GPU Acceleration of Near-Minimal Logic Minimization

Ibrahim Savran
Computer Science and Engineering,
University of South Carolina, SC 29208
Email: savran@email.sc.edu

Jason D. Bakos
Computer Science and Engineering,
University of South Carolina, SC 29208
Email: jbakos@email.sc.edu

***Abstract*—In this paper, we describe a GPU-accelerated implementation of a logic minimization heuristic based on the near minimal approach. This algorithm has three key kernel computations, and the current version of our implementation, we adapted one of these kernels for GPU execution. In this paper we report our results gained from using NVIDIA's CUDA development framework and an NVIDIA Tesla GPUs, achieving a nearly 10X speedup as compared to a software implementation executed on a Xeon 5500-series processor.**

***Index Terms*—Logic Synthesis, GPU Computing, Heterogeneous Computing, SIMD.**

I. Introduction

As VLSI integration levels increase, greater emphasis is being placed on integrating complex special-purpose logic within system-on-chip designs in order to support specific applications such as real-time signal processing, graphics acceleration, and high-speed packet-switched off-chip interfaces [1]. Such designs often include large-scale logic subsystems. In order to support design flows for such designs, there is a need for high-performance logic synthesis tools.

A substantial portion of logic synthesis tools is to synthesize combinational logic, where HDL is interpreted as boolean functions, which must ultimately be minimized and converted into a netlist of gates that meet timing and area constraints.

Logic optimization is divided into two-level and multi-level logic minimization. In two-level minimization the combinational logic is represented in two level form, such as sum-of-product form (SOP) or product-of-sum form (POS). In order to reduce the degree of fan-in, logic described in two-level form is converted into a multi-level form using an interactive refinement algorithm. However, in this design flow, performing minimization of the two-level logic is by far the most computationally expensive step.

For minimizing two-level logic, there are several methods that produce the optimal (or exact) solution [2-5]. Due to the exponential nature of this problem, optimal solvers are typically limited to functions with up to one hundred products [2]. Therefore, most of the practical applications rely on heuristic minimization methods [3-6].

While heuristic algorithms are much faster than the exact ones, they are still extremely expensive to compute for minimizing large-scale logic functions solving [3, 7]. Furthermore, the heuristic algorithms display diversity in realizations. That is, no single heuristic algorithm is consistently better than the others for all logic functions. There are classes of functions where one heuristic algorithm is better than the others [4].

Graphics Processor Units (GPUs) are well-suited for executing algorithms that exhibit a high-degree of data-level parallelism and little control dependence [8]. This is because GPUs are optimized to provide high throughput computation and tolerate frequent long-latency accesses to memory.

While there is a substantial body of work for parallelizing logic minimization heuristics for multi-processor and cluster-based platforms [9,10], to the best of our knowledge, GPU implementation of these algorithms has yet to be explored. In this paper, we describe our preliminary efforts to explore this area. The heuristic that we target is a cube-based approach, which is of the latest generation of minimization algorithms (map-based and table-based algorithms are the first two generations).

Using only a single previous-generation GPU, our current implementation achieves a speedup of nearly 10X over single-threaded software on a state-of-the-art processor (Xeon 5500).

II. Background

In this section we briefly introduce the theoretical foundations of logic minimization.

Definitions: Let f be a Boolean function $f : B^n \rightarrow B$. A variable is a symbol representing a single coordinate of a Boolean space B^n (i.e x,y). A *literal* is a variable or its negation, (e.g. $x, \bar{y}, z$). A *cube* is a conjunction of literals (e.g. xy, $xy\bar{z}$). A *full cube* is a don't care assignment to every variable.

If $B \in \{0, 1\}$, f is called completely specified Boolean function. A completely specified function can be represented by a set of cubes. This representation is known as a two-level *sum of product* representation (SOP).

A *cover* is a set of cubes that potentially contain don't care variable assignments. A cover is *minimum* if it contains the minimum possible number of cubes that cover all the terms in the on-set.

An incompletely specified Boolean function f is defined $f : B^n \rightarrow \{0, 1, -\}$. In such formulations, the function can be described has having an on set and off set, which are enumerations of each variable assignments that yields an output of true and an output of false, respectively. In this case,

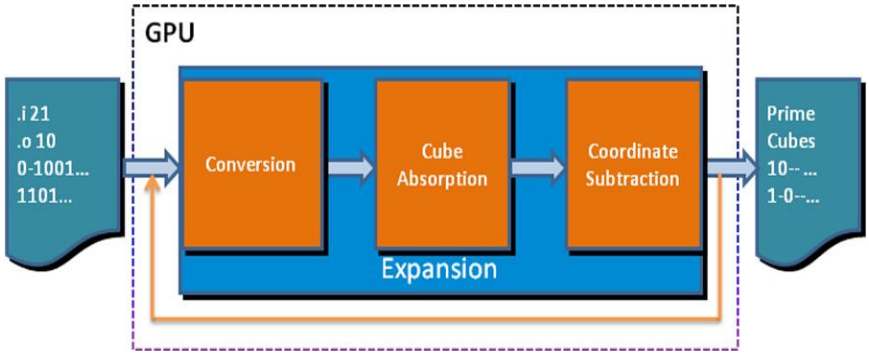

Fig. 1. Expansion Algorithm

the non-specified inputs are assumed to be don't cares. In other words, the on-set, off-set, and don't care-set are respectively F, R, D, where $F : B^n \rightarrow \{1\}, R : B^n \rightarrow \{0\}, D : B^n \rightarrow \{-\}$.

The algorithm that we target in this paper performs on-set expansion, a method that is based on one of the cube based logic minimization algorithm [6].

Figure 1 depicts the top-level steps of our target algorithm. The input is a logic function description that is specified by its corresponding on-set and off-set. An example input for a four-variable function is shown below:

$ON - SET = \{0\,0\,1\,-, 0\,1\,0\,1\}$

$OFF - SET = \{1\,1\,1\,1, 0\,1\,1\,1\}$

In the conversion step, a single cube of the on-set, c, is selected, and a new set of cubes, BYPROD, is created by comparing c to each member of the off-set, and generating a cube where each variable is assigned to X if there is a match and the value from the off-set if there is a mismatch. Following the example above, if $c = 0101$, the resultant BYPROD set would be $\{1\,-\,1\,-, -\,-\,1\,-\}$.

In the cube absorption step, the size of BYPROD is reduced by absorbing any members that are covered by other members. In the example, the second cube, $-\,-\,1\,-$, would absorb the first cube, $1\,-\,1\,-$, creating a new BYPROD $= \{-\,-\,1\,-\}$.

In the coordinate subtraction step, each member of BYPROD is subtracted from a full cube, $-\,-\,-\,-$. When subtracting, each variable of the cube that is not a don't care is inverted. In the example, this would give $-\,-\,0\,-$, which covers c.

III. Approach

In our implementation, we adapted each of these three sub-steps to the CUDA programming model. (see the algorithm of Expansion Kernel)

We instance one thread for each member of the off-set (organized as 32 threads per block). During the conversion step, the BYPROD set is generated and stored in shared memory.

During the absorption step, each member of the BYPROD set must be compared against every other member. Recall that the size of the BYPROD set is equal to the size of the off-set, allowing each thread to associate itself with a member of BYPROD. However, in order for each thread to compare its member of BYPROD to every other thread's member, the threads will need to communicate across block boundaries. As a result, before this comparison can be performed, a copy of the BYPROD set must be stored to global device memory. The comparison operation requires that each thread loop for $|BYPROD|$ iterations. The results of this comparison are stored in shared memory.

For the coordinate subtraction step, the remaining elements of the the BYPROD set are subtracted from a full cube, and the results are stored back to device memory in order to send it back to the host.

Algorithm: Expansion Kernel

1) $\forall$ cube c_i of Set F
   a) q_j = Convert(c_i,r_j) /*BYPRODUCT Computation*/
   b) $B = B \cup q_j$;
   /*Sub-kernel Absoption */
   c) $\forall\, p_k$ of set B where $k \neq j$ do
      i) if p_k absorbs q_j remove q_j from the set B;
      /*Sub-kernel Coordinate Subtraction*/
   d) Do $Coor.Subt$ on B from the cube $\{- - - \ldots\}$ (full cube)
   e) Select the first Prime Cube from the result set of the last Sub-kernel function

IV. Results

We compared the performance of our accelerated minimization implementation to a single-threaded, CPU-only for a sample set of benchmark input functions gathered from Microelectronics Center of North Carolina (MCNC) [11]. The benchmarks were selected by choosing the largest benchmarks available.

Table 1 lists our results. The run times shown in the table are measured from the launch of the top-level application to completion, whichfor the accelerated implement, includes all host-GPU communication, I/O, and host-side processing. All runtimes are arithmetic means over ten runs.

Our test platform is a Dell PowerEdge R710 server containing a Xeon 5500-series processor, connected to a Tesla S1070

#	Benchmark Name	GPU Time (in ms)	x86 Time (in ms)	Ratio	# of On-set	# of Of-set
1	alu4.pla	446.83	769.60	1.72	644	667
2	pdc.pla	4438.03	33981.05	7.65	520	6255
3	spla.pla	1703.78	11248.51	6.60	749	2947
4	apex4.pla	3347.80	21454.57	6.41	1731	2779
5	cordic.pla	898.96	4894.55	5.45	245	818
6	duke2.pla	296.18	522.35	1.76	914	1191
7	ex1010.pla	766.84	2008.13	2.62	742	1209
8	misex3.pla	1946.03	4852.24	2.49	1290	2027
9	table3.pla	5746.96	53693.34	9.34	643	6366
10	table5.pla	6072.71	59295.42	9.76	606	6985

TABLE I
TIME COMPARISON OF THE ALGORITHM

server via dual eight-lane PCIe cables. Note that we used only one GPU for these tests.

As shown in the table, our speedups ranged from approximately 2X to 10X depending on the benchmark tested. In general, the performance achieved was a linear function of the size of the off-set.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we described a GPU-accelerated implementation of a logic minimization heuristic. Our strategy for parallelizing this algorithm was based on the observation that the complexity of the three primary kernels of the algorithm were related to the size of the off-set. As such, we associate the off-set with the threads blocks.

This effort also demonstrates that GPUs can be effective for accelerating non-arithmetic combinatorial algorithms, as opposed to numerical floating-point-based algorithms for which GPUs are generally believed to be best suited.

We believe there is much room for improvement. In particular, the performance can be improved by implemented the outermost loop, i.e. choosing each c from the on-set, entirely on the GPU. This will greatly reduce the effect of host-GPU communication and synchronization. Another possible improvement is to utilize the GPU's constant memory for storing the off-set, since this set is invariant during the course of this algorithm. Finally, because the algorithm mainly consists of SIMD operations, it can potentially be implemented in different ways. For example, instead of assigning every thread to find a prime at a time, it is possible to create a kernel for each cube in the set F. This idea is more amenable for multi-GPU implementation.

REFERENCES

[1] J.E. Stine, J. Grad, I. Castellanos, J. Blank, V. Dave, M. Prakash, N. Iliev, N. Jachimiec, "A framework for high-level synthesis of system on chip designs," Proc. 2005 IEEE International Conference on Microelectronic Systems Education, 2005 (MSE '05), June 12-14, 2005, pp 67-68.

[2] T. Sasao, Worst and Best Irredundant Sum-of-Product Expressions, IEEE Trans. Comp., Vol. 50, No 9, 2001, pp. 935-947.

[3] A. Mishchenco, T. Sasao, Large-Scale SOP minimization Using Decomposition and Functional Properties, DAC 2003, pp 149-154.

[4] P.P. Tirumalai, J.T. Butler, Minimization Algorithms for Multiple-Valued Programmable Logic Arrays, IEEE Transactions on Computers, Vol. 40, No 2, 1991, pp.167-177.

[5] R.K. Brayton, G.D. Hachtel, C.T. McMullen and A.L. Sangiovanni-Vincentelli, "Logic Minimization Algorithms for VLSI Synthesis" Boston, MA, Klewer Academic Publishers, 1984.

[6] S. Kahramanli, S. Gunes, S. Sahan, F. Basciftci, A new Method Based on Cube Algebra for the Simplification of Logic Functions, Arabian Journal for Sci. and Eng., Vol 32, No 1B, 2007, pp 101-114.

[7] P. Fier, H. Kubtov, "Two-Level Boolean Minimizer BOOM-II," Proc. 6th Int. Workshop on Boolean Problems (IWSBP'04), Freiberg, Germany, Sep. 23-24, 2004, pp 221-228.

[8] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In SIGGRAPH, 2004.

[9] J. Bobba, A.M. Kumar, V. Kamakoti Parallel Partitioning Techniques for Logic Minimization using Redundancy Identification, In: HiPC 2003.

[10] K. De, P. Banerjee, Parallel Logic Synthesis Using Partitioning, International Conference on Parallel Processing 1994, (ICPP 1994), Aug. 15-19 1994, Vol 3, pp 135-142.

[11] S. Yang, Logic synthesis and optimization benchmarks, ver. 3.0, Tech. Rep., Microelectronics Center of North Carolina, 1991.