Balancing locality and concurrency: solving sparse triangular systems on GPUs

Andrea Picciau, Gordon E. Inggs, John Wickerson, Eric C. Kerrigan, George A. Constantinides
Department of Electrical and Electronic Engineering
Imperial College London, UK
{a.picciau13, g.inggs11, j.wickerson, e.kerrigan, g.constantinides}@imperial.ac.uk

***Abstract*—Many numerical optimisation problems rely on fast algorithms for solving sparse triangular systems of linear equations (STLs). To accelerate the solution of such equations, two types of approaches have been used: on GPUs, concurrency has been prioritised to the disadvantage of data locality, while on multi-core CPUs, data locality has been prioritised to the disadvantage of concurrency.**

In this paper, we discuss the interaction between data locality and concurrency in the solution of STLs on GPUs, and we present a new algorithm that balances both. We demonstrate empirically that, subject to there being enough concurrency available in the input matrix, our algorithm outperforms Nvidia's concurrency-prioritising CUSPARSE algorithm for GPUs. Experimental results show a maximum speedup of 5.8-fold.

Our solution algorithm, which we have implemented in OpenCL, requires a pre-processing phase that partitions the graph associated with the input matrix into sub-graphs, whose data can be stored in low-latency local memories. This preliminary analysis phase is expensive, but because it depends only on the input matrix, its cost can be amortised when solving for many different right-hand sides.

I. Introduction

A sparse triangular system of linear equations (STL) is a system of equations of the form

$$L\boldsymbol{x} = \boldsymbol{b}, \qquad (1)$$

where $\boldsymbol{x}$ is the result vector and $\boldsymbol{b}$ is the right-hand side vector, both in $\mathbb{R}^n$, and L is a lower or upper triangular matrix in $\mathbb{R}^{n\times n}$ with a significant number of zeroes.

Many algorithms such as those used in managing smart buildings [1] and those used in data mining for medical research [2], demand fast solutions to STLs. Fast solving is particularly important for iterative optimisation algorithms such as the preconditioned gradient method [3], in which it may be necessary to solve (1) several thousand times with different values of $\boldsymbol{b}$.

Sparse data structures have led to two opposing ways of partitioning data to parallelise the solution of STLs: fine-grain partitioning and coarse-grain partitioning.

An example of fine-grain partitioning is provided by Nvidia's CUSPARSE library [4], which solves STLs using the CUDA programming language for GPUs. The approach assigns each row of the matrix L to a work-item (also called a thread in CUDA). By an analysis of the dependencies between rows, CUSPARSE partitions the rows into several *levels* (using what is called the *level-set* algorithm). It determines an execution schedule that ensures that the levels are executed in a sequence respecting the dependencies, but allows all of the rows in the same level to execute in parallel. CUSPARSE performs its partitioning without regard for the positions of the rows in L, only their dependencies, and so levels may contain rows that are far apart in L. As such, the CUSPARSE approach maximises *concurrency*, but can exhibit poor *data locality*.

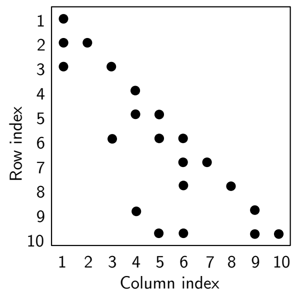
Coarse-grain partitioning approaches, such as Mayer's algorithms for multi-core CPUs [5], work by dividing L into large blocks, with each block containing roughly the same number of non-zero entries. A dependency analysis between the blocks allows certain blocks to be processed in parallel by different CPU cores, but within each individual block, matrix entries are processed sequentially. Because each CPU core accesses only the memory locations associated with one block at a time, the accesses can be coalesced. As such, Mayer's approach exhibits good *data locality* but may not take full advantage of the available *concurrency*.

In this paper, we show that, when solving STLs on GPUs, we can obtain significantly better performance than CUSPARSE by incorporating elements of coarse-grain partitioning. Our approach matches the structure of the input matrix to the structure of the GPU. This is achieved by partitioning the L matrix in (1) into blocks that fit into the GPU's local memory.

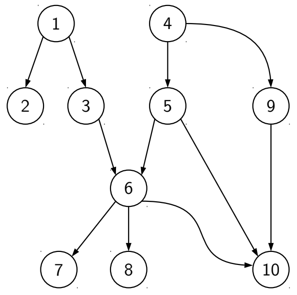
The partitioning is carried out during a preliminary analysis phase. Then, to solve the STL, each sub-graph is associated with a work-group and its data is stored in local memories (also called shared memories in CUDA). The analysis phase finds opportunities for both concurrency and data locality, and partitions the graph in a way that balances the two. The analysis phase is time-consuming, but its result is valid for any matrix with the same sparsity pattern and for any right-hand side vector, so if the solution phase is to be carried out many times, its cost can be amortised.

We compare our algorithm against CUSPARSE, using both artificial matrices with specific properties that we generated automatically and non-artificial test cases from the Florida matrix collection [6]. Our experimental results show that, our algorithm, which is implemented in OpenCL, outperforms CUSPARSE in 71 % of non-artificial test cases and is up to 5.8 times faster than the same library, which is optimised and provided by the GPU vendor.

In summary, we make the following contributions:



(a) *Sparsity pattern of the input matrix.*



(b) *Graph associated with the input matrix.*

Figure 1: An example sparse lower triangular matrix with 10 rows and 10 columns and the graph associated with it. In the representation of the sparsity pattern, the non-zero entries are shown with dots.

- we propose an algorithm for balancing the data locality and concurrent execution when solving STLs;
- we implement this balanced algorithm on GPUs, using OpenCL;
- we evaluate our algorithm against the state-of-the-art CUSPARSE algorithm [4].

II. Background

The properties of sparse data structures have been studied by associating the L matrix in (1) with a graph. In particular, in this paper we discuss lower triangular matrices with all the values on the diagonal equal to one, and $\boldsymbol{b}$ and $\boldsymbol{x}$ vectors stored using dense data structures. Nonetheless, our approach can be generalised to any triangular matrices. Within the framework of graph theory, the operation of the algorithms to solve STLs can be related to the traversal of the associated graph [7]. How this graph traversal is best carried out depends on the properties of the graph.

Given a lower triangular matrix $L \in \mathbb{R}^{n\times n}$, the graph G associated with it is $G := (\mathcal{V}, \mathcal{E})$. The set of vertices $\mathcal{V} := \{1, 2, \ldots, n\}$ has as many elements as the number of rows and columns in the matrix L. A non-zero entry at row i and column $j \neq i$ in the matrix is represented by a directed edge (j, i) from vertex j to vertex i in the graph.

If a graph is composed of a number of parts that are not connected to each other, those parts are called weakly connected components, or *disjoint components*.

In the context of STLs, an edge (j, i) represents the direct dependency of the variable x_i on the variable x_j. Let us consider the STL in (1) and let G be the graph associated with the matrix L. The system of equations can be solved by forward substitution [7]. For example, in case L has a sparsity pattern like that in Figure 1, we can write the following equation for x_6:

$$l_{6\,6}x_6 = b_6 - l_{6\,3}x_3 - l_{6\,5}x_5,$$

which shows that the variable x_6 depends on the variables x_3 and x_5. For the graph in Figure 1, we can see that there are in fact two edges incident to vertex 6, which start from vertex 3 and vertex 5 respectively.

III. Related work

The solution of STLs has been examined both in the context of MIMD architectures, such as multi-core CPUs and HPC clusters, and SIMD architectures, such as GPUs.

Mayer's algorithms [5], which are specific for multi-core CPUs, require the execution of the threads in compute units to be synchronised. In particular, if rows and columns are divided into s_M parts, the execution of the threads is stopped $2s_\mathrm{M} - 1$ times. With this technique, the computation carried out in a CPU core is independent from that in another CPU core. Although this technique is little applicable to GPUs, because it does not exploit the concurrency implicit in the structure of the graph, the results show that by partitioning the matrix into blocks one can make better use of cache memories.

In the context of GPUs, the level-set algorithm [4], which is implemented in the Nvidia CUSPARSE library, is carried out in two phases: a preliminary analysis phase and a solve phase. The analysis phase associates vertices in the matrix graph with levels. So, if rows and columns are divided into s_C parts, the execution of the threads is stopped at most s_C times. The number of levels and vertices per level is only determined by the sparsity pattern of the matrix. For example, if the graph is a chain of vertices, each connected to the next with an edge, there will be only one vertex per level. To overcome this limitation, a technique for transforming the matrix was introduced [8]. This technique increases the number of vertices per level and reduces the number of levels needed. However, it is not guaranteed that the transformation can always increase the amount of concurrency that can be extracted from the matrix.

On GPUs, it was observed that the synchronisation of threads by stopping their execution in all of the compute units is extremely time-consuming [4], [8], [9], and that sparse linear algebra algorithms on GPUs tend to be bandwidth-bound, so the use of local memory to save bandwidth is necessary [9]. Also, while the design aim of GPU architectures is to hide memory latency with parallelism, not all applications are concurrent enough to achieve this aim efficiently [10]. Moreover, in the context of multi-core CPUs and cluster computers, it was shown that matching the structure of the compute to that of the graph improves the scalability of algorithms [11].

In this work, we propose a solution technique for STLs in two phases: the analysis phase, which is illustrated in Section V, and a solve phase, which is illustrated in Secion IV. During the solve phase, numerical operations are carried out according to the output of the analysis phase. The aim of the analysis phase is to maximise the performance of the solve phase.

IV. The solve phase

The solve phase computes the solution to an STL by putting into effect the information given by the analysis phase. In Section IV-A we describe the inputs to our solution algorithm, and the required properties of a partition of the G graph. Then, in section IV-B we illustrate our solution algorithm, while in Section IV-C we derive a performance model that characterises the execution time of the solve phase.

A. Inputs to the solution algorithm

The analysis phase outputs a partition of the set of vertices $\mathcal{V}$ into s disjoint subsets $\mathcal{V}_1, \mathcal{V}_2, \dots, \mathcal{V}_s$, such that

$$\mathcal{V} = \mathcal{V}_0 \cup \mathcal{V}_1 \cup \mathcal{V}_2 \cup \cdots \cup \mathcal{V}_s ,$$

where $\mathcal{V}_0$ is the sub-set of vertices in the G graph with no entering or exiting edges. Given a partition of the input graph, we say that an edge (j, i) is an *internal* edge if both vertex i and vertex j are in the same set $\mathcal{V}_k$ with with $k = 1, 2, \dots, s$. Otherwise, it is an *external* edge.

The idea behind the solution algorithm is to divide the traversal of the graph into *sub-graphs*, each one associated with one of the $\mathcal{V}_k$ sets. For each sub-graph, if memory accesses are suitably organised, the traversal of an internal edge will show better data locality than that of an external edge. However, for the traversal to be carried out correctly, the partition must be feasible according to the following definition.

Definition (Feasible partition)**.** A partition of the G graph is feasible if both following conditions are met.

1) If an edge goes from a vertex in $\mathcal{V}_k$ to a vertex in $\mathcal{V}_p$, then $p \geq k$.
2) The number of vertices in the set $\mathcal{V}_k$, n_k, must not be greater than a positive integer n_{max}.

In particular, Condition 1 implies that there is no cyclical dependency between variables associated with vertices in different $\mathcal{V}_k$ sets. Condition 2 implies that the number of vertices in each sub-graph is bounded. If data is represented in the single precision floating point format, the upper bound of the number of vertices in $\mathcal{V}_k$, is equal to

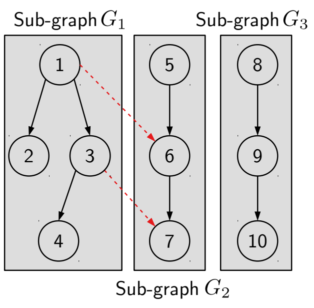
$$n_{\text{max}} := \left\lfloor \frac{M_{\text{cap}}}{4} \right\rfloor , \qquad (2)$$

where M_{cap} is the capacity of the local memory of a compute unit in bytes.

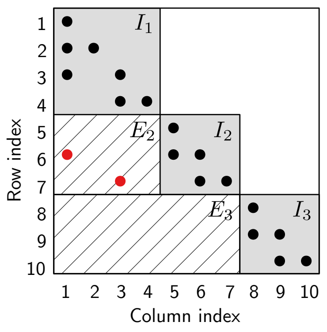
If the partition is feasible, it is possible to permute rows and columns of the matrix so to have indices in the same set $\mathcal{V}_k$ close together, while keeping a triangular structure. This permutation $Q \in \mathbb{R}^{n \times n}$ can be carried out by reordering the vertices of the original graph by sub-graph index. For example, the new indices of the rows and columns in the set $\mathcal{V}_1$ will be smaller than those of the rows and columns in $\mathcal{V}_2$. A lookup table can be used to keep track of the change in the indices.

By applying the permutation Q to to (1), we obtain

$$\underbrace{\begin{bmatrix} I_0 & & & & & \\ & I_1 & & & 0 & \\ & E_2 & I_2 & & & \\ 0 & \multicolumn{2}{c}{E_3} & I_3 & & \\ & \multicolumn{3}{c}{\vdots} & \ddots & \\ & \multicolumn{4}{c}{E_s} & I_s \end{bmatrix}}_{L'} \underbrace{\begin{bmatrix} \boldsymbol{x_0'} \\ \boldsymbol{x_1'} \\ \boldsymbol{x_2'} \\ \boldsymbol{x_3'} \\ \vdots \\ \boldsymbol{x_s'} \end{bmatrix}}_{\boldsymbol{x'}} = \underbrace{\begin{bmatrix} \boldsymbol{b_0'} \\ \boldsymbol{b_1'} \\ \boldsymbol{b_2'} \\ \boldsymbol{b_3'} \\ \vdots \\ \boldsymbol{b_s'} \end{bmatrix}}_{\boldsymbol{b'}} \qquad (3)$$



(a) *An example partitioned graph.*



(b) *The block form corresponding to the example graph.*

Figure 2: An example graph, partitioned into three sub-graphs, and the sparsity pattern of the corresponding matrix with the block form in (3). In the representation of the sparsity pattern, the non-zero entries are shown with dots, while in the representation of the graph external edges are represented with dashed lines.

where $L' := Q^T L Q$, $\boldsymbol{x'} := Q^T \boldsymbol{x}$, and $\boldsymbol{b'} := Q^T \boldsymbol{b}$. In (3), the blocks I_k and E_k are associated with $\mathcal{V}_k$. As shown in Figure 2, the elements in block E_k are associated with external edges, while the elements below the diagonal in block I_k are associated with internal edges. The vectors $\boldsymbol{x_k'}$ and $\boldsymbol{b_k'}$ are defined in $\mathbb{R}^{n_k}$, where n_k is the number of vertices in $\mathcal{V}_k$. Also, I_k is a n_k-by-n_k lower triangular real matrix, while matrix E_k is a real matrix with n_k rows and $\sum_{p=1}^{k-1} n_p$ columns. By construction, as required by Condition 2, n_k is always smaller or equal to n_{max}.

Equation (3) can be written in terms of $\boldsymbol{x_k'}$, which gives

$$E_k \begin{bmatrix} \boldsymbol{x_1'} \\ \boldsymbol{x_2'} \\ \vdots \\ \boldsymbol{x_{k-1}'} \end{bmatrix} + I_k \boldsymbol{x_k'} = \boldsymbol{b_k'} \qquad \text{for } k = 1, 2, \dots, s \, , \qquad (4)$$

where $E_1 = 0$.

B. Solution algorithm

Our solution algorithm computes the solution of s equations of the form in (4). Each equation is associated with a set $\mathcal{V}_k$ and is carried out on the GPU by a separate work-group. Although one can solve the s equations in sequence, from $k = 1$ to $k = s$, some of the equations may be independent since some sub-graphs may not have external edges that connect them. For example, in Figure 2, sub-graph G_1 and sub-graph G_3 are not connected, so their traversal can be carried out in parallel. In general, we can partition the set of sub-graphs into n_{levels} sub-sets of independent sub-graphs

$$\mathcal{L}_1 \cup \mathcal{L}_2 \cup \cdots \cup \mathcal{L}_{n_{\text{levels}}} ,$$

where the set $\mathcal{L}_p$ is called the sub-graph level of index p, for $p = 1, 2, \dots, n_{\text{levels}}$.

Our solution technique for (4) is outlined in Algorithm 1. A GPU kernel is executed n_{levels} times, each time for a different

```
Input: $\boldsymbol{b}'$, $\mathcal{L}_1$, $\mathcal{L}_2$, ..., $\mathcal{L}_{n_{\text{levels}}}$, $\mathcal{T}_{kq}$ and $\mathcal{T}'_{kq}$   $\forall q,\ k = 1, 2, \ldots, s$
Output: $\boldsymbol{x}'$
 1: $\boldsymbol{x}' \leftarrow \boldsymbol{b}'$
 2: for $p \leftarrow 1, 2, \ldots, n_{\text{levels}}$
 3:     for all $g \in \mathcal{L}_p$ do in parallel
 4:         $\boldsymbol{x}_{\text{loc}} \leftarrow \boldsymbol{x}'_g$
 5:         for $q \leftarrow 1, 2, \ldots, n'_{\text{slots},g}$
 6:             for all $(j, i) \in \mathcal{T}'_{gq}$ do in parallel          ▷ SIMD
 7:                 $x_{\text{loc},i} \leftarrow x_{\text{loc},i} - l_{ij}\, x_j$
 8:         for $q \leftarrow 1, 2, \ldots, n_{\text{slots},g}$
 9:             for all $(j, i) \in \mathcal{T}_{gq}$ do in parallel           ▷ SIMD
10:                 $x_{\text{loc},i} \leftarrow x_{\text{loc},i} - l_{ij}\, x_{\text{loc},j}$
11:         $\boldsymbol{x}'_g \leftarrow \boldsymbol{x}_{\text{loc}}$
```

Algorithm 1: Solution of a lower triangular sparse linear system. The vector $\boldsymbol{x}_{\text{loc}}$ is stored in local memory, and it is defined in $\mathbb{R}^{n_k}$, where n_k is the number of vertices in $\mathcal{V}_k$. In the pseudocode, $\mathcal{L}_p$, with $p = 1, 2, \ldots, n_{\text{levels}}$ are the sub-graph levels. Also, $\mathcal{T}_{gq}$ and $\mathcal{T}'_{gq}$ are the q-th time slot of internal and external edges respectively, in sub-graph G_g.

sub-graph level, in sequence. At each kernel execution, a work-group is associated with a sub-graph. In the algorithm, a vector $\boldsymbol{x}_{\text{loc}} \in \mathbb{R}^{n_k}$ is used, its data stored in local memory. The values of $\boldsymbol{x}_{\text{loc}}$ are initialised with $\boldsymbol{b}'_k$, then external and internal edges of sub-graph G_k are processed. The computation in the two cases is of the same type: a sequence of multiply-and-add operations specified by the schedule obtained during the analysis phase, which is organised in time slots. For sub-graph G_k, numerical operations associated with external edges are distributed over $n'_{\text{slots},k}$ time slots, while numerical operations associated with internal edges are distributed over $n_{\text{slots},k}$ time slots. In this paper, we represent the q-th time slot in sub-graph G_k with $\mathcal{T}_{kq}$ when referring to internal edges, and $\mathcal{T}'_{kq}$ when referring to external edges. The difference between the two types of numerical operations is in the memory location accessed. This is because, when processing internal edges, most memory accesses are in local memory, which has a lower latency compared to global memory. At the end of the algorithm, $\boldsymbol{x}_{\text{loc}}$ is copied to $\boldsymbol{x}'_k$. The worst-case computational complexity of the algorithm is $O(n_{\text{edges}})$, where n_{edges} is the number of edges in the graph.

C. Performance model

In this subsection, we derive a performance model for Algorithm 1 that we use to work out a technique to partition the G graph. The purpose of the performance model is not to predict accurately the execution time, but to characterise the execution time in terms of graph attributes.

In Algorithm 1, at each sub-graph level, some numerical operations that are associated with either internal or external edges are carried out. These numerical operations are shown at line 7 and line 10 in Algorithm 1, where l_{ij} is the entry of matrix L in row i and column j. The time to excute a numerical operation depends on the location of each of the operands. If (j, i) is an internal edge, then x_i and x_j are both in local memory, their values can be accessed with relatively low latency. On the other hand, if (j, i) is an external edge, x_j is in global memory, while x_i is in local memory. The access latency of the value x_j depends on whether this value is in the cache or not but, on average, this latency is greater than that of data in local memory. Moreover, cache hits when accessing the location of l_{ij} depend on how the matrix is stored.

Let τ be the average time to carry out all numerical operations in a time slot associated with internal edges. Also, τ can be measured as the average time to execute an iteration at line 8 in Algorithm 1. Similarly, let τ' be the average time to carry out all numerical operations associated with external edges. Moreover, let α be the average time needed to synchronise the execution of the algorithm at the end of each sub-graph level. Then, a performance model for Algorithm 1 is

$$t_{\text{solve}} \approx n_{\text{levels}}\alpha + \sum_{p=1}^{n_{\text{levels}}} \max_{g \in \mathcal{L}_p} \left\{ \tau' n'_{\text{slots},g} + \tau n_{\text{slots},g} \right\}, \qquad (5)$$

where $n_{\text{slots},g}$ and $n'_{\text{slots},g}$ are the number of external and internal time slots respectively, and n_{levels} is the number of sub-graph levels.

In this performance model, we approximated the execution time of each iteration at lines 5 and 8 with τ' and τ respectively. However, the actual execution time varies depending on sub-graph and time-slot index. Also, we neglected the dependency of t_{solve} on the number of compute units in the GPU. This requires the number of sub-graphs in each sub-graph level to be not much greater than the number of compute units. Although this hypothesis does not hold in the general case, we observed it to be reasonable in most practical cases. Furthermore, in (5) we neglected the execution time of the data transfer from global to local memory and vice versa, which is executed in lines 4 and 11. The hypothesis in this case is that there is enough compute per sub-graph to amortise the overhead of the memory copy.

V. The Analysis Phase

The aim of the analysis phase is to find a partition of the G graph that gives the smallest possible t_{solve} in (5). To achieve this, the analysis first tries to minimise the number of sub-graph levels n_{levels} while it guarantees feasibility according to the definition in Section IV-A. The analysis phase pre-processes the graph G associated with the matrix L in (1), it partitions G into sub-graphs, and returns a schedule of numerical operations to be executed during the solve phase. The analysis phase is divided into two stages: the partitioning stage and the scheduling stage.

A. Partitioning stage

The partitioning stage partitions the G graph into s sub-graphs. The value of s is defined as follows:

$$s := \left\lceil \frac{n}{n_{\text{max}}} \right\rceil,$$

where n is the number of vertices in the G graph, and n_{max} is that defined in (2). The aim of the partitioning is to obtain a small value of n_{levels} in (5). Because the number of sub-graphs

s is fixed, the technique we use tries to put as many sub-graphs as possible in the first set $\mathcal{L}_1$, and making sure that the partition is feasible at the end, according to the definition in Section IV. The partitioning stage is divided into three steps.

The first step of the partitioning stage is to partition the G graph into into its n_{comp} disjoint components. This can be done using Tarjan's algorithm [12], which has complexity $O(n + n_{\text{edges}})$, where n is the number of vertices and n_{edges} is the number of edges in the G graph.

The second step of the partitioning stage is to merge together disjoint components with few vertices using a heuristic algorithm. First, all disjoint components are sorted with respect to their number of vertices, from the smallest to the largest. Then, starting from the smallest, disjoint components are merged together, such that the number of vertices of each merged component is never greater than $n_{\max}$. After this step, the matrix L in (1) is permuted into a block diagonal form similar to that in (3):

$$L'' = P^T L P = \begin{bmatrix} \boxed{L''_0} & & & & & \\ & \boxed{L''_1} & & & 0 & \\ & & \boxed{L''_2} & & & \\ & & & \boxed{L''_3} & & \\ & 0 & & & \ddots & \\ & & & & & \boxed{L''_{n_{\text{comp}}}} \end{bmatrix} \quad (6)$$

where L''_k with $k = 0, 1, \ldots, n_{\text{comp}}$ are lower triangular matrix blocks. Each block is associated with one of the resulting components of the graph G. In particular, L''_0 is a diagonal block, and it is associated with the vertices of the graph G that have neither entering nor exiting edges. At the end of the merging, all conditions for the feasibility of the partition are met except possibly Condition 2, so further processing of the components is needed. Because of the block diagonal structure of L'', the processing of each resulting component is independent from the others, therefore the third step of the partitioning stage can be carried out in parallel on a multi-core CPU.

The third step of the partitioning stage is a heuristic that partitions those disjoint components that have a number of vertices greater than $n_{\max}$ into sub-graphs. If n_c is the number of vertices in the graph component C, this component is to be split into

$$n_{\text{split}} := \left\lceil \frac{n_q}{n_{\max}} \right\rceil$$

sub-graphs.

The basic operation of the partitioning heuristic is to associate a given vertex with a sub-graph. This is done by making sure that the partition is kept feasible. The procedure is illustrated in Algorithm 2, where v is the index of the vertex, and p a starting sub-graph index. First, it is checked that the inequality $n_p < n_{\max}$ holds. If this is true, associating vertex v with sub-graph G_p does not violate Condition 2, otherwise, the sub-graph index is incremented and the condition is checked again. If associating vertex v with any sub-graph violates Condition 2, then the procedure terminates with an error.

```
function ASSOCIATE_SUBG($v$, $p$)
    $k \leftarrow p$
    while $n_k \geq n_{\max}$
        if $k < n_{\text{split}}$
            $k \leftarrow k + 1$
        else return error
    $\mathcal{V}_k \leftarrow \mathcal{V}_k \cup \{v\}$
    return success
```

Algorithm 2: Function that associates a vertex v to a sub-graph. The function looks for a sub-graph whose index is between p and n_{split}, and that contains less than $n_{\max}$ vertices. If there is no such sub-graph, the function terminates with an error.

The partitioning heuristic is illustrated in Algorithm 3. The heuristic uses two containers of vertices: V_{next} and V_{cur}, on which it is possible to execute three basic operations, which are the addition of an element (*put*), the sorting according to a criterion κ (*sort*), and counting the number of elements (*numel*).

In line 3 of Algorithm 3, V_{next} is initialised with the root vertices of the component to partition, which are those vertices that do not have any parent but do have children. Initially, the root vertices are distributed across n_{init} initial sub-graphs, where n_{init} is that computed in line 4. The root vertices are associated with these initial sub-graphs according a sorting criterion κ.

We execute the partitioning heuristic with different criteria κ until a feasible partition is found. Initially, the criterion is descending order of out-degree, whose aim is to have vertices with a high out-degree in the same sub-graph of their children whenever possible. However, If a feasible partition is not found immediately, it is because the execution of the *associate_subg* routine was terminated with an error, which can occur either in line 10 or line 21. When an error occurs, n_{init} is halved and the execution of the heuristic is started from the beginning, but if n_{init} becomes equal to zero, then the whole heuristic terminates with an error. Whenever this happens, we repeat the execution of the heuristic and change κ: from descending order of out-degree to ascending order of out-degree, and then to ascending order of vertex index. The worst-case computational complexity of each iteration of the partitioning algorithm is $O(n + n_{\text{edges}})$, where n is the total number of vertices in the graph and n_{edges} is the number of edges.

At each iteration of the partitioning heuristic, in line 19, a vertex k is taken from the container V_{cur}, and it is associated with the sub-graph with the highest possible index. This is described in line 21. If the *associate_subg* routine is succesful, the children of the vertex k are considered, and those whose parents are all assigned to sub-graphs are put in the container V_{next}. When all the vertices in V_{cur} have been processed, the container V_{next} is sorted and its content are moved to V_{cur}. Then, as shown in line 18, V_{next} is emptied before the following iteration.

At the end of the partitioning stage, the permutation Q is obtained, so that the L matrix can be permuted in the block form illustrated in (3). A number of vertices smaller or equal to

```
Input: $C$, $n_{\text{split}}$, $n_{\max}$, $\kappa$
Output: $\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_{n_{\text{split}}}$
 1: $\mathcal{V}_k \leftarrow \emptyset \quad \forall k = 1, 2, \ldots, n_{\text{split}}$
 2: $V_{\text{next}} \leftarrow \not{V}$
 3: $V_{\text{next}} \leftarrow \text{put}(r, V_{\text{next}}) \quad \forall r \in \mathcal{W} \mid \nexists (j, r) \in \mathcal{Y}, j \in \mathcal{W}$
 4: $n_{\text{init}} \leftarrow \min\{n_{\text{split}}, \text{numel}(V_{\text{next}})\}$
 5: if $n_{\text{init}} > 0$
 6:     $V_{\text{next}} \leftarrow \text{sort}(V_{\text{next}}, \kappa)$
 7:     $p \leftarrow 0$
 8:     for all $k \in V_{\text{next}}$
 9:         if $p > n_{\text{init}}$ then $p \leftarrow 1$, else $p \leftarrow p + 1$
10:         $e \leftarrow \text{ASSOCIATE\_SUBG}(k, p)$
11:         if $e =$ error
12:             $n_{\text{init}} \leftarrow \lfloor n_{\text{init}}/2 \rfloor$ and go to line 5
13:         else for all $v \in \mathcal{W} \mid \exists (k, v) \in \mathcal{Y}$
14:             if $\nexists (j, v) \in \mathcal{Y} \mid j \notin \mathcal{V}_k \quad \forall k = 1, 2, \ldots, n_{\text{split}}$
15:                 $V_{\text{next}} \leftarrow \text{put}(v, V_{\text{next}})$
16:     while $\text{numel}(V_{\text{next}}) \neq 0$
17:         $V_{\text{cur}} \leftarrow \text{sort}(V_{\text{next}}, \kappa)$
18:         $V_{\text{next}} \leftarrow \not{V}$
19:         for all $k \in V_{\text{cur}}$
20:             $p \leftarrow \max\{q \in \mathbb{N} \mid \exists (j, k) \in \mathcal{Y}, j \in \mathcal{W}_q\}$
21:             $e \leftarrow \text{ASSOCIATE\_SUBG}(k, p)$
22:             if $e =$ error
23:                 $n_{\text{init}} \leftarrow \lfloor n_{\text{init}}/2 \rfloor$ and go to line 5
24:             else for all $v \in \mathcal{W} \mid \exists (k, v) \in \mathcal{Y}$
25:                 if $\nexists (j, v) \in \mathcal{Y} \mid j \notin \mathcal{V}_k \quad \forall k = 1, 2, \ldots, n_{\text{split}}$
26:                     $V_{\text{next}} \leftarrow \text{put}(v, V_{\text{next}})$
27:     if $\exists v \in \mathcal{W} \mid v \notin \mathcal{V}_k \quad \forall k = 1, 2, \ldots, n_{\text{split}}$
28:         $n_{\text{init}} \leftarrow \lfloor n_{\text{init}}/2 \rfloor$ and go to line 5
29:     else terminate with success
30: else terminate with error
```

Algorithm 3: Heuristic algorithm that partitions a graph component $C := (\mathcal{W}, \mathcal{Y})$, into n_{split} sub-graphs $G_1, G_2, \ldots, G_{n_{\text{split}}}$. Each sub-graph has at most $n_{\max}$ vertices. The algorithm uses the containers of vertices V_{cur} and V_{next}, on which the *put*, *sort* and *numel* operations are defined. In the pseudocode, $\not{V}$ represents the empty container and κ is the sorting criterion.

$n_{\max}$ is associated with each sub-graph. Also, the sequence of the equations in (4) is known, together with the sub-graph levels $\mathcal{L}_p$. In fact, one can build a graph where a vertex represents a sub-graph and an edge represents one or more external edges in the graph G following the partitioning, as shown in Figure 3. In this example, the edge between sub-graph G_1 and sub-graph G_3 shows that there is at least one edge between vertices associated with sub-graph G_1 and vertices associated with sub-graph G_3. Also, sub-graphs 1, 2, 4 and 6 belong to the set $\mathcal{L}_1$ because they do not depend on any other sub-graph. Similarly, sub-graphs 3 and 5 belong to the set $\mathcal{L}_2$ because they are connected to sub-graphs in the set $\mathcal{L}_2$ and they do not depend on each other, therefore in this example $n_{\text{levels}} = 2$. Since Condition 1 for feasibility is always satisfied if the partitioning is successful, this type of graph is always a directed acyclic graph.

The partitioning stage gives the sequence of the equations in (4) and coarse-grained information about data locality. However, it does not give any indication about the order within which the numerical operations required for each equation

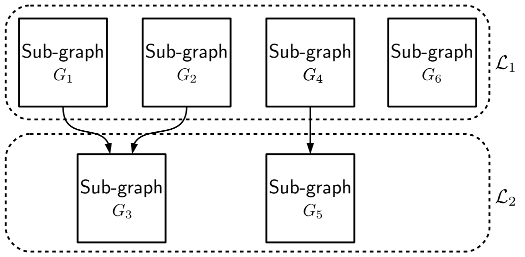


Figure 3: Example graph representing the relationship between sub-graphs. Each vertex in this graph represents a sub-graph and each edge represents an external edge in the original G graph after the partitioning. Sub-graphs are into two sets: $\mathcal{L}_1$ and $\mathcal{L}_2$. There is no path between sub-graphs in each of the sets, because this type of graph is always a directed acyclic graph if the partition is feasible.

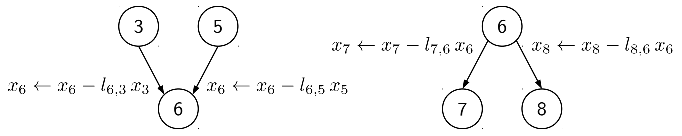

3
5
6
$x_6 \leftarrow x_6 - l_{6,3}\, x_3$
$x_6 \leftarrow x_6 - l_{6,5}\, x_5$
6
7
8
$x_7 \leftarrow x_7 - l_{7,6}\, x_6$
$x_8 \leftarrow x_8 - l_{8,6}\, x_6$

Figure 4: Examples of the two edge patterns in a sub-graph. On one hand, if two or more edges are directed towards a vertex, the numerical operations associated with them are in read-write conflict. On the other hand, if two or more edges are exiting a vertex, the numerical operations associated with them can be carried out in parallel with the SIMD paradigm.

are to be carried out. That information, which in Algorithm 1 is represented by the sets $\mathcal{T}_{gq}$ and $\mathcal{T}'_{gq}$, is obtained by the scheduling stage.

B. Scheduling stage

During the scheduling stage, each edge in the sub-graphs $G_1, G_2, \ldots, G_s$ is mapped to a time slot. Edges with the same time slot represent multiply-and-add operations that can be executed at the same time, with the SIMD paradigm. However, the computation represented by each sub-graph requires further analysis in order to minimize conflicting memory accesses. For example, consider the situation represented in the left half of Figure 4. The variable x_6 is obtained by the execution of two operations in any order, but if the two numerical operations are carried out at the same time, a read-write access conflict occurs. This is because each operation reads the value of the variable x_6 and overwrites it with a new value. In general, this issue occurs whenever two or more edges are directed towards the same vertex.

Our solution to this problem consists of delaying one or more of the operations and assigning the edges to different time slots. This is carried out with the routine shown in Algorithm 4. There is no read-write access conflict in the situation shown in the right half of Figure 4: in this case, the operations overwrite the value of two different variables. Since the two operations can be done at the same time, this is also an opportunity for data-level parallelism.

```
1:  function FIX_CONFLICTS($v$)
2:      for all $(j,v) \in \mathcal{E} \mid j \in \mathcal{V}_k$
3:          if $\exists(q,v) \in \mathcal{E} \mid (q,v) \in \mathcal{T}_{kp},\ (j,v) \in \mathcal{T}_{kp}$
4:              $t \leftarrow \min\{h \in \mathbb{N} \mid h > p,\ \nexists(u,v) \in \mathcal{T}_{kh} \quad \forall u \in \mathcal{V}_k\}$
5:              $\mathcal{T}_{kt} \leftarrow \mathcal{T}_{kt} \cup \{(j,v)\}$
6:      for all $(j,v) \in \mathcal{E} \mid j \notin \mathcal{V}_k$
7:          if $\exists(q,v) \in \mathcal{E} \mid (q,v) \in \mathcal{T}'_{kp},\ (j,v) \in \mathcal{T}'_{kp}$
8:              $t \leftarrow \min\{h \in \mathbb{N} \mid h > p,\ \nexists(u,v) \in \mathcal{T}'_{kh} \quad \forall u \notin \mathcal{V}_k\}$
9:              $\mathcal{T}'_{kt} \leftarrow \mathcal{T}'_{kt} \cup \{(j,v)\}$
10:     return success
```

Algorithm 4: Function that checks and fixes possible read-write conflicts in numerical operations. In the pseudocode, v is a vertex in sub-graph G_k, so $v \in \mathcal{V}_k$. The function always terminates with success.

The scheduling algorithm we propose, which is illustrated in Algorithm 5, assigns edges to time slots while taking advantage of any opportunity for data-level parallelism. Also, it avoids any read-write access conflict by delaying operations if necessary by using the routine in Algorithm 4. Because it associates edges with time slots rather than vertices with levels, this scheduling algorithm is different from the level-set algorithm [4]. Also, since each vertex is associated with only one sub-graph during the partitioning stage, sub-graphs can be processed in parallel. The worst-case computational complexity of the scheduling algorithm is $O(n + 2n_{\text{edges}})$, where n is the total number of vertices in the graph and n_{edges} is the number of edges.

Similarly to the partitioning heuristic, Algorithm 5 uses two containers of vertices: V_{next} and V_{cur}. In line 2, V_{next} is initialised with those vertices that belong to sub-graph G_k and do not have any parent vertex that belongs to the same sub-graph. The input and output edges of these vertices are associated with the first time slots $\mathcal{T}'_{k1}$ and $\mathcal{T}_{k1}$ respectively, then, at each iteration of the algorithm, a vertex v is taken from the container V_{cur}, and the *fix_conflicts* routine is executed on it. In line 18, the input edges of v are checked to find the latest time slot, and next the children vertices of v are then added to V_{next} if all of their input edges have been associated with time slots. When all the vertices in V_{cur} have been processed, the elements of V_{next} are put in V_{cur} and V_{next} is emptied in preparation for a new iteration. The scheduling algorithm always terminates with success if the partition of the G graph is feasible.

At the end of the scheduling stage, all information is available to carry out the execution of the solve phase. Each vertex of the graph is associated with a sub-graph, which is processed by a compute unit during the solve phase, and each edge is associated with a time slot, so the sequence of numerical operations to carry out is also specified. Compared to CUSPARSE's level-set algorithm [4], our analysis phase can results in fewer external edges. For example, in Figure 5, our analysis phase resulted in five internal edges and five external edges, while the level-set algorithm resulted in ten external edges. Since numerical operations associated with internal edges require fewer accesses to global memory, we argue that our approach improves data locality compared to CUSPARSE's level-set algorithm. However, with our algorithm, the maximum number of operations that can be carried out in one time slot is $n_{\max}$, which is usually less than the number of vertices in the G graph, n. Instead, with CUSPARSE's algorithm, the maximum number of operations that can be carried out in one time slot it n. For this reason, we argue that our approach reduces concurrency compared to the level-set algorithm.

```
Input: $G_k$
Output: $\mathcal{T}_{k1}, \mathcal{T}_{k2}, \ldots, \mathcal{T}_{kn_{\text{slots},k}}, \mathcal{T}'_{k1}, \mathcal{T}'_{k2}, \ldots, \mathcal{T}'_{kn'_{\text{slots},k}}$
1:  $V_{\text{cur}} \leftarrow \not{V}$
2:  $V_{\text{cur}} \leftarrow \text{put}(r, V_{\text{next}}) \quad \forall r \in \mathcal{V}_k \mid \nexists(j,r) \in \mathcal{E},\ j \in \mathcal{V}_k$
3:  $\mathcal{T}_{kp} \leftarrow \emptyset \quad \forall p$
4:  $\mathcal{T}'_{kp} \leftarrow \emptyset \quad \forall p$
5:  for all $v \in V_{\text{cur}}$
6:      for all $(j,v) \in \mathcal{E} \mid j \notin \mathcal{V}_k$
7:          $\mathcal{T}'_{k1} \leftarrow \mathcal{T}'_{k1} \cup \{(j,v)\}$
8:      FIX_CONFLICTS($v$)
9:      for all $(v,i) \in \mathcal{E} \mid i \in \mathcal{V}_k$
10:         $\mathcal{T}_{k1} \leftarrow \mathcal{T}_{k1} \cup \{(v,i)\}$
11:         if $\nexists(j,i) \in \mathcal{E} \mid (j,i) \notin \mathcal{T}_{kp},\ (j,i) \notin \mathcal{T}'_{kp} \quad \forall p$
12:             $V_{\text{next}} \leftarrow \text{put}(i, V_{\text{next}})$
13: while $\text{numel}(V_{\text{next}}) \neq 0$
14:     $V_{\text{cur}} \leftarrow V_{\text{next}}$
15:     $V_{\text{next}} \leftarrow \not{V}$
16:     for all $v \in V_{\text{cur}}$
17:         FIX_CONFLICTS($v$)
18:         $t \leftarrow 1 + \max\{q \in \mathbb{N} \mid \exists(j,v) \in \mathcal{T}_{kq}\}$
19:         for all $(v,i) \in \mathcal{E} \mid i \in \mathcal{V}_k$
20:             $\mathcal{T}_{kt} \leftarrow \mathcal{T}_{kt} \cup \{(v,i)\}$
21:             if $\nexists(j,i) \in \mathcal{E} \mid (j,i) \notin \mathcal{T}_{kp},\ (j,i) \notin \mathcal{T}'_{kp} \quad \forall p$
22:                 $V_{\text{next}} \leftarrow \text{put}(i, V_{\text{next}})$
23: terminate with success
```

Algorithm 5: Algorithm that assigns the edges of a sub-graph G_p to time slots while making sure no read-write conflict occurs. The algorithm uses the containers of vertices V_{cur} and V_{next}, on which the *put* and *numel* operations are defined. In the pseudocode, $\not{V}$ represents the empty container, while $\mathcal{T}_{kp}$ and $\mathcal{T}'_{kp}$ are the p-th time slot of internal and external edges respectively, for sub-graph G_k.

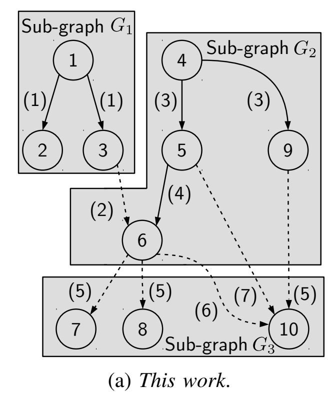
VI. Experimental results

To study how the performance of the solver changes with the sparsity pattern of L, we carried out five experiments which we divided into two categories: experiments with automatically generated test cases and experiments with test cases from the Florida matrix collection [6].

The solve phase was run 100 times per matrix in every experiment, each time with a different, randomly generated right hand side, and the mean of the results was taken. We included the data transfer from the host memory to the device memory in the measurement. The single-precision floating point format was used for matrix entries in both cases.

A. Automatically generated test cases

Automatically generated test cases are matrices we generated that have specific properties, with the aim of studying the performance of our algorithms in function of specific graph attributes. For each of these matrices, we executed the software



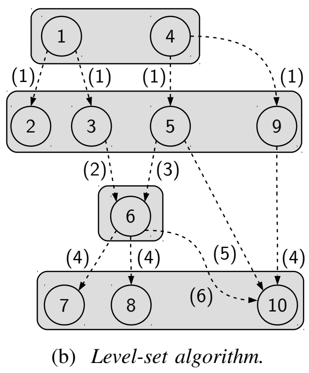


Figure 5: Comparison between the result of the analysis phase on an example graph, with $n_{\max} = 4$, and the analysis phase as carried out by the level-set algorithm [4]. Time slots are shown between parentheses, and external edges are shown with dashed lines. Our analysis resulted in five external edges and five internal edges, while CUSPARSE's analysis results only in external edges.

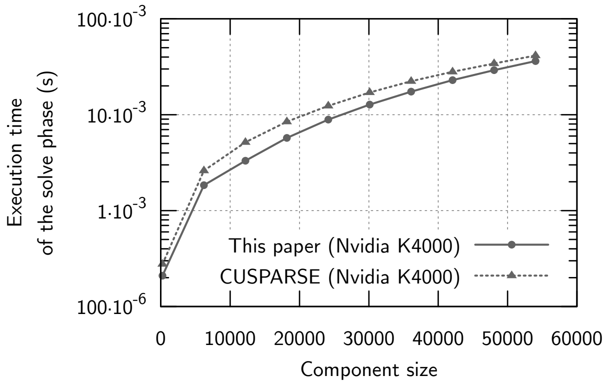


Figure 6: Execution time of the solve phase with randomly generated block-diagonal matrices, each having 16 disjoint components.

on an Nvidia Quadro K4000 GPU [13]. We measured the execution time of our solve phase, which was carried out with an OpenCL kernel, and we compared it to the execution time of the corresponding CUSPARSE library function [4] on the same GPU. On these automatically generated cases, we carried out three experiments.

The objective of the first experiment was to measure how the execution time of our algorithm scales with the number of vertices in the disjoint components of a graph. To achieve this, we generated matrices with 16 disjoint components, but with component size that ranged from 250 to 55000. We used *awk* to generate block-diagonal lower triangular matrices, with an arbitrarily chosen density of non-zero elements in the blocks of $0.1\,\%$. Figure 6 summarises the results of the experiment. The OpenCL kernel outperformed CUSPARSE, the performance difference increasing as component size increased. The maximum speedup achieved was 2.5 fold.

In the same experiment, we also measured the execution

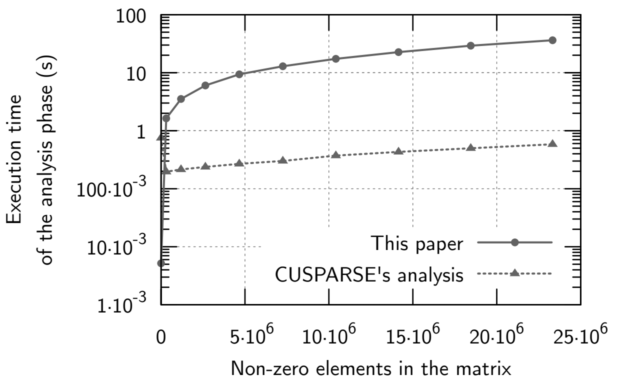


Figure 7: Execution time of the analysis phase with randomly generated block-diagonal matrices, each having 16 disjoint components.

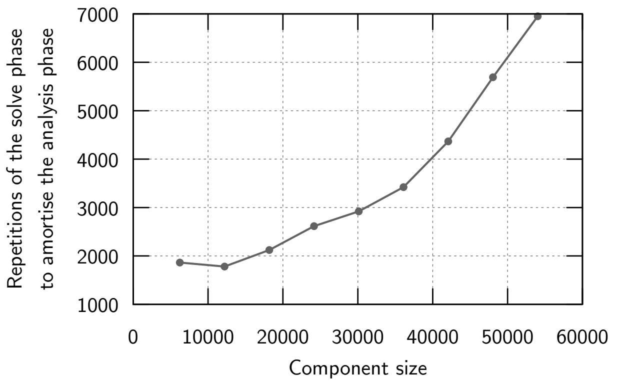


Figure 8: Number of times the solve phase should be repeated before the execution time of the analysis phase is amortised, and the total execution time is less than CUSPARSE's. Each of the test cases has 16 disjoint components of the given size.

time of the analysis phase, which was carried out only once per data point on an Intel Xeon X5650 CPU. Our algorithms for the analysis were implemented in C++. Figure 7 shows that the execution times of both CUSPARSE's analysis and our algorithms for the analysis grow linearly with the number of non-zero elements in the matrix. However, we observed that the slope was 166 times steeper with our algorithms. Because of this, we see that the solve phase has to be repeated a number of times before the execution time of the analysis phase is amortised. This number increases with the size of the components, as is illustrated in Figure 8, but is below the tens of thousands required in large scale optimization problems [14].

The objective of the second experiment was to measure how the execution time of our algorithm scales with the number of disjoint components in a graph. We generated block-diagonal matrices with disjoint components of size 6000, that is below $n_{\max}$ for the Nvidia Quadro K4000 GPU. The number of disjoint components ranged from 8 to 512, while the density of non-zero elements in the block was $1\,\%$ and was chosen arbitrarily. Figure 9 show that our OpenCL kernel outperformed CUSPARSE's function in all cases. Also, the execution time

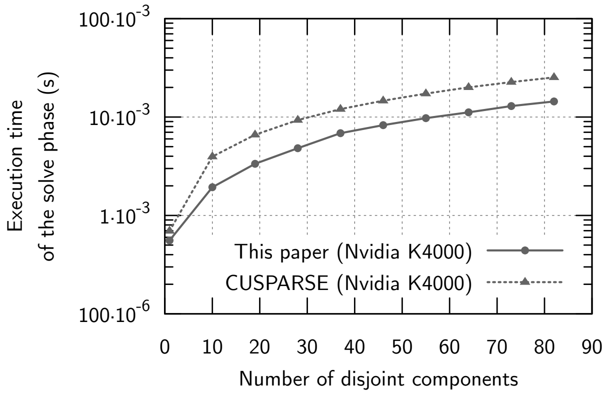


Figure 9: Execution time of the solve phase with randomly generated block-diagonal matrices, each having components of size 6000.

Table I: Value range of Features of the test cases from the Florida Matrix Collection [6] and range of their values.

Matrix feature	min	max
Number of rows/columns	10261	525824
Number of non-zero elements	760	4356551
Number of disjoint components	1	5000
Size of disjoint components	2	20360

of the OpenCL kernel was linear with respect to the number of disjoint components. The maximum speedup achieved was 2 fold.

B. *Test cases from the Florida matrix collection*

The Florida matrix collecton [6] is a standard benchmark set for sparse linear algebra algorithms that contains non-artificial test cases. For each of the matrices, the zero-fill-in incomplete LU decomposition was computed [3], so that the resulting triangular factors had the same sparsity patterns as the lower half of the input matrix. Then, both analysis and solve phases were executed on the lower triangular factor. We considered the first 200 matrices, starting from that with the fewest rows and columns, such that the execution time of the solve phase was known with a confidence interval smaller than $1\,\%$. Table I summarises the features of the lower triangular factors that were used. We considered matrices whose graph is composed of a single, large component as well as matrices whose graph is composed of many, small disjoint components.

On the Nvidia Quadro K4000 GPU, the average execution time of the OpenCL kernel was compared to the execution time of CUSPARSE's function, as shown in Figure 10, where each point corresponds to a test case. Points above the diagonal line correspond to test cases in which the OpenCL kernel outperformed CUSPARSE's function, while points below the diagonal line correspond to test cases in which the OpenCL kernel was outperformed. In $71\,\%$ of all test cases, the execution time of the OpenCL kernel was less than that of CUSPARSE's function. In particular, the worst speedup was achieved on a test case named *circuit_3*. In this test case, the matrix was

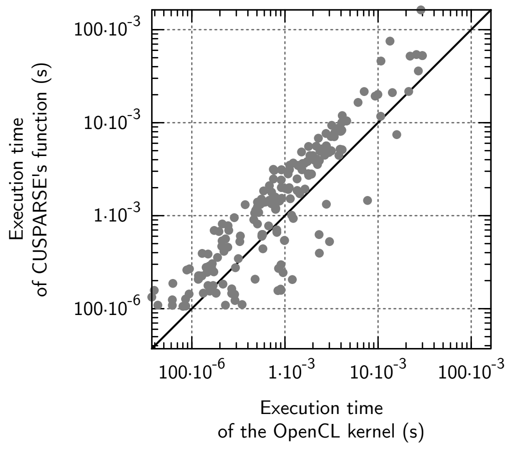


Figure 10: Comparison between the execution time of the OpenCL kernel and the execution time of CUSPARSE's function [4] on 200 test cases from the Florida matrix collection [6]. The points above the diagonal, which are $71\,\%$ of the total, represent test cases in which the OpenCL kernel outperforms CUSPARSE's function. The software was executed on the Nvidia Quadro GPU.

not partitioned because the vertices that did not belong to the set $\mathcal{V}_0$ were fewer than $n_{\max}$, and also the fraction of non-zero elements was $0.019\,\%$, which meant the G graph had few edges. Because there was not enough compute, the data transfer from global to local memory could not be amortised and the OpenCL kernel was 5.9 times slower than CUSPARSE's algorithm. Instead, the best speedup achieved was 5.8-fold, on a test case named *human_gene2*. For this test case, our analysis phase partitioned the graph into three sub-graphs, of which two belonged to the first sub-graph level. With our algorithm, only $26\,\%$ of the edges in the graph were external, compared to CUSPARSE's algorithm.

In the last experiment, we investigated how the architectural attributes of the GPU impact on the performance of our algorithm. For this, we used to two different GPUs: an Nvidia Quadro K4000 and an AMD Firepro W5000. The two GPUs chosen have about the same peak performance, which is $1.26\,\mathrm{TFLOPS}$ for the Nvidia GPU and $1.27\,\mathrm{TFLOPS}$ for the AMD GPU [15], [13]. However, while the AMD Firepro has a local memory capacity of $32\,\mathrm{kB}$, the Nvidia Quadro has a local memory capacity of $48\,\mathrm{kB}$. Also, as specifications of the two GPUs in Table II show, the compute units of the Nvidia Quadro GPU contain more than double the processing elements of the AMD GPU. On the other hand, the AMD Firepro GPU contains three times as many compute units as the Nvidia Quadro GPU.

Figure 11 illustrates the results of the experiment, where points above the diagonal line correspond to test cases in which the Nvidia GPU outperforms the AMD one, while points below the diagonal line correspond to test cases in which the Nvidia GPU is outperformed. In $74\,\%$ of all test cases, the execution time on the Nvidia GPU is less than that of the AMD GPU.

In particular, the maximum speedup achieved with the Nvidia

Table II: Specifications of the GPUs used to carry out the experiments. *CU* stands for compute units, *PE* for processing elements, and *LM* for local memory capacity.

Platform	Clockrate	CU	PE/CU	LM
Nvidia Quadro K4000	0.82 GHz	4	192	48 kB
AMD Firepro W5000	0.83 GHz	12	64	32 kB

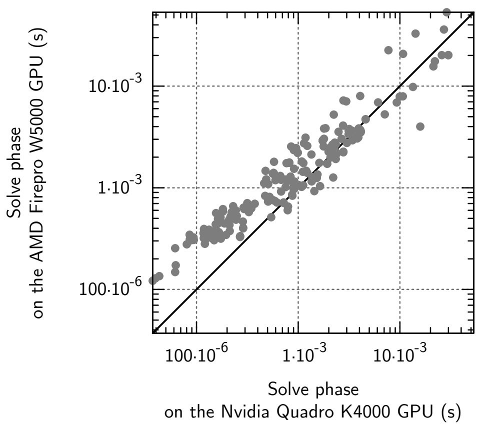


Figure 11: Comparison between the execution time of the OpenCL kernel on the Nvidia Quadro K4000 and the AMD Firepro W5000 GPUs on 200 test cases from the Florida matrix collection [6]. The points above the diagonal, which are 74 % of the total, represent test cases in which the Nvidia GPU outperforms the AMD GPU.

GPU against the AMD GPU was 4-fold, on a test case named *fd15*. For this test case, because of the more capacious local memory, the Nvidia GPU was able to carry out all numerical operations using a single compute unit without partitioning the graph. Hence, all edges in the graph were internal edges on the Nvidia GPU. Instead, for the AMD GPU, the graph was partitioned, and the computation was carried out on two sub-graph levels. The maximum speedup achieved with the AMD GPU with respect to the Nvidia GPU was 3-fold, on a test case named *mc2depi*. For both GPUs, the graph associated with *mc2depi* was composed of 47 sub-graphs, all on the same sub-graph level. Because of the AMD GPU has three times as many compute units as the Nvidia GPU, the AMD GPU carried out all computations three times faster.

VII. Conclusion

When solving STLs, the structure of the input matrix has a significant impact on performance. Like our experimental results show, the same algorithm carried out on the same data with two GPUs of comparable peak performance can have significantly different execution times.

Our experimental results show that reducing concurrency to the advantage of data locality can lead to better performance when solving STLs using GPUs. Our approach is different from that of the level-set algorithm [4], which aims to maximise concurrency, as well as that used in multi-core CPUs [5], which aims to balance the workload across the CPU cores by maximising data locality. Instead, our algorithm partitions the graph associated with the STL by taking into consideration the capacity of the local memories inside the GPU, with the aim of balancing data locality and concurrency. The partitioning is done by a pre-processing algorithm, or analysis phase, which is carried out once per a given sub-graph. Although the analysis phase is more time consuming than that of the level-set algorithm, its result can be re-used for many STL solves.

Our future work focuses on how to reduce the impact of the analysis time on the overall execution of calling software through mapping incremental changes to the matrix structure to incremental changes to the partioning.

Acknowledgements

The support of Siemens Corporate Research, EPSRC grants EP/I020357/1 and EP/K015168/1, the Royal Academy of Engineering and Imagination Technologies is gratefully acknowledged.

References

[1] U. Münz *et al.*, “Overview of recent control technologies for future power systems: An industry perspective,” *Automatisierungstechnik*, pp. 869–882, Nov. 2015.

[2] G. Wu *et al.*, “A preconditioned conjugate gradient algorithm for GeneRank with application to microarray data mining,” *Data Mining and Knowledge Discovery*, vol. 26, no. 1, pp. 27–56, 2013.

[3] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Society for Industrial and Applied Mathematics, 2003.

[4] M. Naumov, “Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU,” NVIDIA Corporation, Technical report 2011–001, Jun. 2011.

[5] J. Mayer, “Parallel algorithms for solving linear systems with sparse triangular matrices,” *Computing*, vol. 86, no. 4, pp. 291–312, Sep. 2009.

[6] T. A. Davis and Y. Hu, “The University of Florida sparse matrix collection,” *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1–25, Dec. 2011.

[7] T. A. Davis, *Direct Methods for Sparse Linear Systems*, ser. Fundamentals of Algorithms. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2006, vol. 2.

[8] B. Suchoski *et al.*, “Adapting sparse triangular solution to GPUs,” in *Proceeding of the 41st International Conference on Parallel Processing Workshops (ICPPW)*, Sep. 2012, pp. 140–148.

[9] V. W. Lee *et al.*, “Debunking the 100x GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU,” in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA ’10. New York, NY, USA: ACM, 2010, pp. 451–460.

[10] S.-Y. Lee and C.-J. Wu, “CAWS: Criticality-aware warp scheduling for gpgpu workloads,” in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT ’14. New York, NY, USA: ACM, 2014, pp. 175–186.

[11] Harshvardhan *et al.*, “KLA: A new algorithmic paradigm for parallel graph computations,” in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT ’14. New York, NY, USA: ACM, 2014, pp. 27–38.

[12] R. Tarjan, “Depth-first search and linear graph algorithms,” *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, Jun. 1972.

[13] “Nvidia professional graphics solution,” Online, Nvidia Corporation, Jul. 2013. [Online]. Available: http://www.nvidia.co.uk/content/PDF/line_card/6660-nv-prographicssolutions-linecard-july13-final-lr.pdf

[14] S. Cafieri *et al.*, “On the iterative solution of KKT systems in potential reduction software for large-scale quadratic problems,” *Computational Optimization and Applications*, vol. 38, no. 1, pp. 27–45, 2007.

[15] “AMD Firepro W5000,” Online, AMD Corporation, 2014. [Online]. Available: http://www.amd.com/documents/2795_W5000_DataSheet_R3.pdf