This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Exploring High-Throughput Computing Paradigm for Global Routing

Yiding Han, Dean Michael Ancajas, Koushik Chakraborty, and Sanghamitra Roy

***Abstract*— With aggressive technology scaling, the complexity of the global routing problem is poised to grow rapidly. Solving such a large computational problem demands a high-throughput hardware platform such as modern graphics processing units (GPUs). In this paper, we explore a hybrid GPU–CPU high-throughput computing environment as a scalable alternative to the traditional CPU-based router. We introduce net-level concurrency (NLC), which is a novel parallel model for router algorithms and aims to exploit concurrency at the level of individual nets. To efficiently uncover NLC, we design a scheduler to create groups of nets that can be routed in parallel. At its core, our scheduler employs a novel algorithm to dynamically analyze data dependencies between multiple nets. We believe such an algorithm can lay the foundation for uncovering data-level parallelism in routing, which is a necessary requirement for employing high-throughput hardware. Detailed simulation results show an average of 4× speedup over NTHU-Route 2.0 with negligible loss in solution quality. To the best of our knowledge, this is the first work on utilizing GPUs for global routing.**

***Index Terms*— Global routing, graphics processing unit, high throughput computing.**

I. Introduction

GLOBAL routing problem (GRP) is one of the most computationally intensive processes in VLSI design. Since the solution of the GRP is used to guide further optimizations before tape-out, it also becomes a critical step in the design cycle. Consequently, both the execution time and the solution quality of the GRP substantially affect the chip timing, power, manufacturability, as well as the time-to-market.

Aggressive technology scaling introduces several additional constraints in the GRP, significantly increasing the complexity of this important VLSI design problem [1], [2]. Alpert *et al.* [3] point out that at 32 nm there will be 4–6 metal widths and 20 thicknesses across 12 metal layers. Furthermore, IBM envisions an explosion in design rules beyond 22 nm that will make GRP a multiobjective problem [4]. Unfortunately, current CPU-based routers will prove to be inefficient for the increasingly complex GRPs, as these routers only solve simple optimization problems [5], [6].

Tackling this huge computationally complex problem would require a platform that offers high-throughput computing such as a graphics processor unit (GPU). Traditionally, a GPUs computing bandwidth is used to solve massively parallel problems. GPUs excel in applications that repeatedly apply a set of operations on a big dataset, involving single-instruction multiple-data (SIMD) style parallel codes. Several existing VLSI computer-aided design problems have seen successful incarnation in GPUs, delivering more than 100× speedup [7]–[9]. However, the canonical GRP does not fit well into such an execution paradigm because routing algorithms repeatedly manipulate shared data structures such as routing resources. This sharing of resources disrupts the data-independence requirement of traditional GPU applications. Hence, existing task-based parallel routing algorithms must be completely revamped to make use of the GPU bandwidth.

In the light of these technology trends, we propose a hybrid GPU–CPU routing platform that enables a collaborative algorithmic framework to combine data-level parallelism from GPUs with thread-level parallelism from multicores. This paper specifically addresses the scalability challenges posed to current global routers. To date, there have been very few works that parallelize the GRP by using multicore processors [10], [11]. However, none of these is designed to exploit high-throughput computing platforms such as the GPU.

Exploiting the computation bandwidth of GPUs for the GRP is a nontrivial problem, as the overhead of sharing resources hurts the overall performance. In this paper, we use a fundamentally new mode of parallelism to uncover the performance potential of the GPU. We propose a novel net-level concurrency (NLC) model to efficiently consider the data dependencies among all simultaneously routed nets. This model enables parallelism to scale well with technology and computing complexity.

Following are the major contributions of this paper to global routing research.

1) GPU-CPU hybrid routing: We propose an execution model that allows cooperation of the GPU and the CPU to route multiple nets simultaneously through NLC. To the best of our knowledge, this is the first work on utilizing GPUs for global routing. The GPU global router uses a breadth first search (BFS) heuristic while the CPU router uses A* maze routing. Together, they provide two distinct classes in the routing spectrum. The high-latency low-bandwidth problems are tackled by the CPU, whereas the low-latency high-bandwidth problems are solved by the GPU. We believe this classification is the key to efficiently tackle the complexity increase of the GRP on massively parallel hardware.

Manuscript received May 15, 2012; revised September 26, 2012; accepted November 4, 2012. This work was supported in part by the National Science Foundation under Grant CNS-1117425.

The authors are with the Electrical and Computer Engineering Department, Utah State University, Logan, UT 84322 USA (e-mail: bitcars@gmail.com; dbancajas@gmail.com; koushik.chakraborty@usu.edu; sanghamitra.roy@usu.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2012.2234489

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

2) Scheduler: We develop a scheduler algorithm to explore NLC in the GRP. The scheduler produces concurrent routing tasks for the parallel global routers based on net dependencies. The produced concurrent tasks are distributed to the parallel environments provided by the GPU and multicore CPU platforms. The scheduler is designed to dynamically and iteratively analyze the net dependencies, hence limiting its computational overhead.
3) GPU Lee algorithm: We propose a Lee algorithm based on BFS path finding on a GPU. This algorithm utilizes the massively parallel architecture for routing and backtracing. Our approach is able to find the shortest weighted path and achieves high computational throughput by simultaneously routing multiple nets.

The remainder of this paper is organized as follows. In Section II, we briefly discuss related literature. Section III introduces our design spectrum for a GPU–CPU hybrid global router. Section IV describes the overview of our GPU–CPU router. We discuss the challenges of a scalable parallel routing algorithm in Section V. Section VI presents our detailed scheduler design, while our GPU–CPU router implementations are described in Section VII. We show our results in Section VIII, and conclude in Section IX.

II. Related Work

In 2007 and 2008, the International Symposium on Physical Design (ISPD) held two global router competitions. These contests promoted the development of many recent global routers. These routers typically employ a collection of well-studied routing techniques. Roughly, we can categorize them into two types, sequential and concurrent. The sequential techniques apply maze routing followed by a negotiation-based rip-up and reroute (RRR) scheme. RRR was originally introduced in Pathfinder for field-programmable gate arrays to route macro cells [12]. Consequently, it has been used to recursively reduce overflows in the following routers: 1) FGR [13]; 2) NTHU-Route 2.0 [14]; 3) NCTUgr [15]; 4) NTUgr [16]; and 5) Archer [17]. The concurrent algorithms apply integer linear programming (ILP) to achieve an overflow-free solution [18]–[20].

GRIP [20] currently holds the best solution quality in open literature of all ISPD 2007 and 2008 benchmarks. However, its pure ILP-based approach requires a significantly longer runtime compared to the negotiation-based RRR scheme. Even the runtime reported in their parallel PGRIP [11] router was still relatively high.

When problems approach a larger scale, sequential global routers are more popular because the negotiation-based RRR scheme offers a much better tradeoff between solution quality and runtime. This scheme can even be applied in addition to an ILP-based approach to reduce runtime [18]. However, the downside of the sequential approach is the heavy dependency of solution quality on the routing order of nets.

For this reason, parallelization of negotiation-based RRR scheme is difficult. Routing multiple nets simultaneously could jeopardize the routing order and create race conditions on shared routing resources among threads. Recently, Liu *et al.* [10] adopted a collision-aware global routing algorithm with bounded-length maze routing on a quad-core system. They implemented a rudimentary workload-based parallelization technique with a simple collision-awareness algorithm. Their router has achieved good speedups on four threads.

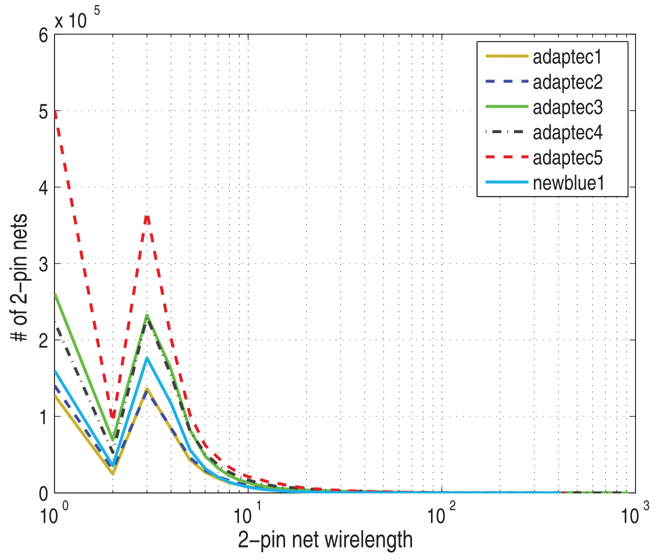

Fig. 1. Wire-length distribution indicating the coexistence of a large number of long and short wires. Wire length is measured in Manhattan distance between two pins.

This paper is the first to use a high-throughput hybrid environment (GPU–CPU) to tackle the GRP. As technology develops, the boundary between SIMD and single-instruction single-data will shrink [21]. This paper lays the foundation for using a hybrid high-throughput environment for global routing.

III. Tackling GRP With GPU–CPU Hybrid System

In this section, we introduce the design motivation and the spectrum of our GPU–CPU hybrid system for global routing.

A. Wire-Length Distribution of GRP

Technology scaling continues to pack more transistors in a chip, and circuits become increasingly complex. The routing benchmarks provided in ISPD 2007 and ISPD 2008 show that modern GRPs typically come in considerably large scales. Each problem generally packs a number of subproblems in the magnitude between 10^6 and 10^7, while the difficulty of each subproblem, defined as the length of the two-pin net, varies in a wide range.

We illustrate the distribution of subproblem difficulties in Fig. 1. The diagram represents the histogram of the difficulties of all subproblems from six benchmarks in the ISPD 2007 suite. The subproblems are defined as the two-pin nets acquired by decomposing the multipin nets. The difficulty of each subproblem is characterized with the Manhattan distance between the two pins. The peaks on the left side of the diagram show that significant amounts of the subproblems are easy to solve. Considerable numbers of difficult subproblems

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

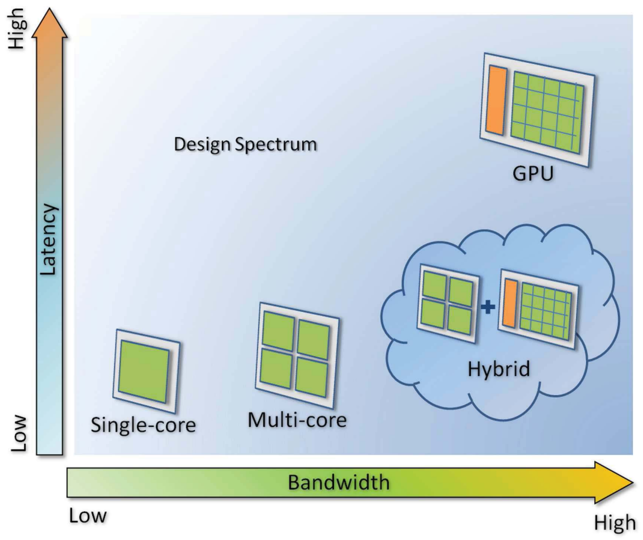

Fig. 2. Conceptual picture of computational bandwidth and latency of existing computing platforms.

still exist, forming a long-tail distribution on the right side of the diagram. Given such properties of the GRP, we aim to maximize the computational throughput with the available hardware.

B. GPU–CPU Hybrid

A conceptual design spectrum is demonstrated in Fig. 2, where we compare the latency and bandwidth of the existing computing platforms. The single-core system can solve each routing problem with low latency, but falls short when the problems come in extremely large numbers. In comparison, the multicore system provides a larger computational bandwidth with similar latency, making it the most common choice in parallel global routing [10], [11]. However, the GPU platform can easily prevail in a bandwidth contest by routing a large number of nets simultaneously. The latency for a GPU solution is likely to be longer due to the additional traffic between the GPU and the CPU.

Given the long-tail distribution of the GRP, a GPU–CPU hybrid solution appears to be attractive. The short nets, which are also the majority of the entire workload, are a good fit for the GPU platform to utilize the broad computational bandwidth. The long nets are simultaneously assigned to the multicore platform to exploit parallelism with lower latency. We target to design such a heterogeneous parallel model to approach the GRP with a wide-bandwidth low-latency computing platform.

In this paper, we implement the proposed GPU–CPU hybrid model for parallel global routing. Our experimental results show the heterogeneous parallel model can yield significant speedup across different benchmarks compared to the single-core implementation while delivering similar routing quality.

IV. Overview of GPU–CPU Global Routing

In this section, we give an overview of our GPU-CPU global router.

A. Problem Definition

The GRP is defined as follows. There is a grid graph G that is composed of a set of vertices V and edges E. Each vertex v_i in V corresponds to a rectangular cell, while each edge e_{ij} in E represents a connection between two adjacent vertices i and j (or a boundary between two cells). There is also a set of nets N, for which every n_i in N is made up of a set of pins P. Each pin in a net coincides with a vertex in V. The capacity c_{ij} of an edge between vertices i and j represents the number of nets that can pass through that edge. The demand d_{ij} represents the current number of nets passing through the edge. Overflow of an edge is then defined as the difference $d_{ij} - c_{ij}$. A net n_i is routed when we find a path connecting all the pins of the net utilizing edges of graph G. The wire length of a net is determined by the number of edges it crossed to route all its pins. A solution to the GRP is achieved when all the nets n_i in N are routed.

B. Objective

Like other global routers [10], [11], [13]–[18], [22], [23], the GPU–CPU global router has three major objectives. First is the minimization of the sum of overflows among all edges. Second is the minimization of the total wire length of routing all the nets, and third is the minimization of the total runtime needed to obtain a solution.

Typically in recent works, the three objectives are used to evaluate the effectiveness of a global router. The primary goal for global routing is to resolve congestion and achieve an overflow-free solution while producing timing-friendly routes. However, the timing enclosures on critical nets are often addressed as a soft constraint. The reason is twofold. First, introducing additional timing constraints to an already complex routing problem can overstress the global router, which may choke the routing process [3]. Second, other timing optimization processes can address the timing closure issues from routed solutions. For example, buffer insertion, placement, and via incremental synthesis can be used to avoid the timing closure problems identified by the router [24].

C. Design Flow

The flow of our global router is shown in Fig. 3. The initial global routing solution is generated as the following: we first project a multilayer design on a 2-D plane and use FLUTE 3.0 [25] to decompose all multipin nets into sets of two-pin subnets. Consequently, we apply edge shifting [26] on all the nets to modify their initial topologies. We then perform initial routing and create the congestion map.

The initial routing is a quick phase that initializes the connectivity of nets. We use two main strategies to minimize the overhead of this stage. First, the routing instances are spread out on all CPU cores for parallel routing. In this concurrency model, we ignore the resource sharing among nets and try to minimize the synchronization overhead. Second, we use a uniform cost function to compute each edge cost. Specifically, the edge cost falls under two broad categories, i.e., nonoverflow and overflow, each represented with a constant

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

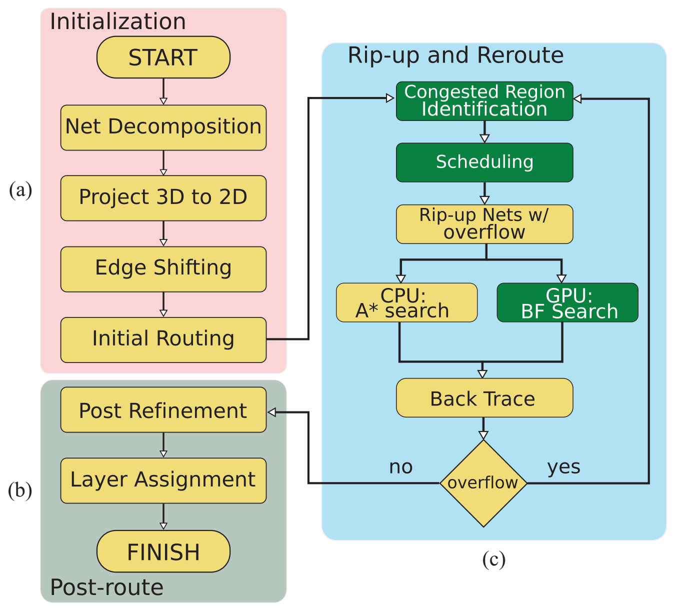

Fig. 3. Global router design flow. (a) Initialization, while (b) post-routing phase. Steps in (a) and (b) are also present in other CPU-based routers. (c) RRR, and we enhanced this section by using a scheduler. Our contributions are highlighted in dark background shading.

cost. As a result, with an A* search algorithm and bounding box constraints, the initial solutions are found very quickly. Typically, the initial routing phase takes less than 10 s to finish on a quad-core processor, while the main RRR phase takes several minutes.

Subsequently, the router enters the RRR phase. Now we apply the negotiation-based scheme to iteratively improve the routing quality by RRR the subnets that pass through the overflowing edges. We route each subnet within a bounding box, whose size is relaxed if the subnet is unable to find a feasible path (Section VII-D). The order of the subnets to be RRR is determined through congested region identification (CRI), which is an algorithm that collects subnets that are bounded within the congestion region (Section VII-E).

The main phase completes when no overflow is detected. Then we apply layer assignment to project the 2-D routing plane back onto the original multilayer design. The layer assignment technique is similar to that described in [13].

D. Global Routing Parallelization

The parallel global router strives for high throughput by maximizing the number of simultaneously routing nets. However, the negotiation-based RRR scheme is strongly dependent on the routing order. Routing nets simultaneously regardless of their order might cause degradation in solution quality and performance.

We tackle this problem by examining the dependencies among nets, and extracting concurrent nets from the ordered task queue. These nets can then be routed simultaneously without jeopardizing the solution quality. We now discuss challenges of extracting concurrent nets (Section V) and tackling them through a novel scheduler (Section VI).

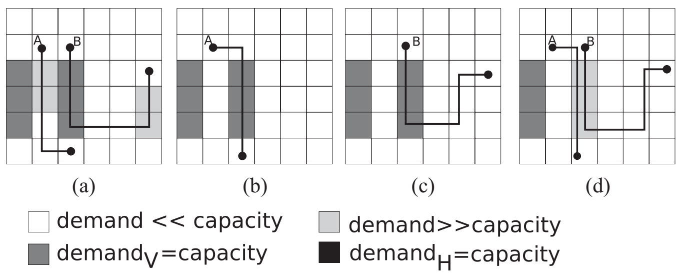

Fig. 4. Parallel router must have consistent view of resources. (a) Before RRR. (b) and (c) Viewpoint of each thread. They unknowingly allocate conflicted resources. (d) Overflow is realized at the end of R&R when both threads back track.

V. Enabling Net-Level Parallelism in Global Routing

In this section, we will explain the requirements of enabling data-level parallelism in the GRP, which is a necessary step for exploiting high-throughput platforms such as GPUs.

A. Challenge in Parallelization of Global Routing

There are two main approaches in parallelizing the GRP. First, the routing map can be partitioned into smaller regions and solved in a bottom-up approach using parallel threads [11], [27], [28]. Second, individual nets can be divided across many threads and routed simultaneously. We call this the NLC. NLC can substantially achieve better load-balancing and uncover more parallelism. However, it also comes at the cost of additional complexity.

Unlike partitioning, NLC allows sharing of routing resources between multiple threads. Consequently, to effectively exploit NLC, one must ensure that threads have current usage information of the shared resources. Without such information, concurrent threads may not be aware of impending resource collisions, leading to unintentional overflow and degradation in both performance and solution quality [10]. This phenomenon is demonstrated in Fig. 4, where both the threads consume the lone routing resource resulting in an overflow.

B. Achieving NLC

Unfortunately, collision awareness alone cannot guarantee reaping performance benefits by employing NLC on high-throughput platforms. This is best explained through Fig. 5. In this example, we assume that there is a task queue and that threads continually get a net to route from this queue. We also assume that nets in the queue are ordered on the basis of the congestion of their path. Finally, there are two layers (horizontal and vertical) available for routing with demands indicated as demand$_H$ and demand$_V$. When multiple threads are processing a congested region, as in Fig. 5(b), concurrent routing will cramp limited resources to specific threads, sacrificing performance and solution quality. This problem is exacerbated with increasing number of threads and circuit complexity. However, we observe that we can recover substantial performance by allowing threads to concurrently

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

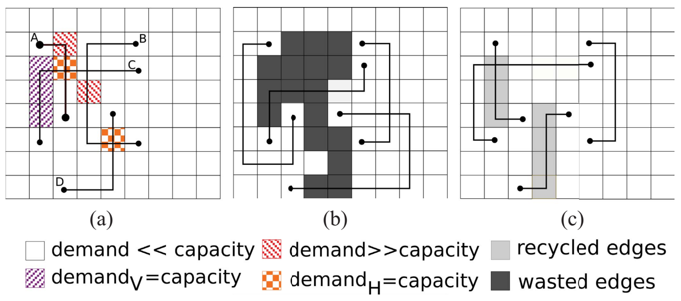

Fig. 5. Collision awareness alone can hurt routing solution. (a) Four-thread router processing a particular congested region, one net per thread. (b) Routing solution generated via collision-aware algorithm. Some resources are wasted as a result of overhead of collision awareness because threads are discouraged to route on cells (black, green, yellow, and blue cells) that were previously used by another thread. (c) With proper scheduling, only one thread is processing this particular region and some of the resources are recycled. The remaining threads are routing other congested areas in the chip (not shown).

route in different congested regions. In other words, a single thread can process the region shown in Fig. 5(c).

Therefore, the key to achieving high throughput in NLC is to identify congested regions and co-schedule nets that have minimal resource collisions. In the next section, we will discuss our proposed scheduler design.

VI. Scheduler

In this section, we describe our scheduler that dynamically examines the dependencies among nets in an ordered task queue, and extracts concurrent nets to be dispatched to the GPU and CPU routers for parallel routing.

A. Scheduler Overview

Fig. 6 shows our preliminary design overview of a global router on a hybrid platform [29]. This design aims to exploit NLC within the traditional RRR global routing algorithms.

The heart of this approach lies in the scheduler design, which identifies the concurrent nets to be routed in parallel. Since nets can share routing resources (e.g., tiles on the routing grid), concurrent nets are chosen in a manner that reduces resource conflicts where multiple routing threads attempt to use a given routing resource. Consequently, this approach restricts the level of parallelism that can be exploited during the global routing.

The scheduler dynamically populates two task queues with sets of nets that can be routed simultaneously. These task queues separately serve CPU and GPU threads. The task queues are decoupled from each other, to ease the load balancing and synchronization between the CPU and the GPU. Nets for GPU threads are chosen in a manner such that the entire routing for those nets can be efficiently done in the available shared memory in the GPU architecture. On an nVIDIA Fermi architecture, which limits the shared memory to 48 kB, nearly 99% of the nets from the ISPD benchmark suites can be routed on the GPU. As explained in Section VII-B, longer nets are assigned to the CPU, as routing these nets can uncover several other candidate nets for parallel routing.

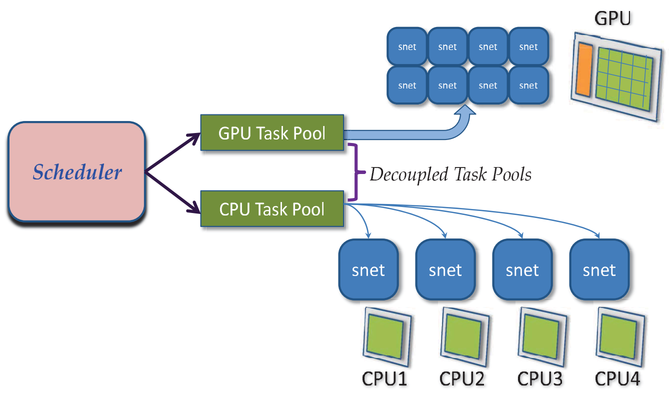

Fig. 6. Overview of GPU–CPU router. Concurrent subnets (snet) are distributed to GPU and CPU task pools.

B. Nets Data Dependency

In this section, we explain the concept of nets data dependencies, and discuss our parallel model that examines this dependency to exploit parallelism.

In each RRR iteration, we first create an explicit routing order for all the nets located within the congested regions. This order is derived by comparing the areas and aspect ratios of the nets. We assign higher priorities to nets that cover larger areas with smaller aspect ratios, and route them first. Typically, it is easier to find overflow free routes for these nets. Therefore, assigning routing resources to these nets before smaller nets can minimize the overall congestion and the wire length [13], [14].

By enforcing this explicit routing order, the routing process essentially creates data dependencies among the nets. Specifically, the routing order dictates different priorities for all nets when reassigning their routing resources. For example, when two nets both need the same resources to find paths, the one routed first has a higher priority to obtain those resources. If the order is changed, a degraded solution is likely to occur. For this reason, a conventional RRR process is typically implemented sequentially to ensure the net data dependencies.

In our parallel model, we try to exploit parallelism by routing nets that do not have a dependency violation among each other. Since net data dependencies only confine nets that have shared routing resources, the key that allows us to parallelize the routing process while maintaining the data dependencies is to examine the shared routing resources among the nets. If no shared resource exists, then we can safely exploit parallelism by routing these independent nets simultaneously.

Honoring the net data dependencies is crucial for our parallel model. The existing task-based parallel global router does not examine the net dependency when exploiting concurrency [10]. As a result, more than 41% of the subnets are affected by collisions in shared routing resources. This model does not suit well in a GPU-based concurrency framework. Because of the lack of synchronization mechanism for thread blocks in the GPU hardware, we need to avoid resource collisions on the GPUs device memory.

In this paper, we propose a scheduler to generate independent routing tasks for the parallel global routers. The data

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

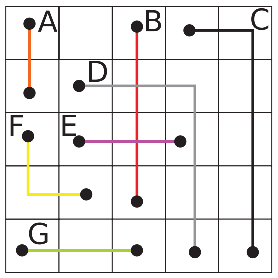

Fig. 7. Routing problem with nets overlapping each other.

dependency is iteratively analyzed, thereby limiting its analysis overhead while providing precise dependency information. First, the data dependency among nets is constructed in a dependency graph. Then we exploit parallelism by routing the independent nets. The parallelism created by our model can exploit massively parallel hardware without causing resource collisions or jeopardizing the routing order. As a result, our GPU–CPU parallel global router achieves deterministic routing solutions.

C. Net Dependency Construction

In this subsection, we present the scheduler algorithm to construct the net dependencies. As an example, a selection of two-pin nets (subnets) is illustrated in Fig. 7. In this example, we assume that each net's bounding region is of the same size as their covering area, and that the routing order derived is

$$D > C > F > B > E > G > A.$$

We now explain how to construct the dependency graph, and exploit the available concurrency without causing conflict in routing resource or order. This approach is mainly divided into the following three steps.

1) Tile Coloring: In this step, each tile identifies its occupancy by iterating through the ordered subnets. The first subnet region that covers the tile is considered as its occupant. Using the example from Fig. 7, the results of the colored tiles are shown in Fig. 8(a). We can observe that most of the map is colored by subnet D because it has the highest priority in routing order. Given this color map, each subnet can visualize the other subnets that it is overlapping with, hence determining its dominant subnets.

2) Finding Available Concurrency: With the help of the color map, we can easily find subnets that can be routed by checking if it occupies all of its routing regions. If not, then there is a dependency on other subnets and it must wait until the dependency is resolved. In Fig. 8(a), subnets A and D occupy all of their routing regions. Hence, they can be scheduled together.

Now we introduce the concept of dependency level. This metric is used to determine the urgency of routing certain subnets. We score the dependency level as the number of routing regions that one subnet invades. For instance, in Fig. 8(a) subnet D scores 5 because it invades the area of five subnets: i.e., B, C, E, F, and G.

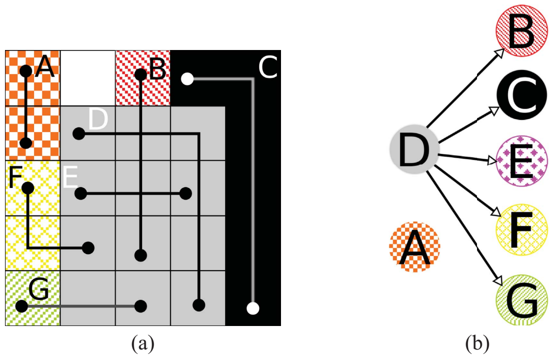

Fig. 8. Results after the first iteration. (a) Coloring of tiles: bigger nets dominate ownership over smaller ones. Only A and D can be routed together because other nets are dependent on D. (b) Net dependencies are derived from the color map.

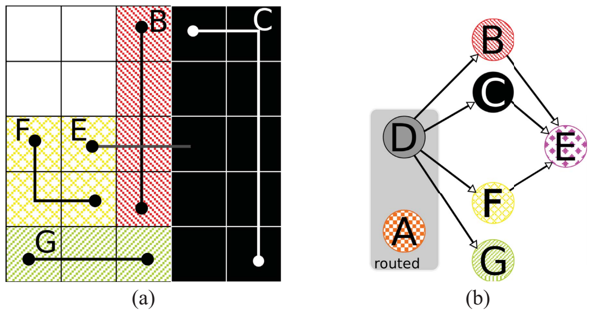

Fig. 9. Results after the second iteration. (a) After D and A are routed, nets C, B, F, and G can be routed together because they have no dependencies. (b) More detailed dependencies are revealed in the graph.

From Fig. 8(b) we can make an interesting observation on the dependencies among all subnets. Subnets B, C, E, F, and G are dependent on subnet D but we cannot identify the dependencies between them. The algorithm intentionally leaves these detailed dependencies for future computation, so as to reduce complexity while extracting the available concurrency in a timely manner.

3) Tile Recoloring: In this step, the algorithm uncovers detailed dependencies by reconstructing the color map. After the scheduled subnets are routed, the color map must be reconstructed to resolve previous dependencies. Fig. 9(a) shows the new color map when subnets A and D are already routed. Recoloring only needs to consider subnets that were dominated by A and D. In this case, they are $\{C, F, B, E, G\}$ for D's region, and $\emptyset$ for A.

The dependency graph is updated using the new map. Fig. 9(b) shows the new graph that reveals a new dependency between subnets B, E, C, and F. Similar to the previous iteration, the dependency graph indicates subnets B, C, F, and G can be scheduled once subnet D has finished, while E can only be scheduled after B, C, and F are completed.

D. Implementation and Optimization

The above three steps are recursively applied until all dependencies are resolved. The scheduler thread and the global router threads execute in parallel with a producer–consumer relationship. The scheduler keeps constructing the

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

dependency graph with the given task queue and producing concurrent nets to the router threads. The routers consume these nets simultaneously with different priorities indicated by the dependency levels, and return the completed nets to release more concurrent nets.

The efficiency of the scheduler algorithm affects the available computational throughput for routing. The complexity of this algorithm increases as the region size and net count increases. In practice, we reduce the problem size using the identified congestion region (Section VII-D). Instead of exploring concurrency on the entire routing map, the search area is restricted to only the congestion regions, hence easing the computation load of tile coloring.

In addition, a task window is used to restrict the number of nets being examined for concurrency in each iteration. In some cases, congestion regions contain a large number of nets. The task window can effectively limit the search space and speed up the dependency tree construction (Section VII-C).

VII. Implementation

In this section, we will discuss details of our GPU–CPU router implementation. We focus on several key issues such as the maze routing implementation on the GPU (Section VII-A), efficient GPU implementation of the scheduler (Section VII-C), directional CRI algorithm (Section VII-D), our bounding box expansion method (Section VII-E), and distribution of nets among CPU and GPU threads (Section VII-F).

A. Maze Routing Implementation on GPU

1) Parallel Lee Algorithm on GPU: Our GPU router uses the parallel Lee algorithm [30] to find the weighted shortest paths. This widely applied approach, although known for its high memory usage and slow search speed, is an attractive candidate for implementation on parallel systems.

Typically, the front wave expansion scheme of the Lee algorithm can be parallelized, allowing us to simultaneously explore the vertices at the same depth, which are defined as frontiers. This concurrency model enables us to utilize the GPUs large number of threads in a single block to exploit fine-grain parallelism during the concurrent frontiers expansion. This concurrency model is shown in Fig. 10.

In addition, we can safely perform multiple wavefront expansions and backtraces in parallel without causing collisions. This level of parallelism is coarse grain, and hence is exploited by the GPUs grid blocks. We should note that no global synchronization mechanism is required among the GPU thread blocks when routing multiple nets in parallel, since the nets are mutually independent. We illustrate this concurrency model with Fig. 11.

Another factor that makes the Lee algorithm appealing to GPU architecture is the use of simple grid arrays to represent the routing map. No complex data structure is required to store the pending route candidates. In our implementation, we use a fixed-sized flat memory to store the costs of expanding paths.

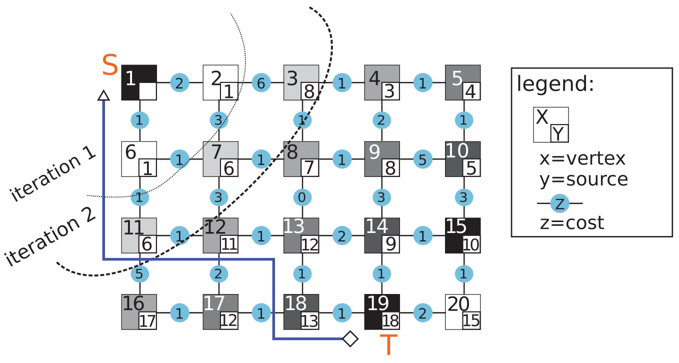

Fig. 10. Pathfinding in a GPU. We propagate from the source node. The BFS fills up the entire search region, and continues until all frontiers are exhausted. Then we backtrace from the target node to find the shortest weighted path.

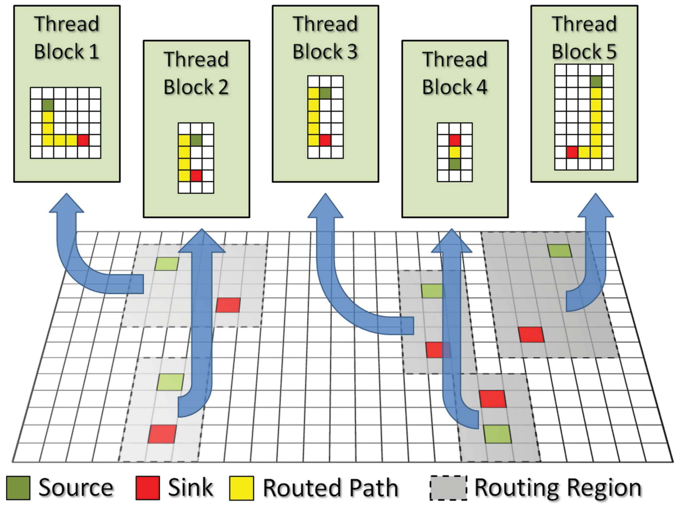

Fig. 11. GPU routing overview. Each thread block finds route for a single set of source and sink. The routing is done locally on the shared memory of each thread block.

2) Algorithm Description: We now explain our GPU-based parallel Lee algorithm in detail. Like the sequential version, our GPU-based Lee algorithm is divided into two phases.

1) Wave propagation: In this phase, propagation starts from the source tile. The search stops when the frontiers are all exhausted, thereby guaranteeing that all possible paths within the search region are traversed. Details of our wave propagation approach is described in Algorithm 1, with the explanation of terms in Table I. The algorithm describes the routing kernel executed on each GPU thread. We map each GPU thread to a tile on the routing grid. Each tile is indexed using the corresponding GPU thread ID (tid). If the tile is identified as a frontier tile, then the GPU thread will try to propagate to the neighboring tiles, which are indexed as nid in the algorithm.
2) Backtracing: In this phase, the GPU kernel essentially reverses the propagation direction. Starting from the Sink tile, we trace the route with the least cost and mark the traversed edges as the resulting path until the Source tile is reached. In the end, only one successful routing path is returned by the kernel.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Algorithm 1 GPU Lee Algorithm Kernel

```
 1: tid ← getThreadID
 2: if Frontier[tid] then
 3:   Frontier[tid] ← false
 4:   for all neighbors nid of tid do
 5:     Edge ← Vertex(tid, nid)
 6:     if Edge exists then
 7:       AddedCost ← TileCost[tid] + EdgeCost[Edge]
 8:       if AddedCost < TempTileCost[nid] then
 9:         TempTileCost[nid] ← AddedCost
10:       end if
11:     end if
12:   end for
13: end if
14: SYNCHRONIZE_THREADS()
15: if TileCost[tid] > tempTileCost[tid] then
16:   TileCost[tid] ← tempTileCost[tid]
17:   Frontier[tid] ← true
18:   DONE ← false
19: end if
20: tempTileCost[tid] ← TileCost[tid]
```

TABLE I
GPU LEE ALGORITHM NOTATIONS

Term	Description
Frontier	Boolean list that marks the frontier tiles in the current iteration. Contains source vertex initially.
EdgeCost	Array that stores cost of all edges.
Vertex	Function that returns the Edge between two Vertices.
AddedCost	Intermediate variable to store the new tile cost.
TempTileCost	Array that stores cost of traversing tiles. Initialized as infinite (Inf.).
TileCost	Array that stores minimum cost of traversed tiles. Initialized as Inf.
DONE	Boolean indicating all frontiers are explored.

We integrate the propagation and backtracing phases into one CUDA kernel function instead of two separate ones. Doing so reduces the overhead of loading intermediate data between the shared memory and device memory, and the overhead of additional kernel launch.

Our algorithm is fundamentally different from the previously proposed BFS algorithms for GPUs [31], [32] for the following reasons: 1) our algorithm tackles the weighted shortest path problem, and 2) we route an individual two-pin net within each thread block. Therefore, we attain performance boost by routing a large amount of nets concurrently.

B. GPU Memory Arrangement

The performance of a GPU application is largely dependent on its memory arrangement. In this section we explain our GPU memory arrangement to enable high-throughput maze routing on the GPU.

In our GPU Lee algorithm, the costs of traversing tiles are arranged in grid arrays, which are stored in the shared memory. This arrangement is demonstrated in Fig. 11. The size of the grid is determined by the bounding box of the routing net, hence it is partial to the complete routing grid. We let individual blocks update the local grid on the shared memory, without synchronization to the global device memory. This arrangement has a much higher efficiency, but comes at the cost of generality. Because of the shared memory size limitation, the number of tiles that can be traversed by each thread block is constrained to about 2500. Fortunately, this size is reasonable for the GRP in most cases. According to our observation, more than 99% of all two-pin nets can be fitted within this area.

We store the vertices topology and edge cost data in the GPU texture memory. The texture memory is allocated on the device memory. But with texture binding, this memory is cached and optimized for read-only data. Fetching from texture memory provides high bandwidth on memory space with good spatial locality (i.e., if a cell is visited, then its neighbors are also traversed). Hence, we bind the vertices and edge costs array with 3-D-texture and 1-D-texture memories, respectively. Each cell in the vertices in 3-D-texture points to six different adjacent cells: –X, –Y, –Z, +X, +Y, +Z. We use these directions to identify the edge index, which locates the cost of the edge from the edge cost texture.

C. Scheduler

The efficiency of the scheduler is critical to the overall throughput of our GPU–CPU hybrid global router. On the one hand, the scheduler needs to be able to produce enough concurrent nets for all parallel threads to consume. On the other hand, the scheduler needs to be light enough (weight) to deliver the workload in a timely manner. In this section, we introduce an efficient scheduler design.

1) Scheduler on GPU: We implement the scheduler algorithm on a GPU. The algorithm is parallelizable on a fine-grain level, with each GPU thread dedicated to color a single tile. This implementation also has a low latency due to the small amount of data packages that are copied between the CPU and the GPU. In addition, transferring the computation to the GPU allows us to free significant CPU resources, which can be dedicated to maze routing. We show these results in Section VIII-C.

2) Algorithm Optimization: The complexity of the scheduler algorithm increases as the region size and net count increases. In practice, we reduce the problem size using the identified congestion region (Section VII-D). Instead of exploring concurrency on the entire routing map, the search area is restricted to only the congestion regions, hence easing the computation load of tile coloring.

In addition, a task window is used to restrict the number of nets being examined for concurrency in each iteration. The task window has a significant impact on the overall performance of our parallel router. A short window usually leads to less trade-off in performance, but might yield insufficient workloads, whereas a long window might over examine the available concurrency, and introduce degradation in performance.

Fig. 12 shows the distribution of workload versus the parallel window size on the ISPD 2007 newblue2 benchmark.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

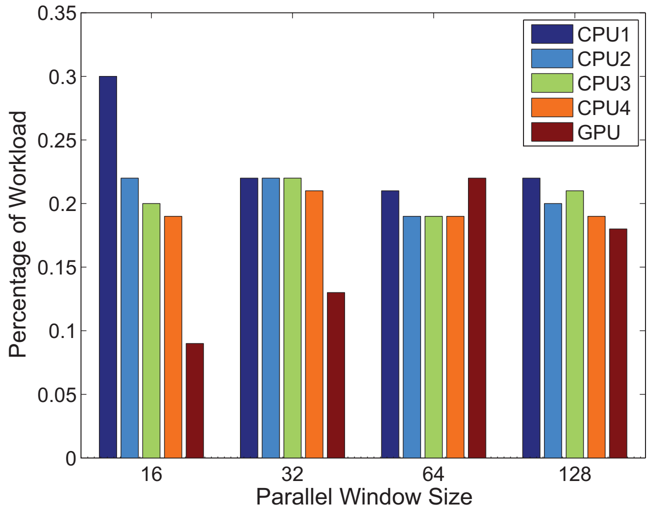

Fig. 12. Workload distribution with different task window. With the increasing size of parallel window, workload is easier to be balanced amongst CPUs and GPU, but it also comes at higher overhead.

On the left side of the figure, insufficient workloads due to a small window size causes most nets being distributed onto a single thread. For this reason, the GPU thread has a very low utilization rate. This problem is alleviated by increasing the size of the task window. However, we should note that, when the window size is too wide, the overhead of the scheduler itself begins to dominate, hence degrading the overall router performance.

D. Congested Region Identification

To discuss our CRI algorithm, we first formalize the problem of finding congested regions by explaining several terminologies. The first step in the identification of congested regions is the generation of a congestion map from the initial routing result. A congestion map is an $X \times Y$ matrix of congestion values. Each element in the map b_{ij} is the average congestion of the top and right edges of cell (i, j). Congestion $m_{i,j;k,l}$ is the congestion of edge between cell (i, j) and (k, l). If we let $d_{i,j;k,l}$ and $c_{i,j;k,l}$ be the corresponding demands and capacities of that edge then we can specify $m_{i,j;k,l}$ as

$$m_{i,j;k,l} = \frac{d_{i,j;k,l}}{c_{i,j;k,l}}.$$

Consequently

$$b_{ij} = \frac{m_{i,j;i+1,j} + m_{i,j;i,j+1}}{2}.$$

All b_{ij}s are scaled with respect to the maximum such that $0 \le b'_{ij} \le 1$, where b'_{ij} is the scaled value. Table II lists the remaining notations and their descriptions.

In order to accurately identify the congested region, we use a directional expansion algorithm to adaptively expand to the region in the directions that result in the highest congestion. Fig. 13 best explains this situation. We begin by taking the most congested tiles (red cells in the figure) and adaptively expand until the average congestion for the region is below a certain threshold. We divide the congestion value into several congestion levels, much like NTHU-Route [14]. The number of congestion levels we seek to model will dictate the size of each region. In our example, we have four congestion levels. After these regions are found, we route the nets inside them in parallel based on Section V. Pertinent details are presented in Algorithm 2, and Table II lists the notations and their descriptions.

TABLE II
ALGORITHM NOTATIONS

Term	Description
r	Rectangle with bottom left coordinates (i, j) and top right (k, l)
$\text{Ave}_i(r)$	Average congestion value inside the expanded rectangle r in the direction of i side(s)
$\mathcal{L}_{\mathcal{B}}(\text{level})$	Returns the lower bound value for a particular congestion level
expand4sides(r)	Expands region r in all directions
expand3sides(r)	Expands three sides of region r toward the maximum congestion
expand2sides(r)	Expands two sides of region r toward the maximum congestion
expand1sides(r)	Expands one side of region r toward the maximum congestion

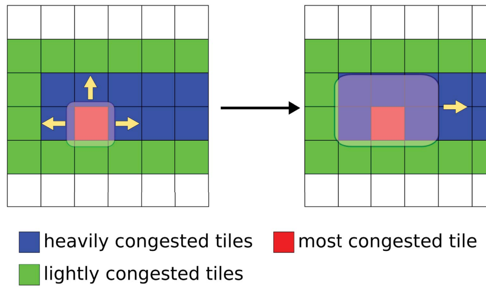

Fig. 13. Directional expansion algorithm adaptively expanding in the directions with the highest congestion.

E. Bounding Box Expansion

Bounding box is widely applied in global routers. This technique constrains the searching region of two-pin nets within a rectangle, reducing the space complexity of maze routing and decreasing the solution wire length.

In our GPU–CPU router, bounding box is a crucial component for the scheduler to determine nets dependencies. However, in order to allow nets to explore a larger solution space, the size of the bounding box is expanded as the RRR iteration proceeds. As the remaining overflow decreases, we can relax the constraint of the bounding box and allow the maze router to obtain overflow free route at the cost of longer wire length. But as the constraint of bounding box relaxes to more than 10 times as large as the original, we stop the expansion again to avoid excessively long routes.

We choose an adaptive method, rather than a fixed parameter function, to expand the bounding box. The search area constraint is relaxed according to the percentage of remaining overflow. We keep the size of the bounding box unchanged until 99% of all overflow is resolved. This phase passes very

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Algorithm 2 Directional Expansion Algorithm

```
$r = \max(b_{ij})$
$\forall\ 0 \le i \le X;\ 0 \le j \le Y$
1: for level= 1 to 4 do
2:   while $\text{Ave}_4(r) > \mathcal{L}_{\mathcal{B}}(\text{level})$ do
3:     expand4sides($r$)
4:   end while
5:   while $\text{Ave}_3(r) > \mathcal{L}_{\mathcal{B}}(\text{level})$ do
6:     expand3sides($r$)
7:   end while
8:   while $\text{Ave}_2(r) > \mathcal{L}_{\mathcal{B}}(\text{level})$ do
9:     expand2sides($r$)
10:  end while
11:  while $\text{Ave}_1(r) > \mathcal{L}_{\mathcal{B}}(\text{level})$ do
12:    expand1sides($r$)
13:  end while
14: end for
```

fast because the searching areas are small. Then we linearly increase the size of the bounding box as the RRR iteration proceeds, until its size reaches the upper limit.

F. Workload Distribution Between the GPU and CPU

In this section, we introduce the heuristic used for workload balancing. The scheduler dispatches workloads among the CPUs and the GPU for optimum computational throughput. Typically, the CPU routers achieve a single solution with lower latency than the GPU router, but the latter can achieve much higher throughput by routing multiple nets in a single kernel call. Urgent nets, which release several subsequent nets to be routed concurrently, are more likely to be scheduled on the CPUs. In addition, nets with large bounding boxes are also routed in the CPU due to the shared memory limits on the GPU. The detailed scheduling heuristic is based on the following criteria.

1) Routing region size: The GPU router has constraints on the size of the routing region for each two-pin net. If a workload exceeds the area limitation, it will be scheduled on a CPU.
2) Net size preference: We sort the concurrent nets with respect to their problem size. The CPU maze router consumes workload from the larger end of the queue, while the GPU consumes from the smaller end.
3) Lower-bounded scheduling on the GPU: Since the GPU router strives for a high bandwidth, it only schedules nets to route if the number of available workloads meet a certain lower boundary. Typically, we set this number to be one-fourth of the current parallel window size.

These criteria can dynamically consume the available workloads as quickly as possible. The heterogeneous structure of a GPU–CPU hybrid system makes it difficult to predict a perfectly balanced schedule. Hence we allow certain router threads to wait in idle. This is especially the case when the number of concurrent nets is small.

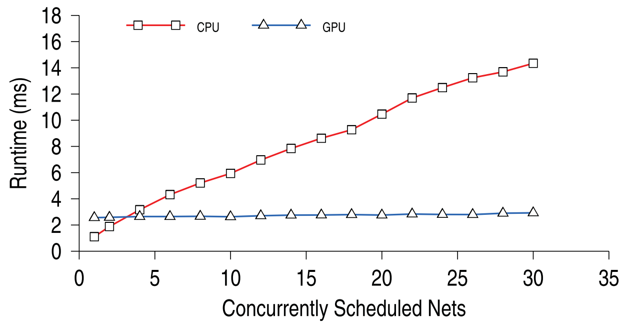

Fig. 14. Runtime comparison between CPU A*Search and GPU BFS.

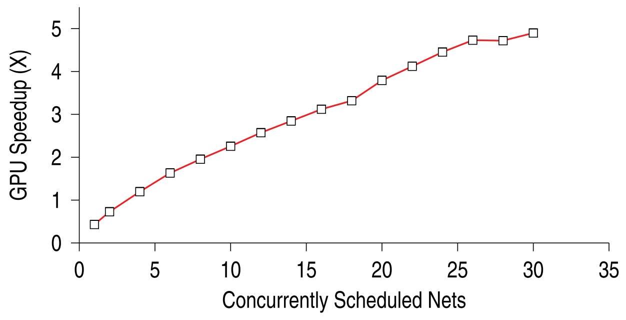

Fig. 15. Speedup of GPU BFS over CPU A*Search.

VIII. Results

We implement our GPU–CPU hybrid router in C++ on an Intel Quad-core 2.4-GHz machine with 8 GB of RAM. The GPU we use is an Nvidia Geforce GTX 470 with the Fermi architecture. The C++ code is compiled with the Intel C++ Compiler 11.1; CUDA code is compiled with the CUDA Toolkit 4.0. The multithreading framework is designed using pthread. We use ISPD 2007 2-D benchmarks and ISPD 2008 3-D benchmarks for evaluation in the experiments shown below. In these benchmarks, each edge or via usage translates to a unit wire length, except for ISPD 2007 benchmarks, in which a via counts as three-unit wire length.

A. GPU and CPU Router

We first compare the routing throughput between our GPU and CPU routers. In this experiment, we schedule the same nets to the GPU router and a single CPU router, and record the average wall clock time for both routers. To make it a fair comparison, the process includes routing, backtracing, and data transfers between the GPU and the system memory.

The results are shown in Figs. 14 and 15. The x-axis in both figures represents the number of concurrent nets being scheduled on either CPU or GPU platform. The runtime comparison in Fig. 14 shows a linear increase of CPU router runtime with the growing number of routing nets. Interestingly, the GPU runtime slope is much flatter than the CPU. Consequently, the CPU router has a much shorter latency when routing individual nets, while the GPU can deliver a much higher bandwidth when scheduled with multiple nets.

The relative speedup of the GPU router over the CPU router is illustrated in Fig. 15. We can observe about 5× speedup when both the routers are scheduled with 30 nets,

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

TABLE III
WIRELENGTH AND RUNTIME COMPARISON WITH NTHU-ROUTE 2.0

	Parallel Router (Four-Core)		Parallel Router (Four-Core + GPU)		NTHU 2.0		Speedup	
	WL[a]	Runtime[b]	WL[a]	Runtime[b]	WL[a]	Runtime[b]	Four-core	Four-core + GPU
adaptec1	5.43E6	2.90	5.44E6	2.07	5.34E6	9.95	3.43	4.81
adaptec2	5.29E6	0.70	5.30E6	0.63	5.23E6	2.1	3.00	3.33
adaptec3	1.31E7	3.53	1.31E7	3.05	1.31E7	10.86	3.08	3.56
adaptec4	1.24E7	0.95	1.24E7	0.65	1.22E7	2.5	2.63	3.85
adaptec5	1.55E7	9.98	1.55E7	8.12	1.55E7	21.9	2.19	2.70
newblue1	4.70E6	2.87	4.70E6	1.88	4.65E6	6.2	2.16	3.30
newblue2	7.79E6	0.66	7.81E6	0.65	7.57E6	1.1	1.67	1.69
newblue5	2.38E7	5.38	2.38E7	4.42	2.32E7	19.1	3.55	4.32
newblue6	1.80E7	4.12	1.80E7	3.78	1.77E7	17.5	4.25	4.63
bigblue1	5.63E6	3.10	5.63E6	2.86	5.59E6	13.1	4.22	4.58
bigblue2	9.10E6	2.31	9.05E6	2.09	9.06E6	8.4	3.64	4.02
bigblue3	1.30E7	1.09	1.30E7	1.01	1.31E7	4.4	4.04	4.37
Average	1.01[c]	–	1.01[c]	–	1	–	3.15	4.01

[a]Wirelength in terms of edges consumed.
[b]Expressed in minutes.
[c]Normalized to the NTHU-Rs wirelength.

which is a typical number of available concurrent nets that we can extract using the proposed scheduling approach. However, the theoretical speedup of the GPU router grows with larger number of concurrent nets. We have observed a speedup of 73× if both routers are scheduled with 1000 concurrent nets (not shown).

B. Comparison With NTHU-Route 2.0

We compare the performance our global router under different configurations, and use the solutions of ISPD 2008 routing contest winner NTHU-Route 2.0 as a reference. NTHU-Route 2.0 is a state-of-the-art global router based on a single-threaded RRR scheme. The router uses CRI to create net ordering for the RRR process, during which a multisource and multisink maze routing technique is use to create a competitive solution quality. We gather the NTHU-Route 2.0 solutions with the same hardware platform for the comparison.

In Table III, we show the comparison of our global router with NTHU-Route 2.0. The runtime is measured in minutes. Our parallel router generates high-quality routing solutions with wire length within an average of 1.1% increase to that reported by NTHU-Route 2.0. Noticeably, the additional GPU router introduces negligible overhead in the resultant wire length. In the runtime comparison, the parallel router utilizing four CPU threads achieves an average speedup of 3.15× compared to NTHU-Route 2.0, while 4× average speedup is achieved with the additional GPU router. These results prove the effectiveness of our concurrency model in solving the GRP with NLP on a high-throughput hardware while the concurrency has little affect in the solution quality.

We show the results of the hard-to-route problems in Table IV. These problems are unable to find overflow-free solutions. We use a fixed time budget, typically 1 h, for these benchmarks to find a solution. In addition, if the route finds out that the previous 10 iterations lead to no solution improvement, it terminates the RRR process. These results

TABLE IV
WIRE LENGTH AND OVERFLOW COMPARISON WITH NTHU-ROUTE 2.0 ON HARD-TO-ROUTE PROBLEMS

	Our GR			NTHU-R2		
	MO[a]	TO[b]	WL[c]	MO	TO	WL
newblue3	183	31484	108.5	204	31454	106.49
newblue4	4	168	138.9	4	138	130.46
newblue7	4	78	352.3	2	62	353.35
bigblue4	2	178	232.9	2	162	231.04

[a]Most overflow.
[b]Total overflow.
[c]Wirelength in terms of edges consumed.

show that for the unroutable benchmarks, our global router generates competitive solutions in terms of overflow reduction.

C. GPU-Based Scheduler

In this section, we show the impact of our scheduling algorithm on the overall performance and study the primary performance bottleneck of the parallel router.

Comparing the results from Section VIII-B against Section VIII-A, we notice that the speedup achieved in the actual routing instances are less than the theoretical value. To understand the cause of this performance gap, we use a much more efficient GPU-based scheduler. The GPU-based approach significantly reduces the runtime overhead of the scheduling process, and potentially directs more computational efforts to the routing threads. We use this experiment to study whether a more efficient scheduling process can benefit the overall performance.

The results are shown in Table V. We organize the results in two columns for both the four-core and four-core with GPU routing cases. The first column reports the routing runtime with the GPU-based scheduling algorithm; the second one reports its speedup comparing to the CPU-based scheduling algorithm (results in Table III).

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

TABLE V
PERFORMANCE SENSITIVITY TO THE FASTER GPU-BASED SCHEDULING ALGORITHM

	4-core		4-core+GPU	
	Runtime^a	Speedup^b	Runtime^a	Speedup^b
adaptec1	2.51	1.16	2.09	0.99
adaptec2	0.63	1.11	0.62	1.02
adaptec3	3.19	1.11	2.98	1.02
adaptec4	0.94	1.01	0.93	1.02
adaptec5	8.33	1.20	7.84	1.04
newblue1	2.57	1.12	2.20	1.17
newblue2	0.64	1.03	0.64	1.02
newblue5	7.01	1.14	6.53	1.08
newblue6	4.81	1.19	4.73	1.14
bigblue1	3.38	1.16	3.28	1.12
bigblue2	2.78	1.10	2.73	1.04
bigblue3	2.33	1.02	2.32	0.99

^{a}Expressed in minutes.
^{b}Speedup compared to runtime listed in Table III.

TABLE VI
RRR STATE SPEEDUP COMPARISON TO OUR GLOBAL ROUTER IN SEQUENTIAL CONFIGURATION

	1-core	4-core		4-core + GPU	
	t_{RRR}	t_{RRR}	Spdup	t_{RRR}	Spdup
adaptec1	7.30	1.77	4.12X	1.35	**5.41X**
adaptec2	0.71	0.25	2.84X	0.24	2.96X
adaptec3	6.48	1.63	3.99X	1.42	4.56X
adaptec4	0.26	0.11	2.36X	0.10	2.60X
adaptec5	28.79	6.46	4.46X	5.97	4.82X
newblue1	7.23	2.22	3.26X	1.85	3.91X
newblue2	0.44	0.26	1.69X	0.26	1.69X
newblue5	14.78	4.41	3.36X	3.92	3.77X
newblue6	11.88	3.19	3.71X	3.13	3.80X
bigblue1	9.83	2.57	3.82X	2.47	3.98X
bigblue2	6.04	2.03	2.96X	1.98	3.05X
bigblue3	2.72	1.03	2.64X	1.03	2.64X
Average	–	–	3.27X	–	3.60X

Unfortunately, the more efficient GPU-based scheduler brings only marginal gain to the overall speedup. No concrete performance boost is observed in either four-core or four-core with GPU case. The speedup of adaptec1 and bigblue3 are slightly decreased because of allocating additional GPU resources to the scheduling process. These results show that the lack of concurrent nets is the primary bottleneck that limits the speedup we can achieve with the proposed framework.

In conclusion, our router relies heavily on the amount of independent nets to exploit parallelism. Regardless of the scheduling approach, the available resources are often limited in the actual routing instances. This limitation leads to an insufficiency of nets to be scheduled in the routing thread pool. The lack of nets forces the inactive threads to wait for the active ones to finish routing due to the nets' ordering and resource dependencies.

D. Achieved Speedup

Finally, we examine the speedups achieved by our parallel model in Table VI, where the runtime of the RRR stage is compared under two configurations of our parallel router: four-core and four-core with GPU. We use the GPU-based scheduler for this set of comparisons. As a result, the parallel router with four CPU threads achieves an average speedup of 3.27×, while an average speedup of 3.60× is gained with the additional GPU.

From these results, we can see that the additional GPU thread brings a noticeable speedup over the four-core case. For example, an additional 129% performance improvement is observed by using the GPU router in the adapetc1 benchmark. These results indicate a strong potential for high computational throughput with our proposed approach. Unfortunately, on average, the GPU–CPU router only brings 10.1% additional performance improvement, indicating that the speedup that we achieve is also dependent on the properties of the benchmarks. Specifically, as mentioned in Section VIII-C, since the lack of independent nets is the primary bottleneck of our performance gain, not all benchmarks have the same level of freedom to schedule independent nets for parallel routing. As a result, the speedups achieved by our CPU–GPU router on the 12 benchmarks fall within a wide range, from 1.69× to 5.41×.

IX. CONCLUSION

As technology continues to scale, computational complexity of many EDA algorithms is growing rapidly. Exploiting the computational bandwidth of high-throughput platforms such as the GPU is a prominent direction for future EDA. In this paper, we presented a hybrid GPU–CPU high-throughput computing environment as a scalable alternative to the traditional CPU-based router. We showed that the traditional GRP needs to be revamped for exploiting the new computing environment. The key to our method is using the NLC guided by a scheduler. The scheduler analyzes data dependencies between nets and dynamically generates concurrent routing tasks for the computing environment. Detailed simulation results showed an average of 4× speedup over NTHU-Route 2.0 with negligible loss in solution quality. Our framework is a concrete step toward developing next-generation global routers geared for high throughput compute architectures.

REFERENCES

[1] D. Cross, E. Nequist, and L. Scheffer, "A DFM aware, space based router," in *Proc. Int. Symp. Phys. Design*, 2007, pp. 171–172.

[2] N. Kaul, "Design planning trends and challenges," in *Proc. Int. Symp. Phys. Design*, 2010, p. 5.

[3] C. J. Alpert, Z. Li, M. D. Moffitt, G.-J. Nam, J. A. Roy, and G. Tellez, "What makes a design difficult to route," in *Proc. Int. Symp. Phys. Design*, 2010, pp. 7–12.

[4] R. C. Johnson, "IBM warns of 'design rule explosion' beyond 22-nm," *EE Times*, Mar. 2010.

[5] P. Groeneveld, "Going with the flow: Bridging the gap between theory and practice in physical design," in *Proc. Int. Symp. Phys. Design*, 2010, p. 3.

[6] M. D. Moffitt, J. A. Roy, and I. L. Markov, "The coming of age of (academic) global routing," in *Proc. Int. Symp. Phys. Design*, 2008, pp. 148–155.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

[7] J. Croix and S. Khatri, "Introduction to GPU programming for EDA," in *Proc. IEEE Int. Conf. Comput.-Aided Design*, Nov. 2009, pp. 276–280.

[8] K. Gulati and S. Khatri, "Toward acceleration of fault simulation using graphics processing units," in *Proc. Design Autom. Conf.*, Jun. 2008, pp. 822–827.

[9] K. Gulati and S. P. Khatri, "Accelerating statistical static timing analysis using graphics processing units," in *Proc. Asia-Pacific Design Autom. Conf.*, 2009, pp. 260–265.

[10] W.-H. Liu, W.-C. Kao, Y.-L. Li, and K.-Y. Chao, "Multi-threaded collision-aware global routing with bounded-length maze routing," in *Proc. Design Autom. Conf.*, 2010, pp. 200–205.

[11] T.-H. Wu, A. Davoodi, and J. T. Linderoth, "A parallel integer programming approach to global routing," in *Proc. Design Autom. Conf.*, 2010, pp. 194–199.

[12] L. McMurchie and C. Ebeling, "PathFinder: A negotiation-based performance-driven router for FPGAs," in *Proc. ACM 3rd Int. Symp. Field-Program. Gate Arrays*, Feb. 1995, pp. 111–117.

[13] J. A. Roy and I. L. Markov, "High-performance routing at the nanometer scale," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 6, pp. 1066–1077, Jun. 2008.

[14] Y.-J. Chang, Y.-T. Lee, J.-R. Gao, P.-C. Wu, and T.-C. Wang, "NTHU-route 2.0: A robust global router for modern designs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 29, no. 12, pp. 1931–1944, Dec. 2010.

[15] K.-R. Dai, W.-H. Liu, and Y.-L. Li, "Efficient simulated evolution based rerouting and congestion-relaxed layer assignment on 3-D global routing," in *Proc. Asia-Pacific Design Autom. Conf.*, 2009, pp. 582–587.

[16] H.-Y. Chen, C.-H. Hsu, and Y.-W. Chang, "High-performance global routing with fast overflow reduction," in *Proc. Asia-Pacific Design Autom. Conf.*, 2009, pp. 570–575.

[17] M. M. Ozdal and M. D. F. Wong, "Archer: A history-driven global routing algorithm," in *Proc. IEEE Int. Conf. Comput.-Aided Design*, San Jose, CA, Nov. 2007, pp. 488–495.

[18] M. Cho, K. Lu, K. Yuan, and D. Z. Pan, "BoxRouter 2.0: Architecture and implementation of a hybrid and robust global router," in *Proc. IEEE Int. Conf. Comput.-Aided Design*, Aug. 2007, pp. 503–508.

[19] B. Korte, D. Rautenbach, and J. Vygen, "BonnTools: Mathematical innovation for layout and timing closure of systems on a chip," *Proc. IEEE*, vol. 95, no. 3, pp. 555–572, Mar. 2007.

[20] T.-H. Wu, A. Davoodi, and J. T. Linderoth, "GRIP: Scalable 3D global routing using integer programming," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 1, pp. 72–84, Jan. 2011.

[21] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: A many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 27, pp. 18:1–18:15, Aug. 2008.

[22] M. D. Moffitt, "MaizeRouter: Engineering an effective global router," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 11, pp. 2017–2026, Nov. 2008.

[23] Y. Xu, Y. Zhang, and C. Chu, "FastRoute 4.0: Global router with efficient via minimization," in *Proc. Asia South Pacific Design Autom. Conf.*, 2009, pp. 576–581.

[24] C. J. Alpert and G. E. Tellez, "The importance of routing congestion analysis," in *Proc. Design Autom. Conf.*, 2010, pp. 1–14.

[25] C. Chu and Y.-C. Wong, "FLUTE: Fast lookup table based rectilinear Steiner minimal tree algorithm for VLSI design," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 1, pp. 70–83, Jan. 2008.

[26] M. Pan and C. Chu, "FastRoute: A step to integrate global routing into placement," in *Proc. IEEE Int. Conf. Comput.-Aided Design*, Aug. 2006, pp. 464–471.

[27] J. Hu and S. S. Sapatnekar, "A survey on multi-net global routing for integrated circuits," *Integr., VLSI J.*, vol. 31, no. 1, pp. 1–49, 2001.

[28] S. Khanna, S. Gao, and K. Thulasiraman, "Parallel hierarchical global routing for general cell layout," in *Proc. 5th Great Lakes Symp. VLSI*, Washington, DC, 1995, pp. 212–215.

[29] Y. Han, D. M. Ancajas, K. Chakraborty, and S. Roy, "Exploring high throughput computing paradigm for global routing," in *Proc. IEEE Int. Conf. Comput.-Aided Design*, Mar. 2011, pp. 298–305.

[30] I.-L. Yen, R. Dubash, and F. Bastani, "Strategies for mapping Lee's maze routing algorithm onto parallel architectures," in *Proc. 7th Int. Parallel Process. Symp.*, Apr. 1993, pp. 672–679.

[31] P. Harish and P. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *Proc. High Perform. Comput.*, 2007, pp. 197–208.

[32] L. Luo, M. Wong, and W. M. Hwu, "An effective GPU implementation of breadth-first search," in *Proc. Design Autom. Conf.*, Anaheim, CA, Jun. 2010, pp. 52–55.

Yiding Han received the B.A.Sc. degree in control science and engineering from the Huazhong University of Science and Technology, Wuhan, China, in 2006, and the Masters degree in electrical engineering from Utah State University, Logan, in 2009. He is currently pursuing the Ph.D. degree with the Electrical and Computer Engineering Department, Utah State University.

His current research interests include general purpose GPU computing, EDA algorithms, VLSI routing, and computer architecture.

Dean Michael Ancajas received the M.S. degree in electrical engineering from the University of the Philippines, in 2010, focusing on computer architecture. He is currently pursuing the Ph.D. degree with the Electrical and Computer Engineering Department, Utah State University, Logan.

His current research interests include computer architecture and reliability of network on chips. His paper published at the 30th IEEE International Conference on Computer Design (ICCD 2012).

Mr. Ancajas was a recipient of a Best Paper Award.

Koushik Chakraborty received the B.Tech. degree from the Indian Institute of Technology Kanpur, Kanpur, India, and the Ph.D. degree from the University of Wisconsin-Madison, Madison, in 2000 and 2008, respectively, all in computer sciences.

He is currently an Assistant Professor with the Electrical and Computer Engineering Department, Utah State University, Logan, UT. His current research interest include interdisciplinary computer architectures and VLSI CAD. He has authored or co-authored over 30 peer-reviewed papers in journals and conferences.

Dr. Chakraborty was a recipient the Best Paper Award at the 2012 IEEE International Conference on Computer Design, and was nominated for the Best Paper Award at 2011 IEEE/ACM Design Automation and Test in Europe and the 2010 IEEE International Conference on VLSI Design. He is the Reviewer of the IEEE TVLSI, IEEE TPDS, ACM TECS, and IEEE Design and Test. His research is funded by the National Science Foundation and Micron Foundation.

Sanghamitra Roy received the M.S. degree in computer engineering from Northwestern University, Evanston, IL, in 2003, and the Ph.D. degree in electrical and computer engineering from the University of Wisconsin-Madison, Madison.

She is currently an Assistant Professor with the Department of Electrical and Computer Engineering, Utah State University. Her doctoral research was sponsored by Intel Strategic CAD labs and National Science Foundation. She has authored or co-authored over 30 peer-reviewed papers in top-tier journals and conferences, and a book chapter in *VLSI Design Automation*. Her current research interests include VLSI circuit design and optimization, and exploring reliability aware novel circuit styles, and architectures.

Dr. Roy was a recipient of the Best Paper Award nominations at the IEEE Design Automation & Test in Europe 2011, the IEEE/ACM International Conference on Computer Aided Design 2005, and the IEEE 23rd International Conference on VLSI Design 2010. She is a Reviewer of the IEEE TVLSI, the IEEE TCAD, Integration - the *VLSI Journal*, IET Computers and Digital Techniques, the IEEE International Conference on VLSI Design, the IEEE/ACM Design Automation Conference, and the IEEE/ACM International Symposium on Quality Electronic Design.