Parallel Cross-Layer Optimization of High-Level Synthesis and Physical Design

James Williamson[1], Yinghai Lu[2], Li Shang[1], Hai Zhou[2,3], Xuan Zeng[3]
[1]ECEE, University of Colorado, Boulder, U.S.A., [2]EECS, Northwestern University, U.S.A.
[3]State Key Lab of ASIC & System, Microelectronics Dept., Fudan University, China

***Abstract*—Integrated circuit (IC) design automation has traditionally followed a hierarchical approach. Modern IC design flow is divided into sequentially-addressed design and optimization layers; each successively finer in design detail and data granularity while increasing in computational complexity. Eventual agreement across the design layers signals design closure. Obtaining design closure is a continual problem, as lack of awareness and interaction between layers often results in multiple design flow iterations. In this work, we propose parallel cross-layer optimization, in which the boundaries between design layers are broken, allowing for a more informed and efficient exploration of the design space. We leverage the heterogeneous parallel computational power in current and upcoming multi-core/many-core computation platforms to suite the heterogeneous characteristics of multiple design layers. Specifically, we unify the high-level and physical synthesis design layers for parallel cross-layer IC design optimization. In addition, we introduce a massively-parallel GPU floorplanner with local and global convergence test as the proposed physical synthesis design layer. Our results show average performance gains of 11X speed-up over state-of-the-art.**

I. Introduction

The mainstream hierarchical design methodology of modern VLSI CAD is to separate the design flow into a sequence of design optimization steps ranging in abstraction from system-level design exploration down to the physical design. Reusing prior work reduces the complexity of each design step, and abstraction makes early-stage design optimization feasible without being overwhelmed by the low-level design details. However, with the increasing role of physical effects such as interconnects, process variation, and power and thermal profiles in the final design's cost, this separation between layers makes the overall CAD process more difficult and error prone. Primarily, this is because the physical effects can only be obtained accurately after performing the physical design. Without knowing physical information during the early design stages, inaccurate or even incorrect decisions are often made early on with irreversible impact during the rest of the IC design process. Iterations of the entire design flow are therefore required to meet design closure and thus the overall process is expensive, resulting in serious design closure failings, increased time to market, and increased IC cost.

To solve the design closure problem, accurate physical information must be obtained to guide the high-level synthesis process [1]. While unifying the high- and physical-levels has shown promise, it has suffered from extreme computational complexities incurred when calculating the physical ramifications of each potential high-level move. One major merit of a unified framework is that it iteratively improves the overall design quality by using incremental algorithms for high-level synthesis and floorplanning. Incremental improvements are based on previous results; so, it is much easier for these algorithms to maintain optimization continuity and data locality. However, since the low-level physical information such as wire length and power consumption must be taken into consideration for every candidate high-level move, the design space to be explored by each iterative improvement stage is huge. By our own study, the computational requirement of searching this space heavily dominates the high-level optimizations, needing over 93 percent of total execution time. Thus, even when incremental adjustment is applied the computational cost is still prohibitively high.

To make practical the idea of unifying layers of the CAD process, available parallelism within each design layer must be identified, extracted, and mated to a suitable compute architecture. With the recent trend toward parallel computing driven by the emergence of powerful multi-/many-core microprocessors and the supporting parallel programming environments, parallel CAD research has been rejuvenated. In [2], a parallel-moves placement strategy was pipelined, with each pipeline stage occupying its own processing core. Recent studies leveraging the emerging many-core GPU with NVIDIA's CUDA platform [3] have focused on CAD problems that are inherently data parallel [4], [5], [6], [7], [8], [9]. For example, in [10], a well known analytical cell placer [11] was paralleled on a many-core GPU, exploiting high data parallelism. In [12], the high-level global placement layer of a multi-level analytical placer [13] was again accelerated through a GPU co-processor.

Though most of the studies provide significant speedup, they bear the following two constraints. First, these studies target only a single layer in the design space for parallelization. Secondly, their solution is formed for one specific architecture, namely either multicore CPU or the GPU, to exploit parallelism in accelerating a pre-existing flow. Further, these recent studies have shown little success for CAD problems with complicated control characteristics, and none of these recent studies consider cross-layer optimization of the IC design automation flow. This absence in the literature can be attributed to the complexities of the heterogeneous control, data parallelism, and communication characteristics required for a cross-layer parallel optimization approach.

For the first time, we introduce parallel cross-layer optimization for unified high-level and physical synthesis. The major contributions of this work include:

- This is the first work for parallel cross-layer optimization. Though we use unified high-level and physical synthesis as our case study, our parallel flow can be applied to other multi-level flows.
- We leverage the parallel computational power from both the CPU and GPU, and we fit them to meet the heterogeneous computational requirements across the design layers.
- We take advantage of unique conditions present in our parallel cross-layer flow, and further optimize the traditional simulated annealing floorplanning approach to achieve a 24 percent speedup without sacrificing quality of results.
- We apply high- and physical-level synthesis to the parallel cross-layer optimization technique and show on average 11X speed-up compared to state-of-the-art work.

The rest of the paper is organized as follows. In Section II, we introduce parallel cross-layer optimization with the nondeterministic transactional model and discuss its application in combined high-level and physical level synthesis. Section III describes how to apply our parallel cross-layer optimization algorithm to a heterogeneous computing system, Section IV introduces our novel massively parallel GPU floorplanner, and in Section V we demonstrate the experimental result. Finally, Section VI concludes our work.

II. Parallel Cross-Layer Optimization

In this work, we deliver a parallel algorithm solution for cross-layer power optimization of unified high-level and physical synthesis using a nondeterministic transactional model. Through the cross-layer optimization framework, we globally optimize the decisions made in the individual layers to produce an IC holistically minimized for power. We investigate how to uniformly specify parallel high-level and physical-synthesis algorithms, how to design cross-layer interaction and optimization between high-level and physical synthesis, and how to expose the intra-layer and cross-layer parallelisms to speed up the overall optimization flow.

This research is supported in part by the NSF under awards CCF-0954157 and CCF-0811270.

We adapt the framework of [14] to the idea of parallel high-level and physical-level synthesis. The input to the cross-layer optimization algorithm is a CDFG G, an input arrival (and output sampling) period T_s, and a library L of function units (FUs) for data-path implementation. With the given input, it explores the design space by doing an incremental search from initial solutions of each different combination of candidate supply voltages and control steps. Upon completion, it produces an RTL circuit whose total power consumption and estimated area are optimized.

Our algorithm starts by generating a candidate set P of valid combination of supply voltages and control steps. $\forall p \in P$, we store a synthesized data path and physical solution, a corresponding weighted cost considering power consumption and area, and the current supply voltage and control step. These properties are denoted as $p.sol$, $p.cost$, $p.voltage$ and $p.cstep$. Additionally, we add a control flag $p.flag$ to each solution to monitor the progress of the algorithm. The algorithm initializes the solution with a fully-paralleled assignment with the fastest available FU from the library that implements each operation. An as-soon-as-possible (ASAP) schedule is then generated for the initial solution to determine whether it meets its timing requirements. Starting from this family of initial solutions, an iterative improvement phase attempts to improve each candidate architecture by reducing the switched capacitance, while still satisfying the sample period constraints. In each iteration of the improvement phase, a high-level test movement is generated. To measure its impact at the physical level, a floorplan for the modified HLS solution is generated, and lower-level physical information such as area, wire length, and wire capacitance is evaluated. The quality of the current HLS solution is then evaluated with the updated physical information. If the test solution is better, it will be kept. Such incremental exploration is performed for each supply-voltage and control-step configuration combination.

Notice that the order of examining supply voltage and control step pairs (V_{dd}, C_s) is not important, as long as all the pairs are explored. In order to explore the parallelism and communication characteristics across high-level and physical layers in the cross-layer optimization algorithm, we leverage the mechanism of the nondeterministic transactional model. In the nondeterministic transactional model, an algorithm is specified as an initialization followed by a loop of guarded commands. A guarded command is composed of a boolean condition, called the guard, and an assignment. When a guard is true, the corresponding assignment can be executed. If multiple guards are true, one or more commands can be arbitrarily selected for execution. Parallelism is exposed, and heterogeneity is encouraged. Selection repeats until none of the guards are true, whereby the algorithm produces the result. Such a model was adopted by [15] in developing a parallel min-cost flow solver.

Leveraging the nondeterministic transactional model, the proposed parallel high- and physical synthesis algorithm is expressed in Figure 1. Figure 1 consists of an initialization and a loop of three groups of guarded commands. The initialization generates the initial solutions for $p \in P$ and assesses their costs. After the initialization phase, the algorithm enters the **do**-loop and executes until all the guarded commands become invalid. The **do**-loop is the main body of our algorithm corresponding to the iterative-improvement phase. In Figure 1, there are three groups of guarded commands, with execution conditional on the set flag for each $p \in P$. These guarded-command groups are HLS-movement generation, floorplan generation, and cost evaluation. Notice that, in each group of guarded commands, there are $|P|$ commands, where $|P|$ is the cardinality of valid voltage-supply and control-step pairs. The first group of commands generate incremental HLS moves based on the current $p.sol$. Such moves include module rebinding, resource sharing/splitting, followed by rescheduling in order to meet the timing constraint.

The second group of guarded commands carries out physical synthesis based on the updated data path for p to accurately evaluate the power consumption due to the test HLS move. In our current algorithm, floorplanning is performed for physical synthesis; which determines the specific physical location of, shape of, and interconnect between, the actual hardware functional units in a composed data path. Notice that for each configuration $p \in P$, our flow

```
Generate and initialize P and set best as φ
do
  ∃p ∈ P : p.flag = SYN → Try generate HLS movement for p.
    if succeeded
      Update p.sol and p.flag := PHY
    else
      p.flag := BRK
    fi
  ∃p ∈ P : p.flag = PHY
    → Do floorplan for p, update p.sol and set p.flag := EVL
  ∃p ∈ P : p.flag = EVL → Update the cost p.cost
    if p.cost improved
      p.flag := SYN
    else
      p.flag := BRK
    fi
od
Output p with the lowest cost in P as best
```

Fig. 1. Nondeterministic transactional algorithm for unified high-level and physical synthesis.

generates the corresponding floorplan and then updates $p.sol$. With the physical synthesis done and the physical information now known, the algorithm returns to the high-level and activates the third group of guarded commands. These commands now accurately evaluate the power consumption of the configuration p. If the cost was improved due to the combined HLS move and updated physical layout, its flag is set as SYN, indicating that further improvement is possible. Otherwise, its flag is set as BRK, indicating that no further incremental improvement attempt on the specific p is needed.

The algorithm terminates when all the flags for P are set to BRK, which indicates that no better solution is found or the number of HLS movements has reached the user-defined threshold. On exit, our algorithm outputs the lowest cost configuration among P as the $best$, which is our decided upon result.

Our unified high-level and physical-level synthesis algorithm in Figure 1 demonstrates and leverages the three benefits of the nondeterministic transactional model. Firstly, the second group of guarded commands correspond to the floorplanning action, which can be further specified using finer guarded commands. The abstraction and composability of guarded commands helps us to more clearly specify the cross-layer algorithm. Secondly, the commands on the physical layer interact with the high-level synthesis commands through examining $p.flag$ in their guards. Fusion of optimizations across both high-level and physical layers is attained, which is the primary tenet of our parallel cross-layer optimization technique. Finally, abundant parallelism is exposed in Figure 1. At first look, commands are interdependent on $p.flag$. However, when we expand the group of guarded commands, we find that each command in the group can be executed independently, since commands treat each (V_{dd}, C_s) pair individually. The nondeterministic transactional model tells us that we can execute any of the transactional commands arbitrarily as long as the guards are satisfied. Due to the data isolation of each $p \in P$, the commands in each group can be executed in parallel. In summary, $|P|$ parallelism is available in our algorithm, where $|P|$ is the number of valid (V_{dd}, C_s) configuration pairs. In our experiments, $|P|$ is often on the order of 1000 and higher, depending on the benchmark, which suits extremely well to a multi-core/many-core architecture as we will see in the following section.

III. Mapping of Unified Cross-Layer Optimization to Heterogeneous Architectures

In this section, we discuss how the parallel cross-layer optimization technique is applied to a heterogeneous computing system, which is composed of a multicore CPU and multiple high-performance Nvidia GPUs. We will discuss in detail the appropriate programming considerations encountered and their corresponding performance tradeoffs.

A. Overall Framework

When implementing the high-level and physical level synthesis algorithm described in Figure 1, we need to partition the three

groups of guarded commands in between the CPU and the GPUs in order to best utilize the heterogeneous computing system. The first and third groups of commands, which correspond to the HLS operations, are mapped to CPU threads so as to facilitate high-performance sequential execution. The second group of commands, which corresponds to the physical synthesis, is mapped to the four GPUs for issuing as bulk CUDA kernels. Such partitioning is done mainly for three reasons. First, we cannot decouple a HLS move from its successor or predecessor move for parallelization due to the obvious dependency; but, we can alternatively serially generate many candidate (V_{dd}, C_s) configurations of the same move iteration, and concurrently evaluate the cost for each on a highly data-parallel GPU architecture. Second, serial dependencies and high control complexity in the HLS mates best with a sequential device that leverages deep pipelines, high instruction-level parallelism, and high instructions-per-second throughput. Moreover, the computational cost of these HLS operations takes only a small portion of the whole algorithm flow, so there is not much advantage in attempting parallel execution. Third, floorplanning operations are composed of many finely-grained and independent data manipulations; as such we are free to floorplan all HLS configurations concurrently, which mates extremely well with a high-throughput and data-parallel architecture like the GPU.

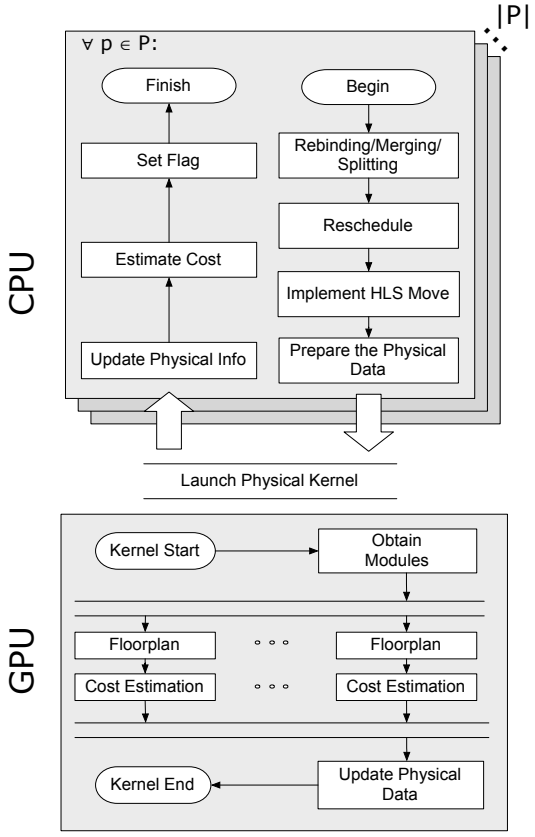


Fig. 2. Control flow sequences of high-level synthesis (HLS) and physical design.

With the guarded commands appropriated to the most suitable architectures, we can more closely examine the algorithms defined by them. We implement the loop body of Figure 1 with an iterative improvement structure. The control flow of high-level and physical synthesis in one iterative improvement iteration is shown in Figure 2. For each iteration, we address every (V_{dd}, C_s), i.e. p, configuration in P. For a single configuration p, we first generate a high-level incremental movement by rebinding, merging or splitting the functional module. Rescheduling is then carried out for current p to meet the timing requirement. The high-level RTL changes are recorded and the module and connectivity information is updated for physical design. After high-level movements are generated for each configuration $p \in P$, their physical information is updated all together by launching a floorplanning kernel on the GPU. Leveraging the massive number of cores in GPU, we can produce the floorplan for all $|P|$ configurations concurrently. The detail of the GPU floorplanning kernel will be described in Sections III-C and IV. After the floorplan and the associated physical information is updated by the GPU, we retrieve the physical information to send to the CPU. The power cost function for each configuration is then updated one by one. With the new cost, we can decide whether configuration p gains improvement or not, which is the last step in the current iteration. This process is carried out repeatedly until no more improvement can be achieved in all configurations.

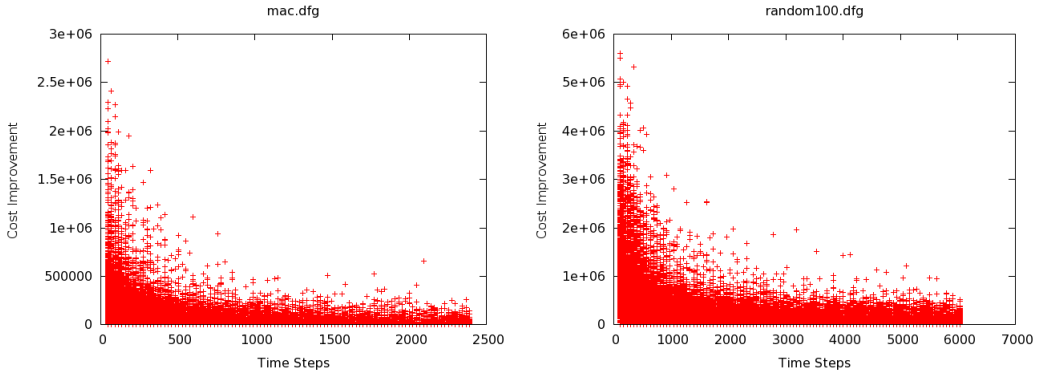
B. Parallel Control and Communication Characteristics

From system-level design exploration to physical synthesis, parallel CAD algorithms targeting different IC-design optimization layers exhibit distinct run-time behaviors, e.g. unique computation and communication characteristics. We now focus on how to best satisfy communication between layers, or equivalently, their synchronization.

In this work, we perform HLS and physical synthesis together in a single heterogeneous system, e.g. a CPU with GPU co-processor, with communication occurring across the system PCI-express bus. The bottleneck of the communication is the data transfer rate between the CPU and the GPU. Additional overhead from the GPU is also incurred from each GPU kernel launch. Our task is then to minimize transfer frequency, and maximize data transfer size. We do this by organizing the data of a CPU/GPU transaction to be large enough to overcome these overheads, but not so large as to stall either CPU or GPU computation. To this end, we coalesce small frequent communications into larger and more infrequent ones, and in doing so we pipeline communication with computation in both the CPU and GPU. The CPU iteratively generates and buffers candidate configurations intended for GPU physical design and cost evaluation, as they are too small to efficiently transfer alone. When enough data path candidates have been generated to make the PCI-express transaction profitable, a GPU kernel is launched. Meanwhile, the CPU generates further candidate configurations while the GPU performs the physical evaluation layer. Due to the hard architectural constraint of the PCI-express bus, even with our best efforts we are forced to tolerate some imbalance in execution times between the CPU and GPU. In the terms of our nondeterministic transaction framework outlined in Section II, high-level guarded command groups one and three communicate with group two physical-level commands by exchanging module type and connection information from the CPU and physical information from the GPU. Because we only issue the data for the GPU only once per HLS optimization iteration, communication granularity between layers is effectively coarsened, the number of GPU kernel launches are minimized, and computation can be overlapped with communication.

C. GPU-Driven Physical Design

For our algorithm in Figure 1, we bind each of the guarded commands in the third group to a computing thread of the GPU. Many threads are then batch executed in GPU kernels. Concurrently, each GPU thread in a kernel derives the floorplan for its candidate (V_{dd}, C_s) configuration and further evaluates the physical information from the floorplan such as total area and wire length. Since there can be thousands of configurations, and the GPU has hundreds of cores, multiple configurations are mapped to a single core in the GPU to be alternatively executed in groups of threads that in CUDA are called *warps*. Warps dictate the thread granularity at which the GPU hardware thread manager issues instructions, and they are sized to groupings of 32 threads in most recent Nvidia devices. In this way, we can make the maximum use of every computing core of the GPU; keeping them busy. We use simulated annealing (SA) for our floorplanner, as it is the most popular approach used in the floorplanning physical design stage [16], [17], [18], [19]. Sequence pairs are used to represent the configuration of the floorplan because they can be organized as arrays in GPU memory, which is an appropriate data access pattern for GPU. Similarly, the module block and net information for each (V_{dd}, C_s) configuration is also stored as arrays in the GPU global memory, due to their size. In order to



(a) MAC benchmark (b) RANDOM100 benchmark

Fig. 3. Cost improvement values seen throughout the floorplanning of all (V_{dd}, C_s) configurations at a single iterative improvement phase.

facilitate SA move generation, a method of randomization is needed. Currently, the CUDA programming model does not support efficient on-the-fly random-number generation during the kernel execution. Therefore, an array of random values is generated by the CPU and preloaded along with the data path information into the GPU memory, which is then subdivided for exclusive use in individual threads. After floorplanning is complete, the GPU kernel calculates a final cost describing power usage due to interconnect length and overall area of the circuit, for each of the configurations. All configuration results are stored in an array and passed back to the host CPU, which, now aware of the effects of every HLS move, can select the lowest-power result.

D. Multicore and Multi-GPU Implementation

With the techniques introduced by the previous subsections, the parallel high-level and physical synthesis flow has been completely specified. However, we can leverage the multicore CPU and multiple GPU cards available to further speed up the algorithm without compromising the quality of the result.

Using the multithreading mechanism, we create several threads and partition the three groups of guarded commands in Figure 1 according to the candidate configurations as $P = \{P_1, \ldots, P_k\}$. Here k is the number of threads created. Each thread i is in charge of its own partition of candidate configurations P_i. Since there is no data dependency between each individual configurations, the guarded commands for high-level moves can be executed in parallel across the threads. More importantly, since the CUDA programming model requires that each GPU processor in use be controlled through a unique CPU thread identification, multithreading in the CPU allows for the employment of multiple GPU cards for parallel floorplanning. As is previously mentioned, the number of valid (V_{dd}, C_s) configurations $|P|$ is usually on the order of a thousand while the number of cores that a GPU contains is on the order of a hundred. It is desirable that multiple GPUs are leveraged to make full use of the abundant parallelism in concurrent floorplan evaluation of $|P|$ configurations. For example, in our implementation, through four computing threads in the multicore CPU, we can drive four Tesla C1060 GPUs – by which we leverage up to 960 computing cores in total for floorplanning on the (V_{dd}, C_s) configuration set. With the exception that each thread now works on its own partition P_i instead of the whole configuration set P, the program flow for each thread is identical as that in Figure 2.

IV. Massively-Parallel Physical Design on GPUs

In this section, we propose a novel massively-parallel GPU floorplanner. In floorplanning, optimization attempts halt when further randomized moves begin to show little to no cost improvement, i.e., convergence occurs. As an example, Figures 3(a) and 3(b) show this convergence point occurring at roughly 600 and 1500 iterations respectively; i.e. when the improvement curves begin to flatten. To exploit massive GPU parallelism, this work concurrently floorplans up to thousands of (V_{dd}, C_s) configurations, using one GPU thread per floorplan. Due to the atomicity of the GPU kernel, an already converged GPU thread must wait for all other GPU threads in the kernel to converge before transferring results to the CPU. Early converging GPU threads must wait, attaining poor result improvement per cycle spent during this time. Late converging GPU threads add little overall benefit to the GPU kernel, while blocking converged threads from sharing results with the CPU. To address the above observations, we introduce a method to dynamically decide the global (kernel) termination of floorplan optimizations, utilizing local convergence test results of the individual GPU threads. The algorithm for local and global tests for convergence per GPU thread is shown in Figure 5, and is further detailed in the following two paragraphs.

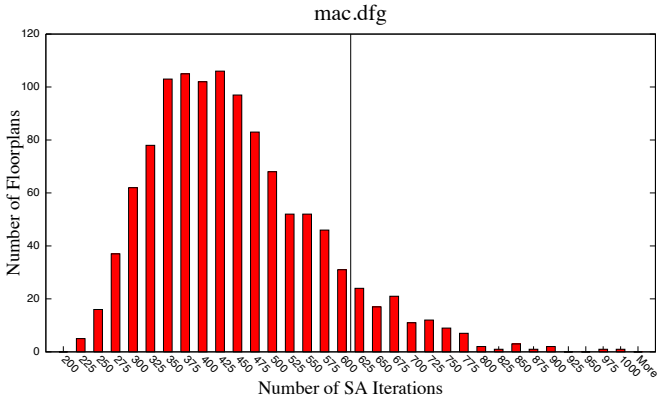


Fig. 4. Number of iterations needed to pass our two-level test for floorplan convergence for all 1,155 (V_{dd}, C_s) configurations at a single iterative improvement phase for the MAC benchmark. The vertical line indicates the global termination signal.

Our approach to determine local convergence for a single GPU thread is as follows. The local test for convergence is defined by two levels. The first level is satisfied when most moves stop yielding an improved cost. The most recent 60 moves are examined, and if 5 percent or less improve the cost, the second level test is activated. The second level examines the magnitude of the benefit of move improvements, and compares these against the most profitable move ever encountered. If all improved moves have 10 percent or less of the maximum profit ever encountered, both levels are satisfied and the local test for convergence is passed. The converged GPU thread then atomically increments a counter stored in the GPU global memory. This shared variable will be further explained in the paragraph below. Local test threshold values were determined from empirical profiling of the benchmark suite. Larger history lengths were found to improve accuracy, but increased memory usage and computing time.

We now explain our approach to determine global convergence for a GPU kernel. Before a round of floorplanning on the GPU, each (V_{dd}, C_s) configuration in a GPU kernel has completed an equal number of high-level perturbations and so has a similar CDFG structure. Therefore the majority of GPU threads converge relatively close in time to one another, with a much smaller percentage being

latent outliers. This is evidenced by Figure 4, where a convergence distribution of the MAC benchmark can be seen. Figure 4 shows that the majority of MAC configurations meet the local convergence test within the same 150 iteration period (between 350 and 500 iterations). The vertical line dividing columns in Figure 4 indicates when 90 percent of MAC configurations have met our local convergence test. Other benchmarks showed similar distributions as MAC. Once a GPU thread has converged, it continues to optimize the floorplan until the GPU kernel is globally terminated. Doing so ensures that all GPU resources are in use for the entirety of the GPU kernel lifetime. In Figure 4 the MAC benchmark passes the global test in roughly 600 iterations. When the shared counter in global memory reaches 90 percent of $|P|$, the GPU kernel is globally terminated. Stopping here is advantageous because the remaining configurations likely would have yielded little improvement while adding a great expense. The lack of degradation in our results, shown by Table I, supports this assertion. Through testing of combined local and global influences, subtle design relationships affecting convergence can be successfully accounted for, as well as maximizing use of GPU compute power.

```
if threadIdx = 0
  count_SHARED := 0
fi
move_BEST, flag_CONV := ∞, false
do
  generate floorplan move and evaluate its cost move_COST
  if flag_CONV = false
    HISTORY ← move
    if move_COST < move_BEST
      move_BEST := move_COST
    fi
    ∀moves ∈ HISTORY ∧ move_COST < 0
      IMPRV ← move
    if |IMPRV| ≤ 5% × |HISTORY|
      ∀moves ∈ IMPRV ∧ move_COST ≤ 10% × move_BEST
        count_SHARED := count_SHARED + 1
        flag_CONV := true
    fi
  fi
  if count_SHARED ≥ 90% × |P|
    return
  fi
od
```

Fig. 5. CUDA GPU floorplan algorithm to test for local GPU thread and global GPU kernel convergence.

V. Experimental Results

In this section, we present the results of the parallel high-level and physical level synthesis, as described in the previous sections. The whole program framework follows that of [14]. We implemented both a sequential version and the proposed parallel version in the C++ programming language and CUDA SDK [3], and will compare their running time and performance. We further evaluate the proposed GPU floorplanner with local and global convergence test against a traditional SA GPU floorplanner. Both GPU floorplanners leverage our parallel cross-layer optimization technique. As this work is the first to perform parallel cross-layer optimization, we cannot compare with other prior GPU implementations in this area.

The experiments are conducted on a Linux workstation with an Intel quad-core Nehalem $2.13GHz$ processor and 4GB of memory. Our workstation also contains four Nvidia Tesla C1060 cards for the GPU experiments. Both the sequential and parallel programs are run with the same set of benchmarks introduced in [14]. The boundary constraint for data paths is set as 1.8 times the fastest clock period achievable. The voltage supply space is searched from $1V$ to $5V$ with the increment of $0.1V$, as is suggested in [1]. For both programs, the number of passes of incremental improvement for each configuration p is set as 50.

First, we examine the quality and efficiency of the proposed massively parallel GPU floorplanner. Table I shows the speed-up and quality comparisons for our parallel GPU floorplanner against a traditional parallel SA floorplanner. In Table I, we run a single iterative-improvement phase $\forall p \in P$ for each benchmark using both floorplanners and compare the results. The experiments for Table I were run with a single Nvidia Tesla C1060 GPU. Since power is a function of final circuit frequency, which is determined in the HLS, we report only the energy of the floorplanned configurations. In order to encompass the effect of the proposed GPU floorplanning with local and global convergence test over all the configuration pairs, we report the averaged energy across for all the configurations in each benchmark. With the local and global convergence test schemes, we see improved speed-up in all benchmarks, with comparably minimized energies in the resulting floorplans. Our results show an average speed-up of nearly 25 percent with increased energy of less than one tenth of 1 percent, indicating result quality is unaffected. These results show our new scheme is able to successfully detect the appropriate stopping condition unique to each benchmark, optimizing for maximized performance benefit with minimized increase in energy.

Next, we examine the performance of the whole parallel high- and physical-level cross-layer power optimization algorithm, compared with the traditional sequential version. Tables II and III give the comparisons of the quality of synthesized designs and run-times between the sequential program and our (multi-) GPU program. We first check the consistency of resulting quality of our parallel algorithm. From Table II, we can observe that the GPU-parallel programs achieve indistinguishable design quality, in terms of total power and area, to the sequential version's. This is due to the fact that all the programs explore the same design space, and the algorithm for high-level synthesis is identical. The tiny differences in Table II are induced by the non-determinacy of the simulated annealing algorithm, e.g. for the same design, it is possible that the floorplanner gives a slightly different result each time it is called.

After examining result consistency, we now analyze the performance of parallel cross-layer optimization, as is listed in Table III. Note that the run-time and speed-up listed in Table III are the total time of the program, including sequential switching-activity simulation. The column labeled with "#config" shows the number of valid (V_{dd}, C_s) configurations for the current benchmark. Since our workstation has four GPU cards, we tested the performance of our parallel program using one to four cards, the results of which are indicated in Table III as columns "GPU 1" to "GPU 4". The single GPU program achieves average speed-up of $3.53X$ over the CPU version. Although a GPU has a great number of cores (Tesla C1060 has 240 cores), the computational power of one GPU core is far less powerful than that of the modern CPU processor. Furthermore, the memory access speed of the CPU is much faster than that of the GPU. However, by leveraging the abundant parallelism in the cross-layer optimization algorithm, the collective power of the many-core GPU overwhelms that of the CPU, achieving better performance. This is indicated by the speed-up shown in our multi-GPU program. As we observe from Table III, near linear speed-up is achieved. For some small benchmarks such as MAC and PAULIN, the speed-up is worse than the others, caused by two major reasons. First, for these small cases, the sequential switching-activity simulation takes a larger part in the whole program. Secondly, the number of valid (V_{dd}, C_s) configurations in these cases is relatively smaller, which indicates limited parallelism and therefore limited potential for performance increase. On average, a speed-up of $11.13X$ is achieved with four GPU cards.

VI. Conclusions

In this work, we proposed and implemented the parallel cross-layer optimization technique across high-level and physical syntheses. We derived our optimization framework using the nondeterministic transactional model with UNITY. Leveraging the heterogeneous parallel-computational power in current multi-/many-core processors, increased exploration of design space was achieved with our experiments showing an 11X average speedup while delivering comparable results. We believe that the proposed parallel cross-layer optimization technique is a potential method for alleviating unacceptable compute complexities of the design closure problem.

TABLE I
GPU FLOORPLANNING: TRADITIONAL SA VS. PROPOSED GPU SA

Benchmark	Traditional SA		Proposed GPU SA		Improvement	
	Time (s)	Average Energy (pJ)	Time	Average Energy	Speedup	Energy (%)
MAC	17.84	2214.56	15.75	2215.20	1.13X	100.02
IIR77	67.10	3422.75	51.63	3433.46	1.29X	100.31
ELLIPTIC	32.52	2837.97	26.60	2847.84	1.22X	100.34
PAULIN	20.58	1242.33	19.61	1241.98	1.04X	99.97
PR1	51.93	2693.80	42.43	2708.64	1.22X	100.55
PR2	83.64	4029.71	63.11	4019.39	1.32X	99.74
DCT_IJPEG	53.64	2925.21	41.39	2916.09	1.29X	99.69
DCT_DIF	58.78	2222.45	47.54	2219.13	1.23X	99.85
CHEMICAL	35.21	2592.33	29.16	2593.79	1.20X	100.06
WDF	75.17	2301.84	60.91	2304.28	1.23X	100.11
DCT_WANG	83.45	1820.14	63.70	1837.53	1.31X	100.96
JACOBI_SM	107.15	3646.65	78.85	3661.61	1.35X	100.41
DCT_LEE	99.44	3061.91	78.84	3067.20	1.26X	100.17
RANDOM100	141.16	3715.22	107.78	3683.77	1.30X	99.15
Avg.					1.24X	100.09

TABLE II
DIFFERENCE OF RESULT ON BENCHMARKS

Benchmark	[14]		GPU 1 Diff (%)		GPU 2 Diff (%)		GPU 4 Diff (%)	
	Area (mm^2)	Power (W)	Area	Power	Area	Power	Area	Power
MAC	1.69	3.07	0.00	-1.14	0.59	2.00	0.00	0.65
IIR77	4.24	1.85	2.35	-0.51	-2.30	-0.46	-3.00	-0.56
ELLIPTIC	3.32	2.97	-0.30	0.16	-0.90	-0.16	-0.31	1.68
PAULIN	0.94	0.89	-1.21	-3.69	1.27	-1.80	-1.06	1.60
PR1	3.77	2.03	2.12	-0.62	-0.26	0.58	0.00	-1.90
PR2	5.58	2.05	0.00	1.46	-1.35	0.58	-0.35	0.07
DCT_IJPEG	3.64	3.37	-1.92	0.98	-0.82	0.54	2.74	-0.51
DCT_DIF	2.51	1.11	0.40	0.21	0.79	1.03	-0.55	-1.00
CHEMICAL	3.64	2.44	2.74	0.73	-0.82	0.17	0.53	-0.27
WDF	2.18	0.93	-0.27	0.07	-0.50	-0.31	-1.45	1.47
DCT_WANG	4.26	1.03	-1.42	-0.47	0.95	-0.34	-0.58	0.12
JACOBI_SM	5.03	1.73	-0.23	0.38	-0.20	1.21	0.13	0.35
DCT_LEE	3.90	1.04	-1.36	0.85	0.79	-0.61	1.84	-0.50
RANDOM100	8.98	2.60	-0.77	-0.74	0.25	-0.67	0.79	-0.14
Avg.	3.83	1.94	0.01	-0.17	-0.18	0.13	-0.09	0.03

TABLE III
RUN-TIME SPEED-UP ON DIFFERENT BENCHMARKS. (TIME UNIT: SECOND)

Benchmark	$\|P\|$	[14]	GPU 1		GPU 2		GPU 4	
		Time	Time	Speedup	Time	Speedup	Time	Speedup
MAC	1155	377.43	194.95	1.94X	114.22	3.30X	75.91	4.97X
IIR77	2578	4711.19	1259.27	3.74X	685.16	6.88X	392.32	12.01X
ELLIPTIC	1264	1873.43	617.81	3.03X	382.46	4.90X	251.79	7.44X
PAULIN	1827	263.70	158.82	1.66X	85.60	3.08X	50.31	5.24X
PR1	2033	3900.44	1021.56	3.82X	550.37	7.09X	328.89	11.86X
PR2	2718	7248.53	1864.50	3.89X	1022.15	7.09X	580.87	12.48X
DCT_IJPEG	1377	4131.89	1092.63	3.78X	617.93	6.69X	400.54	10.32X
DCT_DIF	2818	3391.65	922.10	3.68X	488.64	6.94X	279.67	12.13X
CHEMICAL	1377	2062.77	631.88	3.26X	382.71	5.39X	269.28	7.66X
WDF	3557	4840.82	1379.58	3.51X	726.89	6.66X	395.78	12.23X
DCT_WANG	2814	7400.68	1842.65	4.02X	988.22	7.49X	576.30	12.84X
JACOBI_SM	2840	11584.48	2652.55	4.37X	1422.61	8.14X	811.72	14.27X
DCT_LEE	4396	7405.26	1891.16	3.92X	992.69	7.46X	535.85	13.82X
RANDOM100	4803	14298.03	3360.62	4.25X	1171.42	8.07X	961.85	14.87X
Avg.	2540	11807.34	2951.36	3.53X	1152.32	6.47X	862.08	11.13X

REFERENCES

[1] A. Raghunathan and N. K. Jha, "Scalp: An iterative-improvement-based low-power data path synthesis system," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, pp. 1260–1277, Dec 1997.

[2] A. Ludwin, V. Betz, and K. Padalia, "High-quality, deterministic parallel placement for fpgas on commodity hardware," in *ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*, 2008.

[3] "Compute unified device architecture." [Online]. http://www.nvidia.com/cuda.

[4] Z. Feng and P. Li, "Multigrid on gpu: tackling power grid analysis on parallel simt platforms," in *Int. Conf. on Computer-Aided Design*, 2008.

[5] D. Chatterjee, A. DeOrio, and V. Bertacco, "Event-driven gate-level simulation with GP-GPUs," in *Design Automation Conf.*, 2009.

[6] D. Chatterjee, A. DeOrio, and V. Bertacco, "GCS: High performance gate-level simulation with GP-GPUs," in *Design Automation and Test in Europe*, 2009.

[7] U. D. Bordoloi and S. Chakraborty, "Accelerating system-level design tasks using commodity graphics hardware: A case study," in *International Conf. on VLSI Design*, 2009.

[8] Y. S. Deng, B. D. Wang, and S. Mu, "Taming irregular EDA applications on GPUs," in *International Conf. on Computer-Aided Design*, 2009.

[9] M. Nanjundappa, H. D. Patel, B. A. Jose, and S. K. Shukla, "SCGPSim: A fast systemC simulator on GPUs," in *Asia South-Pacific Design Automation Conf.*, 2010.

[10] G. Flach, M. Johann, R. Hentschke, and R. Reis, "Cell placement on GPUs," in *SBCCI*, 2007.

[11] N. Viswanathan and C. C.-N. Chu, "Fastplace: Efficient analytical placement using cell-shifting, iterative local refinement and a hybrid net model," in *International Symposium on Physical Design*, 2005.

[12] J. Cong and Y. Zou, "Parallel multi-level analytical global placement on graphics processing units," in *Int. Conf. on Computer-Aided Design*, 2009.

[13] T. Chan, J. Cong, and K. Sze, "Multilevel generalized force directed method for circuit placement," in *International Symposium on Physical Design*, 2005.

[14] Z. Gu, J. Wang, R. P. Dick, and H. Zhou, "Incremental exploration of the combined physical and behavioral design space," in *Proc. of the Design Automation Conf.*, (Anaheim, CA), pp. 208–213, June 2005.

[15] Y. Lu, H. Zhou, L. Shang, and X. Zeng, "Multicore parallel min-cost flow algorithm for cad applications," in *Design Automation Conf*, 2009.

[16] D. F. Wong and C. L. Liu, "A new algorithm for floorplan design," in *Proc. of the Design Automation Conf.*, pp. 101–107, 1986.

[17] H. Zhou and J. Wang, "ACG–adjacent constraint graph for general floorplans," *Int. Conf. on Computer-Aided Design*, 2004.

[18] X. Tang and D. F. Wong, "FAST-SP: A fast algorithm for block placement based on sequence pair," in *ASP-DAC*, pp. 521–526, 2001.

[19] J.-M. Lin and Y.-W. Chang, "TCG-S: orthogonal coupling of $p*$-admissible representations for general floorplans," in *Design Automation Conf*, 2002.