A Parallel GPU Implementation of the *TimberWolf* Placement Algorithm

Ahmad Al-Kawam and Haidar M. Harmanani
Department of Computer Science and Mathematics
Lebanese American University
Byblos, 1401 2010, Lebanon

***Abstract*—GPUs have been gaining acceptance in the electronic design automation field as attractive platforms for implementing and accelerating computationally extensive applications. Researchers agree that it is critical that EDA algorithms exploit future platforms and explore the use of parallel algorithms as we move to the *manycore* era. This paper describes the implementation of the *TimberWolf* placement algorithm using CUDA and demonstrates the applicability of GPUs in accelerating electronic design automation tools. The algorithm has been implemented on a Xeon Workstation using C, and achieved a substantial *acceleration* on an *Nvidia* Tesla C2070 card.**

I. Introduction

The electronic design automation field has been successful in meeting the *time-to-market* challenge by bridging the gap between system specification and manufacturing. However, most of the problems that arise in VLSI CAD are $\mathcal{NP}$-complete and often require either heuristic solutions or partitioning into smaller tractable problems. Although this approach has successfully worked in various areas such as synthesis, it often compromised the results quality. Over the past two decades, EDA tools have matured in various domains and thus there is a fundamental need to exploit and extend such tools in order to explore scalable design methodologies [3].

Graphics Processing Units (GPUs) are now ubiquitous general purpose *manycore* devices that have been designed as graphics accelerators. GPUs are being increasingly used for high-performance general non-graphics applications due to their low cost and easy accessibility. Another factor that has contributed to GPUs recent popularity is the emergence of parallel computing platforms adapted to GPUs such as CUDA [12] or OpenCL [7]. A GPU includes several multi-threaded SIMD *streaming multiprocessors* (SMs) that are composed of multiple *CUDA cores* that share registers. The *Tesla C2070* from *Nvidia* has 14 SMs, 448 cores, 6 GB memory, and 32,768. A developer writes in CUDA scalar code for one function designed to be executed on the device, namely a *kernel*. At runtime, many *threads* are created by *blocks* and bundled into a *grid* to run the same kernel concurrently on the device. Each block is assigned to a SM. Within each block, threads are executed by groups of 32 called *warps*.

This paper presents a parallel CUDA-implementation of the *TimberWolf* placement application [15]. The method illustrates the applicability of GPUs in the electronic design automation field. The remainder of the paper is organized as follows. Section II presents the placement problem and briefly reviews recent trends in EDA hardware acceleration. Section III describes the *TimberWolf* algorithm while section IV describes the algorithm's GPU parallel implementation. We present our experimental results in section V and conclude with remarks in section VI.

II. The Placement Problem

Placement is a fundamental task in physical design, and is a crucial step in the VLSI design cycle that consists of *partitioning*, *floorplanning*, *placement*, and *routing*. The placement phase starts with a placement region, a set of standard cells or macros where each block has a specific shape, the number of terminals for each block, and the circuit netlist. The objective of the placement is to determine the location of each block in the *netlist* while minimizing the achievable routable design and improving the circuit performance. A placement is acceptable if 100% routing can be achieved within a given area where routing is the process of assigning actual tracks to wires that connect ports. Placement may include additional constraints such as the reduction of thermal hotspots as well as power consumption.

The placement problem has been shown to be $\mathcal{NP}$-hard; even the simple case of placing a circuit with only unit-size modules and 2-pin nets along a straight line with the objective to minimize total wire length is $\mathcal{NP}$-complete [5]. Typically, the number of cells to be placed is very large, and therefore, it is impractical to take a brute force approach. Over the past three decades, various efficient heuristic placement algorithms have been proposed. There has been recently a renewed interest in the circuit placement problem due to the emerging interconnect issues in deep sub micron design.

Despite the recent advances in general purpose GPUs (GPGPUs), it is very uncommon to find commercial electronic design automation tools that harness the power and speed of the GPUs [11]. There has been, on the other hand, a recent resurgence in academic research that tackles parallel acceleration for electronic design automation algorithms with a focus on GPUs. This is especially true in the case of FPGA placement problem whose computational time has been increasing exponentially due to the continuous increase in FPGA capacity which has been growing faster than CPU speed [1, 4, 6, 9, 14]. Most of the FPGA GPU research harnessed the GPU's computational power for accelerating computationally intractable problems. Other researchers tackled mainstream

978-1-4799-8828-0/15 $31.00 © 2015 IEEE
DOI 10.1109/ITNG.2015.144

VLSI design automation problems. For example, Suresh et al. [16] and Bertacco et al. [2] used GPUs for event-driven simulation of digital circuits on GPUs, and succeeded in achieving a 13x speedup on average. Han et al. [8] proposed a floorplanning algorithm using GPUs. The algorithm achieved 6−88x speedup for a range of MCNC and GSRC benchmarks. Li et al. [10] proposed a parallel fault simulator that exploits the high degree of parallelism supported by GPUs.

III. *TimberWolf* ALGORITHM

TimberWolf is a stochastic standard cell placement algorithm based on simulated annealing that was initially proposed by Sechen et al. [15]. *TimberWolf* operates in two stages. During the first stage, the algorithm explores the solution space by moving cells between rows while allowing modules overlaps using the following moves: (a) M_1: move a module from one location to another that can be in the same or in a different row; (b) M_2: pairwise interchange two modules that can be in different rows. (c) M_3: change the orientation of a module by mirroring its x-coordinates. During every iteration, the algorithm randomly selects a move based on the following probabilities: $\text{Prob}_{M_1} = 0.8$ and $\text{Prob}_{M_2} = 0.2$. The algorithm applies M_1 and M_2 within a *range limiter* which is a rectangular window of height H_T and width W_T. For M_1, the window is centered at the center of the randomly selected cell. A random location within the window will be chosen as the destination of the cell. For M_2, a swap will be attempted only if the window can be positioned such that it contains both centers of the two randomly selected cells. Initially, W_T and H_T are large enough to include the whole chip. The *range limiter* shrinks as the temperature decreases as a function of $\log(T)$. If M_1 is selected but the new configuration is rejected, then M_3 will be attempted for the same cell with a probability of $\text{Prob}_{M_3} = 0.1$.

The second stage is triggered when the vertical span of the *range limiter window* is less than the *center-to-center* spacing between the rows. During the second stage, the algorithm allocates feed-through cells and eliminates cell overlaps. This is accomplished through a process that sorts all cells within a row according to the x-coordinate of the centers before they are replaced from the left edge of the row. During this stage, the algorithm does not allow moves of type M_1 while moves of type M_2 interchanges adjacent cells only if they are in the same row. After a move is carried out, the cost is evaluated and the move is accordingly accepted or rejected. The algorithm attempts a bounded number of moves during every temperature where the optimal number of iterations is empirically determined. The algorithm starts at a very high initial temperature which is reduced according to the following schedule $T_{i+1} = \alpha(T_i) \times T_i$. Initially, the temperature is rapidly decreased using a cooling factor of $\alpha(T_i) = 0.8$ and then at a slower pace where $\alpha(T_i) = 0.9$ before going back to $\alpha(T_i) = 0.8$. The algorithm terminates when $T < 1$.

Algorithm 1 Parallel_Move()

Input: Set of all cells V

```
1:  r ← RANDOM(0,1)
2:  if ( r ≤ 0.8) then                          ▷ Do M1 with probability of 0.8
3:      number_of_threads = p                   ▷ GPU threads is set to the number of
    positions
4:      cell1 ← RANDOM(0,k)
5:  end if
6:  for all (threads) do
7:      if ( pos(thread_id) is inside range) then      ▷ Thread's position is
    inside the range limiter
8:          pos ← pos(thread_id) ▷ set destination to the 1st valid position
9:      end if
10: end for
11: if (rowpop(pos) == max_row_pop ) then                          ▷ Row is full
12:     Reject
13: else if ( pos not empty) then
14:     Make_Space
15:     Pos ← cell1
16: else
17:     Pos ← cell1
18: end if
19: if (reject) then
20:     r ← RANDOM(0,1)
21:     if (r ≤ 0.1) then                          ▷ if M1 is rejected, apply M3
22:         do_M3
23:     end if
24: else                                    ▷ Apply M2 with probability of 0.2
25:     number_of_threads = k                  ▷ GPU threads = number of cells
26:     cell1 = RANDOM(0,k)
27: end if
28: for all (threads) do
29:     if ( cell(thread_id) is inside range) then
30:         cell2 = cell(thread_id)
31:         temp ← cell1                                          ▷ Swap Cells
32:         cell1 ← cell2
33:         cell2 ← temp
34:     end if
35: end for
```

IV. GPU *TimberWolf* PARALLELIZATION

We implemented the *TimberWolf* algorithm using a 2D lattice. Each location on the lattice is occupied by a *cell*. The lattice has a predefined maximum number of columns that corresponds to the maximum number of cells per row. The number of rows is equal to the number of cells which is equal to the worst-case height of a placement that may result from having all rows of length one. The lattice is large enough to represent any possible *floorplan*. The size of the matrix would then be equal to the number of placed cells multiplied by the maximum capacity of each row.

A. Multithreaded Neighborhood Transformations

TimberWolf is a computationally expensive algorithm where the optimal number of neighborhood moves depends on the size of the circuit. In general, as the temperature decreases, it becomes more difficult to find an acceptable move within the *range limiter*. For example, 100 moves per cell are recommended for a circuit with 200 cells. If one assumes an initial temperature, $T_0 = 125$ then 2.34×10^6 configurations will be evaluated. The number of configurations increases to 247.5×10^6 for a $3,000$-cell circuit and 700 moves per cell [13].

Algorithm 2 Find_Span()

Input: Set of all cells V
Output: Vertical and Horizontal Spans

```
1: Number_of_threads = nNets
2: for all (threads) do
3:     C ← nNet_Cells
4:     while (C > 0) do
5:         if ( row(Net_Cell (C) ) > max_row) then
6:             max_row ← row(Net_Cell (C)
7:         end if
8:         if ( row(Net_Cell (C) ) < min_row) then
9:             min_row ← row(Net_Cell (C)
10:        end if
11:        if ( col(Net_Cell (C) ) > max_col) then
12:            Max_col ← col(Net_Cell (C)
13:        end if
14:        if ( col(Net_Cell (C) ) < min_col) then
15:            min_col ← col(Net_Cell (C)
16:        end if
17:        C ← C − 1
18:    end while
19:    span(thread_id) = max_row − min_row + max_col  − min_col
20: end for
```

We accelerate the neighborhood transformations using multithreading. The algorithm spawns n threads that correspond to the number of cell locations on the lattice. For synchronization purposes, `thread 0` selects a placed cell and then all other threads select randoms location within the *range limiter* as explained in section III. For neighborhood transformation M_2, the destination cell location must be a placed cell. The algorithm checks with each thread until a valid destination cell is found before interchanging the cells. Neighborhood transformation M_1 requires that the destination cell location be an empty location. Furthermore, there should not be any empty rows between the source and the destination. The algorithm checks with each thread at a time until a valid location is found. Otherwise, M_3 will be applied.

B. Cost Function Evaluation

TimberWolf evaluates the solution after every move using the following cost function:

$$C = \sum_{i \in Nets} (w_i^H \times X_i + w_i^V \times Y_i) + w_2 \sum_{i \neq j} [O_{ij}]^2 + w_3 \sum_{rows} |l_R - \overline{L_R}|$$

Where:

- w_i^H and w_i^V are horizontal and vertical weights, respectively. For any net i, if the horizontal and vertical spans are given by X_i and Y_i then the estimated length of the net i is $(X_i + Y_i)$.
- w_2 is a penalty weight for module overlaps.
- w_3 is a weight that indicates unevenness as uneven distribution of row lengths results in wastage of chip area.

It should be noted that the cost function in the second stage of the algorithm is effectively the first part of the cost function only since there is no cell overlap, and the last component is constant since cells are not allowed to change rows.

During the cost function computation, the algorithm computes the *horizontal* and *vertical* spans of the net which are defined as the smallest rectangle that encloses all of the pins comprising the net in the x and y direction, respectively. This is the largest bottleneck in the *TimberWolf* algorithm, since there is a large number of nets, and each net has a considerable amount of connections. The algorithm loops through all the nets, and through all their connections to calculate the span of each net. This is computed in parallel using n threads that cooperate to calculate the vertical and horizontal spans using all cells that are connected to that net. The net is next multiplied by the cost of each net so that to yield the total interconnection cost. This parallel function reduces the number of iterations of the number of connections of each net, which is usually much smaller than the number of nets. The bounding area is calculated by multiplying the effective number of rows by the effective row length.

TABLE I
COST FUNCTION EVALUATION SPEED-UP

Circuit	Sequential	Parallel	Speedup
fract	0'30	0'22	1.36
primary1	0'308	0'160	1.93
industry1a	2'599	0'460	5.65
industry1	2'609	0'470	5.55
primary2	6'039	0'620	9.74
biomed	38'340	0'711	53.92

V. RESULTS

We implemented the proposed parallel algorithms using C on a Xeon 2.67 GHZ Workstation with an *Nvidia* Tesla C2070 GPU. We tested the sequential and GPU accelerated *TimberWolf* algorithms on the MCNC benchmarks whose characteristics are shown in Table II. The parallel algorithm was able to find a solution for the largest benchmark, *biomed*, that has $6,417$ cells in around 14 minutes. The sequential algorithm, in contrast, needed more than a day to find the same solution. The parallelization of the cost function achieved a substantial speed-up, especially in the case of large benchmarks where it was 53 (Table I). In general, the parallel implementation achieved an average speed-up of 55 across all benchmark examples as shown in Table III and Figure 1.

It was noticed that as the number of move attempts, M, increased, the speed-up also increased. The rationale is that the parallel cost evaluation function is much faster than the sequential cost evaluation function. And since the cost function needs to be evaluated in every metropolis loop and at every temperature, parallelization would substantially improve the

TABLE II
CHARACTERISTICS OF MCNC BENCHMARKS

Circuit	#Cells	#Pads	#Nets	#Pins	#Rows
fract	125	24	147	462	6
primary1	752	81	904	5526	16
primary2	2907	107	3029	18407	28
industry1	2271	814	2594	8513	15
industry1a	2271	580	2479	8513	15
biomed	6417	97	5742	26947	46

TABLE III
PARALLEL PLACEMENT RESULTS FOR THE MCNC BENCHMARKS

Name	Sequential (s'ms)	Parallel (s'ms)	Speedup
Fract	13'556	10'215	1.33
primary1	1488'711	76'670	19.42
Industry1	12516'977	223'33	56.12
Industry1a	13078'770	223'642	58.48
Primary2	30407'376	331'255	91.79
Biomed	24+ hour	841'903	102.73+

overall performance of the algorithm. The cooling schedule was determined as follows: $T_{init} = 65536.0$, which is the largest possible float value; α was set to start at 0.8 and then increases to 0.96 before it decreases again to 0.8. Finally, the metropolis loop multiplier, M, was set to 100. Although this configuration of M is nowhere near the minimum accepted value and should be typically set to the number of cells, it was necessary to set it at such a low value in order for the sequential algorithm to finish in a relatively short time.

VI. CONCLUSION

We presented in this paper a GPU-accelerated implementation for the *TimberWolf* placement algorithm. The algorithm proposes a multithreaded implementation for exploring the search space as well as for parallelizing the cost function. The proposed parallel implementation was able to achieve substantial speedups over the sequential algorithm. This demonstrates that the high degree of parallelism that was exploited by our implementation can be used to harness the power of the GPUs in electronic design automation in order to produce more powerful and efficient results.

REFERENCES

[1] M. An, J. G. Steffan, and V. Betz. Speeding Up FPGA Placement: Parallel Algorithms and Methods. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 178–185, May 2014.

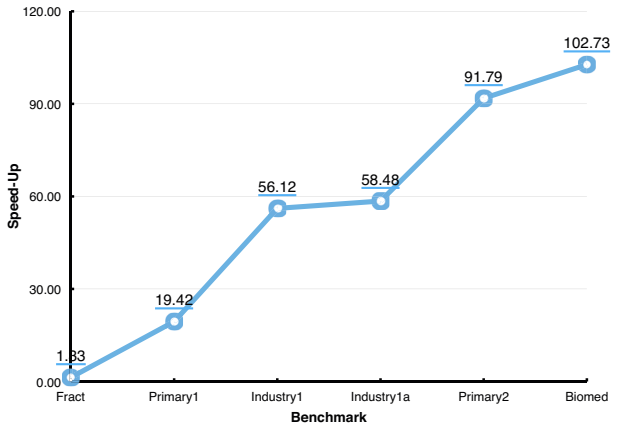

Fig. 1. Achieved Speed-up for Benchmark Examples

[2] V. Bertacco and D. Chatterjee. High Performance Gate-Level Simulation with GPGPU Computing. In *VLSI Design, Automation and Test*, pages 1–3, 2011.

[3] R. Brayton and J. Cong. NSF Workshop on EDA: Past, Present, and Future (Part 2). *IEEE Design & Test of Computers*, 27(3):62–74, 2010.

[4] A. Choong, R. Beidas, and Z. Jianwen. Parallelizing Simulated Annealing-Based Placement Using GP-GPU. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 31–34, 2010.

[5] C. Chu. Placement. In L.-T. Wang, Y.-W. Chang, and K.-T. Cheng, editors, *Electronic Design Automation: Synthesis, Verification, and Test*. Morgan Kaufmann, 2009.

[6] C. Fobel, G. Grewal, R. Collier, and D. Stacey. GPU Approach to FPGA Placement Based on Star+. In *IEEE International New Circuits and Systems Conference (NEWCAS)*, pages 229–232, 2012.

[7] Khronos Group. The OpenCL Specification Version 2.0, http://www.khronos.org/registry/cl/specs/opencl-2.0.pdf, 2014.

[8] Y. Han, K. Chakraborty, S. Roy, and V. Kuntamukkala. Design and Implementation of a Throughput-Optimized GPU Floorplanning Algorithm. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 16(3):23:1–23:21, 2011.

[9] B. Huang and H. Zhang. Application of Multi-Core Parallel Computing in FPGA Placement. In *International Symposium on Instrumentation and Measurement, Sensor Network and Automation*, pages 884–889, 2013.

[10] M. Li and M. Hsiao. 3D Parallel Fault Simulation With GPGPU. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 30(10):1545–1555, 2011.

[11] A. Maghazeh, U.D. Bordoloi, P. Eles, and Z. Peng. General Purpose Computing on Low-Power Embedded GPUs: Has it Come of Age? In *International Conference on Embedded Computer Systems*, pages 1–10, 2013.

[12] NVIDIA. *CUDA C Programming Guide, Version 6.5*, 2014.

[13] S. Sait and H. Youssef. *VISI Physical Design Automation: Theory and Practice*. World Scientific Publishing, New York, 1999.

[14] M. Sanjabi, A. Jahanian, S. Amanollahi, and N. Miralaei. ParSA: Parallel Simulated Annealing Placement Algorithm for Multi-Core Systems. In *International Symposium on Computer Architecture and Digital Systems*, pages 19–24, 2012.

[15] C. Sechen and A. Sangiovanni-Vincentelli. The TimberWolf Placement and routing Package. *IEEE Journal of Solid-State Circuits*, 20(2):510–522, 1985.

[16] L. Suresh, N. Rameshan, M.S. Gaur, M. Zwolinski, and V. Laxmi. Acceleration of Functional Validation Using GPGPU. In *IEEE Symposium on Electronic Design, Test and Application*, pages 211–216, 2011.