Parallel Statistical Analysis of Analog Circuits by GPU-accelerated Graph-based Approach

Xue-Xin Liu, Sheldon X.-D. Tan, and Hai Wang
Dept. Electrical Engineering, University of California, Riverside, CA 92521

Abstract—**In this paper, we propose a new parallel statistical analysis method for large analog circuits using determinant decision diagram (DDD) based graph technique based on GPU platforms. DDD-based symbolic analysis technique enables exact symbolic analysis of vary large analog circuits. But we show that DDD-based graph analysis is very amenable for massively threaded based parallel computing based on GPU platforms. We design novel data structures to represent the DDD graphs in the GPUs to enable fast memory access of massive parallel threads for computing the numerical values of DDD graphs. The new method is inspired by inherent data parallelism and simple data independence in the DDD-based numerical evaluation process. Experimental results show that the new evaluation algorithm can achieve about one to two order of magnitudes speedup over the serial CPU based evaluations and 2–3 times speedup over numerical SPICE-based simulation method on some large analog circuits.**

I. Introduction

It is well known that analog and mixed-signal circuits are very sensitive to the process variations as many matchings and regularities are required. This situation becomes worse as technology continues to scale to 90 nm and below owing to the increasing process-induced variability [1], [2]. For example, due to an inverse-square-root-law dependence with the transistor area, the mismatch of CMOS devices nearly doubles for each process generation less than 90 nm [3], [4]. To consider the impacts of process variations on circuit performance. Monte-Carlo based statistical approach is the most reliable solutions to this problem. But the prohibitive computational costs of Monte Carlo method perverts it from solving large analog circuits.

Modern computer architecture has shifted towards designs that employ multiple processor cores on a chip, so called multi-core processor [5], [6]. The graphic processing unit (GPU) are one of the most powerful many-core computing systems in mass-market use. For instance, NVIDIA Telsa T10 chips have a peak performance of over 1 TFLOPS versus about 80–100 GFLOPS of Intel i5 series Quad-core CPUs [7]. 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) [8]. Accordingly, the introduction of new parallel programming interfaces for general purpose computations, such as Computer Unified Device Architecture (CUDA) [9], Stream SDK [10] and OpenCL [11], has made GPUs powerful and attractive for developing high-performance tools of solving practical engineering problems. Parallelization on GPU platforms is an emerging strategy to improve the efficiency of Monte-Carlo based statistical analysis method. But traditional numerical simulators based on LU decomposition such as SPICE is difficult to be parallelized on GPUs due to irregular memory access and huge memory-intensive operations.

Graph-based symbolic technique is a viable tool for calculating the behavior or characteristic of analog circuits [12]. The introduction of determinant decision diagrams based symbolic analysis technique (DDD) allows exact symbolic analysis of much larger analog circuits than all the other existing approaches [13], [14]. Furthermore, with hierarchical symbolic representations [15], [16], exact symbolic analysis via DDD graphs essentially allows the analysis of arbitrary large analog circuits. Once the small-signal characteristics of circuits are presented by DDDs, evaluation of DDDs, whose CPU time is proportional to the size of DDDs, will give exact numerical values. One important observation is that the DDD-based simulation is very amenable for parallel computing as the main computation is distributed to each DDD node (via graph traversals) and the data dependency is very simple due to the simple binary graph structure.

In this article, we develop efficient parallel graph-based simulation technique based on GPU computing platforms for Monte-Carlo based statistical analysis of analog circuits. We design novel data structures to represent the DDD graphs in the GPUs to enable fast memory access of massive parallel threads for computing the numerical values of DDD graphs. The new method is inspired by inherent data parallelism and simple data independence in the DDD-based numerical evaluation process. Experimental results show that the new evaluation algorithm can achieve about one to two orders of magnitudes speedup over the serial CPU based evaluations of analog circuits and 2–3 times speedup over numerical SPICE-based simulation method on some large analog circuits. Further more, 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 [17], [18].

This paper is organized as follows. Section II outlines DDD-

This research was supported in part by NSF grants under No. CCF-1116882, No. CCF-1017090, No. OISE-1130402, and No. OISE-0929699.

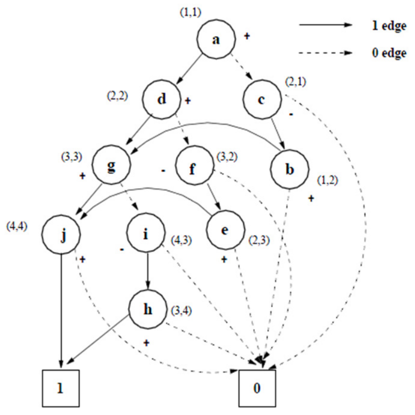


Fig. 1: DDD representation for matrix M.

based symbolic analysis techniques. Then, we introduce the flow of the proposed GPU Monte Carlo simulation. Section IV describes the proposed GPU parallel algorithm, followed by several numerical examples in section V. Last, Section VI concludes the paper.

II. DDD 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.

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

We can express each element using a node in a diagram and each product term using a path going through four nodes. For every node, it has a value of itself and a sign attached to it. The sign of each product term is decided by multiplying the sign of every node in the corresponding path. When the matrix M is an circuit matrix (such as modified nodal analysis (MNA) matrix), the value for each node 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 nodes, namely the *0-terminal* node and the *1-terminal* node. Each non-terminal DDD node 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 node 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 nodes pointed by the *1-edge* and *0-edge* of a_i. The *1-terminal* node represents expression 1, whereas the *0-terminal* node represents expression 0. For example, node h (in Fig. 1) represents expression h, and node i represents expression $-ih$, and node g represents expression $gj - ih$. We also say that a DDD node g represents an expression defined the DDD subgraph rooted at g. For each node, there are two values, v_{self} and v_{tree}. In (2), v_{self} represents the value of the element itself, which is D_{a_i}; while the v_{tree} 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 node (a in our example) to the *1-terminal* including all symbols and signs of the nodes 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 node represents the sum of these product terms. Size of a DDD is the number of DDD nodes, 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.

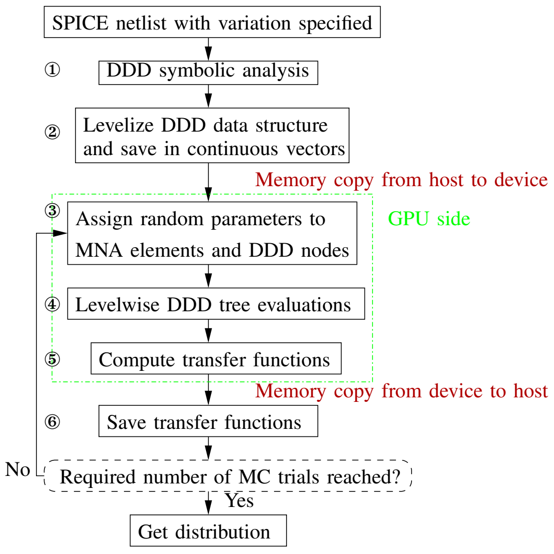
III. The proposed graph-based parallel statistical analysis

In this section, we first provides an overview of our graph-based GPU-based parallel statistical analysis before the detailed explanation.

As mentioned before, in DDD-based analysis, computing numerical value of the determinant of the DDD essentially boils down to the depth-first traversal of the graph. 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 statistical frequency analysis of analog circuits, evaluation of a DDD node at different frequency points and different Monte-Carlo runs can be performed in parallel. We show that all those parallelism will be explored by the new statistical analysis approach on GPU platforms.

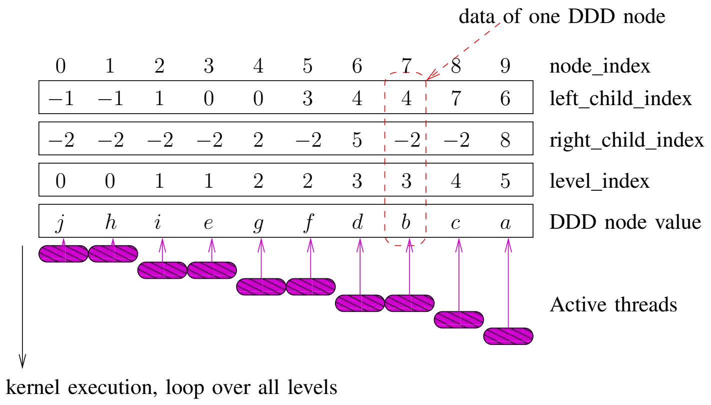
A. The overall algorithm flow

Fig. 2 gives the overall flow of our statistical method. The whole algorithm has two main parts, the CPU part (host) and GPU part (device) as clearly marked in the figure. CPU part mainly reads the netlist, generate the original DDD tree structures and builds new continuous DDD vector array structure (for GPU) and outputs the final numerical results. GPU part takes care of the main parallel DDD evaluation and communicates with CPU. The new program reads input netlist containing variation information of the relevant circuit devices. Then, the analyzer builds the MNA (modified nodal analysis) matrix and DDD binary tree data structure [13] as shown in step ①.



SPICE netlist with variation specified
①
DDD symbolic analysis
②
Levelize DDD data structure and save in continuous vectors
Memory copy from host to device
③
Assign random parameters to MNA elements and DDD nodes
GPU side
④
Levelwise DDD tree evaluations
⑤
Compute transfer functions
Memory copy from device to host
⑥
Save transfer functions
No
Required number of MC trials reached?
Yes
Get distribution

Fig. 2: The flow of GPU-based parallel Monte Carlo analysis.



0	1	2	3	4	5	6	7	8	9	node_index
-1	-1	1	0	0	3	4	4	7	6	left_child_index
-2	-2	-2	-2	2	-2	5	-2	-2	8	right_child_index
0	0	1	1	2	2	3	3	4	5	level_index
j	h	i	e	g	f	d	b	c	a	DDD node value

Fig. 3: Levelized continuous storage of a DDD, and levelwise GPU evaluation of the DDD in Fig. 1.

B. New continuous and levelized DDD structure

To prepare for the GPU computing, we need to build new data structures from the original binary tree DDD structures. This will be done in the CPU as the construction only needs to be performed once and traversal of original DDD linked trees is still sequential in nature and will be difficult to handle in GPU, as labeled ② in Fig. 2.

For GPU computing, the main challenge is to allow fast memory access by threads or reduce memory traffic as much as possible by using shared memory (or texture memory) within blocks so that GPU cores can be busy all the time. In GPU, fast global memory access by threads can be done by coalesced memory access where a half warp (or a warp) of threads (16 or 32 threads respectively) can read their data from the global memory in one read access. Coalesced memory access requires that data are arranged continuously in memory and consecutive with respective to involved thread indexes. As a result, we need to remap the linked DDD trees into a memory-continuous data structure.

The second issue is that we do not need to perform the DDD node evaluation for all the DDD nodes. Only those nodes whose children have been evaluated should be computed by threads (one thread for one DDD node). This can be done by sorting the DDD nodes by their *level*. Two DDD nodes have same level if they have the same number of edges on their longest path to the *1-terminal*. For instance, node g and node f has the same level in Fig. 1. DDD nodes at the same level can be computed in parallel in GPU. As we can see, the largest level of DDD nodes will be bounded by the numbers of non-zeros in a determinant. But practically, number of level can be much less than the number of non-zeroes. For instance, we have 5 levels in the DDD shown in Fig. 1 versus 10 nonzero elements.

In the new DDD structure, all the DDD nodes at the same level will be put in continuous and consecutive memories (mainly the v_{self} and future v_{tree} values) and be assigned to threads (one DDD node per thread) at the same time (one kernel launch). The level assignment can be done by simple depth-first traversal of the DDD graph. After this, we can allow the continuous memory for all the DDD nodes for one level starting from the lowest level until the highest level. We use the DDD example in Fig. 1 again to illustrate the new data structure shown in Fig. 3. For each value associated with a DDD node such as its value (v_{self}), left child index, right child index, level index, sign (not shown), a linear array will be generated based on the level indexes of DDD node. For example, node b in Fig. 1 becomes the 7-th element in the vector, and the index of its children, g and 0-terminal, are 4 and -2 accordingly. Note that, by our definition, 1-terminal's index is -1, and 0-terminal's is -2. Those arrays then will be copied into GPU memory for future DDD evaluation.

Fig. 3 also shows the execution pattern of GPU threads during DDD evaluation where we start with the DDD nodes in the lowest level and continue one level at a time until we hit the highest level. Since all nodes of the same level have been reorganized into one continuous memory segment, the active GPU threads working on them can achieve coalesced read/write access and also minimize the occurrence of branch divergence. As we observed, consecutive and levelwise data format improves the performance of GPU by 2–3× for large sized circuits.

IV. New parallel GPU-based Monte-Carlo analysis method

A. Random number assignment to MNA elements and DDD nodes

For statistical analysis, we need to generate variations from devices into the elements of the determinant and then into the data in the continuous DDD data structure. Due to MNA formulation, each device may appear 4 positions in a MNA matrix. Hence we track and save the MNA stamp patterns of circuit devices, and also their locations in DDD, during DDD construction. These data are transferred to GPU texture memory as texture memory are read-only and can be accessed much faster than GPU global memory.

Algorithm 1 Parallel random value assignment for DDD nodes

```
1:  for all Monte Carlo runs do // launch threads in grids
2:      Assign random numbers to involved device parameters
        and stamp MNA elements.
3:      Save each DDD node's admittance, capacitance, and
        inductance components as $R[k] = \{g, c, l\}$.
4:      for all DDD nodes do // launch threads in grids
5:          Load frequency values to $f$.
6:          for all frequencies do // launch threads in a block
7:              $v_{\text{self}}[i] = R[k].g + j \cdot (R[k].c \cdot f[i] + R[k].l/f[i])$
8:          end for
9:          Save $v_{\text{self}}$.
10:     end for
11: end for
```

Next, in random number assignment, CURAND libray is used to generate variations on nominal values of circuit parameters in GPU kernel function. We need to make sure that one device variation, which may appear in 4 position in the MNA will take the same value and this also reflect on the f the four DDD nodes will reflect the same change. This is done in Line 2 and Line 3 of the pseudo-code in Algorithm 1. The variations introduced in our experiments are Gaussian random values, whose means and deviations can be specified by users from input netlist.

Note that since we perform the frequency domain analysis, we need to evaluate the MNA and DDD on all frequency points of interest. To enable coalesced memory access to compute DDD values for many frequencies, as Line 5 and Line 9, the DDD continuous structure will be further changed so that all frequency responses of the same element or node reside in consecutive memory addresses. We observe that this frequency related calculation is very suitable for intra-block GPU computing as all the threads in a block can share the same DDD information (except for the frequency values).

In GPU, the threads are organized into grids (can be two dimensional) and number of grids can be as large as 64K and each grid contains a block and each block can have as many as 1024 threads (in current GPU families from NVIDIA) and they can be organized in 3 dimensions. Threads in a block can communicate via shared or texture memory and can be explicitly synchronized. In our problem, the dimension of the grid is set to $N_{MC} \times |DDD|$, i.e., the number of Monte Carlo runs times the number of DDD nodes (assume that it is less than 64K) and each block of this grid contains TILE_DIM threads, where TILE_DIM is multiply of 16 to enable coalesced access on neighboring frequency responses and is also set with consideration of available GPU resources per block. In practice, we set TILE_DIM $=$ 256. So we can allow to compute 256 frequency responses for one DDD node. Notice that all the three FOR loops in Algorithm 1 will be replaced by massive thread launches in parallel. The two outer FOR loops are parallelized at grid level, and the innermost FOR loop is parallelized at block level. Hence, the DDD node values v_{self} are computed for all Monte Carlo

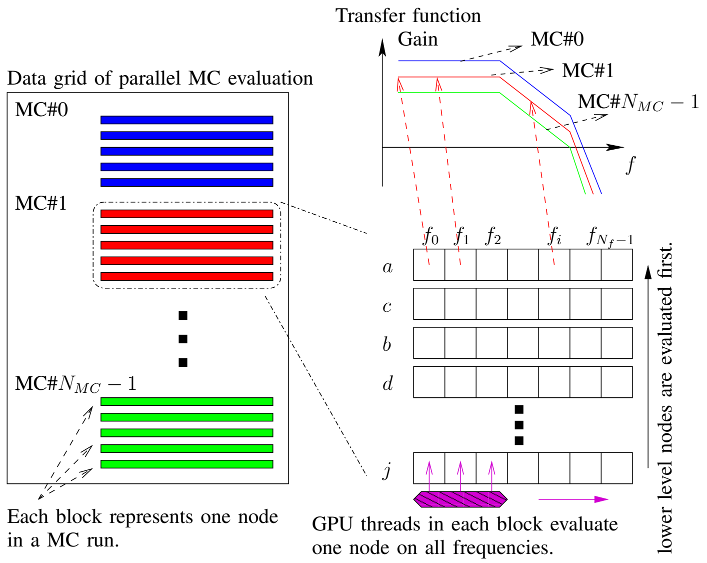

Transfer function
Gain
MC#0
MC#1
MC#$N_{MC}-1$
f
Data grid of parallel MC evaluation
MC#0
MC#1
MC#$N_{MC}-1$
f_0 f_1 f_2 f_i f_{N_f-1}
a
c
b
d
j
Each block represents one node in a MC run.
GPU threads in each block evaluate one node on all frequencies.
lower level nodes are evaluated first.

Fig. 4: GPU parallel evaluation of the DDD in Fig. 1.

runs and all frequency points in their respective blocks and threads (Line 7). If number of frequency points is larger than TILE_DIM, the innermost FOR loop will be kept inside the kernel function. But instead of loop over each frequency point, we loop over TILE_DIM frequency points every time.

The number of Monte Carlo runs in each kernel launch is determined by the GPU specification and the allocated resources, such as global memory, to each Monte Carlo calculation. For a typical μA741 circuit whose DDD contains 6205 nodes and 2400 evaluated frequency points, the Tesla C2070 can allow 20 Monte Carlo runs in parallel. In case more runs are required, the steps from ③ through ⑥ in Fig. 2 are repeated as many times as needed.

B. Parallel evaluation of DDDs

The evaluation of DDD is a process that computes the final numerical value of the determinant it represents. This procedure is labeled with ④ in Fig. 2. As we previously discussed in Section III-B, the data structure of DDD has been remapped to GPU friendly continuous and consecutive arrays and are sorted by level to enhance efficiency of evaluation.

Similar to the GPU calculation of DDD node values mentioned in previous subsection, we also launch independent blocks for different Monte Carlo runs and different DDD nodes, and use each thread block to calculate values of each node's v_{tree} for all frequency points, which is depicted in Fig. 4.

Algorithm 2 lists the main flow of this algorithm. To ensure that the nodes are evaluated from bottom to top, the first FOR loop iterates the level index from 0 to the maximum level in the DDD, and launches kernel function on the DDD nodes of the specific level, one at a time. Note that we keep this FOR loop in CPU control, instead of moving it inside the GPU kernel, in order to accomplish inter-block synchronization. This is necessary because we deploy the evaluation of different nodes in different thread blocks, and, if there is no synchronization,

Algorithm 2 Parallel Monte Carlo evaluation of DDDs

```
1:  for level=0 to top_level do // CPU host iteration
2:      for all Monte Carlo runs do // launch threads in grids
3:          for all DDD nodes do // launch threads in grids
4:              if node.level == level then
5:                  Load $v_{\mathrm{self}}$ of the current node, and $v_{\mathrm{tree}}$ of its children.
6:                  for all frequencies do // launch threads in a block
7:                      Evaluate $v_{\mathrm{tree}}$ for the current node by Eq. (2) on all frequencies.
8:                  end for
9:                  Save current node's $v_{\mathrm{tree}}$.
10:             end if
11:         end for
12:     end for
13: end for
```

it is possible that a node of higher level gets evaluated before its children. Moreover, CUDA only provides synchronization among threads in a block, the kernel has to be finished if all blocks in the kernel grid are required to be synchronized. Therefore, in our implementation, the index of current level is passed into the kernel function as an argument, and the kernel will evaluate those thread blocks with the same level indicated by the argument index.

The coalesced memory access to the node's v_{self} and its children's v_{tree} values are also ensured in the load and save operations in Line 5 and Line 9, because during the evaluation of the current node on all frequencies, the k-th thread will work on the k-th frequency, and all threads in a warp execute the same code path. Consequently, such a kernel launching exhibits a highly data intensive pattern, and reduces global memory traffic at the same time.

V. Experimental results

To show the performance of the proposed GPU parallel Monte Carlo simulation, we test the program on several industrial benchmark circuit netlists. For running time comparisons, we also measure the time cost by the CPU version of DDD evaluation and HSPICE.

All of our programs are implemented in C++, with NVIDIA CUDA for the GPU computation part. All running time are sampled from a Linux server with an 2.4 GHz Intel Xeon Quad-Core CPU, and 36 GBytes memory. The GPU card installed on this server is Tesla C2070, which contains 448 cores running at 1.15 GHz and up to 5 GBytes global memory.

Now let us investigate one typical example in detail. Fig. 5 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. 6, is used for DDD symbolic analysis and numerical evaluation. The AC analysis is performed with the variation of several circuit components for Monte Carlo simulation. Several Monte Carlo samples of the

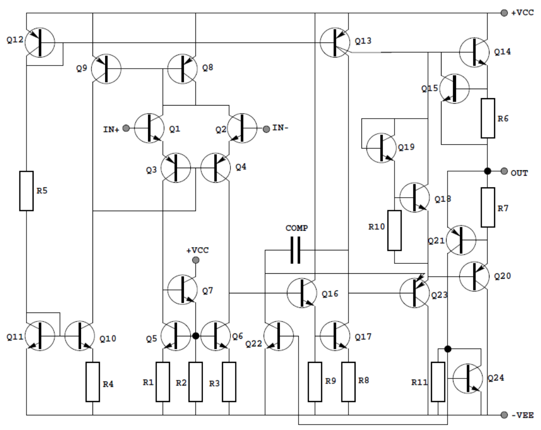


Fig. 5: The circuit schematic of μA741

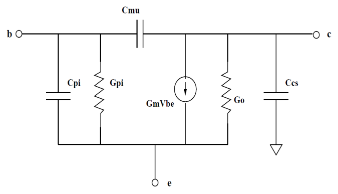


Fig. 6: The small signal model for bipolar transistor

magnitude response are plotted in Fig. 7. The 3-db bandwidth of all the statistics is calculated and shown in the histogram in Fig. 8. In this example, the nominal 3-db frequency is 1.2 kHz. As we can observe from Fig. 8, the histogram of the bandwidth frequency is similar to the Gaussian distribution.

Next, we study the speedup and scalability of the GPU and CPU DDD based Monte Carlo simulations. The measurements of time taken by both programs running on the same RC tree circuit are shown in Table I, where different number of Monte Carlo runs are tested. It is obvious that the speedup of GPU method over the CPU one is significant. Also, when the number of Monte Carlo runs increases, GPU running time

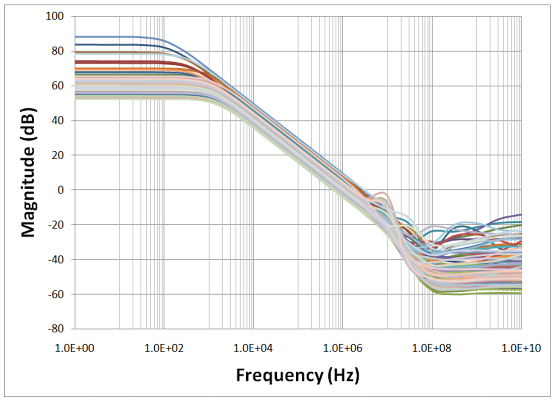


Fig. 7: The cluster of frequency responses of the tested μA741 circuit

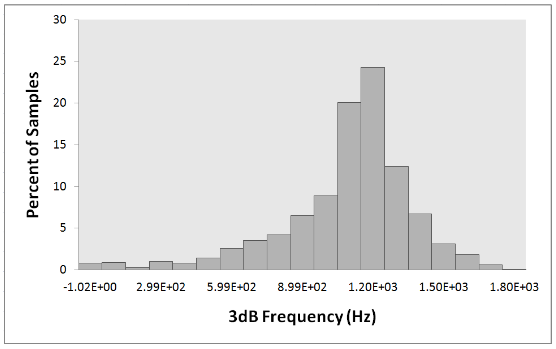


Fig. 8: Histogram diagram of the 3-db points for all these results

TABLE I: Performance comparison of CPU serial and GPU parallel DDD evaluation for RC tree circuit

# MC Runs	GPU time (s)	CPU time (s)	Speedup
1	1.98	23.0	11
2	2.08	46.2	22
4	2.21	90.5	41
8	2.50	183.8	73
16	3.03	364.1	120
32	4.76	725.3	152
64	8.68	1442	166
128	17.42	2910	167

does not multiply as fast as the CPU version does, provided that the GPU resources can accommodate parallel execution of these Monte Carlo evaluations in one kernel launch. Hence, in this way, all the GPU streaming multiprocessors are kept busy and the throughput is maximized, which results in a striking speedup over the CPU serial version.

Last, we list the results of all benchmark tests in Table II. The information of the circuits and their DDD representation is also included in the same table. The 2nd through 5th column record number of nodes in circuit, number of elements in the MNA matrix, number of DDD nodes in the generated DDD graph, number of determinant product terms, respectively.. The last three columns summarize the run-time of GPU parallel algorithm, serial algorithm and the HSPICE. The number of Monte Carlo runs for all tests is set to 128. It is clear from this table that the GPU-accelerated version outperforms its CPU counterpart, and also achieves 2–3 times speedup over the commercial HSPICE on a variety of test circuits.

TABLE II: Performance comparison of GPU, CPU, and HSPICE Monte Carlo simulations

circuit name	# cir. nodes	# cir. devices	$\lvert DDD \rvert$	DDD terms	GPU time (s)	CPU time (s)	HSPICE time (s)
bigtst	32	112	642	2.68×10^7	19.7	3143	38.4
ccstest	9	35	109	260	0.80	108	2.5
rlctest	9	39	119	572	1.05	145	2.6
vcstst	12	46	121	536	0.73	104	3.8
ladder21	22	64	64	28657	2.10	365	5.1
ladder100	101	301	301	9.27×10^{20}	30.6	3965	42.5
rctree1	40	119	211	1.15×10^8	5.55	928	11.3
rctree2	53	158	302	4.89×10^{10}	17.42	2910	46.1
μA741	23	89	6205	363914	59.1	6243	73.6

VI. Conclusion

A new parallel statistical analysis method for large analog circuits using determinant decision diagram (DDD) based graph technique is proposed. To make it amenable for massively threaded based parallel computing GPU platforms, we designed novel data structures to represent the DDD graphs in the GPUs to enable fast memory access of massive parallel threads for computing the numerical values of DDD graphs. The new method is inspired by inherent data parallelism and simple data independence in the DDD-based numerical evaluation process. Experimental results show that the new evaluation algorithm can achieve about one to two order of magnitudes speedup over the serial CPU based evaluations and 2–3× speedup over numerical SPICE-based simulation method on some large analog circuits.

References

[1] R. Rutenbar. Next-generation design and EDA challenges. In *Proc. Asia South Pacific Design Automation Conf. (ASPDAC)*, January 2007. Keynote speech.

[2] S. Nassif. Model to hardware correlation for nm-scale technologies. In *Proc. IEEE International Workshop on Behavioral Modeling and Simulation (BMAS)*, Sept 2007. Keynote speech.

[3] H. Masuda, S. Ohkawa, A. Kurokawa, and M. Aoki. Challenge: Variability characterization and modeling for 65- to 90-nm processes. In *Proc. IEEE Custom Integrated Circuits Conf.*, 2005.

[4] J. Kim, K.D. Jones, and M.A. Horowitz. Fast, non-monte-carlo estimation of transient performance variation due to device mismatch. In *Proc. IEEE/ACM Design Automation Conference (DAC)*, 2007.

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

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

[7] David B. Kirk and Wen-Mei Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 2010.

[8] Dominik Göddeke. General-purpose computation using graphics harware. http://www.gpgpu.org/, 2011.

[9] NVIDIA Corporation. CUDA (Compute Unified Device Architecture), 2011. http://www.nvidia.com/object/cuda_home.html.

[10] AMD Inc. AMD Steam SDK. http://developer.amd.com/gpu/ATIStreamSDK, 2011.

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

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

[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*, 19(1):1–18, Jan. 2000.

[14] C.-J. Shi and X.-D. Tan. 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, April 2001.

[15] 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*, 19(4):401–412, April 2000.

[16] 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*, 24(8):1241–1250, August 2005.

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

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