Graph-based Parallel Analysis of Large Analog Circuits Based on GPU Platforms *

Jianan Lu†, Zhigang Hao‡, and Sheldon X.-D. Tan†
† Department of Electrical Engineering, University of California, Riverside, CA 92521
‡ School of Electrical and Electronic Engineerin, Shanghai Jiao Tong University, China
jlu@ee.ucr.edu, zhigang.g.hao@gmail.com, stan@ee.ucr.edu

ABSTRACT

In this paper, we propose a new parallel analysis method for large analog circuits using determinant decision diagram (DDD) based graph technique. DDD-based symbolic analysis technique enables exact symbolic analysis of vary large analog circuits. Once the circuit small-signal characteristics are presented by DDDs, evaluation of DDDs will give exact numerical values. In this paper, we develop efficient parallel DDD evaluation techniques based on general purpose GPU (GPGPU) computing platform to explore the parallelism of DDD structures. We propose two parallelization algorithms and their performance are compared. Experimental results show that the new evaluation algorithm can achieve about one to two order of magnitudes speedup over the serial CPU based evaluations on some large analog circuits.

1. INTRODUCTION

Graph-based symbolic technique is a viable tool for calculating the behavior or the characterization of an analog circuit. Traditional symbolic analysis tools typically are used to calculate the behavior or the characteristic of a circuit in terms of symbolic parameters [9]. The introduction of determinant decision diagrams based symbolic analysis technique allows exact symbolic analysis of much larger analog circuits than all the other existing approaches [13, 14]. Furthermore, with hierarchical symbolic representations [17,18], exact symbolic analysis via DDD graphs essentially allows the analysis of arbitrary large analog circuits. Some recent advancement in DDD ordering technique and variants of DDD allow even larger analog circuits to be analyzed [15, 16]. Once the circuit small-signal characteristics are presented by DDDs, evaluation of DDDs, whose CPU time is proportional to the size of DDDs, will give exact numerical values. However, with large networks, the DDD size can be huge and the resulting evaluation can be very time consuming.

Modern computer architecture has shifted towards designs that employ multiple processor cores on a chip, so called multi-core processor or chip-multiprocessors (CMP) [6, 7]. The graphic processing unit (GPU) are one of the most powerful many-core computing systems in mass-market use [1, 12]. For instance, Nvidia Telsa T10 chip has a peak performance of over 1 TFLOPS versus about 80-100 GFLOPS of Intel i5 series Quad-core CPUs [10]. In addition to the primary use of GPUs in accelerating graphics rendering operations, there has been considerable interest in exploiting GPUs for general purpose computation (GPGPU) [4]. The introduction of new parallel programming interfaces for general purpose computations, such as Computer Unified Device Architecture (CUDA) [3], Stream SDK [2] and OpenCL [5], has made GPUs powerful and attractive choice for developing high-performance numerical, scientific computation and solving practical engineering problems.

In this article, we develop efficient parallel DDD evaluation techniques based on general purpose GPU (GPGPU) computing platform to explore the parallelism of DDD structures. We propose data structures to present the DDD graphs in the GPUs for massively threaded parallel computing of the numerical values of DDD graphs. The new method explores data parallelism in the DDD numerical evaluation process where DDD graphs are traversed in depth-first fashion. Experimental results show that the new evaluation algorithm can achieve about one to two order of magnitudes speedup over the serial CPU based evaluations of some analog circuits. The proposed parallel techniques can be used for the parallelization of many more decision diagrams based applications such as logic synthesis, optimization and formal verifications, which are based on binary decision diagrams (BDDs) and its variants [8, 11].

This paper is organized as follows: Section 2 reviews DDD-based symbolic analysis techniques. Section 3 briefly review the GPU architectures and CUDA computing. Section 4 introduces the new parallel algorithm with the results in section 5. And lastly Section 6 concludes the paper.

2. DDDS AND DDD-BASED ANALYSIS

The DDD technique uses directed binary graphs to represent a determinant where the paths in the graph represents the product terms from determinant. Since the number of paths in a graph can be much larger than the number of nodes, DDD representation enables symbolic analysis of much larger analog circuits than before [13]. The concept of DDD representation is briefly reviewed as follows. The determinant of a matrix can be expressed as the symbolic product terms from the subset of all elements in the matrix. For example, consider the following matrix determinant.

$$det(M) = \begin{vmatrix} a & b & 0 & 0 \\ c & d & e & 0 \\ 0 & f & g & h \\ 0 & 0 & i & j \end{vmatrix} = adgj - adhi - aefj - bcgj + cbih \quad (1)$$

We can express each element using a vertex in a diagram and each product term using a path going through four vertices. For every vertex, it has a value of itself and a sign attached to it. The sign of each product term is decided by multiplying

*This work is funded in part by NSF grant under No. CCF-0448534, in part by NSF grant under No. OISE-0929699 and in part by NSF grant under No. CCF-1017090.

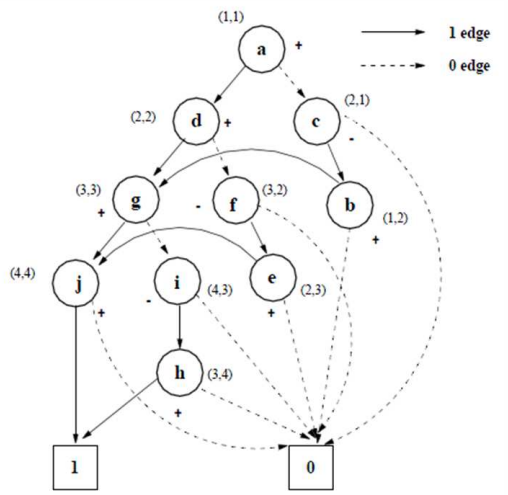

Figure 1: DDD representation for matrix M.

the sign of every vertex in the corresponding path. When the matrix M is an circuit matrix (such as modified nodal analysis (MNA) matrix), the value for each vertex represents the RLC value for the element in the MNA matrix. The diagram for the above matrix is shown in Fig. 1.

A DDD is a signed, rooted, directed acyclic graph with two terminal vertices, namely the *0-terminal* vertex and the *1-terminal* vertex. Each non-terminal DDD vertex is labeled by a symbol in the determinant denoted by a_i (a to j in Fig. 1), and a positive or negative sign denoted by $s(a_i)$. It originates two outgoing edges, called *1-edge* and *0-edge*. Each vertex a_i represents a symbolic expression $D(a_i)$ defined recursively as follows:

$$D(a_i) = a_i \cdot s(a_i) \cdot D_{a_i} + D_{\overline{a}_i}, \tag{2}$$

where D_{a_i} and $D_{\overline{a}_i}$ represent, respectively, the symbolic expressions of the vertices pointed by the *1-edge* and *0-edge* of a_i. The *1-terminal* vertex represents expression 1, whereas the *0-terminal* vertex represents expression 0. For example, vertex h (in Fig. 1) represents expression h, and vertex i represents expression $-ih$, and vertex g represents expression $gj - ih$. We also say that a DDD vertex g represents an expression defined the DDD subgraph rooted at g. For each vertex, there are two values, *vself* and *vtree*. In (2), *vself* represents the value of the element itself, which is D_{a_i}; while the *vtree* represents the value of the whole tree (or subtree), which is $D(a_i)$.

A *1-path* in a DDD corresponds with a product term in the original DDD, which is defined as a path from the root vertex (a in our example) to the *1-terminal* including all symbols and signs of the vertices that originate all the *1-edges* along the *1-path*. In our example, there exist five *1-paths* representing five product terms: $adgj$, $adhi$, $aefj$, $bcgj$, and $cbih$. The root vertex represents the sum of these product terms. Size of a DDD is the number of DDD vertices, denoted by $|DDD|$.

Once a DDD has been constructed, its numerical values of the determinant it represents can be computed by performing the depth-first type search of the graph and performing (2) at each node, whose time complexity is linear function of the size of the graphs (its number of nodes). The computing step is call *Evaluate(D)* where D is a DDD root.

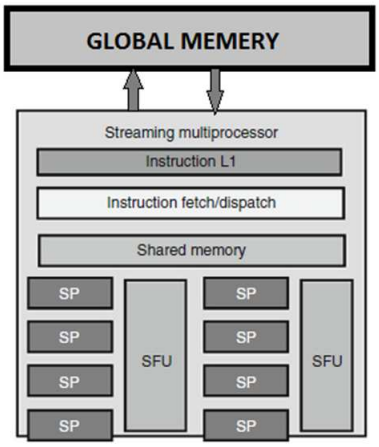

Figure 2: Structure of streaming multiprocessor.

3. REVIEW OF GPU ARCHITECTURES

CUDA (short for Compute Unified Device Architecture) is the parallel computing architecture for Nvidia many-core GPU processors. The architecture of a typical CUDA-capable GPU is consisted of an array of highly threaded streaming multiprocessors (SM) and comes with up to 4 GB DRAM, referred to as global memory. Each SM has eight streaming processor (SP) and two special function units (SFU) and possesses its own shared memory and instruction cache. The structure of a streaming multiprocessor is shown in Fig. 2.

As the programming model of GPU, CUDA extends C into CUDA C and supports such tasks as threads calling and memory allocation, which makes programmers able to explore most of the capabilities of GPU parallelism. In CUDA programming model, threads are organized into blocks; blocks of threads are organized as grids. CUDA also assumes that both the host (CPU) and the device (GPU) maintain their own separate memory spaces in DRAM, referred to as host memory and device memory, respectively. For every block of threads, a shared memory is accessible to all threads in that same block. And the global memory is accessible to all threads in all blocks. Developers can write programs running millions of threads with thousands of blocks in a parallel approach. This massive parallelism forms the reason that programs with GPU acceleration can be multiple times faster than their CPU counterparts.

One thing to mention is that for current GPU, a multiprocessor has eight single-precision floating point ALUs (one per core) but only one double-precision ALU (shared by the eight cores). Thus, for applications whose execution time is dominated by floating point computations, switching from single-precision to double-precision will decrease performance by a factor of approximately eight. However, this situation is being improved. More recent GPU from Nvidia can already provide much better double-precision performance than before.

4. NEW GPU-BASED DDD EVALUATION

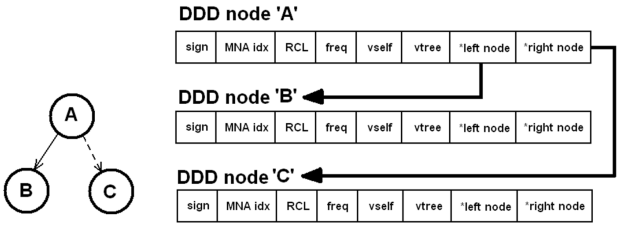

Figure 3: Illustration of the data structure for serial method

In this section, we present the new GPU-based DDD evaluation algorithm. Before the details of GPU-based DDD evaluation method, we first discuss the new DDD data structure for GPU parallel computing.

4.1 New data structure

To achieve the best performance on GPU, linear memory structure is preferable. For CPU serial computing, the data structure is based on dynamic links in a linked binary tree. For parallel computing, the data will be stored in linear arrays which can be more efficiently accessed by different threads based on thread ids.

As we discussed above, the DDD representation stores all product terms of the determinant of the MNA matrix in a binary linked tree structure. The vertex in the tree structure is known as DDD node that represents element in MNA matrix which is identified by its index. For each DDD node, the data structure includes the sign value, the MNA index, the RCL values, corresponding frequency value, *vself*, and *vtree*. In the serial approach, these values are stored in a data structure and connected through links, as shown in Fig. 3. On the other side, in the parallel approach, all of these data are stored separately in corresponding linear arrays and each element is identified by the DDD node index (not necessarily the same as the MNA element index). Figure 4 illustrates the new data structure.

Two choices are available for *vself* data structure. One is similar to the data structure of *vtree*. The *vself* value for each DDD node is stored consecutively. This data structure is called the linear version of *vself* data structure. The other method is as shown in Fig. 4. The array is organized per MNA element. Due to the fact that some of the DDD nodes share the same MNA element value, the second data structure is more compact in memory than the linear version. So it is called the compact version of *vself* data structure. The compact version is suitable for small circuits because it reduces the global memory traffic when computing *vself*. However, for large circuits, the calculation of *vtree* dominates the time cost. And we can implement a strategy to reduce the global memory traffic for computing *vtree* using the linear version of *vself* data structure to further improve the GPU performance. Therefore, for larger circuits, the linear version is preferable. The performance comparison is discussed later in the next chapter.

4.2 Algorithm flow

The parallel evaluating process consists of two stages. First, the *vself* values for all DDD nodes are computed and stored. In this stage, a set of 2D threads are launched on GPU

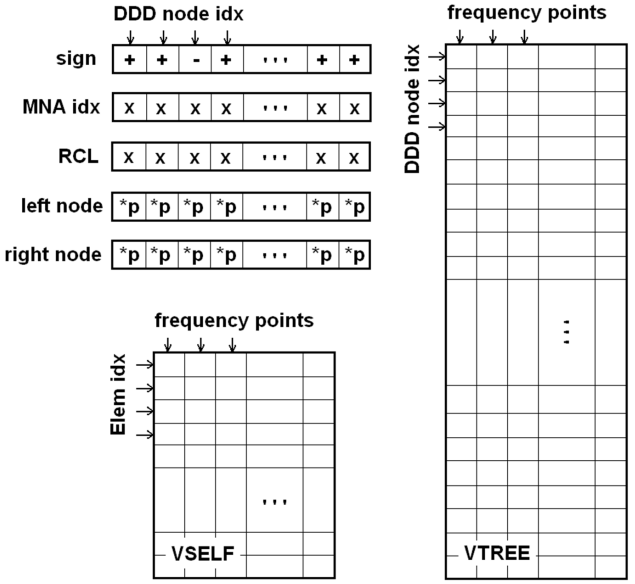

Figure 4: Illustration of the data structure for parallel method

devices. The X-dimension of the 2D threads represents different frequencies; the Y-dimension represents different elements (for compact *vself*) or DDD nodes (for linear *vself*). Therefore, all elements (or DDD nodes) can be computed under all frequencies in massively parallel manners. In the second stage, we simultaneously launch GPU 2D-threads to compute all the *vtree* values for DDD nodes based on (2). Notice a DDD node *vtree* value becomes valid when all its children's *vtree* values are valid. Since we compute all the *vtree* for all the nodes at the same time, the correct *vtree* values will automatically propagate from the bottom of the DDD tree to the top node. The number of such *vtree* iterative computing are decided by the number of layers in DDD tree. A layer represents a set of DDD nodes whose distance from *1-terminal* or *0-terminal* are the same. The number of layers equal to the longest distance between non-terminal nodes and *1-terminal*/*0-terminal*. Algorithm 1 shows the flow of parallel DDD evaluation using compact *vself* data structure.

Line 3-4 load frequency index and element index respectively with CUDA built-in variables (Thread.X and Thread.Y are simplified notations). These built-in variables are the mechanism for identifying data within different threads in CUDA. The line 5-6 compute the *vself* with the RCL value of the element under given frequency. From line 8, loop for computing *vtree* is entered. Line 13-14 load *vtree* values for left/right branch using function *Then()/Else()*. Line 15-26 explains themselves. Line 27 computes *vtree* with *vself* and *Left/Right* and ends the flow.

4.3 Coalesced memory access

The GPU performance can be further improved by making proper use of coalesced global memory access to prevent the global memory bandwidth from being performance limitation. Coalesced memory access is one efficient method reducing global memory traffic. When all threads in a warp execute a load instruction, the hardware detects whether the threads access the consecutive global memory address.

Algorithm 1 Parallel DDD evaluation algorithm flow

```
1: if Launch GPU threads for each node then
2:    {Computing vself:}
3:    FreqIdx ← Thread.X
4:    ElemIdx ← Thread.Y
5:    (R, C, L) ← GetRCL(ElemIdx)
6:    vself ← (R, C * Freq + L/Freq)
7: end if
8: for all lyr such that 0 ≤ lyr ≤ NumberOfLayers do
9:    {Computing vtree:}
10:   if Launch GPU threads for each node then
11:      FreqIdx ← Thread.X
12:      DDDIdx ← Thread.Y
13:      Left ← Then(DDDIdx)
14:      Right ← Else(DDDIdx)
15:      if is 0 − terminal then
16:         Left ← (0, 0)
17:         Right ← (0, 0)
18:      else
19:         if is 1 − terminal then
20:            Left ← (1, 0)
21:            Right ← (1, 0)
22:         end if
23:      end if
24:      if sign(DDDIdx) < 0 then
25:         vself ← −1 * vself
26:      end if
27:      vtree ← vself * Left + Right
28:   end if
29: end for
```

In such case, the hardware coalesces all of these accesses into a consolidated access to the consecutive global memory. In the implementation of GPU-accelerated DDD evaluation, such favorable data access pattern is fulfilled for the linear version of *vself* data structure to gain performance enhancement. The *vself* data structure is in a linear pattern so that the *vself* values for a given DDD node under a series of frequency values are stored in coalesced memory. Therefore, threads, in the same block, with consecutive thread index will access consecutive global memory locations, which ensure that the hardware coalesces these accessing process in just one reading operation. In this example, this technique reduces the global memory traffic by a factor of 16. However, for the compact version of *vself* data structure, the *vself* values are stored per elements, which means that for consecutive DDD nodes, their respective *vself* values are not stored in consecutive locations. So, for the compact version of *vself* data structure, the global memory access is not coalesced. The performance comparison for both of versions is discussed in experimental result section.

5. EXPERIMENTAL RESULTS

We have implemented both CPU serial version and GPU version of the DDD-based evaluation programs using C++ and CUDA C, respectively.

The serial and parallel versions of programs have been tested under the same hardware/software conditions. The hardware platform is Linux server with two Intel Xeon E5620 2.4GHz Quad-Core CPUs, 36 GB memory, equipped with Nvidia Tesla S1070 1U GPU card (4 T10 GPUs). The software environment is Linux version 2.6.18-194.26.1.el5, gcc version 4.1.2 20080704 (Red Hat 4.1.2-48), CUDA version

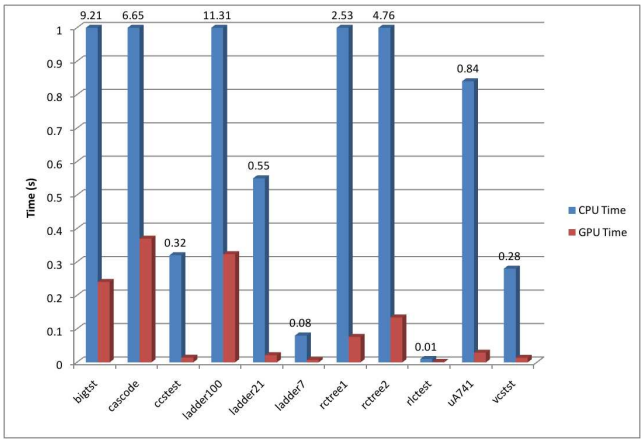

Figure 5: Performance comparison

3.2.

For the purpose of performance comparison, the programs with CPU-serial and GPU-parallel algorithm are both tested for the same set of circuits. The testing circuits include: uA741 (a bipolar opamp), Cascode (a CMOS Cascode opamp), ladder7, ladder21, ladder100 (7-, 21-, 100-section cascade resistive ladder networks), rctree1, rctree2 (two RC tree networks), rlctest, vcstest, ccstest, bigtst (some RLC filters).

In the two implementations, the same DDD construction algorithm is shared. The numerical evaluation process is done under serial and parallel version separately. The performance comparison for each of the given circuit is listed in Table 1 and illustrated in Fig. 5. In our experimental results, the overhead for data transferring between host and GPU devices are not included as their costs can be amortized over many DDD evaluation processes and can be partially overlapped with GPU computing in more advanced parallelization implementation. The statistics information for DDD representation is also included in the same table. The first column indicates the name of each circuit tested. The second to fourth columns represent the number of nodes in circuit, the number of elements in the MNA matrix and the number of DDD nodes in the generated DDD graph, respectively. The number of determinant product terms is shown in fifth column. CPU time is the time cost for the calculation of DDD evaluation in serial algorithm. The GPU time is the computing time cost for GPU-parallelism (the kernel parts). The final column summerizes the speedup of parallel algorithm over serial algorithm.

From Table 1, we can make some observations. For a variety of circuits tested in the experiment, the GPU-accelerated version outperforms all of their counterparts. The maximum performance speedup is 38.33 times for *bigtst*. The time cost of the serial version is growing fast along the increasing of circuit size (nodes in the circuit). On the other side, however, the GPU-based parallel version performs much better for larger circuits. And more importantly, the larger the circuit is, the better performance improvement we can gain using GPU-acceleration. This trend is illustrated in Fig. 6. This result implies that the GPU-acceleration is suitable to overcome the performance problem of DDD-based numerical evaluation for large circuits.

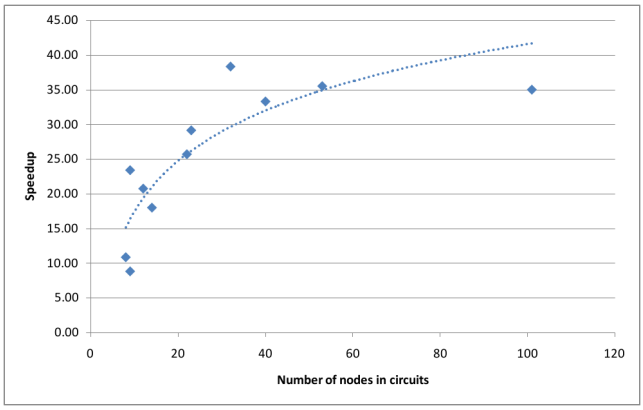

Figure 6: The performance speedup of GPU-acceleration vs. circuits size (number of nodes)

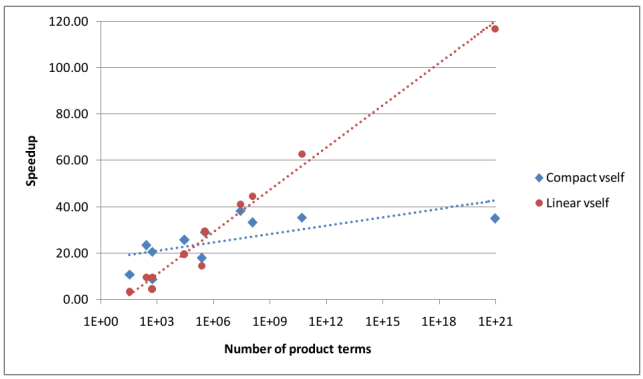

Figure 7: Performance comparison for two approaches of vself data structure (the x-axis is on logarithm scale)

In this experiment, both of data structures for storing *vself* are implemented. The performance comparison is listed in Table 2. The GPU parallel version under both of the two data structures for *vself* outperforms the serial version. And the performance speedup is clearly related to the number of product terms in MNA matrix, as shown in Fig. 7. For small circuits with less MNA matrix product terms, the compact version of *vself* is more efficient due to the lowering of global memory traffic when calculating *vself*. However, for large circuits with bigger number of MNA matrix product terms, the linear version of *vself* outperforms the compact version owing to the effect of coalesced memory access as discussed in the prior chapter.

6. CONCLUSIONS

In this paper, a GPU-based parallel analysis method for large analog circuits has been proposed. Its performance is compared to its CPU-based serial counterpart. The GPU-based DDD evaluation performs better, especially for larger circuits. Two data structures have been proposed and their performance has been compared for circuits with different number of product terms in MNA matrix. Experimental results show that the new evaluation algorithm can achieve about one to two order of magnitudes speedup over the serial CPU based evaluations on some large analog circuits. The proposed parallel techniques can be also used for the parallelization of other decision diagrams based application such Binary Decision Diagrams (BDDs) for logic synthesis and formal verifications.

7. REFERENCES

[1] "AMD developer center," http://developer.amd.com/GPU.

[2] "AMD Steam SDK," http://developer.amd.com/gpu/ATIStreamSDK.

[3] "CUDA (Compute Unified Device Architecture)," http://www.nvidia.com/object/cuda_home.html.

[4] "General-purpose computation using graphics harware," http://www.gpgpu.org/.

[5] "Open computing language (OpenCL)," http://www.khronos.org/opencl.

[6] "Intel multi-core processors, making the move to quad-core and beyond (White Paper)," 2006, http://www.intel.com/multi-core.

[7] "Multi-core processors—the next evolution in computing (White Paper)," 2006, http://multicore.amd.com.

[8] R. E. Bryant, "Binary decision diagrams and beyond: enabling technologies for formal verification," in *Proc. Int. Conf. on Computer Aided Design (ICCAD)*, 1995.

[9] G. Gielen, P. Wambacq, and W. Sansen, "Symbolic analysis methods and applications for analog circuits: A tutorial overview," *Proc. of IEEE*, vol. 82, no. 2, pp. 287–304, Feb. 1994.

[10] D. B. Kirk and W. mei W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. San Francisco, CA: Morgan Kaufmann Publishers Inc., 2010.

[11] S. Minato, *Binary Decision Diagrams and Application for VLSI CAD*. Boston: Kluwer Academic Publishers, 1996.

[12] "Nvidia Corporation," http://www.nvidia.com.

[13] C.-J. Shi and X.-D. Tan, "Canonical symbolic analysis of large analog circuits with determinant decision diagrams," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 1, pp. 1–18, Jan. 2000.

[14] ——, "Compact representation and efficient generation of s-expanded symbolic network functions for computer-aided analog circuit design," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 7, pp. 813–827, April 2001.

[15] G. Shi, "Computational complexity analysis of determinant decision diagram," *IEEE Trans. on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 57, no. 10, pp. 828 –832, 2010.

[16] ——, "A simple implementation of determinant decision diagram," in *Proc. Int. Conf. on Computer Aided Design (ICCAD)*, 2010, pp. 70 –76.

[17] S. X.-D. Tan, W. Guo, and Z. Qi, "Hierarchical approach to exact symbolic analysis of large analog circuits," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 8, pp. 1241–1250, August 2005.

[18] X.-D. Tan and C.-J. Shi, "Hierarchical symbolic analysis of large analog circuits via determinant decision diagrams," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 4, pp. 401–412, April 2000.

Table 1: Performance comparison of CPU-serial and GPU-parallel DDD evaluation for a set of circuits

circuit	# nodes	# elements	# DDD nodes	# terms	CPU time (s)	GPU time (s)	speedup
bigtst	32	112	642	2.68×10^7	9.21	0.240	38.33
cascode	14	76	2110	2.32×10^5	6.65	0.369	18.00
ccstest	9	35	109	260	0.32	0.014	23.40
ladder100	101	301	301	9.27×10^{20}	11.31	0.323	35.00
ladder21	22	64	64	28657	0.55	0.021	25.69
ladder7	8	22	22	34	0.08	0.007	10.86
rctree1	40	119	211	1.15×10^8	2.53	0.076	33.30
rctree2	53	158	302	4.89×10^{10}	4.76	0.134	35.51
rlctest	9	39	119	572	0.01	0.001	8.82
uA741	23	89	6205	363914	0.84	0.029	29.14
vcstst	12	46	121	536	0.28	0.013	20.74

Table 2: Performance comparison for two implementations of *vself* data structure

circuit	# terms	CPU time (s)	GPU time (s)		speedup	
			w/ compact vself	w/ linear vself	w/ compact vself	w/ linear vself
bigtst	2.68×10^7	9.21	0.240	0.223	38.33	41.21
cascode	2.32×10^5	6.65	0.369	0.452	18.00	14.70
ccstest	260	0.32	0.014	0.033	23.40	9.65
ladder100	9.27×10^{20}	11.31	0.323	0.097	35.00	116.92
ladder21	28657	0.55	0.021	0.028	25.69	19.40
ladder7	34	0.08	0.007	0.025	10.86	3.20
rctree1	1.15×10^8	2.53	0.076	0.057	33.30	44.71
rctree2	4.89×10^{10}	4.76	0.134	0.076	35.51	62.93
rlctest	572	0.01	0.001	0.002	8.82	4.40
uA741	363914	0.84	0.029	0.029	29.14	29.27
vcstst	536	0.28	0.013	0.029	20.74	9.62