implemented was to be able to compare with the one were the host also does some minor computations, such as finishing the reduction of the internal products and update the scalars α and β. Such implementation imposes stricter synchronization between both parts; on the other hand, the *batched* version is more flexible. However, the smaller computations (once done in the host) come with significant overhead.

The batched version of the method can be seen in Algorithm 8, where some kernel calls were simplified do make it more readable. There are many ways to implement this method. For instance, instead of launching the single thread grid to perform the scalar division, it can be replaced with redundant computations (passing each required operand to the kernels).

Algorithm 8 Batched PCG using GPU.

```
$q =$ MVproduct<<<$B, TpB$>>>($matrix, p$)
$r =$ aXpBYKernel<<<$B, TpB$>>>($1.0, b, -\alpha, q$)
$k = maxIteration = norm = initNorm = 0$
**while** $maxIteration \leq N$ and $norm/initNorm > Tolerance$ **do**
    $maxIteration = k +$ determineBatchSize($N, k$)
    **while** $k < maxiteration$ **do**
        $z =$ Solve($P, r$)          ▷ Decides the kernel to call
        $rho1 =$ DotProdKernel<<<B,TpB>>>($r, z$)
        **if** $k = 0$ **then**
            Copy<<<$B, TpB$>>>($p, z$)
        **else**
            $\beta =$ div<<<$1, 1$>>>($\rho1, \rho2$)
            $p =$ aXpBYKernel<<<$B, TpB$>>>($1, z, \beta, p$)
            Copy<<<$1, 1$>>>($\rho2, \rho1$)
        $q =$ MVproduct<<<$B, TpB$>>>($matrix, p$)
        $norm =$ DotProdKernel<<<$B, TpB$>>>($r, r$)
        $\alpha =$ DotProdKernel<<<$B, TpB$>>>($p, q$)
        $\alpha =$ div<<<$1, 1$>>>($\alpha, norm$)
        $x =$ aXpBYKernel<<<$B, TpB$>>>($1.0, x, \alpha, p$)
        $r =$ aXpBYKernel<<<$B, TpB$>>>($1.0, r, -\alpha, Ap$)
        $k = k + 1$
    Synchronize and copy $norm$ to the host
    $norm = \sqrt{norm}$
```

Name	Size	Bandwidth	NNZ	Density
ibmpg1	16327	72	75827	2.844534e-04
ibmpg2	126905	408	542895	3.370997e-05
ibmpg3	850626	1233	3651332	5.046311e-06
ibmpg4	952618	1722	4054056	4.467373e-06
ibmpg5	540220	582	2691958	9.224163e-06
ibmpg6	834252	843	4125504	5.927649e-06
thupg1	4974178	1914	21469972	8.677384e-07
thupg2	8988782	2631	39072262	4.835784e-07
thupg3	11777726	3005	51173098	3.689086e-07
thupg4	15208774	3457	66341492	2.868118e-07
thupg5	19230583	4042	85231241	2.304698e-07

TABLE I: Properties of benchmark matrices.

IV. Evaluation

The implementations developed, were applied over real problems (benchmarks), and for purposes of this work, a set of artificial matrices were generated form a regular grid assumption, to compare the performances when facing regularity or not. IBM made a set of benchmarks available, obtained from real designs, each with their own characteristics, resulting in several ranks of difficulties [1]. Later on, researchers from EDA Lab of Tsinghua University complemented the set with larger benchmarks, extended from a smaller original design [29]. The table I summarizes the properties of the matrices obtained (where the headings NNZ and Density are respectively, the number of nonzeros and its division by the total number of elements (Size^2)). Note that their bandwidth is computed after the Reverse Cuthill-McKee permutation is applied, the other properties don't change.

The generation of matrices obtained from regular grids was done using the `laplacian` method, available in the following repository: https://github.com/lobpcg/blopex. The generated set tries to fit the matrices obtained from IBM benchmarks, in terms of size and bandwidth (and fill-in by consequence). Their sizes and bandwidths are very close to the respective IBM matrix, although the number of nonzeros (NNZ) is slightly higher as can be seen in Table II.

Name	Size	Bandwidth	NNZ	Density
laplace1	16344	72	97012	3.631690e-04
laplace2	127100	410	834940	5.168496e-05
laplace3	850915	1235	5606679	7.743429e-06
laplace4	952266	1722	6338360	6.989729e-06
laplace5	540096	582	3413876	1.170323e-05
laplace6	834570	843	5277984	7.577789e-06

TABLE II: Properties from artificial matrices.

A. Setup

The implementations developed are in C and CUDA 10.1 programming interface. The hardware platform consists of a Linux server with two 6-Core Intel(R) Xeon(R) E5-2620 v2 CPUs, allowing Hyper-Threading (2 threads per core), running at 2.10 GHz. The system contains 32GB of RAM and 16 GB of swap. Two GPUs are available K40 and K20, although only the K40 was used. It contains 15 multiprocessors, each with 192 CUDA Cores, making a total of 2880 CUDA Cores. The maximum clock rate is 0.75 GHz. The warp size is the standard of 32 threads, with a limit of 2048 per multiprocessor, and 1024 per block. The amount of memory available is 11 GB of global memory, 65 KB of constant memory, and 49 KB of shared memory per block. The CUDA capacity is the same in both GPUs, 3.5. Not supporting single or multi-device cooperative kernel launching.

B. Direct solvers

The implemented solver (GPUBandSolver) is compared with CHOLMOD version 3.0.6 [17], a high-performance library for sparse Cholesky factorization, which, allows GPU acceleration. It is a hybrid solver since the GPU acceleration is optional.

The comparisons are subject to restrictions in terms of computations performed. As said before, the symbolic analysis is a crucial step in the whole process, helping to reduce memory and the number of computations required on the following steps; therefore, each method should operate on the matrix after Reverse Cuthill-McKee permutation is applied to equalize the comparisons. In this manner, the number of operations performed by each method round is of the same order since the fill-in pattern will be the the same in both cases.

The Table III, comprises the results of the implemented method and of the CHOLMOD, when running over the IBM benchmark set and the generated one. The larger (Tsinghua) was not used, since both methods fail when the Reverse Cuthill-McKee ordering is imposed, due to excess use of memory.

It can be seen that the developed method performance has a strong dependence on the bandwidth and size of the matrix. Even if regularity is given (generated set) or not (IBM set), the

Name	GPUBandSolver		CHOLMOD			
			CPU	CPU+GPU		
	Time	Mem.	Time	Time	Proportions	Mem.
ibmpg1	0.218	0.009	———	———	—————	0.012
ibmpg2	2.656	0.414	1.357	1.225	(80.4+19.6)%	0.341
ibmpg3	**79.64**	8.391	**57.3**	**33.075**	(61.6+31.4)%	5.772
ibmpg4	———	13.123	136.866	79.126	(57.0+**43.0**)%	9.253
ibmpg5	15.695	2.515	6.053	4.548	(76.4+23.6)%	1.478
ibmpg6	43.070	5.626	21.939	12.745	(65.9+34.1)%	3.628
laplace1	0.218	0.010	0.0325	0.031	(100+0)%	0.011
laplace2	2.676	0.418	2.193	1.509	(64.1+35.9)%	0.481
laplace3	**80.015**	8.414	**98.398**	**29.206**	(61.8+31.2)%	9.120
laplace4	———	13.126	196.800	48.727	(61.2+38.8)%	14.123
laplace5	16.034	2.519	16.635	8.399	(59.4+40.6)%	2.808
laplace6	42.621	5.635	48.432	17.586	(63.6+36.4)%	6.174

TABLE III: Factorization results of the developed solver (GPUBandSolver) and CHOLMOD 3.0.6 (with and without GPU acceleration). The time is in seconds, and memory in GB. The time measures in GPUBandSolver are done to the GPU since it is only GPU based. Memory and flops in CHOLMOD are not influenced by the use of GPU. The built-in method used to measure the time on CHOLMOD didn't work on the first benchmark. When GPUBandSolver uses the fourth benchmark, it requires more memory than the available one on the NVIDIA K40.

Name	CUSPARSE		Batched Conjugate Gradient					
					Diagonal		Inc. Cholesky	
	Time	Iter.	Time	Iter.	Time	Iter.	Time	Iter.
ibmpg1	2.2	1422	1.2	1412	1.11	690	1.72	149
ibmpg2	2.72	2896	1.82	2884	1.36	1209	4.83	334
ibmpg3	269.66	212987	318.21	213067	227.49	146455	791.77	22548
ibmpg4	13.26	7990	14.3	8010	3.8	1557	41.04	507
ibmpg5	8.75	7991	9.76	7967	2.95	1700	34.56	482
ibmpg6	12.97	9177	16.21	9075	4.74	2037	39.05	548

TABLE IV: The CUDA library implementation uses cuSPARSE, cuBLAS, and unified memory. The customized method in this table uses a batch size of 1, and it is presented in three versions; the first with no preconditioner and the other two using the diagonal and the incomplete Cholesky decomposition with no fill-in.

time and memory requirements round the same order. It was expected since the storage format doesn't compress data and therefore inherits this dependency, making, in both cases, the number of operations required nearly the same. However when the matrix is regular (the artificial set), CHOLMOD matrix format (CSR) cannot compress the as much zeros as before, and its CPU version performance becomes more close to the GPUBandSolver's. When the GPU is used on the CHOLMOD, the amount of computations it performed tends grow as the size of the benchmark grows. Always having significant speed up over GPUBandSolver (3.38 $\times$ on the *ibmpg6*).

C. Iterative solvers

The developed Batch Conjugate Gradient method is compared with an implementation made available by NVIDIA on the library from CUDA samples. This implementation uses cuBLAS, cuSPARSE, and unified memory. It suffered a small modification, to equalize the way convergence was tested among the methods so that the conclusions can be stronger.

The implemented method was tested on its three possibilities, with no preconditioner, using the Diagonal or incomplete Cholesky. The results of each can be seen in Table IV. It proves what was mentioned in [28], the incomplete Cholesky with no fill-in might not reduce the execution time. Although it reduced the number of iterations required (approximately by a factor of 10), each of them have a higher computational complexity. The use of Diagonal preconditioner, achieves a middle term result between the previous and the absence of preconditioning, in terms of iteration reduction. It always achieves a performance improvement on the IBM benchmark set.

Some optimizations that can still be done in the preconditioner system solving using the wavefront ordering to help reduce the weight of each iteration. Although, its nature is partially sequential, and when performed on the GPU instills a big penalty on the performance. An implementation that forfeits the batched iterations and brings the residual computations to the host might also achieve better efficiency, since the GPU instills a significant overhead, especially on these small-sized operations.

The artificial matrices have low condition number compared to the respective benchmark (*laplace1* has a condition number of 11 while *ibmpg1* has 1.05e+05), which is decisive for the solution with Conjugate Gradient. For that reason, the performance results have no practical meaning.

The increment of the batch size results in avoiding synchronization, which reduces execution time, but with this static size, it instills more computations than required, increasing runtime. It can be seen that the worst-case scenario is when using a size very close to the required number of iterations. The optimal size equals the required iteration to converge, which is unknown beforehand. The use of a small batch shows a slight improvement. Although, as the number of iterations increases it becomes more evident that the small-sized computations done on the GPU cause a higher impact on the performance, since the overheads for ”unjustified” computations become more significant. The batch iteration cannot offer gains to overcome those overheads.

V. Conclusion

The present work offered an interesting approach to solve the power grid analysis allowing us to verify that the GPU can in fact solve the computational demanding part of this problem, saving the CPU that processing. Nevertheless, using simple and naive approaches of direct and iterative methods, the potential efficiency improvements are not optimal since such approaches are not the best suited to take advantage of the GPU characteristics. For that reason, the results are not as satisfactory as the ones achieved by more complex and clever approached, tailored to the problem at hand, that while appearing to make it more difficult to use the GPU end up taking better advantage of those characteristics.

On one hand it is possible to improve on the limitations of direct methods by using reorderings to reduce fill-in which inherently reduces the number of computations required, as well as the excess use of memory. On the other hand, iterative methods, with more memory available, may accelerate their convergence through the use of preconditioners. However, its use adds a bottleneck to execution and, if not well-chosen and implemented, can be catastrophic, degrading performance.

The performance of both types of developed methods in the examples available is generically similar.

When reviewing the problem of performing the analysis on power grids, the use of direct methods should be focused around the use of fill-in reducing ordering — to perform a reduction of the computational complexity of the method to be used. It is very important to highlight that the direct method must consider beforehand the fill-in reducing process. The direct method used was supported in an implementation based on a right-looking variant of the Cholesky decomposition since all variants have approximately the same amount of floating-point operations, adding the fact that this one was seemingly the one that was more prone to parallelism. This implementation was also limited by the attempt to make use of the GPU as much as possible. It can be concluded that the use of GPU can, in fact, help the CPU, solve this type of problem, but due to its partially sequential nature, performance is degraded when trying to pass the whole process to the GPU. CHOLMOD is a great example of that, viewing the GPU as an auxiliary accelerator, maximizing the GPU utilization.

On the iterative methods, the use of a preconditioner proved useful, reducing the number of iterations. However care must be taken in order to ensure that the extra computations required to solve the preconditioned system, by implementing it efficiently on the GPU. In our implementation we relied on level scheduling to maximize the parallelism on the GPU when using the incomplete Cholesky decomposition. However, the reduction of iterations was not sufficient to overcome the computational step added. The preconditioner used must reduce even more the conditioning number of the resulting system.

The GPU is a very sensitive and complex device, becoming a bit difficult to perform optimizations to achieve better results. It takes time and practice to assimilate the programming concept, all the available resources, and the respective features and drawbacks. It is also recurrent the keep enhancing a given implementations using different mechanisms available (streams, shared memory, texture, etc.).

References

[1] S. R. Nassif, "Power grid analysis benchmarks," in *Proceedings of the 2008 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '08. Los Alamitos, CA, USA: IEEE Computer Society Press, 2008, pp. 376–381. [Online]. Available: http://dl.acm.org/citation.cfm?id=1356802.1356895

[2] J. C. Butcher, *Numerical Methods for Ordinary Differential Equations*, 2nd ed. J. Wiley, 2003. [Online]. Available: http://gen.lib.rus.ec/book/index.php?md5=8AD7183973B059FCFF031D7049FBFE04

[3] G. Allaire, K. Trabelsi, and S. Kaber, *Numerical Linear Algebra*, ser. Texts in Applied Mathematics. Springer New York, 2008. [Online]. Available: https://books.google.pt/books?id=PlUWqQzY-4EC

[4] J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*, Z. Bai, Ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2000.

[5] G. Strang, *Introduction to Linear Algebra*. Wellesley-Cambridge Press, 2016. [Online]. Available: https://books.google.pt/books?id=efbxjwEACAAJ

[6] T. A. Davis, S. Rajamanickam, and W. M. Sid-Lakhdar, "A survey of direct methods for sparse linear systems," *Acta Numerica*, vol. 25, p. 383–566, 2016.

[7] A. Haidar, A. Abdelfatah, S. Tomov, and J. Dongarra, "High-performance cholesky factorization for gpu-only execution," in *Proceedings of the General Purpose GPUs*, ser. GPGPU-10. New York, NY, USA: ACM, 2017, pp. 42–52. [Online]. Available: http://doi.acm.org/10.1145/3038228.3038237

[8] K. He, S. X. Tan, H. Wang, and G. Shi, "Gpu-accelerated parallel sparse lu factorization method for fast circuit analysis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 3, pp. 1140–1150, March 2016. [Online]. Available: doi.ieeecomputersociety.org/10.1109/TVLSI.2015.2421287

[9] G. H. Golub and C. F. Van Loan, *Matrix Computations (3rd Ed.)*. Baltimore, MD, USA: Johns Hopkins University Press, 1996.

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

[11] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 1st ed. Addison-Wesley Professional, 2010.

[12] NVIDIA Corporation, "NVIDIA CUDA C programming guide," https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf, Aug. 2019, version PG-02829-001_v10.1.

[13] N. Bell and M. Garl, "Efficient sparse matrix-vector multiplication on cuda," Tech. Rep., 2008.

[14] Z. Feng and P. Li, "Multigrid on gpu: Tackling power grid analysis on parallel simt platforms," in *2008 IEEE/ACM International Conference on Computer-Aided Design*, Nov 2008, pp. 647–654.

[15] S. Peng and S. X. D. Tan, "Glu3.0: Fast gpu-based parallel sparse lu factorization for circuit simulation," 2019.

[16] P. Amestoy, Enseeiht-Irit, T. A. Davis, and I. S. Duff, "Algorithm 837: Amd, an approximate minimum degree ordering algorithm," *ACM Trans. Math. Softw.*, vol. 30, pp. 381–388, 2004.

[17] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam, "Algorithm 887: Cholmod, supernodal sparse cholesky factorization and update/downdate," *ACM Trans. Math. Softw.*, vol. 35, no. 3, pp. 22:1–22:14, Oct. 2008. [Online]. Available: http://doi.acm.org/10.1145/1391989.1391995

[18] F. Pellegrini, J. Roman, and P. Amestoy, "Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering," *Concurrency: Practice and Experience*, vol. 12, no. 2-3, pp. 69–84, 2000.

[19] T. Davis, J. Gilbert, S. Larimore, and E. Ng, "Algorithm 836: Colamd, a column approximate minimum degree ordering algorithm." *ACM Trans. Math. Softw.*, vol. 30, pp. 377–380, 01 2004.

[20] NVIDIA Corporation, "cuBLAS," https://docs.nvidia.com/pdf/CUBLAS_Library.pdf, Aug 2019, version DU-06702-001_v10.1.

[21] ——, "cuSPARSE," https://docs.nvidia.com/pdf/CUSPARSE_Library.pdf, May 2019, version DU-06709-001_v10.1.

[22] ——, "cuSOLVER," https://docs.nvidia.com/pdf/CUSOLVER_Library.pdf, Aug 2019, version DU-06709-001_v10.1.

[23] H. Anzt, M. Baboulin, J. Dongarra, Y. Fournier, F. Hülsemann, A. Khabou, and Y. Wang, "Accelerating the conjugate gradient algorithm with gpus in cfd simulations," 07 2017, pp. 35–43.

[24] E. Phillips and M. Fatica, "A cuda implementation of the high performance conjugate gradient benchmark," vol. 8966, 04 2015, pp. 68–84.

[25] W.-H. Liu and A. Sherman, "Comparative analysis of the cuthill–mckee and the reverse cuthill–mckee ordering algorithms for sparse matrices," *Siam Journal on Numerical Analysis - SIAM J NUMER ANAL*, vol. 13, pp. 198–213, 04 1976.

[26] A. George and J. W. Liu, *Computer Solution of Large Sparse Positive Definite*. Prentice Hall Professional Technical Reference, 1981.

[27] M. Harris *et al.*, "Optimizing parallel reduction in cuda."

[28] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003.

[29] Z. L. Jianlei Yang and E. L. of Tsinghua University, "THU power grid benchmarks," 2012. [Online]. Available: http://tiger.cs.tsinghua.edu.cn/PGBench/