Optimizing OpenDwarf Applications for CPU and GPU Platforms

Using OpenMP & OpenACC

Thomas C.H. Lux and Kyle Tanous

Computer Science Dept.
Virginia Tech
Blacksburg, VA

***Abstract*— This document outlines the implementation of N-Queens, Breadth First Search, LU-Decomposition, and K-Means as found in the OpenDwarfs [1] repository using both OpenMP and OpenACC. Motivation for this effort arises from the need for benchmark software capable of running easily on multiple platforms in a performant manner. The resulting implementations are similar to their respective OpenCL implementations in terms of computational and communicational behavior, but are optimized to make better utilization of available CPU or GPU resources.**

I. Introduction

Each of N-queens, Breadth First Search (BFS), LU-Decomposition (LUD), and K-means are useful benchmarks for identifying some of the common performance characteristics of parallel computer architectures. As modern trends and new programming paradigms continue to grow, OpenCL remains only one of multiple options for producing parallel codes. In order to ensure that our benchmarks are accurately reporting the performance of a given parallel architecture, it is important to test with near optimal implementations of each parallel benchmark. Yet, it is often useful to run the exact same un-optimized codes on multiple architectures in order to gain greater comparative performance insights.

OpenMP and OpenACC are parallel programming methods that allow programmers to identify optimal parallelization patterns for given problems while leaving compilers to handle the creation of specific parallel codes on any given architecture. Making these four benchmarks available in OpenMP and OpenACC implementations allows for more uniform and effective performance benchmarking of parallel architectures in the future.

The primary contribution of this paper is the proposed addition of seven new applications to the OpenDwarfs suite:

- N-Queens - OpenMP
- BFS - OpenMP & OpenACC
- LUD - OpenMP & OpenACC
- K-Means - OpenMP & OpenACC

[1] github.com/vtsynergy/OpenDwarfs

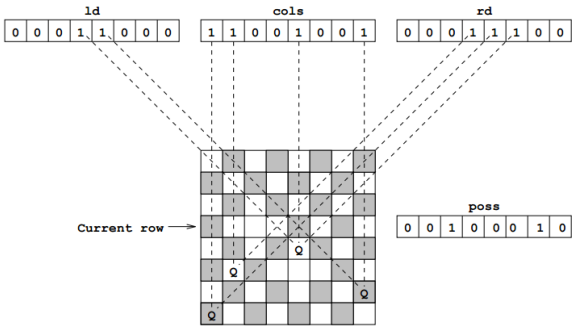

Fig. 1. Illustration of N-Queens solver using masks as in [2]

Additional contributions consist of insights gained during porting of these four dwarfs, especially in terms of mapping OpenCL source code to the OpenMP and OpenACC programming models.

A. N-Queens

The n-queens problem is a well known NP-hard problem. The solution to the n-queens problem consists of finding all possible placements of N queens on a chess board of size NxN where no queen can attack another. This problem captures patterns of backtracking and branch-and-bound, which have seen use in such fields as image processing [1] and bioinformatics.

A common solution to solving the n-queens problem consists of a recursive algorithm to achieve backtracking. In the case of the OpenCL implementation found in the OpenDwarfs suite and the OpenMP/OpenACC implementations outlined in this paper, recursion carries board masks for valid queen placement between levels as outlined in [2].

B. Breadth First Search

Techniques for graph traversal have been the subject of increasing focus with the emergence of data science applications making use of increasingly large-scale graph structures. Efficient techniques for the exploration of such graphs is an important area of focus for such

fields as genomics [3], astrophysics [4], and social network analysis [5].

OpenDwarfs captures the graph traversal paradigm by providing a BFS implementation. BFS algorithms start from the root node and visit all the immediate neighbors. The algorithm continues visiting immediate neighbors in this fashion until the entire graph has been traversed. Such traversal can be used in revealing information about the graph and its vertices (e.g. shortest path between nodes).

C. LU-Decomposition

LUD poses a different computational problem that is more generally placed under the category of dense linear algebra. The actual specification of LUD is straight forward, as well as the resulting implementation. Given a nonsingular square matrix A, find the lower triangular matrix L and upper triangular matrix U such that:

$$A = LU \tag{1}$$

In order to solve this problem, it is sufficient to perform standard gaussian elimination on A, while storing the multiples generated in the gaussian elimination in the lower triangle of the matrix L. Conveniently, this problem only requires the allocation of memory for storing A, as during gaussian elimination the lower triangle of A is converted to 0s. Therefore the lower triangle of A can be used to store L, while the upper triangle of A can be used to store U. This result in the implicit storage pattern:

$$\begin{bmatrix} A_{11} & \cdots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{n1} & \cdots & A_{nn} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ L_{21} & \ddots & & \vdots \\ \vdots & \ddots & & 0 \\ L_{n1} & \cdots & L_{n(n-1)} & 1 \end{bmatrix} \begin{bmatrix} U_{11} & \cdots & \cdots & U_{1n} \\ 0 & \ddots & & \vdots \\ \vdots & & & \vdots \\ 0 & \cdots & 0 & U_{nn} \end{bmatrix} \tag{2}$$

where we only need to store the L_{ij} that are not 0 or 1 and the U_{ij} that are not 0. This computation is parallelizable per-row operation, so for large matrices operations can be performed on each row with complete independence. All operations are floating point operations.

D. K-Means

Similarly to LUD, k-means poses a computational problem capturing the computation patterns of dense linear algebra. K means is an iterative algorithm that attempts to identify a set, S, of k points such that for each k in S:

$$m(k) = \{x_j | k = \min_{k_i \in S} dist(x_j, k_i)\} \tag{3}$$

$$k = \frac{1}{|m(k)|} \sum_{X_j \in m(k)} X_j \tag{4}$$

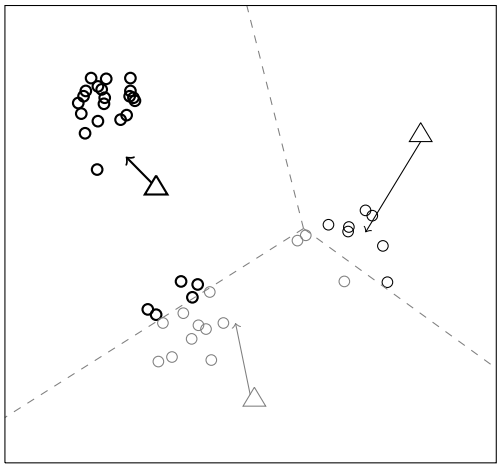

Fig. 2. Example of k-means after initialization. The color of the points (circles) corresponds with the color of the closest cluster center (triangles). The cluster centers are updated to be the mean of their member points, denoted by the arrows extending off of them.

where standard operators over x_i are component-wise, $m(k_i)$ denotes membership of a point x_j to a region defined by a mean value k_i and $dist$ represents the distance between two points. In order to attempt to calculate this set S, the k-means algorithm iteratively assigns x_j to k_i and then updates the location of k_i to be the mean value of all x_j assigned to its region in that step. An illustration of this update can be seen in figure 2. The sets m need to be redefined at each step, and this process repeats for some pre-specified number of iterations. Notice that an arbitrary set of points is not guaranteed to converge definitely on a set S of k means. However, for each step of this algorithm there will be $n \cdot k$ distance calculations performed placing this benchmark squarely in the category of dense linear algebra.

Each iteration of k-means has an independent set of distance calculations, those between each point and each cluster center, that can be performed entirely in parallel. Secondly, after the data point sets for each cluster are produced, the updates to the centers of each cluster are independent as well.

The rest of this paper discusses the motivation for this project, related work, our approach to implementation, and a description of the development, testing, and evaluation environments.

II. Motivation

The OpenCL applications found in the OpenDwarfs suite serve as effective proxy applications for comparative performance of heterogeneous computing platforms by maintaining agnosticism with regard to architecture. This indifference to architecture is obtained by ”unoptimizing” the implemented applications to the point of

not favoring any specific computing platform more than another. Unfortunately, it is this characteristic of OpenDwarfs applications that prevents them from achieving optimal performance and resource utilization between platforms.

Implementation of these applications in OpenMP and OpenACC provides the opportunity to reveal potential performance capabilities of, specifically, CPUs and GPUs. This ability to explore potential performance in terms of specific architectures serves as a valuable supplement to the comparative capabilities of the addressed dwarfs' OpenCL counterparts.

III. Related Work

Thorough exploration has been performed in terms of parallelizing the dwarfs captured by these four applications. Parallelization of the backtracking and branch-and-bound paradigms captured by the n-queens problem has undergone extensive investigation [6], [7] as a result of its practicality in fields such as bioinformatics and image processing. The n-queens problem itself has also been investigated as a means of comparing and evaluating different parallel platforms [8], [9], [10].

Breadth first search captures the graph-traversal paradigm. Similarly to n-queens, BFS has been explored in terms of possible optimizations [11] and as a tool for evaluating parallel programming models [12].

LU-Decomposition and its proposed parallel implementations have existed for some time [13]. Nevertheless, its fundamental importance remains present in modern computing applications. Image processing, computer security, graph processing, dictionary construction, and video text extraction are just a few modern applications that are computationally intensive and heavily utilize the LU-Decomposition. [14], [15], [16], [17], [18] This linear algebra tool will remain a pivotal one for the foreseeable future of computer applications.

K-Means is a heavily studied benchmark for parallel computing. [19], [20], [21], [22], [23]. It is a valuable and generically useful tool for large scale data mining, image segmentation, document retrieval, and pattern classification [24]. Due to the wide set of uses, its importance as a generic parallel computation problem is well accepted. The type of computation done conveniently fits into the larger category of dense linear algebra, because of the intense utilization of vector operations.

Along with studies of parallelization, there also exists literature comparing different parallel processing technologies with specific regard to the applications considered in this paper [25], [26]. These will serve as a baseline for expected performance, but should be superseded by the addition of architecture-specific optimizations. With regard to OpenACC, its performance has previously been demonstrated to lack portability of performance due to high levels of diversity among architectures [27].

With regard to implementation itself, literature exists that presents techniques for fine tuning OpenMP and OpenACC compiler directives [28]. This work informed the optimization process for the OpenMP and OpenACC application implementations.

IV. Approach

Generally, porting of the OpenCL dwarfs consisted of analyzing data management and computational patterns. OpenCL kernels were charactized in terms of communication between the host and the target computing architecture (in the case of execution on a GPU). First, simple sequential versions of the algorithms used in OpenDwarfs n-queens, BFS, LUD, and k-means codes are implemented in standard C++. That is, these versions retain similar runtime behavior and computational structure with regard to their respective OpenCL implementations.

For OpenMP implementations, C++ source code is augmented with `pragmas` that ensure optimal utilization of CPU resources. Choice and location of OpenMP directives are informed by computational intensity, control flow, and data requirements such that CPU cores maintain high levels of throughput throughout the course of execution. This opposed to placement of OpenACC `pragmas`, whose application is informed by the prioritization of potential SIMD operations and minimal memory transfers between the host and accelerator (i.e. GPU).

A. N-Queens

The n-queen solver algorithm as seen in OpenDwarfs can be summarized as follows:

1) Place queens row wise, starting from bottom most row
2) If all queens have been placed:
   - Increment solution counter
3) Else:
   a) Try all columns in the current row
   b) If queen can be placed safely:
      - Recursively find solutions
   c) Else:
      - Backtrack and follow a different branch

The n-queens solver continues in this fashion until it has exhaustively discovered all possible solutions.

The OpenMP implementation of the n-queen solver parallelizes each branching instance of the solution finder from the first row. Dynamic scheduling is used to accomodate differing numbers of solutions resulting from placement of a queen on each column along the starting row. After being assigned a branch to search, a

thread recursively explores potential solutions using bit masks as illustrated in Figure 1.

B. Breadth First Search

Breadth first search is a common algorithm for traversing and searching graph structures. The proposed OpenMP and OpenACC implementations employ highly similar computational patterns, making use of a similar set of mask structures used in recording node travel costs. The BFS algorithm as implemented in OpenDwarfs can be summarized as follows:

1) Initialize structures:
   - List of nodes containing edge count and index
   - Mask for nodes being updated in the current iteration
   - Mask for nodes being updated in the next iteration
   - Mask for visited status of each node
   - List of node travel costs
2) Do until travel costs for all nodes are determined:
   a) For all unvisited nodes updating this iteration (beginning with only root node):
      - Calculate travel cost
      - Flag for update in next pass
   b) For all nodes previously flagged for update
      - Flag for update in next iteration of do loop
      - Mark as visited

The output of this BFS implementation is a list of the lowest travel cost (i.e. shortest route) for every node in the graph with respect to the graph's root.

The OpenMP and OpenACC implementations of this algorithm parallelize both `for` loops. OpenMP directives employ static scheduling despite varying workloads (i.e. neighboring nodes) per iteration, as overhead costs associated with dynamic scheduling outweighed performance increases for graphs with up to 1,000,000 nodes in size.

The OpenACC implementation performs a singular transfer of all initialized data structures and masks from the host CPU to the target GPU. A stop flag is synchronized between the host and target throughout execution of the second `for` loop. Finally, the calculated travel costs for each node are copied from the target to the host. OpenACC kernels are generated for each `for` loop with identical placement compared to OpenMP `pragmas` and take advantage of the highly vectorizable loop operations. Generally, the OpenMP implementation aims to keep cores busy as often as possible, while the OpenACC implementation aims to employ the high vectorization capabilities of GPUs.

C. LU-Decomposition

The algorithm can be summarized as follows:

1) Initialize a square data matrix A
2) For each diagonal element, starting at the top left:
   a) Subtract whatever multiple of the current row from the rows beneath, such that the entire column underneath the n^{th} diagonal becomes 0.
   b) Store the multiples used for each row in place of the 0's in order to reduce storage requirements.
3) Output the lower triangle of A as L, being sure to include the implicit diagonal of 1's, then output the upper triangle as U.

The OpenMP implementation parallelizes loop 2 in the above steps. Given a starting element along the diagonal, each row below can be managed by one thread with full data independence. This however does pose the interesting problem that as more of the matrix L is computed, there are fewer rows beneath the current diagonal element and hence there is less opportunity for parallelism.

The OpenACC implementation refers to the same structure of parallelism, but with the additional copy in and copy out of the matrix to device memory. Compared to the computation time of lud, this data copy should be relatively minimal.

D. K-Means

The algorithm can be summarized as follows:

1) Initialize a set of points that need to be clustered, and randomly initialize the k means to be values in the range of the input points.
2) For each iteration (until max iteration):
   a) Assign each point x_i in the input data set to the one k_i that it is closest to by calculating the distance between that x_i and all k.
   b) Recalculate the new value for each k_i to be the mean of all points from the input data that were assigned to it.
3) Display the k means that have been computed if the user requested they be shown.

The OpenMP implementation of k-means takes advantage of the innate parallelism in calculating distances between points. This means that there is a `pragma` over the for loop that computes step *2a*. The closest clusters to each point can be tracked in an integer array with the same length as the number of points. Each iteration of the loop calculates the distance between one point and each of the clusters, tracking which one is closest. It is also possible to add another parallel section that updates the centers of each cluster, but that is only parallelizable into the number of clusters that are being calculated. For average numbers of clusters (20 or fewer), it is not worth the overhead of parallelizing part *2b*.

The OpenACC `pragma` exhibit similar parallelism, with the primary parallel loop being over the distance calculations between points and clusters. However, since OpenACC works on devices with their own memory, it is necessary that before and after the outer loop (*2*), that the cluster centers and data points be transferred to device memory. That involves an additional data `pragma`, as well as necessary data updates inside of step *2*.

V. Evaluation

Performance of all implementations of the four dwarfs addressed in this paper (N-Queens, BFS, LUD, K-Means) were evaluated on Virginia Tech's Ironclaw machines. Sequential and parallel execution times of OpenMP implementations are compared to OpenCL execution times as reported when being executed on Ironclaw's Intel Xeon E5-2630v3 using 16 cores. OpenACC execution times are compared to OpenCL as reported when being executed on Ironclaw's AMD FirePro W8000 model GPU. Performance metrics are recorded as the mean result of of ten executions. Execution times for OpenMP and OpenAcc implementations are measured using wall clock time while OpenCL employs its built-in profiling methods.

VI. Results

Following are the performance times and observed speedups of the OpenMP, OpenACC, and OpenCL implementations of the four addressed dwarfs with respect to sequential CPU execution. Performance results for n-queens, BFS, LUD, and k-means can be seen in figures 3, 4, 5, and 6 respectively. CPU performance is reported as raw execution time; GPU execution times are split into three components:

- Kernel execution time (GPU Compute)
- Memory transfer times (GPU Mem)
- Unaccounted

Unaccounted time is the remainder of time not characterized by kernel execution or memory management, but is still present in the total execution time of an application.

VII. Conclusions and Discussion

In this paper we present 4 new dwarfs implemented in OpenMP and OpenACC that are capable of benchmarking a wide variety of parallel computing platforms as well as quickly being tuned to peak performance via minimal architecture aware optimizations. The OpenMP implementation of the n-queen solver realized a slight speedup compared to its OpenCL counterpart. In the case of BFS, its OpenMP implementation achieved a slight speedup with respect to sequential execution. This opposed to the OpenCL implementation experiencing slowdown when executed on the CPU. The reported

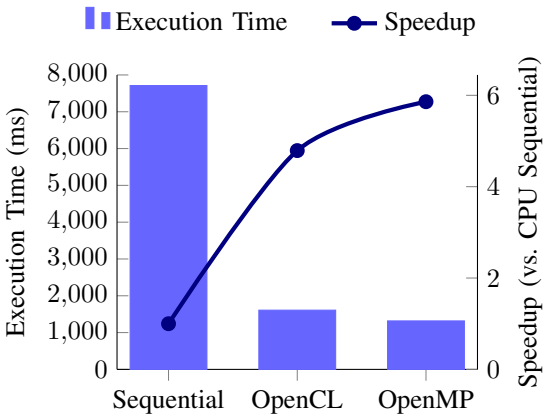

Fig. 3. N-Queens Execution Times and Speedup (N=16)

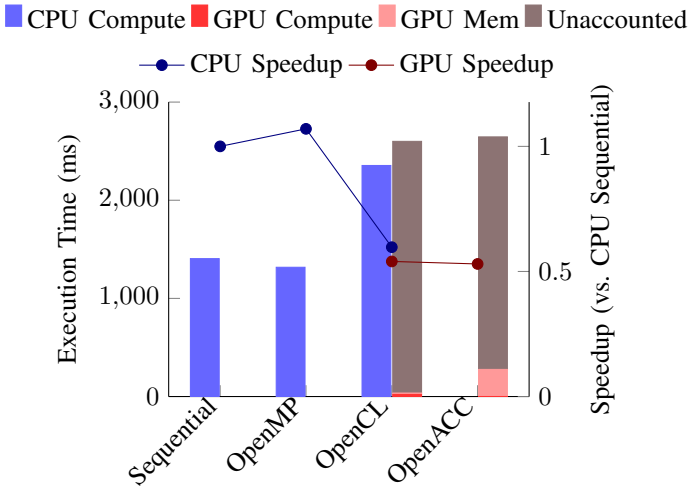

Fig. 4. BFS Execution Times and Speedup (N = 1,000,000)

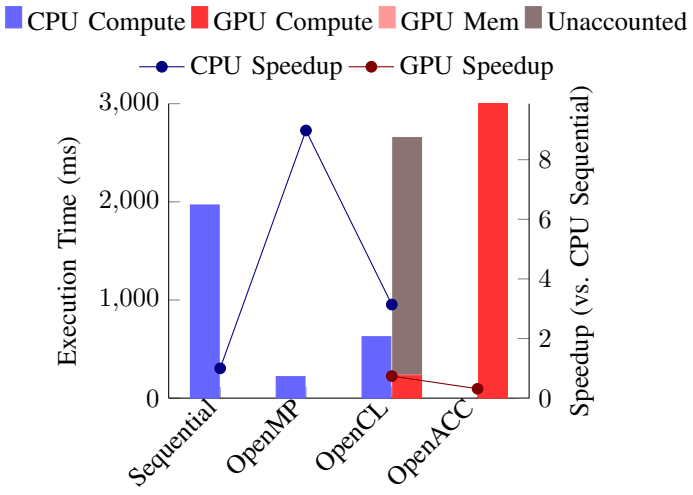

Fig. 5. LUD Execution Times and Speedup (N = 2,048)

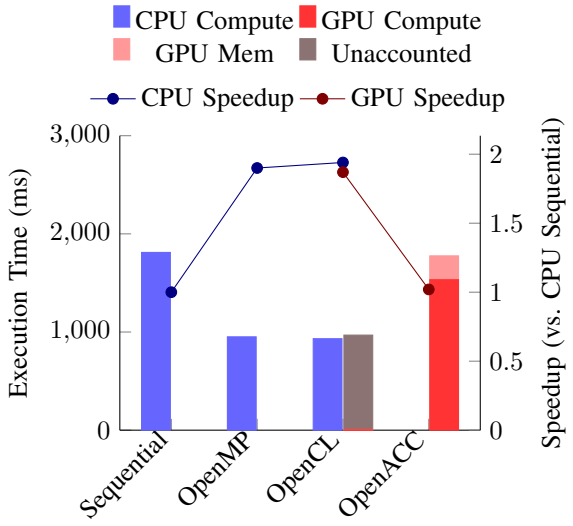

Fig. 6. K-Means Execution Times and Speedup (N = 10,000 x 200)

times for BFS kernel execution on the GPU were minuscule for both OpenCL and OpenACC, but OpenACC outperformed OpenCL (0.28ms and 20.98ms respectively). However, OpenACC reported much higher data transfer operations than OpenCL (14.83ms and 276ms). The OpenACC implementation of LUD significantly underperformed in comparison to the OpenCL implementation both in terms of memory management and kernel execution time. Some testing revealed that this underperformance resulted from poor distribution of the input matrix throughout device memory by OpenACC, making all array accesses extremely expensive. The OpenMP implementation of LUD however experienced a significant speedup over the CPU parallel OpenCL. Finally, for K-Means, the OpenMP and OpenCL CPU codes were almost identical in performance while once again OpenACC struggled to efficiently execute on the GPU. Much of the inefficiency in the execution of OpenACC codes appears to lay in its poor memory locality.

OpenMP produced noticeable speedup for three of the four applications and required significantly simpler source code. This simplicity is achieved primarily by means of eliminating the necessity for explicit memory management statements. Such statements are required by the OpenCL applications found in OpenDwarfs to retain architecture agnosticism. Thus, OpenMP implementations typically only required simple parallel loop directives with occasional use of reductions.

OpenACC achieved faster kernel execution times when compared to their OpenCL counterparts but suffered in terms of memory management. This is not necessarily surprising, as this issue has been addressed before [27].

VIII. Future Work

The OpenCL dwarfs repository still has many more applications that would be useful if implemented in OpenMP and OpenACC. While the benefits of using these simple and common parallel computing frameworks are evident, there is still only a small set of publicly available benchmarks. Future work will likely include further extending the available set of dwarfs conforming to the OpenMP and OpenACC standards. This will allow for more accessible method for architecture-specific testing across a wide variety of common computing devices (i.e. CPUs and GPUs).

Acknowledgement

This work is the culmination of the final project assignment for the Computer Science graduate course CS5234: Advanced Parallel Computation, at Virginia Polytechnic Institute and State University.

References

[1] B. L. Stojkoska, D. P. Davcev, and V. Trajkovik, "N-queens-based algorithm for moving object detection in distributed wireless sensor networks," *CIT. Journal of Computing and Information Technology*, vol. 16, no. 4, pp. 325–332, 2008.

[2] M. Richards, "Backtracking algorithms in mcpl using bit patterns and recursion," Citeseer, Tech. Rep., 1997.

[3] Y. Zheng, J. D. Szustakowski, L. Fortnow, R. J. Roberts, and S. Kasif, "Computational identification of operons in microbial genomes," *Genome research*, vol. 12, no. 8, pp. 1221–1230, 2002.

[4] J. VanderPlas, A. J. Connolly, Ž. Ivezić, and A. Gray, "Introduction to astroml: Machine learning for astrophysics," in *Intelligent Data Understanding (CIDU), 2012 Conference on*. IEEE, 2012, pp. 47–54.

[5] W. Chen, Y. Wang, and S. Yang, "Efficient influence maximization in social networks," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2009, pp. 199–208.

[6] T. G. Crainic, B. Le Cun, and C. Roucairol, "Parallel branch-and-bound algorithms," *Parallel combinatorial optimization*, pp. 1–28, 2006.

[7] J. Jansen and F. Sijstermans, "Parallel branch-and-bound algorithms," *Future Generation Computer Systems*, vol. 4, no. 4, pp. 271–279, 1989.

[8] T. J. Rolfe, "A specimen mpi application: N-queens in parallel," *ACM SIGCSE Bulletin*, vol. 40, no. 4, pp. 42–45, 2008.

[9] V. Aggarwal, I. A. Troxel, and A. D. George, "Design and analysis of parallel n-queens on reconfigurable hardware with handel-c and mpi," in *2004 MAPLD Intl. Conference*, 2004.

[10] K. Thouti and S. Sathe, "Comparison of openmp & opencl parallel processing technologies," *arXiv preprint arXiv:1211.2038*, 2012.

[11] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek, "A scalable distributed parallel breadth-first search algorithm on bluegene/l," in *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*. IEEE, 2005, pp. 25–25.

[12] J. J. Fumero, "A mpi back-end for the openacc accull. exploiting openacc semantics in message passing clusters," 2012.

[13] J. G. van de Vorst, "The formal development of a parallel program performing lu-decomposition," *Acta Informatica*, vol. 26, no. 1, pp. 1–17, 1988.

[14] Q. Su, G. Wang, X. Zhang, G. Lv, and B. Chen, "A new algorithm of blind color image watermarking based on lu decomposition," *Multidimensional Systems and Signal Processing*, pp. 1–20, 2017.

[15] S. Choi, K. Kim, and H. Youn, "A novel key pre-distribution approach for high security and efficiency using lu-decomposition of matrix," *International Journal of Ad Hoc and Ubiquitous Computing*, vol. 23, no. 3-4, pp. 168–181, 2016.

[16] C. Ren, L. Mo, C. Kao, C. Cheng, and D. Cheung, "Clude: An efficient algorithm for lu decomposition over a sequence of evolving graphs," in *Advances in Database Technology-EDBT 2014: Proceedings of the 17th International Conference on Extending Database Technology*. OpenProceedings., 2014.

[17] A. Rotbart, G. Shabat, Y. Shmueli, and A. Averbuch, "Randomized lu decomposition: An algorithm for dictionaries construction," *arXiv preprint arXiv:1502.04824*, 2015.

[18] P. Sudir and M. Ravishankar, "Curved videotext detection and extraction: Lu-decomposition and maximal h-transform based method," in *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014*. Springer, 2015, pp. 675–683.

[19] K. Stoffel and A. Belkoniene, "Parallel k/h-means clustering for large data sets," *Euro-Par99 Parallel Processing*, pp. 1451–1454, 1999.

[20] S. Tzeng, B. Lloyd, and J. D. Owens, "A gpu task-parallel model with dependency resolution," *Computer*, vol. 45, no. 8, pp. 34–41, 2012.

[21] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii, "Scalable k-means++," *Proceedings of the VLDB Endowment*, vol. 5, no. 7, pp. 622–633, 2012.

[22] M.-F. F. Balcan, S. Ehrlich, and Y. Liang, "Distributed k-means and k-median clustering on general topologies," in *Advances in Neural Information Processing Systems*, 2013, pp. 1995–2003.

[23] S. Kantabutra and A. L. Couch, "Parallel k-means clustering algorithm on nows," *NECTEC Technical journal*, vol. 1, no. 6, pp. 243–247, 2000.

[24] W. Zhao, H. Ma, and Q. He, "Parallel k-means clustering based on mapreduce," in *IEEE International Conference on Cloud Computing*. Springer, 2009, pp. 674–679.

[25] C. L. Ledur, C. M. Zeve, and J. C. dos Anjos, "Comparative analysis of openacc, openmp and cuda using sequential and parallel algorithms," in *11th Workshop on parallel and distributed processing (WSPPD)*, 2013.

[26] M. Sugawara, S. Hirasawa, K. Komatsu, H. Takizawa, and H. Kobayashi, "A comparison of performance tunabilities between opencl and openacc," in *Embedded Multicore Socs (MC-SoC), 2013 IEEE 7th International Symposium on*. IEEE, 2013, pp. 147–152.

[27] A. Sabne, P. Sakdhnagool, S. Lee, and J. S. Vetter, "Evaluating performance portability of openacc," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2014, pp. 51–66.

[28] C. Bonati, E. Calore, S. Coscetti, M. D'Elia, M. Mesiti, F. Negro, S. F. Schifano, G. Silvi, and R. Tripiccione, "Design and optimization of a portable lqcd monte carlo code using openacc," *arXiv preprint arXiv:1701.00426*, 2017.