DLINE JOURNALS

Journal of Information & Systems Management

Print ISSN: 2230 – 8776

JISM 2024; 14 (1)

https://doi.org/10.6025/jism/2024/14/1/1-7

# BDD Package Implementation for Computational Distribution in CPU And GPU

Miloš M. Radmanovic[1], Dušan B. Gajic[2]
The Faculty of Electronic Engineering
Aleksandra Medvedeva 14, 18000 Niš
Serbia
{milos.radmanovic@gmail.com}
{dusan.b.gajic@gmail.com}

Received: 19 October 2023
Revised: 9 December 2023
Accepted: 16 December 2023


## ABSTRACT

*Binary Decision Diagram (BDD) construction and manipulation is an important part of CAD tasks. One of the ways to improve BDD package performance is to perform certain BDD operations in parallel with the GPU. The recent development of GPU frameworks for general-purpose programming, such as OpenCL or Nvidia CUDA, has made GPUs a very powerful and attractive option for developing high-performance numerical applications. This paper proposes an efficient implementation of the BDD package that distributes computational workloads over CPUs and GPUs. This implementation takes advantage of various parallelism sources found in the BDD package. The experimental results demonstrate that implementing this solution results in significant computational speedups.*

**Keywords:** *Binary Decision Diagrams, BDD Package, Parallel Implementation, Graphics Processing Unit, GPU Computing*

## 1. Introduction

*Binary Decision Diagrams (BDDs) are the dominant data structure for representing Boolean functions in CAD applications. The application of BDDs is further extended with their use in various areas of computer science and engineering. In practice, the success of BDD representations depends on the ability to efficiently manipulate large BDDs. Therefore, considerable research has been conducted in order to develop more efficient implementations of BDD algorithms [1-5].*

*BDD algorithms are usually built on top of BDD packages. Many BDD package implementations have been developed in a variety of programming languages and most of them are freely available as public domain on the Internet. The choice of a BDD package for a certain application is typically guided by the following package characteristics: functionality, software interface, robustness, reliability, portability, support, and performance. In most cases, the performance of a BDD package is of major concern. Parameters which influence the performance of a BDD package include the choice of the programming language and the software and hardware platforms, BDD node structure, type of garbage collection, unique and operation hash table strategies [6-8].*

*Parallel computing can be used to efficiently solve large scale problems, either by distributing computational loads among processors or by utilizing the large memory in parallel networked workstations. Parallel processing of BDDs can be used both to reduce the BDD algorithm running time and to extend the memory limitations which exist in the traditional single-processor sequential computing.*

*In order to increase the performance of BDD packages, the concept of parallelism has been introduced to the BDD representations and algorithms in several papers. In [9], a parallel algorithm for the construction of BDDs is described. The algorithm is motivated by the fact that the construction of a BDD, for certain large or particularly complex Boolean functions, can be a very time-consuming task. In order to overcome limitations of computational resources, research in [10] presents an approach which distributes the BDD data structure across multiple networked workstations. Further, several techniques are introduced which allow parallelization of depth-first search algorithms on a BDD. Reference [11] presents a parallel algorithm for BDD construction targeted at shared memory multiprocessors and distributed shared memory systems. The results obtained using a shared memory multiprocessors system show speedups of over 2×, with four processors, and up to 4×, with eight processors. Alongside the research on parallel BDD construction, various BDD algorithm parallel implementations were developed for networks of workstations [12-14]. In [15], some key algorithms for performing BDD operations are first described and, afterwards, an approach to their parallelization is described, with a goal to achieve efficient execution of BDD packages on multicore CPUs. The technique of general purpose computing on the GPU (GPGPU) enables parallel processing of non-graphics algorithms using graphics hardware. Only recently, the possibility of using GPUs to solve complex problems in logic design has been explored by researchers, for example in [16-20].*

*Motivated by the existing research on efficient execution of parallel BDD operations on multicore CPUs and possibility of using GPUs, in this paper we propose an efficient implementation of a BDD package using the GPU platform. The proposed implementation exploits the various sources of parallelism that exist in BDD packages. We address several topics considering parallel computations in BDD packages and present their mapping to the GPU architecture. The experimental results confirm that the application of the proposed implementation of a parallelized BDD package leads to significant computational speedups over traditional C/C++ implementations processed on CPUs.*

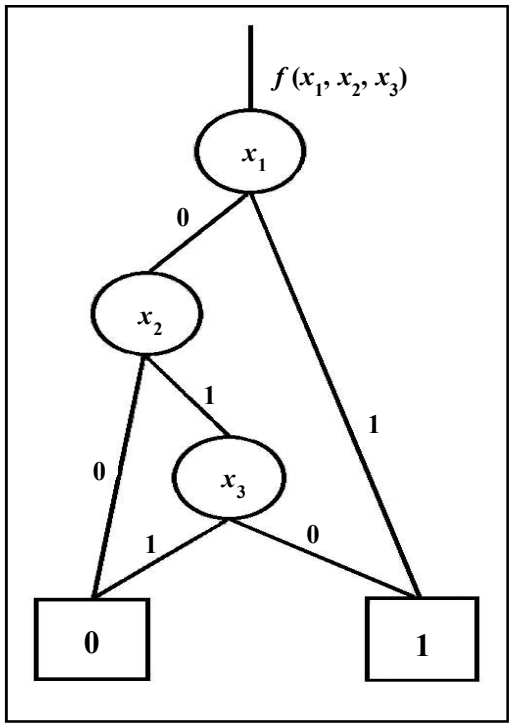


**Figure 1. BDD Representation of the Function Defined by the Truth Table $F = [00101111]^T$**

*The paper is organized as follows: Section 2 shortly introduces the BDD representation of Boolean functions. Section 3 describes the structure of a BDD package and the basic BDD algorithms. Section 4 presents the GPU as a computing platform. Section 5 discusses the operations in the BDD package for which we introduce the GPU processing. Section 6 shows experimental results obtained with the proposed implementation. Finally, Section 7 offers some concluding remarks and directions for future work.*

## 2. Binary Decision Diagrams

*BDDs consist of non-terminal (decision) nodes, 0-edges and 1-edges attached to all non-terminal nodes, a '0' terminal node, and a '1' terminal node, as shown in Figure 1. The non-terminal node with no upper nodes is called a root node. As it can be seen from Figure 1, a variable is related to every non-terminal node, such that every path from the root node to one of the terminal nodes respects the same variable ordering.*

*A Boolean function can be converted into an equivalent function by performing Shannon expansion based on the fixed variable ordering. This new function can be represented by a binary tree. The corresponding BDD is constructed from this binary tree by applying the two reduction rules (redundant node elimination and equivalent sub-graphs sharing). The Boolean operations such as the logical AND, logical OR, etc., can be achieved by using BDD manipulations, which have an average time complexity propositional to the size of BDDs. It is well known that the size of the BDD for a given Boolean function depends on the variable order for the function. The strength of BDDs is that they can represent Boolean function data with high level of redundancy in a compressed form.*

## 3. BDD Packages

*BDD packages are deployed in many software tools, particularly in the area of logic design, and they typically deal with the following common implementation features [1]. A BDD package has three main components [21]:*

- *The BDD algorithm component,*
- *Dynamic variable reordering component,*
- *Garbage collection component.*

*The BDD algorithm component builds the result BDDs for various Boolean operations. The implementation of these algorithms is typically based on the BDD node data structure, unique and operation tables, and depth-first BDD traversal.*

*The decisions made in defining the BDD node data structure have impact on memory space requirements for storing node objects. There are many choices for defining a BDD node object, but every node usually contains: an id, then cofactor, else cofactor, next pointer, and reference counter [22]. The BDD construction is based on applying the traversal in a depth-first manner.*

*The maintance of a BDD representation is improved by storing BDD nodes in a dictionary, called the unique table. The unique table maps a unique triple of (v, g, h) for a BDD node, where v is the variable identifier, g is the node connected to the "1" edge, and h is the node connected to the "0" edge. The unique table is a hash table with the hash collisions resolved by chaining. A hash function is applied to the triple to obtain the index in the unique table of the start of a collision chain of nodes. Comparing the unique triple against the nodes in the collision chain addresses the look up.*

*The efficient implementation of almost all recursive BDD manipulation algorithms is made possible by the operation table. This table is also implemented as a hash table with the collisions resolved by chaining. The collision lists can be kept sorted to reduce the number of memory accesses required on average for the lookup. Table sizes which are prime numbers require an expensive modulo operation. Table sizes that are a power of 2 are often better handled by memory management.*

*As the variable ordering can have significant impact on the size of a BDD, dynamic variable reordering component is a fundamental part of all modern BDD packages. Dynamic variable reordering algorithms are generally based on the shifting algorithm [23]. The BDD variable order changes by exchanging nodes in one level with nodes in the neighbouring levels. Dynamic variable ordering should best be invoked as an asynchronous process that can be activated at any time during the BDD manipulation. Dynamic variable ordering is a complex problem since finding an optimal ordering is NPhard. Futher, small changes in the BDD ordering may have significant impact on both the space and time requirements.*

*BDD computations are memory intensive, especially when large BDDs are involved. They not only require a lot of memory, but also frequent accesses to many small data structures. Furthermore, many intermediate BDD results are created to arrive at a resulting BDD. These computations may have poor memory handling, as there is not a solution to ensure that the accessed BDD nodes are close in memory. It is important to have a garbage collector component [24] to automatically remove BDD nodes that are no longer useful. In modern BDD packages, garbage collector component is based on reference counting and the recycling of nodes for later reuse. Garbage collection is activated when the percentage of the unusable BDD nodes reaches a threshold. Unusable BDD nodes are nodes with zero reference counts. Some of unusable BDD nodes may become usable again (recycled) if they are obtained as results of new sub problems. Thus, in the case when BDD nodes change state between „usable" and „unusable" frequently, garbage collection can reduce the benefit of the operation tables and decrease the overall performance of a BDD package.*

## 4. The GPU Architecture and GPGPU

*Processor frequency progress, which followed the Moore's law for more than four decades, reached a limit in 2003, mostly due to the inability to further solve the problems of heat dissipation and energy consumption. Since then, there are two approaches in the development of computer architectures. The multicore approach, typical for CPUs, seeks to maintain the execution speed of sequential programs while moving into multiple cores. In contrast, the many core approach, found in GPUs, focuses more on the execution throughput of parallel applications. This resulted in a rapid evolution of GPU architectures. The GPU evolution started from fixed-function hardware specialized for rendering computer graphics, which first appeared in 1999, and developed into a massively parallel, scalable, and fully programmable platform characterized by exquisite memory bandwidth and computational power. Due to this, many of the general purpose applications which were processed on CPUs are now re-implemented in order to efficiently harness the GPU resources. For more details on recent changes that made GPGPU possible, see [25, 26, 27, 28].*

*The GPU parallel processing model is based on a large number of processor cores which can directly address into a global GPU memory. The GPU architecture follows the single program, multiple data (SPMD) paradigm [26, 27], features a multi-level memory hierarchy and has simple branching circuits. In SPMD computing, a large number of threads execute in parallel the same function, called a kernel, over different data.*

*Application Programming Interfaces (APIs) most often used for the development of GPGPU programs are Nvidia's CUDA and Open Computing Language - OpenCL. CUDA is a vendor-specific development framework and only supports execution on Nvidia's GPU hardware. Therefore, we give advantage to OpenCL which is hardware agnostic. Further, the OpenCL C programming language, included in the framework, allows development of programs that are both accelerated and portable across a wide set of devices (CPUs, GPUs, Field Programmable Gate Arrays (FPGAs), Digital Signal Processors (DSPs), Cell processors, embedded processors) [28].*

## 5. GPU Acceleration in The BDD Package

*Motivated by the existing work on the parallelization of components in BDD packages, described in Section 1, we explored various sources of parallelism that exists within the algorithms included in BDD packages in order to develop an efficient model of parallel BDD operations on GPUs. The components of the BDD package that take advantage of the GPU processing in our present solution are the BDD algorithm and the garbage collection components.*

*The effectiveness of caching within unique and operation tables of the BDD algorithm component*

*strongly influences the number of subproblems generated in the BDD algorithm task execution.Thus, the hash tables (unique and operation tables) in a BDD package need to support concurrent execution of the hash operation lookup_insert(key). This operation is a crucial component of the Apply procedure which is central to the BDD construction and manipulation [1]. The lookup_insert operation returns the key, if it already exists in the hash table, or, otherwise, inserts the key. Reference [15] shows how this operation can be safely executed in parallel on multicore processors. The lookup_insert operation within the BDD algorithm component in our BDD package is, therefore, implemented as an OpenCL kernel which performs the same function over different keys. Since GPUs use hardware multithreading [25, 26, 27], this automatically allows simultaneous execution of as many lookup_insert operations as there are active GPU threads.*

*The effectiveness of garbage collection component can have significant impact on both space and time requirements of a BDD package. When garbage collector removes unusable BDD nodes, the unique and the operation table entries that reference these nodes must also be removed to eliminate unusable references. If garbage collection is not invoked frequently enough, the memory usage can be greatly increased. An OpenCL kernel for garbage collection is developed so that each GPU thread removes an entry from the hash tables. Since thousands of GPU threads can be active at the same time, this leads to a massively-parallel GPU garbage collection. The transfer of the garbage collection task to the GPU, also allows the CPU to be free to perform other tasks for which it may be more suitable.*

## *6.* Experimental Results

*In this section, we compare the performance of our GPU accelerated BDD package implementation, which incorporates the before-mentioned OpenCL kernels, and a single-threaded C/C++ implementation of the BDD package on the CPU. For the comparison, we use a set of well-known standard benchmarks. Table I presents a view on the performance of the BDD package computations performed in the basic BDD construction algorithm on the CPU and the GPU.*

| Benchmark | in / out / cubes | CPU | GPU |
|---|---|---|---|
| *alu4* | 14 / 8 / 1028 | 0.15 | 0.08 |
| *apex1* | 45 / 45 / 206 | 5.18 | 0.81 |
| *apex2* | 39 / 3 / 1035 | 3.31 | 0.62 |
| *apex5* | 117 / 88 / 1227 | 0.30 | 0.17 |
| *cordic* | 23 / 2 /1206 | 0.06 | 0.04 |
| *cps* | 24 / 109 / 654 | 0.15 | 0.09 |
| *misex2* | 25 / 18 /29 | 0.05 | 0.03 |
| *misex3* | 14 / 14 / 1848 | 0.03 | 0.02 |
| *table3* | 14 / 14 / 135 | 0.02 | 0.02 |
| *table5* | 17 / 15 / 158 | 0.01 | 0.01 |

**Table 1. Comparison of the BDD Construction Times for the BDD Package on the CPU and the GPU**

*The test platform features an Intel i7-920 quad-core processor, operating at 2.66 GHz, and has 4 GBs of DDR3- 2000 RAM. GPU that is used is an Nvidia GeForce GTX 560Ti with 1GB of GDDR5 RAM, composed of 384 streaming processors. The OpenCL kernels are developed using the AMD Accelerated Parallel Programming SDK 2.6.*

*The size of the unique and the operation tables is limited to 8191 entries. The garbage collection*

*is activated if the hash tables exceed the 80%-full marker. All benchmarks are used in the Espresso-mv or pla format [29] and the computation times are reported in seconds.*

*As it can be seen from Table I, the addition of the GPU acceleration to the BDD package brings clear performance benefits over the CPU-only solution. The speedup, in terms of the BDD construction algorithm computation time, for most of the cases is substantial and varies from 6.4× to 1.5×. However, it should be noted that the speedup may not be achieved in some cases, e.g., in the case of the benchmark table3, because the construction of the BDD in this case is not enough computationally-intensive to benefit from the introduction of the GPU.*

## 7. Conclusion

*This paper proposes an implementation of a BDD package which uses the GPU hardware for the acceleration of certain data-parallel operations. The proposed implementation exploits several sources of parallelism that exist within BDD packages. In particular, we discuss the parallel OpenCL implementation of the lookup_insert hash operation, which is of central importance to the BDD algorithm component, and a GPU-accelerated garbage collection component. The experimental results confirm that the application of the proposed implementation, which distributes the BDD package operations over the CPU and the GPU, leads to significant computational speedups. The results presented in the paper may also be helpful in the general study on improvement of BDD packages. Since these first research results look promising, further work on this topic will be devoted to the extension of the GPU acceleration method to the implementation of other operations that are common in the components of BDD packages.*

## References


[1] Brace, K., Rudell, R., Bryant, R. (1990). Efficient implementation of a BDD package. *In Proc. Design Automation Conf*. (pp. 40-45).

[2] Sangavi, J., Ranjan, R., Bryton, R., Sangiovanni-Vincentelli, A. (1996). High performance BDD package based on exploiting memory hierarchy. *In Proc. of the Design Automation Conf*.

[3] Long, D. (1998). The design of cache-friendly BDD library. *In Proc. 1998 IEEE/ACM Intl. Conf. on CAD* (pp. 639-645).

[4] Janssen, G. (2001). Design of pointerless BDD package. 10th Int. Workshop on Logic & Synthesis, Lake Tahoe, USA.

[5] Ebendt, R., Gorschwin, F., Drechsler, R. (2005). Advanced BDD Minimization. Springer, New York.

[6] Janssen, G. (2003). A consumer report on BDD packages. *In Proc. 16th Symp. Integrated Circuits and Systems Design* (pp. 217-223).

[7] Somenzi, F. (2001). Efficient manipulation of decision diagrams. *Int. J. Software Tools for Technology Transfer (STTT)*, 3(2), 171-181.

[8] Sentovich, M. (1996). A brief study of BDD package performance. *In: Proc. of the Formal Methods on CAD* (pp. 389-403).

[9] Kimura, S., Clarke, E. M. (1990). A parallel algorithm for constructing binary decision diagrams. *In IEEE Intl. Conf. on Computer Design: VLSI in Computers and Processors* (pp. 220-223).

[10] Stornetta, T., Brewer, F. (1996). Implementation of an efficient parallel BDD package. In Proc. of Design Automation Conf., Las Vegas, USA (pp. 641-644).

[11] Yang, B., O'Hallaron, D. R. (1997). Parallel breadth-first BDD construction. 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (pp. 145-156).

[12] Caban, D., Milford, D. (1992). A parallel BDD engine for logic verification. In Proc. 5th

*Annual IEEE Int. ASIC Conf. and Exhibit* (pp. 499-502).

[13] Ranjan, R. K., Sanghavi, J. V., Brayton, R. K., Sangiovanni-Vincentelli, A. (1996). Binary decision diagrams on network of workstations. *IEEE Int. Conf. on Computer Design: VLSI in Computers and Processors* (pp. 358-364).

[14] Milvang-Jensen, K., & Alan, J. (1998). BDDNOW: A parallel BDD package. *In Proc. 2nd Int. Conf. on Formal Methods in Computer-Aided Design* (pp. 501-507).

[15] Yuxiong, H. (2009). Multicore-enabling a binary decision diagram algorithm. Intel Software Network. Retrieved from http://software.intel.com/en-us/articles/multicore-enabling-abinary-decision-diagramalgorithm.

[16] Gulati, K., Khatri, S. (2008). Towards acceleration of fault simulation using graphics processing units. In Proc. 45th ACM/IEEE Design Automation Conference (pp. 822-827).

[17] Bertacco, V., Chatterjee, D. (2011). High performance gate-level simulation with GP-GPU computing. Int. Symp. on VLSI Design, Automation and Test, 1-3.

[18] Chatterjee, D., Bertacco, V. (2010). EQUIPE: parallel equivalence checking with GP-GPUs. *IEEE Int. Conference on Computer Design*, 486-493.

[19] Gajic, D., & Stankovic, R. (2011). GPU accelerated computation of fast spectral transforms. Facta Universitatis - Series: *Electronics and Energetics*, 24(3), 483-499.

[20] Gajic, D., Stankovic, R., & Radmanovic, M. (2012). Implementation of dyadic correlation and autocorrelation on graphics processors. *Int. J. Reasoning-based Intelligent Systems (IJRIS)*, 4(1/2), 82-90.

[21] Yang, B. (1999). Optimizing model checking based on BDD characterization (PhD Dissertation). Carnegie Mellon University Pittsburgh, USA.

[22] Yang, B., et al. (1998). A study of BDD performance in model checking. *In Proc. Formal Methods in CAD* (pp. 255-289).

[23] Rudell, R. (1993). Dynamic variable ordering for ordered binary decision diagrams. *In Proc. Int. Conf. on Computer-Aided Design* (pp. 139–144).

[24] Klarlund, N., & Rauhe, T. (1996). BDD algorithms and cache misses. BRICS Report Series RS-96-5, Department of Computer Science, University of Aarhus.

[25] Aamodt, T. (2009). Architecting graphics processors for non-graphics compute acceleration. *In Proc. IEEE PACRIM Conf.* (pp. 963-968).

[26] Ryoo, S., et al. (2008). Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. *In Proc. 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming* (pp. 73–82).

[27] Owens, J., et al. (2008). GPU computing. *Proc. of the IEEE*, 96(5), 279–299.

[28] Gaster, B., et al. (2011). *Heterogeneous Computing with OpenCL*. Elsevier.

[29] Rudell, R. (1993). Espresso Misc. Reference Manual Pages. Retrieved from http://embedded.eecs.berkeley.edu/pubs/downloads/espresso/index.html.