6

Parallel Symbolic Analysis of Large Analog Circuits on GPU Platforms *

Sheldon X.-D. Tan[1], Xue-Xin Liu[1], Eric Mlinar[1] and Esteban Tlelo-Cuautle[2]
[1]*Department of Electrical Engineering, University of California, Riverside, CA 92521*
[2]*Dapartment of Electronics, INAOE*
[1]*USA*
[2]*Mexico*

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 (Gielen et al., 1994). The introduction of determinant decision diagrams based symbolic analysis technique allows exact symbolic analysis of much larger analog circuits than any other existing approaches (Shi & Tan, 2000; 2001). Furthermore, with hierarchical symbolic representations (Tan et al., 2005; Tan & Shi, 2000), exact symbolic analysis via DDD graphs essentially allows the analysis of arbitrarily large analog circuits. Some recent advancement in DDD ordering technique and variants of DDD allow even larger analog circuits to be analyzed (Shi, 2010a;b). Once the circuit's small-signal characteristics are presented by DDDs, the evaluation of DDDs, whose CPU time is proportional to the sizes 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 processors or chip-multiprocessors (CMP) (AMD Inc., 2006; Intel Corporation, 2006). The graphic processing unit (GPU) is one of the most powerful many-core computing systems in mass-market use (AMD Inc., 2011a; NVIDIA Corporation, 2011a). 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 (Kirk & Hwu, 2010). 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 (Göddeke, 2011). The introduction of new parallel programming interfaces for general purpose computations, such as Computer Unified Device Architecture (CUDA) (NVIDIA Corporation, 2011b), Stream SDK (AMD Inc., 2011b) and OpenCL (Khronos Group, 2011), has made GPUs a powerful and attractive choice for developing high-performance numerical, scientific computation and solving practical engineering problems.

*This work is funded in part by NSF grants NSF OISE-0929699, OISE-1130402, CCF-1017090 and part by CN-11-575 UC MEXUS-CONACYT Collaborative Research Grants.

In this chapter, we present an efficient parallel DDD evaluation technique based on general purpose GPU (GPGPU) computing platform to explore the parallelism of DDD structures. We present a new 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 a depth-first fashion. Numerical results show that the new evaluation algorithm can achieve about one to two orders of magnitude speedup over the serial CPU based evaluations of some analog circuits. The presented parallel techniques can be used for the parallelization of many decision diagrams based applications such as logic synthesis, optimization, and formal verification, all of which are based on binary decision diagrams (BDDs) and its variants (Bryant, 1995; Minato, 1996).

This chapter 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, and then the results are demonstrated in Section 5. Lastly, Section 6 summarizes this chapter.

2. DDDs and DDD-based symbolic analysis

Before we introduce our GPU-base parallel analysis method, we first provide a brief overview of determinant decision diagram (DDD) Shi & Tan (2000) in this section.

Determinant decision diagrams (DDDs) was introduced to represent determinants symbolically Shi & Tan (2000). A DDD is essentially a zero-suppressed Binary Decision Diagrams (ZBDDs) — introduced originally for representing sparse subset systems Minato (1993). A ZBDD is a variant of a Binary Decision Diagram (BDD) introduced by Akers Akers (1976) and popularized by Bryant Bryant (1986). BDDs have brought great success to formal verification and testing for combinational and sequential digital circuits Bryant (1986); Hachtel & Somenzi (1996). DDD representation has several advantages over both the expanded and arbitrarily nested forms of a symbolic expression.

- First, similar to the nested form, DDD representation is compact for a large class of analog circuits. A ladder-structured network can be represented by a diagram where the number of vertices in the diagram (called its *size*) is equal to the number of symbolic parameters. As indicated by Shi & Tan (2000), the typical size of DDD is dramatically smaller than that of product terms. For instance, 5.71×10^{20} terms can be represented by a diagram with 398 vertices.
- Second, similar to the expanded form, DDD representation is canonical, i.e., every determinant has a *unique* representation, and is amenable to symbolic manipulations. This property of canonical representation for matrix determinants is similar to BDD for representing *binary functions* and ZBDD for representing *subset systems*.

A key observation is that the circuit matrix is sparse and that for many times, a symbolic expression may share many sub-expressions. For example, consider the following determinant

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

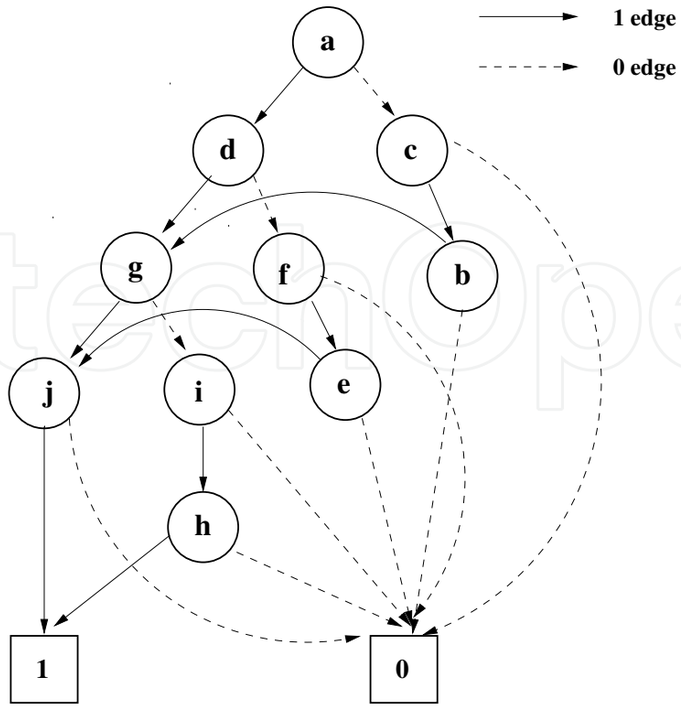


Fig. 1. A ZBDD representing $\{adgi, adhi, afej, cbgj, cbih\}$ under ordering $a > c > b > d > f > e > g > i > h > j$

Note that sub-terms ad, gj, and hi appear in several product terms, and each product term involves a subset (four) out of ten symbolic parameters. Therefore, we view each symbolic product term as a subset, and use a ZBDD to represent the subset system composed of all the subsets each corresponding to a product term. Fig. 1 illustrates the corresponding ZBDD representing all the subsets involved in $\det(\mathbf{M})$ under ordering $a > c > b > d > f > e > g > i > h > j$. It can be seen that sub-terms ad, gj, and ih have been shared in the ZBDD representation.

Following directly from the properties of ZBDDs, we have the following observations. First, given a fixed order of symbolic parameters, all the subsets in a symbolic determinant can be represented uniquely by a ZBDD. Second, every 1-path in the ZBDD corresponds to a product term, and the number of 1-edges in any 1-path is n. The total number of 1-paths is equal to the number of product terms in a symbolic determinant.

We can view the resulting ZBDD as a graphical representation of the recursive application of the determinant expansion with the expansion order $a, c, b, d, f, e, g, i, h, j$. Each vertex is labeled with the matrix entry with respect to which the determinant is expanded, and it represents all the subsets contained in the corresponding sub-matrix determinant. The 1-edge points to the vertex representing all the subsets contained in the cofactor of the current expansion, and 0-edge points to the vertex representing all the subsets contained in the remainder.

To embed the signs of the product terms of a symbolic determinant into its corresponding ZBDD, we associate each vertex v with a sign, $s(v)$, defined as follows:

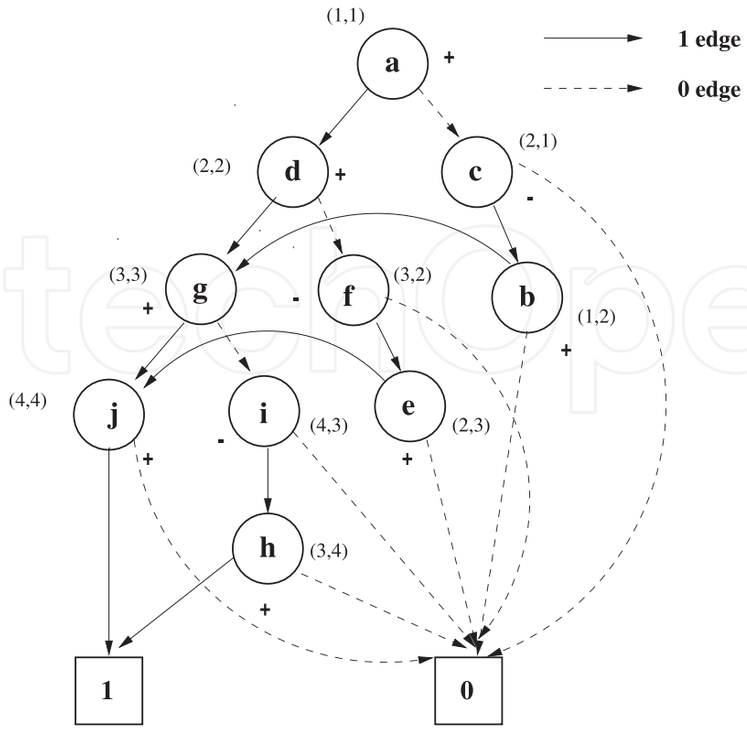


Fig. 2. A signed ZBDD for representing symbolic terms from matrix **M**

- Let $P(v)$ be the set of ZBDD vertices that originate the 1-edges in any 1-path rooted at v. Then

$$s(v) = \prod_{x \in P(v)} \text{sign}(r(x) - r(v)) \, \text{sign}(c(x) - c(v)), \tag{2}$$

  where $r(x)$ and $c(x)$ refer to the absolute row and column indices of vertex x in the original matrix, and u is an integer so that

$$\text{sign}(u) = \begin{cases} +1, & \text{if } u > 0, \\ -1, & \text{if } u < 0. \end{cases}$$

- If v has an edge pointing to the 1-terminal vertex, then $s(v) = +1$.

This is called the *sign rule*. For example, in Fig. 2, shown beside each vertex are the row and column indices of that vertex in the original matrix, as well as the sign of that vertex obtained by using the sign rule above. For the sign rule, we have following result:

Theorem 1. *The sign of a DDD vertex v, $s(v)$, is uniquely determined by (2), and the product of all the signs in a path is exactly the sign of the corresponding product term.*

For example, consider the 1-path *acbgih* in Fig. 2. The vertices that originate all the 1-edges are c, b, i, h, their corresponding signs are $-$, $+$, $-$ and $+$, respectively. Their product is $+$. This is the sign of the symbolic product term *cbih*.

With ZBDD and the sign rule as two foundations, we are now ready to introduce formally our representation of a symbolic determinant. Let **A** be an $n \times n$ sparse matrix with a set of distinct m symbolic parameters $\{a_1, ..., a_m\}$, where $1 \leq m \leq n^2$. Each symbolic parameter a_i

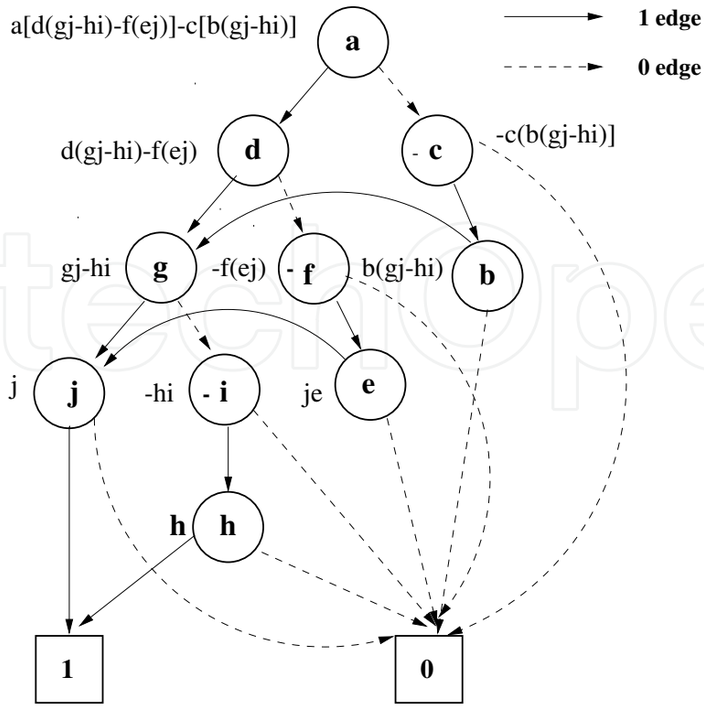


Fig. 3. A determinant decision diagram representation for matrix **M**

is associated with a unique pair $r(a_i)$ and $c(a_i)$, which denote, respectively, the row index and column index of a_i. A *determinant decision diagram* is a signed, rooted, directed acyclic graph with two terminal vertices, namely the 0-terminal vertex and the 1-terminal vertex. Each non-terminal vertex a_i is associated with a sign, $s(a_i)$, determined by the sign rule defined by (2). It has two outgoing edges, called 1-edge and 0-edge, pointing, respectively, to D_{a_i} and $D_{\overline{a}_i}$. A determinant decision graph having root vertex a_i denotes a matrix determinant D defined recursively as

- If a_i is the 1-terminal vertex, then $D = 1$.
- If a_i is the 0-terminal vertex, then $D = 0$.
- If a_i is a non-terminal vertex, then

$$D = a_i s(a_i) D_{a_i} + D_{\overline{a}_i} \tag{3}$$

Here $s(a_i)D_{a_i}$ is the *cofactor* of D with respect to a_i, D_{a_i} is the *minor* of D with respect to a_i, $D_{\overline{a}_i}$ is the *remainder* of D with respect to a_i, and operations are algebraic multiplications and additions. For example, Fig. 3 shows the DDD representation of det(**M**) under ordering $a > c > b > d > f > e > g > i > h > j$.

To enforce the uniqueness and compactness of the DDD representation, the three rules of ZBDDs, namely, zero-suppression, ordered, and shared are adopted. This leads to DDDs having the following properties:

- Every 1-path from the root corresponds to a product term in the fully expanded symbolic expression. It contains exactly n 1-edges. The number of 1-paths is equal to the number of product terms.

- For any determinant D, there is a unique DDD representation under a given vertex ordering.

We use $|DDD|$ to denote the *size of* a DDD, i.e., the number of vertices in the 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 (3) 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.

For each vertex, there are two values, *vself* and *vtree*. As above mentioned, vself represents the value of the element itself; while vtree represents the value of the whole tree (or subtree). For each vertex, the vtree equals vself multiplying vtree of 1-subtree plus vtree of 0-subtree as shown in (3). In this example, the value of the determinant equals vtree of a; while vtree of a equals vself of a multiplying vtree of b plus vtree of c. In a serial implementation, the tree value of a is computed by recursively computing all vtree of subtrees, which is very time-consuming when the tree becomes large.

One key observation for DDD structure is that the data dependence is very clear. The data dependency is very simple: a node can be evaluated only after its children are evaluated. Such dependency implies the parallelism where all the nodes satisfying this constraint can be evaluated at the same time. Also, in frequency analysis of analog circuits, evaluation of a DDD node at different frequency runs can be performed in parallel. In the following section we show how we can explore such parallelism to speed up the DDD evaluation process.

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 more than 4 GBytes 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. 4.

As the programming model of GPU, CUDA extends C into CUDA C, and supports tasks such 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 into 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 some series of CUDA 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

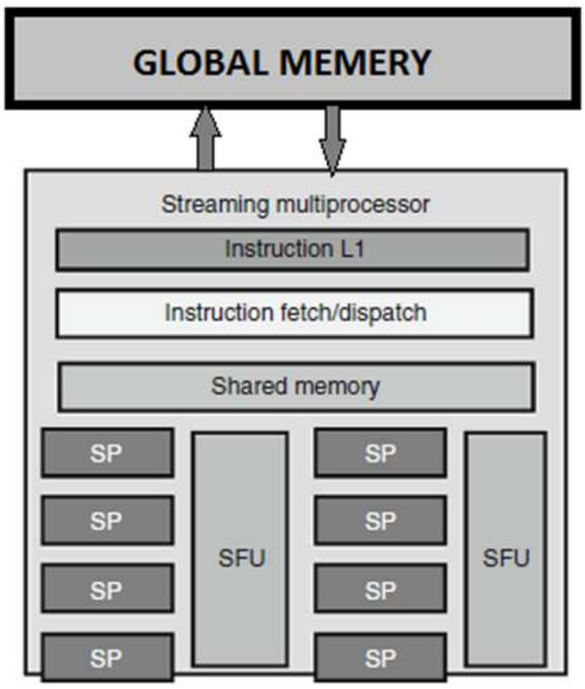


Fig. 4. Structure of streaming multiprocessor.

in NVIDIA T20 series, the Fermi family. These most recent GPU from NVIDIA can already provide much better double-precision performance than before.

4. New GPU-based DDD evaluation

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.

One key observation for DDD structure is that the data dependence is very clear. The data dependency is very simple: a node can be evaluated only after its children are evaluated. Such dependency implies the parallelism where all the nodes satisfying this constraint can be evaluated at the same time. Also, in frequency analysis of analog circuits, evaluation of a DDD node at different frequency runs can be performed in parallel. In the following subsections we show how we can explore such parallelism to speed up the DDD evaluation process.

4.1 New data structure

To achieve the best performance on GPU, linear memory structure, i.e., data stored in consequent memory addresses, 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

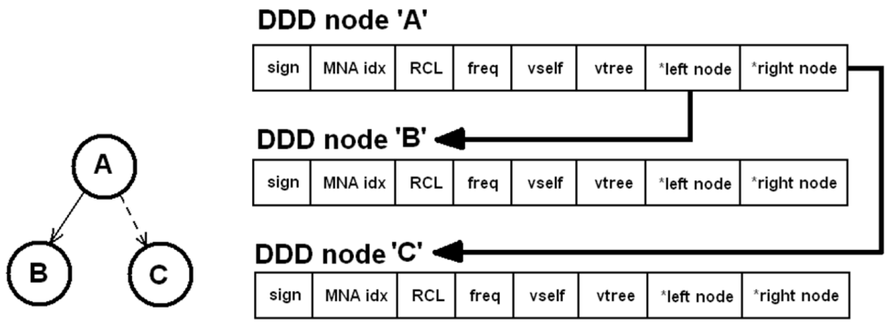


Fig. 5. Illustration of the data structure for serial method

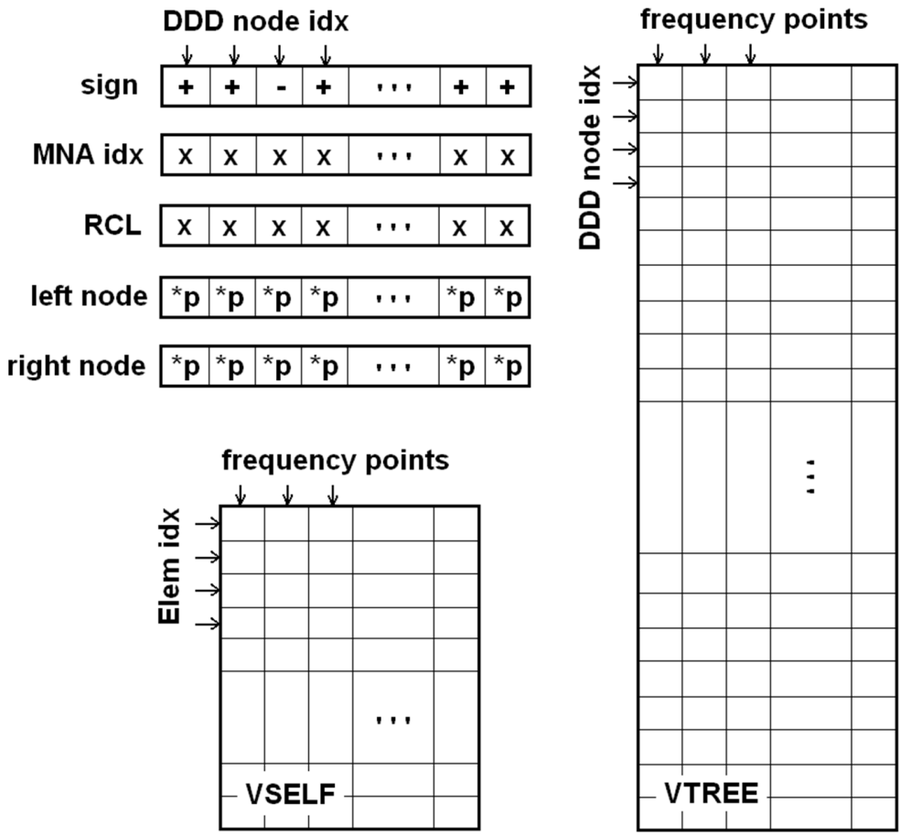


Fig. 6. Illustration of the data structure for parallel method

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. 5. 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 6 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. 6. 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 section.

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 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 (3). Notice that 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 and Line 4 load frequency index and element index respectively with CUDA built-in variables (Thread.X and Thread.Y are our simplified notations). These built-in variables are the mechanism for identifying data within different threads in CUDA. Then, line 5 and Line 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 and Line 14 load *vtree* values for left/right branch using function *Then()/Else()*. Line 15 through Line 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. 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

Algorithm 1 Parallel DDD evaluation algorithm flow

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

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. Numerical 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 and OS configuraions. The computation platform is a Linux server with two Intel Xeon E5620 2.4 GHz Quad-Core CPUs, 36 GBytes memory, equipped with NVIDIA Tesla S1070

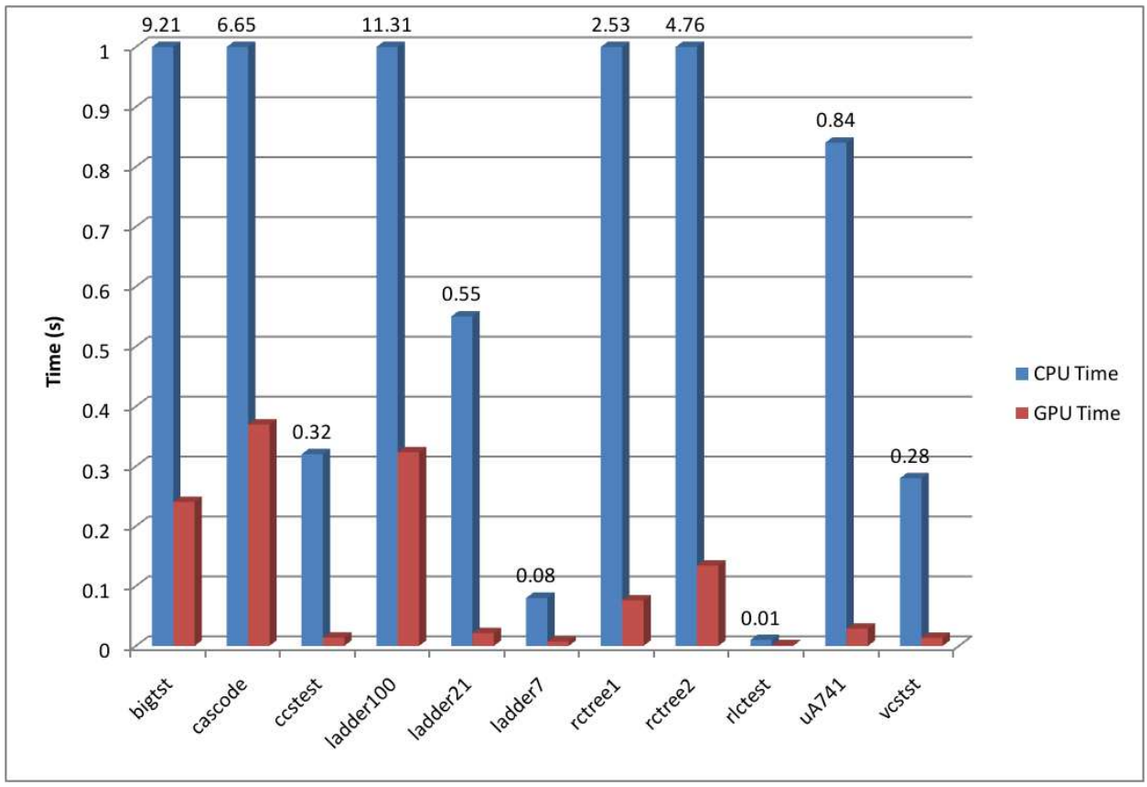


Fig. 7. Performance comparison

1U rack-mounted system (containing four T10 GPUs). The software environment is Red Hat 4.1.2-48 Linux, gcc version 4.1.2 20080704, and CUDA version 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: μA741 (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. 7. 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.

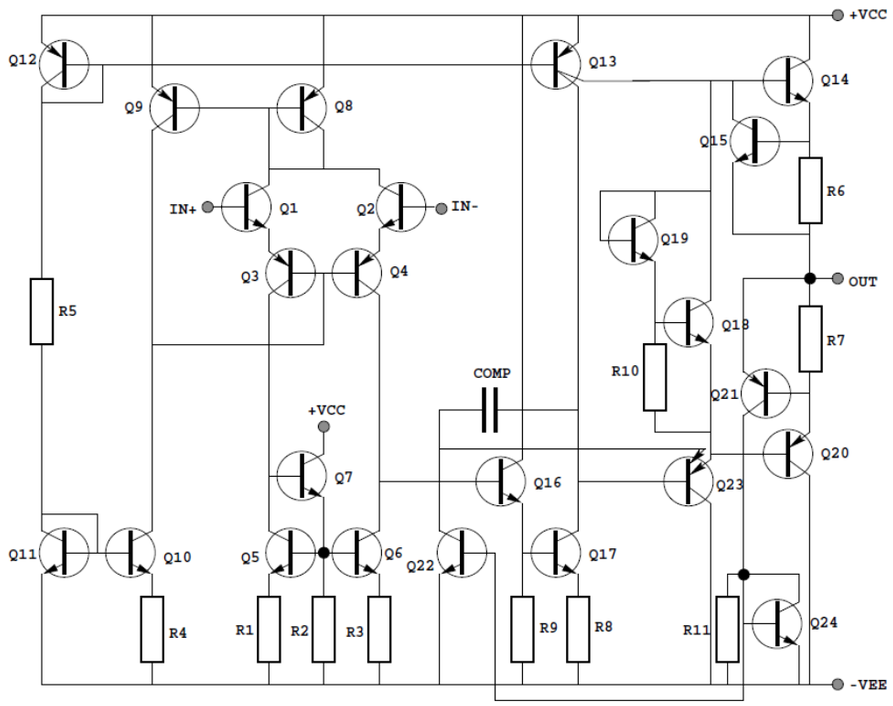


Fig. 8. The circuit schematic of μA741

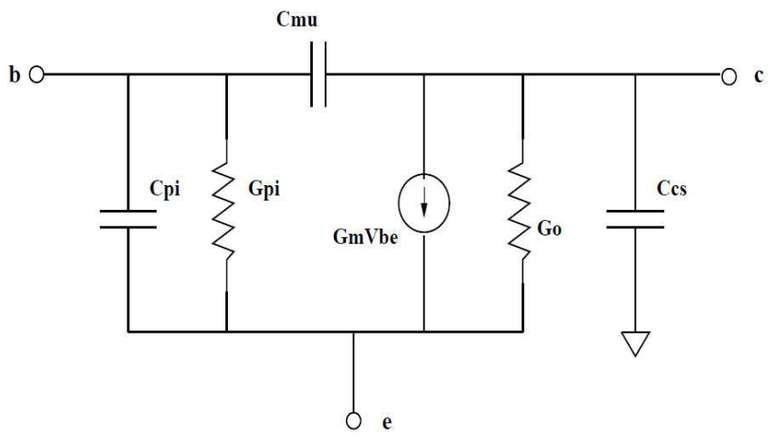


Fig. 9. The small signal model for bipolar transistor

Now let us investigate one typical example in detail. Fig. 8 shows the schematic of a μA741 circuit. This bipolar opamp contains 26 transistors and 11 resistors. DC analysis is first performed by SPICE to obtain the operation point, and then small-signal model, shown in Fig. 9, is used for DDD symbolic analysis and numerical evaluation. The AC analysis is performed using both CPU DDD evaluation and GPU parallel DDD evaluation proposed in our work. In Fig. 10 plots the frequency responses of the gain and the phase at the amplifier's output node from the two comparison methods. It can be observed that GPU parallel DDD evaluation provides the same result as it CPU serial counterpart does. We measured the run

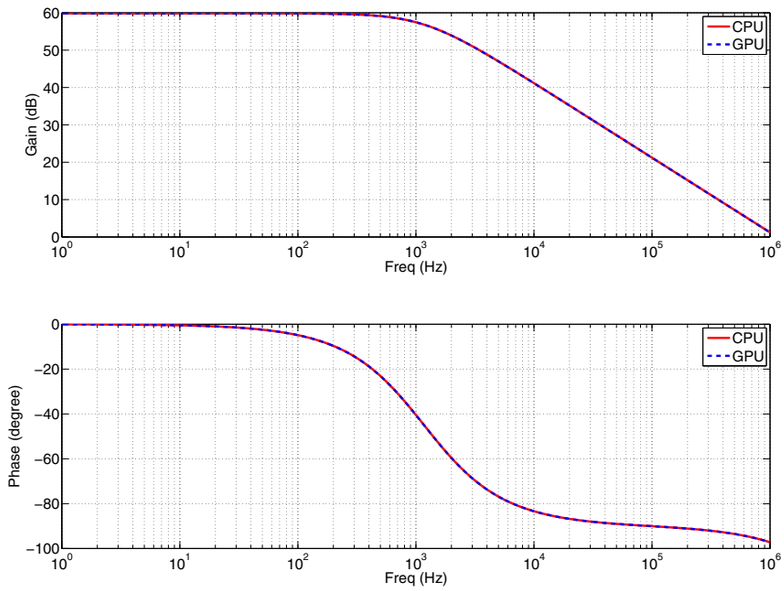


Fig. 10. Frequency response of μA741 amplifier. The red solid curve is the result of CPU DDD evaluation, while the blue dashed line is the result of GPU parallel DDD evaluation.

time of these two methods: the program of CPU evaluation costs 0.84 second, while the GPU parallel version only takes 0.029 second. For this benchmark circuit, we can judge that the parallel computation can easily achieve a speedup of 29 times. As the size of the circuit and the number of DDD nodes grow larger, more speedup can be expected.

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. 11. This result implies that the

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
μA741	23	89	6205	363914	0.84	0.029	29.14
vcstst	12	46	121	536	0.28	0.013	20.74

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

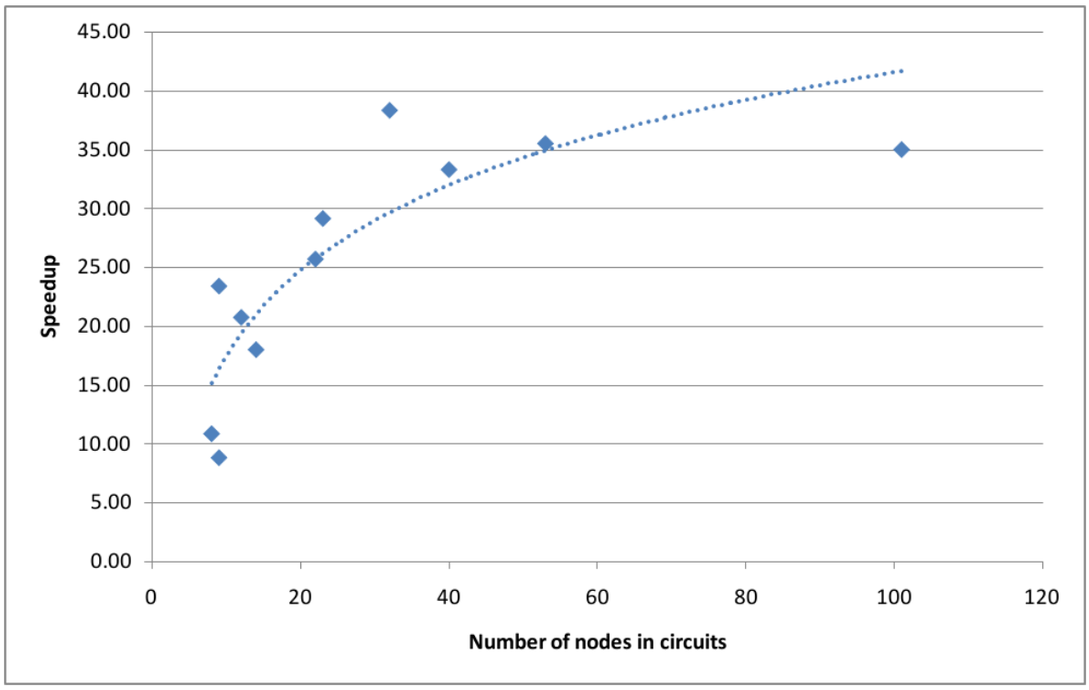


Fig. 11. The performance speedup of GPU-acceleration vs. circuits size (number of nodes)

GPU-acceleration is suitable to overcome the performance problem of DDD-based numerical evaluation for large circuits.

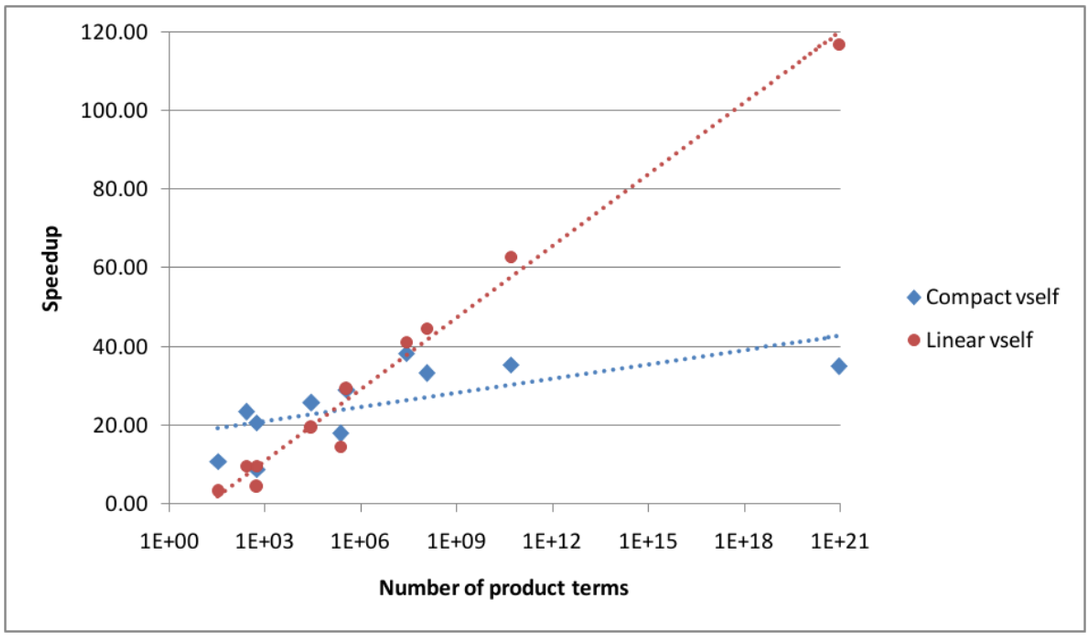


Fig. 12. Performance comparison for two approaches of vself data structure (the x-axis is in logarithm scale)

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. 12. 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 *comp vself* owing to the effect of coalesced memory access as discussed in the prior section.

circuit	# terms	CPU (s)	GPU time (s)		speedup	
			w/ comp vself	w/ linear vself	w/ comp 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
μA741	363914	0.84	0.029	0.029	29.14	29.27
vcstst	536	0.28	0.013	0.029	20.74	9.62

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

6. Summary

In this chapter, a GPU-based graph-based parallel analysis method for large analog circuits has been presented. Two data structures have been designed to cater the favor of GPU computation and device memory access pattern. Both the CPU version and the GPU version's performance has been studied and compared for circuits with different number of product terms in MNA matrix. The GPU-based DDD evaluation performs much better than its CPU-based serial counterpart, especially for larger circuits. Experimental results from tests on a variety of industrial benchmark circuits 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 presented parallel techniques can be also used for the parallelization of other decision diagrams based applications such as Binary Decision Diagrams (BDDs) for logic synthesis and formal verifications.

7. References

Akers, S. B. (1976). Binary decision diagrams, *IEEE Trans. on Computers* 27(6): 509–516.

AMD Inc. (2006). Multi-core processors—the next evolution in computing (White Paper). `http://multicore.amd.com`.

AMD Inc. (2011a). AMD developer center, `http://developer.amd.com/GPU`.

AMD Inc. (2011b). AMD Steam SDK, `http://developer.amd.com/gpu/ATIStreamSDK`.

Bryant, R. E. (1986). Graph-based algorithms for Boolean function manipulation, *IEEE Trans. on Computers* pp. 677–691.

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

Gielen, G., Wambacq, P. & Sansen, W. (1994). Symbolic analysis methods and applications for analog circuits: A tutorial overview, *Proc. of IEEE* 82(2): 287–304.

Göddeke, D. (2011). General-purpose computation using graphics harware, `http://www.gpgpu.org/`.

Hachtel, G. D. & Somenzi, F. (1996). *Logic Synthesis and Verification Algorithm*, Kluwer Academic Publishers.

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

Khronos Group (2011). Open Computing Language (OpenCL), `http://www.khronos.org/opencl`.

Kirk, D. B. & Hwu, W.-M. (2010). *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA.

Minato, S. (1993). Zero-suppressed BDDs for set manipulation in combinatorial problems, *Proc. Design Automation Conf. (DAC)*, pp. 272–277.

Minato, S. (1996). *Binary Decision Diagrams and Application for VLSI CAD*, Kluwer Academic Publishers, Boston.

NVIDIA Corporation (2011a). http://www.nvidia.com.

NVIDIA Corporation (2011b). CUDA (Compute Unified Device Architecture). `http://www.nvidia.com/object/cuda_home.html`.

Shi, C.-J. & Tan, X.-D. (2000). Canonical symbolic analysis of large analog circuits with determinant decision diagrams, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 19(1): 1–18.

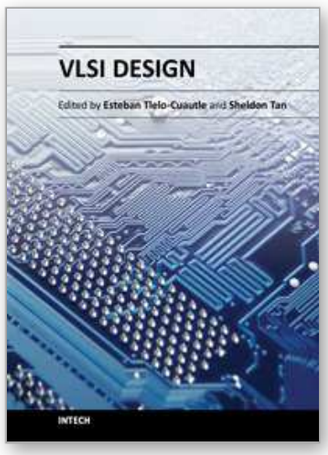
Shi, C.-J. & Tan, X.-D. (2001). Compact representation and efficient generation of s-expanded symbolic network functions for computer-aided analog circuit design, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 20(7): 813–827.

Shi, G. (2010a). Computational complexity analysis of determinant decision diagram, *IEEE Trans. on Circuits and Systems II: Analog and Digital Signal Processing* 57(10): 828 –832.

Shi, G. (2010b). A simple implementation of determinant decision diagram, *Proc. Int. Conf. on Computer Aided Design (ICCAD)*, pp. 70 –76.

Tan, S. X.-D., Guo, W. & Qi, Z. (2005). Hierarchical approach to exact symbolic analysis of large analog circuits, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 24(8): 1241–1250.

Tan, X.-D. & Shi, C.-J. (2000). Hierarchical symbolic analysis of large analog circuits via determinant decision diagrams, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 19(4): 401–412.



VLSI Design

Edited by Dr. Esteban Tlelo-Cuautle

ISBN 978-953-307-884-7
Hard cover, 290 pages
Publisher InTech
Published online 20, January, 2012
Published in print edition January, 2012

This book provides some recent advances in design nanometer VLSI chips. The selected topics try to present some open problems and challenges with important topics ranging from design tools, new post-silicon devices, GPU-based parallel computing, emerging 3D integration, and antenna design. The book consists of two parts, with chapters such as: VLSI design for multi-sensor smart systems on a chip, Three-dimensional integrated circuits design for thousand-core processors, Parallel symbolic analysis of large analog circuits on GPU platforms, Algorithms for CAD tools VLSI design, A multilevel memetic algorithm for large SAT-encoded problems, etc.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Sheldon X.-D. Tan, Xue-Xin Liu, Eric Mlinar and Esteban Tlelo-Cuautle (2012). Parallel Symbolic Analysis of Large Analog Circuits on GPU Platforms, VLSI Design, Dr. Esteban Tlelo-Cuautle (Ed.), ISBN: 978-953-307-884-7, InTech, Available from: http://www.intechopen.com/books/vlsi-design/parallel-symbolic-analysis-of-large-analog-circuits-on-gpu-platforms

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen