Regularity versus Load-Balancing on GPU for treefix computations

David Defour, Manuel Marin

Regularity versus Load-Balancing on GPU for treefix computations

David Defour and Manuel Marin
Univ. Perpignan Via Domitia, DALI F-66860, Perpignan, France
Univ. Montpellier II, LIRMM, UMR 5506, F-34095, Montpellier, France
CNRS, LIRMM, UMR 5506, F-34095, Montpellier, France

***Abstract*—The use of GPUs has enabled us to achieve substantial acceleration in highly regular data parallel applications. The trend is now to look at irregular applications, as it requires advanced load balancing technics. However, it is well known that the use of regular computation is preferable and more suitable when working with these architectures. An alternative to the use of load balancing is to rely on *scan* and other GPU friendly parallel primitives to build the desired result; however implying in return, the involvement of extra memory storage and computation.**

This article discusses of both solutions for treefix operations, which consist of applying a certain operation while performing a tree traversal. They can be performed by traversing the tree from top to bottom or from bottom to top, applying the proper operation at each vertex. It can be accelerated using either load balancing which maintains a pool of tasks while performing only the necessary amount of computation or using a vector friendly representation that will involve twice the amount of computation than the first solution. We will explore these two approaches and compare them in terms of performance and accuracy. We will show that the vectorial approach is always faster for any category of trees, but it raises accuracy issues when working with floating-point data.

I. Introduction

In recent years, processors such as IBM cell SPUs, FPGAs, GPUs, and ASICs were successfully considered to provide speedup on numerous classes of applications. Of these, GPUs stand out as they are produced as commodity processors and exhibiting a number of processing cores doubling every year, revealing the current architectural trend. GPUs were used to improve the performance of regular computations such as those described in [21]. On such highly regular computations, GPUs can outperform a single core CPU by a large factor on average, that could be higher than 400 in some cases [7]. These large speedups are only possible for highly regular and computationally intensive classes of application. More recently, irregular computations on graphs such as list ranking [24] and connected components [13] were also considered. However, in these cases, the observed speedup compared to single core performance is of the order of 5 or less.

Treefix operations were first introduced by Leiserson and Maggs [15] as intermediate steps in a number of higher-level graph analysis algorithms. They defined two basic operations, *Rootfix* and *Leaffix*. Rootfix returns to each vertex of the tree the result of applying a certain operation over all its ancestors; Leaffix returns to each vertex the result of applying an operation over all its descendants. Rootfix and Leaffix have application for example in the Backward-forward sweep algorithm for electrical network analysis [22] or to evaluate the parsimony score of phylogenetic trees [9], [20]. In this article, we explore the available alternatives to accelerate these computations using GPUs.

The usual implementation of Rootfix and Leaffix is based on traversing the tree, from top to bottom or from bottom to top. The vertices are updated as visited, allowing to effectively propagate the accumulated result of the operation through the whole tree as the traversal progresses. The order of visit is relevant. Starting from the root, depth-first or breadth-first traversals are both valid alternatives. Ultimately, Rootfix and Leaffix can be viewed as performing a complete Breadth-first or Depth-first search over a tree, updating the vertices' weights as they are visited.

Successful implementations of parallel Breadth-first search over a general graph on GPU can be found in [8], [10], [14], [16], [17]. All of them rely on *level-synchronization*, i.e. processing every level of the graph in parallel, in order of depth. This is often implemented as an iterative process that performs one iteration per level. Some versions [8], [10], [14] examine every vertex of the graph at every iteration: if the predecessor was visited during the last iteration, then the vertex is visited. These methods perform a quadratic amount of work, as the graph can have, in the worst case, as many levels as vertices. A work efficient versions [16], [17] focus on producing, at each iteration, a vertex or edge *frontier*, including only those elements to be visited or traversed during that iteration. The main advantage of these methods is to exhibit a work efficient scheme, but have to deal with the irregularity of the graph data structure, which involves load imbalance and potential underutilization of SIMD lanes. Different load balancing strategies are applied to improve the performance achieved by these methods.

An alternative for performing Rootfix and Leaffix on a GPU, is to use a parallel-friendly representation of the tree consisting of three arrays based on the Euler-tour ordering. A series of highly regular parallel operations performed over these arrays, such as *scan*, allow to compute the result of Rootfix and Leaffix for a tree with n vertices in $O(\lg n)$ parallel steps, independently of the tree topology. However this methods relies on array of size $2.n$ with two times more computations than load balancing implementations.

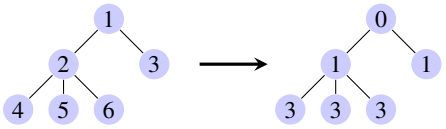
The purpose of this article is to determine the best solution between a work efficient scheme thanks to irregular computation or a solution with regular computation with double the amount of operation to solve the treefix problem on GPUs. It makes the following contributions in the area of parallel computing:

- **Regular vs irregular algorithm comparison**. We present two different approaches that make use of data-parallelism to perform a distinctive operation over trees. One of them leads to an application that is highly regular, the other to one that is highly irregular and compares them in terms of performance.
- **Numerical quality analysis**. We compare the numerical accuracy of both methods when dealing with floating-point data as the amount and the order of operation is different.
- **Rootfix and Leaffix OpenCL implementation**. We provide a vectorial implementation of +Rootfix and +Leaffix in OpenCL. Even if there has been some work on implementing Rootfix and Leaffix in different languages [2], [3], [6], this is, to our knowledge, the first parallel implementation that could run on a GPU.

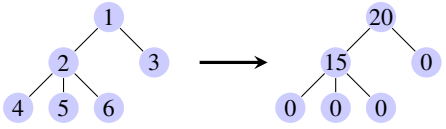
II. Presentation of Rootfix and Leaffix

Leiserson and Maggs [15] formally defined Rootfix and Leaffix as follows: given a weighted tree and a binary operator $\oplus$, Rootfix assigns to each vertex the result of applying $\oplus$ to all of the vertex's ancestors; Leaffix assigns to each vertex the result of applying $\oplus$ to all of the vertex's descendants.

From there, we can define the *+Rootfix* and *+Leaffix* operations, where $\oplus$ is the addition, as assigning to each vertex the sum of its ancestors and the sum of its descendants, respectively. Figure 1 shows an example. In particular, if all the vertices of the tree have weight 1, +Rootfix returns the depth of each vertex, and +Leaffix returns the size of the subtree rooted on every vertex.



(a) +Rootfix



(b) +Leaffix

Fig. 1: Example of +Rootfix and +Leaffix.

A. Parallel algorithm

Regarding the type of trees considered, there are two easy cases of parallelization: balanced binary tree and linked list. For the balanced binary tree, Leiserson and Maggs [15] proposed a randomized algorithm that performs Rootfix and Leaffix on a tree of size n in $O(\lg n)$ parallel steps, applying the contraction technique provided by Miller and Reif [18]. For the linked list, or caterpillar, there exists a $O(\lg n)$ depth algorithm based on symmetry breaking. For other cases, when there is no bound on the number of children, nor on the tree topology, a different algorithm has to be used.

In this article, we consider traversing the tree using parallel Breadth-first search. The tree is expressed as a directed graph of the form $G = (V, E)$, with a set V of n vertices and a set E of $n-1$ directed edges [1]. The adjacency matrix A is defined as follows.

$$A_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$$

We rely on compressed sparse row (CSR) format to store this matrix into two arrays. The array C contains the column indices of the non-zero elements of A arranged in row-major order. The array R contains $n+1$ integers, and entry $R[i]$ is the index in C of the *i-th* row of A.

Algorithm 1 illustrates the usual way of performing +Rootifx using parallel Breadth-first search based on level-synchronization. The algorithm manipulates two queues: one input queue and one output queue. The input queue contains all the vertices to be examined during certain iteration. All these vertices are dequeued in parallel and their children are updated. As updated, the children are placed in the output queue. When all the children have been visited at a given level, the output queue is transferred in the input queue to be consumed by the next iteration. The algorithm proceeds until there are no vertices left to examine.

Algorithm 1 +Rootfix parallel algorithm

Input: Row-offsets array R, column-indices array C, weights array W, queues. Function *LockedEnqueue(vertex)* safely inserts *vertex* at the end of the queue instance.
Output: Array $rootfix[0 \dots n-1]$ holding the result.

```
1:  rootfix[0] ← W[0]
2:  inQ ← {}
3:  inQ.LockedEnqueue(0)
4:  while inQ != {} do
5:      outQ ← {}
6:      for i in inQ do in parallel
7:          for offset in R[i] ... R[i+1] − 1 do
8:              j ← C[offset]
9:              rootfix[j] = rootfix[i] + W[j]
10:             outQ.LockedEnqueue(j)
11:     inQ ← outQ
```

The amount of parallel work that this algorithm can perform depends on the tree topology. The wider the level, the greater the number of parallel tasks than can be assigned for that level.

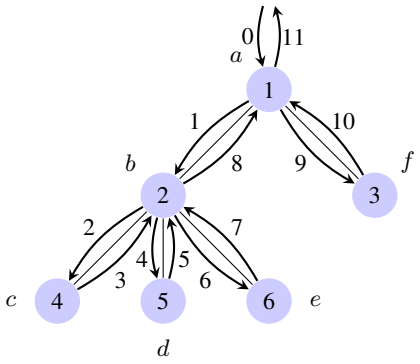
[1] always directed from parent to child

This is related to the *average branching factor*, i.e. the average number of children per vertex. The worst-case scenario is when every vertex has only one child (caterpillar) and then all the vertices have to be examined sequentially.

B. Vectorial algorithm

The implementation of Rootfix and Leaffix for the PRAM machine model was studied by Blelloch [5], who provided a vectorial algorithm. The algorithm uses Euler-tour order, a technique first introduced by Tarjan and Vishkin [23], to compute a vector representation of the tree. The Euler-tour order is generated by replacing every edge in the tree by two directed edges, one in each sense; these edges define an Eulerian path around the tree. As they appear on this path, the edges are placed into an Euler-tour vector E. A downward edge reaching vertex v is labeled ${}_(v$ and an upward edge leaving vertex v is labeled $v_)$. Figure 2 shows an example tree and the corresponding Euler-tour vector. Note that, as we doubled the number of edges, the Euler-tour vector has twice the size of the tree.

The vector tree representation consists of three arrays, ${}_(V$, $V_)$ and W. The array ${}_(V$ holds, for each vertex v, the index of ${}_(v$ in the Euler-tour vector E; the array $V_)$, the index of $v_)$. The array W holds the vertices' weights.



$$E = [{}_(a, {}_(b, {}_(c, c_), {}_(d, d_), {}_(e, e_), b_), {}_(f, f_), a_)]$$

$${}_(V = [0, 1, 2, 4, 6, 9]$$

$$V_) = [11, 8, 3, 5, 7, 10]$$

$$W = [1, 2, 4, 5, 6, 3]$$

Fig. 2: Example tree, Euler-tour ordering and vector tree representation.

These three arrays are used altogether with some regular parallel primitives to compute the result of Rootfix and Leaffix in a parallel fashion. The key primitive is the *scan* operation, that given a binary operator $\oplus$ with identity i, takes the array

$$(x_0, x_1, \ldots, x_{n-1})$$

and returns the array

$$(i, x_0, x_0 \oplus x_1 \ldots, x_0 \oplus x_1 \oplus \ldots \oplus x_{n-2})$$

For $\oplus$ being the addition, the *+scan* operation takes the same input array and returns

$$(0, x_0, x_0 + x_1 \ldots, x_0 + x_1 + \ldots + x_{n-2})$$

There exist many GPUs implementation of this operation as it is a basic building block of many data parallel algorithms. The one in [12] operates in $O(\lg n)$ steps and $O(n)$ operations. This has been further optimized for the NVIDIA Fermi architecture in [11].

Algorithm 2 takes as input an array E of size $2n$, which is used for intermediate computation, and the three arrays ${}_(V$, $V_)$ and W of size n that hold the tree. It produces the result of +Rootfix. A similar algorithm is available for +Leaffix.

Algorithm 2 +Rootfix vectorial algorithm

Input: Array E of size $2n$, arrays ${}_(V$, $V_)$ and W of size n holding the tree.
Output: Array R of size n holding the result.

```
1: //Step 1: Write
2: for i in 0...n do in parallel
3:     E[(V[i]] ← W[i]
4:     E[V)[i]] ← −W[i]
5: //Step 2: Scan
6: Run an inplace +scan on E
7: //Step 3: Read
8: for i in 0...n do in parallel
9:     R[i] ⟵ E[(V[i]]
```

Figures 3 illustrates this algorithm on an example tree. We used the sum as operation applied on integer data. It can be noticed that we could have used any set of values and with any binary operation that forms a *group*. The operation has to be associative, with an inverse and an identity value. As floating-point addition is not associative, these algorithms should not be applied in such cases. However, we will show that in this particular case the error can be bounded.

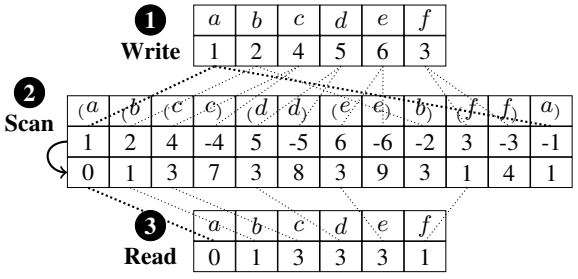


Fig. 3: +Rootfix vectorial algorithm.

III. GPU IMPLEMENTATION

A. Parallel version

In section II-A we showed that +Rootfix can be performed using parallel Breadth-first search over a tree. As Breadth-first search is a common building block for many graph analysis algorithms, there exist several GPU implementations. We used

the one by Merril *et al.* [17], written in CUDA. This version optimizes the neighbor gathering process, which corresponds to the for-loop in line 7 of algorithm 1, to balance load within the CTA. For each vertex being expanded, the row-range bounds are read from the array R (values $R[i]$ and $R[i+1]$). Then, each thread uses the result of a CTA-wide parallel prefix sum over the differences $R[i+1] - R[i]$, to perfectly pack into a buffer, which is shared by the entire CTA, the positions on the array C of the neighbors to be gathered (values $R[i] \dots R[i+1] - 1$). Once the buffer has been filled, each thread in the CTA reads one position on it and gathers the corresponding neighbor from C, leaving no SIMD lane idle during the process. This load balancing strategy allows to achieve a traversal rate about 5 times greater than with other parallel implementations on GPU, as stated by Merril *et al.*. We did not modify the code to make it more suitable to our purposes, more details are available in *et al.*.

B. Vectorial version

We have seen in section II-B that +Rootfix and +Leaffix can be implemented on a PRAM machine using the vector tree representation and the +scan operation. However, there was no GPU implementation available. To perform the test, we developed an OpenCL implementation of +Rootfix and +Leaffix, as this allows us to be platform independent.

The implementation for both operations follows the algorithms by Blelloch and it is built around 3 separate kernels, operating on 3 vectors of size n that represent the input tree $({}_(V, V_), W)$. Once data allocation and data transfer are done, a first kernel *Write* is launched with n work items packed in workgroup sizes that maximize performance. Our test has shown that this corresponds to the maximum allowed for the selected device, which can be queried via *clGetKernelWorkGroupInfo()*. This first kernel is in charge of reading data from input vectors and placing them accordingly in the Euler-tour vector E located in global memory. Then the *Scan* kernel is launched to perform a prefix sum on E. And finally the third kernel *Read* reads the results from E and compute the results for each node. The execution configuration of this third kernel is identical to the first kernel.

All tree kernels are bandwidth limited. Let idx represent the global index of a given OpenCL work item. The Write kernel involves 3 coalesced reads (${}_(V[idx]$, $V_)[idx]$ and $W[idx]$) and 2 uncoalesced writes in the Euler-tour vector E ($E[{}_(V]$ and $E[V_)]$) for both +Rootfix and +Leaffix. The Read kernel involves 1 coalesced read (${}_(V[idx]$), 1 uncoalesced read ($E[{}_(V]$) and 1 coalesced write ($R[idx]$) for both +Rootfix and +Leaffix, plus 2 coalesced reads ($V_)[idx]$ and $W[idx]$) and 1 uncoalesced read ($E[V_)]$) only for +Leaffix. Although it is possible to design an efficient memory access pattern for the Scan kernel, it was not possible to avoid those 'uncoalesced' memory accesses for the Write and Read kernels as the scheme is highly dependent on the tree topology. This has been confirmed by the Nvidia profiler. However, we noticed that GPU with L1 and L2 cache like Fermi were beneficing of relaxed memory access pattern improving memory bandwidth.

TABLE I: Suite of benchmark trees

Name	Nb. of vertices	Depth	Avg. branching factor
af_shell9	504855	490	1030.32
audikw1	943695	236	3998.71
ldoor	952203	784	1214.54
af_shell10	1508065	1098	1373.47
G3_circuit	1585478	705	2248.91
kkt_power	2063494	36	57319.28
nlpkkt120	3542400	123	28800.00
cage15	5154859	81	63640.23
nlpkkt160	8345600	163	51200.00
nlpkkt200	16240000	203	80000.00

On a Fermi architecture, when performing +Rootfix on a tree of 10^7 vertices, the global memory load efficiency of the Read kernel is about 61.5 %, whereas on a pre-Fermi architecture it is about 30 %. For the Write kernel, the difference is of 42.9 % versus 22.9 %.

IV. Tests and results

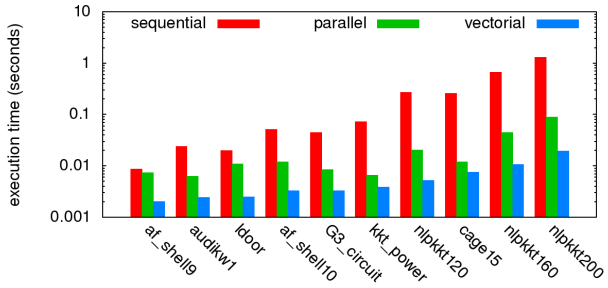
In this section we present the tests we carried out to measure the related performance and accuracy of different +Rootfix and +Leaffix implementations. The results are discussed in light of the different features presented in the tested implementations.

A. Performance

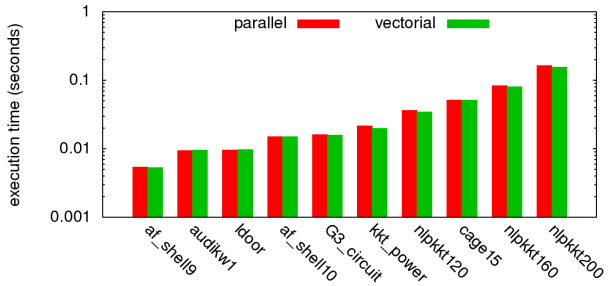
When using Breadth-first search for performing Rootfix over a tree, as the algorithm is completely data-driven, one can expect that the tree topology will have an impact on the performance. Moreover, if a parallel implementation is used, some types of tree will allow more parallelism than others. This is related to the average branching factor, i.e. the ratio between the number of vertices and the number of levels. The larger is this parameter, the wider the tree and thus the greater the number of parallel tasks that can be performed. On the other hand, if we use a vectorial algorithm, the impact of the tree topology over performance should be negligible. To validate this hypothesis and test the proposed implementations, we used a group of benchmarks from the University of Florida Sparse Matrix Collection [4]. This collection, maintained by Tim Davis and Yifan Hu, includes several matrices from different real-life problems on different fields. We selected ten matrices that were considered by the 10^{th} DIMACS Implementation Challenge [1]. For each one of these matrices, we computed a spanning tree of the associated directed graph and used that tree as benchmark. Table I shows the details of the benchmarks generated, including the tree depth and the average branching factor.

For each algorithm running on each benchmark, we measured the total execution time and decoupled it into (a) data transfer time, and (b) computation time. This is motivated by the fact that these algorithms are usually included in iterative scheme where they are called alternatively until a condition is reached. In these cases, data transfer is operated only once. We compared the parallel and vectorial +Rootfix implementations to a purely sequential +Rootfix implementation running on CPU. The machine used for running all our tests is an Intel Xeon E645 CPU with an NVIDIA GeForce GTX670 1344

cores GPU. We used GCC 4.6.3, Cuda 4.2.1 and OpenCL 1.1.



(a) Computation time



(b) Data transfer time

Fig. 4: Related performance of sequential, parallel and vectorial +Rootfix on the GTX670 GPU.

Figure 4 shows the related performance of sequential, parallel and vectorial +Rootfix. The benchmarks are ordered from left to right by increasing number of vertices. We observe in figure 4a that the computation time for the vectorial implementation always grows with the tree size, while for the sequential and parallel implementation there are some cases where a certain tree is processed in less time than another one that has fewer vertices. For example, the parallel implementation needs 21 milliseconds to compute the result for the *nlpkkt120* benchmark, which has 3.54 million vertices, and only 12 milliseconds to compute the result for the *cage15* benchmark, which has 5.15 million vertices.

The data transfer time is almost the same for both the parallel and vectorial implementations, as we see in figure 4b. This is consistent with the fact that they transfer the same amount of data. For a tree of n vertices, the parallel implementation transfers from host to device the CSR representation, consisting of two arrays of size respectively $n-1$ and $n+1$. The vectorial implementation transfers the vector tree representation, consisting of two arrays of size n each. Both implementations transfer from device to host the result in the form of one array of size n.

Figure 5 shows the speedup of parallel and vectorial +Rootfix over sequential +Rootfix. We can see that, when considering only computation time, the speedup achieved by both implementations is quite substantial; it reaches more than 60x on the largest benchmarks analyzed with the vectorial implementation.

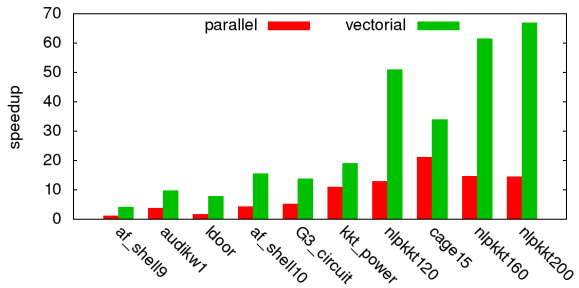


Fig. 5: Speedup of parallel and vectorial +Rootfix over sequential +Rootfix on the GTX670 GPU.

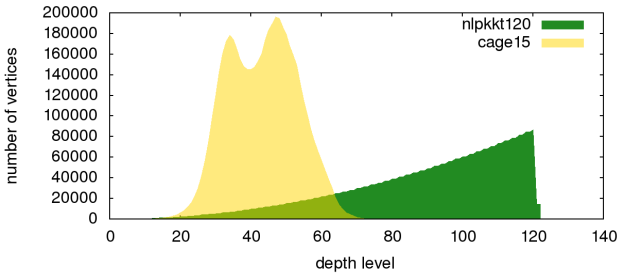
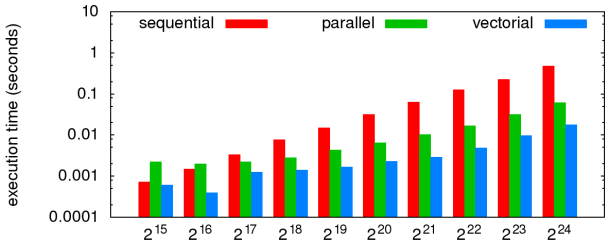


Fig. 6: Comparison of vertex distribution in the *nlpkkt120* and *cage15* benchmarks.

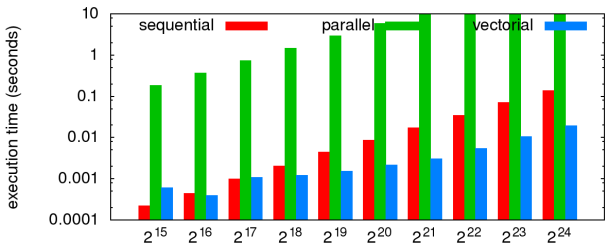
To measure the impact of the tree topology, we looked at the vertex distribution of pairs of benchmarks, like it is plotted in figure 6. The 5.15 million vertices of the *cage15* benchmark are concentrated in fewer levels than the 3.54 million of the *nlpkkt120* benchmark. As a consequence of this, the *nlpkkt120* benchmark takes longer to process, even if it is smaller than the *cage15*. This explains the difference quoted in figure 4.

To quantify the effect of the average branching factor on the two version of the +Rootfix algorithms, we considered two extreme cases of topology: (a) the star, where the root has $n-1$ children, and (b) the linked list, or caterpillar, where every vertex has exactly one child. Figure 8 shows a diagram of both. In the star, the average branching factor is equal to the size of the tree; in the caterpillar, it is equal to one.

We generated a new set of benchmarks composed of stars and caterpillars of sizes varying from 2^{15} to 2^{24} vertices. Figure 7 shows the computation time of sequential, parallel and vectorial +Rootfix on these special topologies. We observe that the parallel implementation performs poorly on the caterpillar, as this algorithm finally needs to process all the vertices sequentially on the GPU. This causes a slowdown compared to the sequential implementation, as the load balancing tasks remains while bringing no benefits. In the star, all the vertices except the root are concentrated on one single level, which correspond to the perfect case for the parallel version. We can notice that, surprisingly, the vectorial implementation is faster by a factor 5 compared to the parallel implementation for the

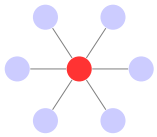


(a) Star

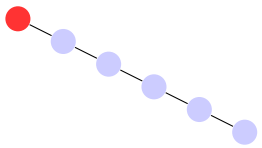


(b) Caterpillar

Fig. 7: Related performance of sequential, parallel and vectorial +Rootfix for star and caterpillar trees on the GTX670 GPU.



(a) Star



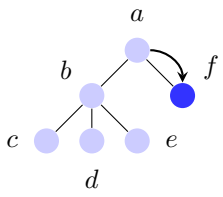
(b) Caterpillar

Fig. 8: Extreme cases of average branching factor and the corresponding tree topology.

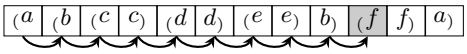
star with 2^{24} nodes. As the branching factor is decreasing, the performance of the parallel version is quickly decreasing leading to a computation time 5000 times greater than the vectorial implementation.

B. Accuracy

When +Rootfix and +Leaffix operate on integer both the parallel and vectorial implementations return the same result as long as no overflow occurs during intermediate computation. However, with floating-point arithmetics, rounding errors may occur for every operation. This is the case with floating-point addition that is not associative. Therefore, we could expect a variation in the result between the parallel and vectorial versions of +Rootfix and +Leaffix. For every vertex v, the +Rootfix parallel algorithm performs only as many operations as the vertex has ancestors. The +Rootfix vectorial algorithm performs as many operations as the number of elements in the Euler-tour vector before the $v_)$ position. When using floating-point arithmetics, we can expect the +Rootfix vectorial algorithm to be less accurate than the +Rootfix parallel algorithm. Figure 9 illustrates the difference in the number of operations for both +Rootfix parallel and vectorial algorithms.



(a) Parallel algorithm



(b) Vectorial algorithm

Fig. 9: Different number of operations when performing Rootfix with different algorithms.

To measure the numerical quality of these algorithms, we use the *relative error*, which is a measure of how far is the observed result from the real result. If x is the real result and $\hat{x}$ the observed result, the relative error e is calculated as follows.

$$e = \frac{|\hat{x} - x|}{|x|}$$

Given a problem and an input data, this measure is linked with the algorithm that produces the result and thus can be used to compare algorithms. The measure of the difficulty of a problem independently of the algorithm used to solve it is given by the *condition number*. The condition number is a measure of how much the result of a problem is changed by small variations in the operands. If we consider the addition of n floating-point numbers $x_0, \ldots, x_{n-1}$, the condition number C is defined as follows:

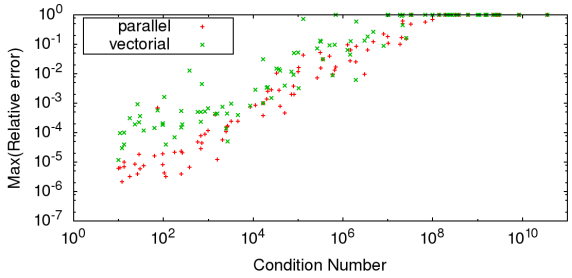
$$C = \frac{\sum_{i=0}^{n-1} |x_i|}{\left|\sum_{i=0}^{n-1} x_i\right|}$$

As a rule thumb, we may lose up to $\lg(C)$ bits of accuracy.

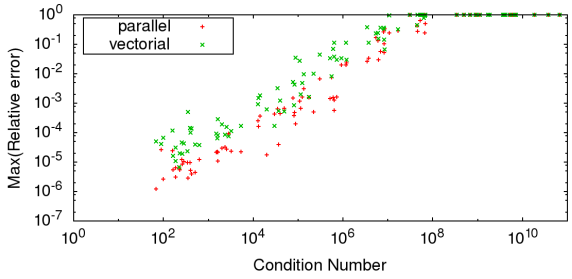
As the result of +Rootfix and +Leaffix is a set of n values, we can use different metrics to quantify the error. We could look at each error individually, the mean error over the n results or the maximum error. In addition, the topology of the tree is impacting the computation scheme and therefore the error. For example, if we consider a linked list (caterpillar), then both +Rootfix and +Leaffix parallel implementations will require a recursive sum of n values with n partial sums. Whereas if we consider a tree with the root and $n-1$ children

(star) then each partial sum generated by +Rootfix will be the result of only one addition.

We choose to evaluate the numerical behavior of both the parallel and vectorial versions of +Rootfix and +Leaffix over a sum of n numbers, which corresponds to a chain of n vertices in a tree. For this set of n numbers we generated 100 random trees of 10.000 nodes with condition numbers from 10 to 10^{10}; then, we measured the relative error of the parallel and vectorial versions of +Rootfix and +Leaffix. We used the algorithm proposed by Ogita *et al.* [19] to generate series of floating-point numbers with a given condition number. We measured the relative error on every node using double-precision to compute the real result and single-precision to compute the observed result. With this measure we captured the numerical behavior of both algorithms on one sum among the n sums that constitute the result. By construction, this is representative of the numerical behavior in function of the condition number of the problem.



(a) +Rootfix maximum relative error.



(b) +Leaffix maximum relative error.

Fig. 10: Related accuracy of parallel and vectorial +Rootfix and +Leaffix algorithms.

Figure 10 shows the maximum relative error as a function of the condition number for the +Rootfix and +Leaffix parallel and vectorial algorithms. We observe that both parallel and vectorial versions of +Rootfix have similar numerical behavior. The large dispersion of points for condition number less than 10^4 may come from the difficulties we had generating vectors with such characteristics. On the other hand, the parallel version of +Leaffix seems better than the vectorial one. It seems that in this case the vectorial version is loosing an extra 2 bits of accuracy compared to the parallel version.

V. Conclusion

In this paper, we have presented two different methods to solve the treefix problem on GPU and compared them. A parallel implementation, that minimizes the number of operations and intermediate storage thanks to load balancing technics and a vector friendly method that involves twice the amount of memory usage and operation than the previous one but exhibit regular computation pattern. We have shown that in terms of performance, regularity is always a better choice over reducing the amount of operations and memory usage. In addition, we have observed that depending on the tree topology, the vectorial implementation is insensitive to it which lead to speed-up factor ranging from 5 to 5000 compared to the load-balancing implementation.

When dealing with floating-point input data, we have seen that the vectorial implementation is introducing rounding error in the final result compared to the parallel implementation. These errors are the consequence of the extra operations and reordering of computations of the vectorial method, which may leads to a 2-bit lost in the worst case. Nevertheless, this accuracy impact has to be formally bounded according to the tree topology, which is planed as future work.

References

[1] 10th dimacs implementation challenge. http://www.cc.gatech.edu/dimacs10/index.shtml, 2012.

[2] The manticore project. http://manticore.cs.uchicago.edu/, 2012.

[3] Scandal project home page. http://www.cs.cmu.edu/ scandal/, 2012.

[4] The university of florida sparse matrix collection. http://www.cise.ufl.edu/research/sparse/matrices/, 2012.

[5] G. E. Blelloch. Prefix sums and their applications. Technical report, Synthesis of Parallel Algorithms, 1990.

[6] M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data parallel haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, DAMP '07, pages 10–18, New York, NY, USA. ACM.

[7] S. Collange, M. Daumas, and D. Defour. Graphic processors to speed-up simulations for the design of high performance solar receptors. In *Application-specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on*, pages 377–382. IEEE, 2007.

[8] Y. S. Deng, B. D. Wang, and S. Mu. Taming irregular eda applications on gpus. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, ICCAD '09, pages 539–546, 2009.

[9] W. M. Fitch. Toward defining the course of evolution: Minimum change for a specific tree topology. *Syst Biol*, 20:406–416, 1971.

[10] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *Proceedings of the 14th international conference on High performance computing*, HiPC'07, pages 197–208, 2007.

[11] M. Harris and M. Garland. *GPU Computing Gems Jade Edition, 1st Edition*, chapter Optimizing Parallel Prefix Operations for the Fermi Architecture. Number 3. MKP, 2011.

[12] M. Harris, S. Sengupta, and J. D. Owens.

[13] K. Hawick, A. Leist, and D. Playne. Parallel graph component labelling with gpus and cuda. *Parallel Computing*, (12):655 – 678.

[14] M. Hussein, A. Varshney, and L. S. Davis. On implementing graph cuts on cuda. *First Workshop on General Purpose Processing on Graphics Processing Units*, 2007/// 2007.

[15] C. Leiserson and B. M. Maggs. Communication-efficient parallel algorithms for distributed random-access machines. *Algorithmica*, 3:53–77, 1988.

[16] L. Luo, M. Wong, and W.-m. Hwu. An effective gpu implementation of breadth-first search. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 52–55, 2010.

[17] D. Merrill, M. Garland, and A. Grimshaw. Scalable gpu graph traversal. *SIGPLAN Not.*, 47(8):117–128, Feb. 2012.

[18] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *26th Symposium on Foundations of Computer Science*, pages 478–489, Portland, Oregon, October 1985. IEEE.

[19] T. Ogita, S. M. Rump, and S. Oishi. Accurate sum and dot product. *SIAM J. Sci. Comput*, 26:2005, 2005.

[20] D. Sankoff. Minimal mutation trees of sequences. *SIAM Journal on Applied Mathematics*, 28(35–42), 1975.

[21] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for gpu computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '07, pages 97–106, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

[22] D. Shirmohammadi, H. Hong, A. Semlyen, and G. Luo. A compensation-based power flow method for weakly meshed distribution and transmission networks. *Power Systems, IEEE Transactions on*, 3(2):753 –762, may 1988.

[23] R. E. Tarjan and U. Vishkin. Finding biconnected componemts and computing tree functions in logarithmic parallel time. In *Proceedings of the 25th Annual Symposium onFoundations of Computer Science, 1984*, SFCS '84, pages 12–20, Washington, DC, USA. IEEE Computer Society.

[24] Z. Wei and J. JaJa. Optimization of linked list prefix computations on multithreaded gpus using cuda. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –8, april 2010.