FastGR : Global Routing on CPU-GPU with Heterogeneous Task Graph Scheduler (Extended Abstract)∗

Siting Liu†, Yuan Pu1, Peiyu Liao1,2, Hongzhong Wu3, Rui Zhang4, Zhitang Chen1, Wenlong Lv5, Yibo Lin2 and Bei Yu1
1 The Chinese University of Hong Kong
2 Peking University
3HiSilicon Technologies Co.
4Huawei Noah’s Ark Lab

Abstract

Running time is a key metric across the standard physical design flow stages. However, with the rapid growth in design sizes, routing runtime has become the runtime bottleneck in the physical design flow. To improve the effectiveness of the modern global router, we propose a global routing framework with GPU-accelerated routing algorithms and a heterogeneous task graph scheduler, called FastGR. Its runtime-oriented version FastGRL achieves 2.489× speedup compared with the state-of-the-art global router. Furthermore, the GPU-accelerated L-shape pattern routing used in FastGRL can contribute to 9.324× speedup over the sequential algorithm on CPU. Its quality-oriented version FastGRLH offers further quality improvement over FastGRL with similar acceleration.

1 Introduction

Routing is essential in the design flow of the modern very-large-scale integration. Modern routing flow is divided into global routing and detailed routing. Global routing produces routing guidance for detailed routing by performing rough routing on a coarse grid graph [Liu et al., 2020]. The efficiency and efficacy of global routing are crucial to the design closure due to its recurrent invocation and guiding role.

The literature has extensively explored shortest path searching with GPU [Djidjev et al., 2015]. However, most work only explores the basic single-source shortest path algorithm. These algorithms are unsuitable for routing since we must route millions of nets while considering numerous objectives and limitations such as wirelength, number of vias, and design rules. Regarding those modern routing challenges, more appropriate GPU kernel algorithms should be designed.

In this work, we propose FastGR, a global routing framework accelerated for CPU-GPU platforms. The framework leverages a GPU-friendly pattern routing algorithm and a task graph scheduler for heterogeneous CPU-GPU systems. By utilizing the processing resources of GPUs, we can further increase the solution quality performance of our global routing framework while incurring a little runtime overhead. We develop two variants of our global routing framework: the runtime-oriented version FastGRL and the quality-oriented version FastGRLH. Experiments show that when compared to the state-of-the-art global router [Liu et al., 2020], our runtime-oriented version FastGRL can achieve 2.489× overall speedup without any quality degradation. The quality-oriented version FastGRLH [Liu et al., 2022] reduces the number of shorts by 27.855% over the runtime-oriented version FastGRL [Liu et al., 2022a] while remaining 1.970× faster than the most advanced global router [Liu et al., 2020].

2 Preliminaries

2.1 Problem Formulation

A grid graph G(V, E) is defined to formulate global routing problem by considering each G-cell as a vertex (v ∈ V) and drawing an edge (e ∈ E) between all the pairs of adjacent G-cells. Figure 1 illustrates the procedure of grid graph construction. We map all the pins into G-cells according to the pin position. In this sample, different colors represent different metal layers. There is a preferred routing direction (horizontal or vertical) for wire edges in each metal layer, represented as the colored solid lines. The black dotted lines mean the via edges in our grid graph. With the grid graph G construction, the global routing problem can be formulated as the minimum accumulated cost path searching problem on G for all the nets defined in VLSI designs.

2.2 Modern Global Router

Pattern routing [Kastner et al., 2002] plays an important role in the modern global routing framework due to its efficiency. Two popular patterns are shown in Figure 2. We illustrate the L-shape and Z-shape pattern routing paths on 2D and 3D routing spaces. As shown in Figure 2, the L-shape pattern includes one single bend point to change the routing direction, while the Z-shape pattern routing path contains two bends.

To determine the execution order of each conflicting pair of tasks, we develop a two-stage task graph scheduler. The first stage is to determine the execution order of the global routing tasks, and the second stage is to manage the execution order of multiple routing tasks in both space and time. Our task graph scheduler is utilized to determine the execution order of each conflicting pair of tasks.

3.2 Task Graph Scheduler

To determine the execution order of the global routing tasks, we develop a two-stage task graph scheduler. The first stage is to create a task conflict graph. The task graph scheduler is then used to determine the order of execution for each conflict edge. Following the task conflict graph generation, we extract one maximum non-conflicting set as the root task batch. All of these tasks can be divided into two groups: the root task batch and the non-root task batch. We assign the execution order to each pair of conflicting tasks as follows.

![Z-shape and L-shape pattern routing](image)

Figure 2: 2D/3D pattern routing; The red path represents one L-shape pattern routing solution, and the blue path is one of the candidate Z-shape pattern routing paths.

3 Algorithms

3.1 Overview

Figure 3 depicts the overall flow of FastGR. To begin, we present a heterogeneous task graph scheduler and use it to manage the execution order of multiple routing tasks in both portions of our global routing framework, the pattern routing stage, and the rip-up and reroute iterations. The conflicting relationship among these tasks is used to form the task graph. It is important to note that a conflict between two routing tasks indicates that they cannot be processed at the same time. Our task graph scheduler is utilized to determine the execution order of each conflicting pair of tasks.

![Pattern Routing Stage and Rip-up and Reroute Iterations](image)

Figure 3: Overall flow of FastGR.

3.2 Task Graph Scheduler

To determine the execution order of the global routing tasks, we develop a two-stage task graph scheduler. The first stage is to create a task conflict graph. The task graph scheduler is then used to determine the order of execution for each conflict edge. Following the task conflict graph generation, we extract one maximum non-conflicting set as the root task batch. All of these tasks can be divided into two groups: the root task batch and the non-root task batch. We assign the execution order to each pair of conflicting tasks as follows.

3.3 GPU-friendly 3D Pattern Routing

Figure 4 shows the programming architecture of our GPU-friendly pattern routing framework for all of these routing tasks during the pattern routing stage. Each batch in Figure 4 represents a single routing task, and each task contains several multi-pin nets. As shown in Figure 2, there are two common patterns in pattern routing approaches, where the L-shape pattern provides two candidate routing paths in 2D space and $L \times L$ candidate routing paths in 3D space. At the same time, there are two bend points in Z-shape patterns named the source bend point, and the target bend point since one connects to the source pin, and the other connects to the target pin. Without loss of generality, we discuss two bend points here and treat L-shape patterns as a special case. The bend point pair can get $M + N$ candidate paths in 2D space, where $M$ represents the width of the bounding box of the net on $G$ and $N$ is the height. Section 3.3 defines the notations used in our GPU-friendly 3D pattern routing algorithms for the two-pin net $P_s \rightarrow P_t$.

We define a GPU-friendly 3D pattern routing algorithm as shown in Figure 5. The left part in Figure 5 illustrates one of the solutions of $P_s \rightarrow P_t$ with two bend points $(B^{s(i)}, B^{t(i)})$, where $i$ means the index of the candidate bend point pair and $1 \leq i \leq M + N$. We define $l_s$, $l_b$, and $l_t$ as the layer of the source pin, the wire connecting two bend points, and the target pin respectively.

The double-bend routing path includes three parts,

- the wire connecting the source point $P_s$ to the source bend point $B^{s(i)}$;
- the vias to change routing metal layers from $l_s$ to $l_b$ and the wire connecting to the target bend point $B^{t(i)}$;
- the vias to change routing metal layers from $l_b$ to $l_t$ and the wire connecting to the target point $T_{t_i}$.

In this sample path, $l_s$ is 1, $l_b$ is 2 and $l_t$ is 4. We denote this candidate double-bend pattern routing path as $P\{P_s, B^{s(i)}_s, B^{t(i)}_b, T_{t_i}\}$. According to the above three parts,
The formal formulation of the path cost is,
\[
c(\mathcal{P}, B^{(i)}_s, B^{(i)}_t, T_{1:i}) = c_w(\mathcal{P}, B^{(i)}_s, l_s) + c_b(B^{(i)}_s, l_s, l_b) + c_b(B^{(i)}_t, l_b, l_t) + c_w(\mathcal{P}, B^{(i)}_t, l_t).
\]  

For each pair of bend points \((B^{(i)}_s, B^{(i)}_t)\), we will generate the candidate flow \(i\) for this bend point pair. \(c^{(i)}(P_s, P_t, l_i)\) represents the minimum cost result of the two-pin net \(P_s \rightarrow P_t\) in the \(i\)th candidate flow with the 3D pattern routing algorithm. The calculation of \(c^{(i)}(P_s, P_t, l_i)\) with the inter-net ordering is,
\[
c^{(i)}(P_s, P_t, l_i) = \min_{0 < l_s, l_b \leq L} \left\{ c_{bc}(P_s, l_s) + c_w(\mathcal{P}, B^{(i)}_s, l_s), c_b(B^{(i)}_s, l_s, l_b), c_b(B^{(i)}_t, l_b, l_t), c_w(\mathcal{P}, B^{(i)}_t, l_t) \right\}.
\]  

We propose a merge step to merge the results of all \(M + N\) candidate flows. We can get the final minimum cost as follows,
\[
c^*(P_s, P_t, l_i) = \min_{1 \leq i \leq M + N} c^{(i)}(P_s, P_t, l_i).
\]  

To better utilize the GPU resources, we reformulate it as the computation graph flow using the vector/matrix addition and minimum operation. Our proposed GPU-friendly pattern routing algorithm for the \(i\)th candidate bend point pair \((B^{(i)}_s, B^{(i)}_t)\) is shown in the right part of Figure 5. The weight of the edge \(P_s \rightarrow B^{(i)}_s\) includes the bottom children cost \(c_{bc}(P_s, l_s)\) and the wire cost to connect \(P_s\) and \(B^{(i)}_s\). The formulation of \(w^{(i)}_s\) of the edge weights \(w^{(i)}\) is,
\[
w^{(i)}_s = c_{bc}(P_s, l_s) + c_w(\mathcal{P}, B^{(i)}_s, l_s), 0 < l_s \leq L.
\]  

The metal layer switch procedure at the source bend point is represented by the connection between \(B^{(i)}_s\) and \(B^{(i)}_t\). Based on the connection, we formulate the entry of the edge weights matrix \(W^{(2)}\) at the \(i\)th row and the \(l_b\)th column as
\[
w^{(2)}_{l_s,l_b} = c_w(B^{(i)}_s, l_s, l_b) + c_w(B^{(i)}_t, l_b, l_t), 0 < l_s, l_b \leq L.
\]  

We can also define the metal layer switch procedure at the target bend point as the connection between \(B^{(i)}_t\) and \(T_{1:i}\). The entry at \(l_t\)th row and the \(l_{bc}\)th column of the edge weights matrix \(W^{(3)}\) is defined as,
\[
w^{(3)}_{l_t,l_{bc}} = c_w(B^{(i)}_t, l_t, l_{bc}) + c_w(B^{(i)}_t, l_{bc}, l_t), 0 < l_t, l_{bc} \leq L.
\]  

The minimum cost \(c^*(P_s, P_t, l_i)\) of the candidate bend point pair \((B^{(i)}_s, B^{(i)}_t)\) can be calculated as Equation (7) referring to Equation (2),
\[
c^*(P_s, P_t, l_i) = \min_{0 < l_s, l_b \leq L} \left\{ w^{(1)}_s + w^{(2)}_{l_s,l_b} + w^{(3)}_{l_t,l_{bc}} \right\}.
\]  

Having all the minimum cost of \(M + N\) candidate bend point pairs, we can finally get the \(c^*(P_s, P_t, l_i)\) using the merge step in Equation (3) which can also be computed as the vector minimum operation on GPU.

Both L-shape and Z-shape patterns are included in the above formulations. Separate routing algorithms can be formulated by limiting bend point pair candidates.

### 3.4 Parallel Rip-up and Reroute Iterations

Each multi-pin net is treated as a separate routing task. Then, we apply our task graph scheduler to these routing tasks based on the conflict relationship. Finally, all these routing tasks follow the execution order determined in the ordered task graph. As a result, utilizing Taskflow [Huang et al., 2019] and the ordered task graph, we can quickly maximize the parallelism of our rip-up and reroute iterations.
Table 2: Runtime speedup (SPD) results on ICCAD 2019 benchmarks. AVG means the averaged one.

<table>
<thead>
<tr>
<th>Bench</th>
<th>FastGR 🟢</th>
<th>CUGR</th>
<th>Overall runtime (s)</th>
<th>FastGR 🟢</th>
<th>SPD</th>
<th>FastGR 🟢</th>
<th>SPD</th>
<th>FastGR 🟢</th>
<th>SPD</th>
</tr>
</thead>
<tbody>
<tr>
<td>18t5</td>
<td>34.125</td>
<td>72.169</td>
<td>2.07×</td>
<td>34.125</td>
<td>1.00×</td>
<td>34.125</td>
<td>1.00×</td>
<td>34.125</td>
<td>1.00×</td>
</tr>
<tr>
<td>18t8m</td>
<td>25.708</td>
<td>51.207</td>
<td>2.00×</td>
<td>25.708</td>
<td>1.00×</td>
<td>25.708</td>
<td>1.00×</td>
<td>25.708</td>
<td>1.00×</td>
</tr>
</tbody>
</table>

Table 3: Solution quality results after global routing. The better performance is marked as bold.

<table>
<thead>
<tr>
<th>Bench</th>
<th>Score</th>
<th>Wirelength</th>
<th># Vias</th>
<th># Shorts</th>
<th>Improved (%)</th>
<th>FastGR 🟢</th>
<th>FastGR 🟢</th>
<th>FastGR 🟢</th>
<th>FastGR 🟢</th>
</tr>
</thead>
<tbody>
<tr>
<td>18t5</td>
<td>16919500</td>
<td>16808000</td>
<td>26988200</td>
<td>26915900</td>
<td>0</td>
<td>856362</td>
<td>857523</td>
<td>3188</td>
<td>3130</td>
</tr>
<tr>
<td>18t8m</td>
<td>40881100</td>
<td>40825200</td>
<td>64359900</td>
<td>64217700</td>
<td>8.5</td>
<td>2174240</td>
<td>2179840</td>
<td>37148</td>
<td>37148</td>
</tr>
<tr>
<td>18t9m</td>
<td>42879000</td>
<td>42827800</td>
<td>64554600</td>
<td>64441200</td>
<td>56.94%</td>
<td>1946740</td>
<td>1957020</td>
<td>129707</td>
<td>129707</td>
</tr>
</tbody>
</table>

4 Experimental Results

4.1 Experimental Setup

The framework was developed in C++/CUDA based on CUGR [Liu et al., 2020]. We conducted the experiments on a 64-bit Linux machine with Intel(R) Xeon(R) Gold 6226R CPU @ 2.90GHz and 1 NVIDIA GeForce RTX 3090 GPU. ICCAD2019 benchmarks[Dolgov et al., 2019] were adopted. We integrated our proposed two types of GPU-friendly pattern routing algorithms into the pattern routing stage separately to illustrate the strength of our methods. Further, the details of sorting scheme and threshold settings can be found in [Liu et al., 2022b].

4.2 Acceleration

We evaluate the total runtime, pattern routing runtime, and runtime of the rip-up and reroute iterations to demonstrate the acceleration performance. Section 4 shows that our proposed FastGR 🟢 can bring an overall 2.489× speedup on the pattern routing stage. Meanwhile, the proposed FastGR 🟢 can still obtain 1.970× speedup with much more candidate routing paths, in which the pattern routing stage achieves 2.070× acceleration on average and 3.157× improvement on the maze routing stage due to the reduction of the number of nets to rip up.

4.3 Evaluation on Hybrid Shape Patterns

Section 4 shows the comparison of the solution quality after global routing. It establishes that the global router with hybrid-shape pattern routing algorithm FastGR 🟢 can beat the global router with L-shape pattern routing algorithm FastGR 🟢 [Liu et al., 2022a] with respect to the solution quality on most designs. Since our hybrid-shape pattern routing algorithm considers Z-shape patterns as the candidate routing paths, the number of vias increases reasonably. To further evaluate the solution performance, Dr.CU [Chen et al., 2019] is applied to conduct detailed routing under the guide of the global routing solution. As for the wirelength, our FastGR framework outperforms CUGR on most designs. Furthermore, FastGR can obtain comparable detailed routing performance with CUGR in many aspects (including the number of vias, the number of shorts, and the number of spacing violations) as listed in [Liu et al., 2022b].

5 Conclusion

In this paper, we propose an efficient global routing framework, FastGR, accelerated for CPU-GPU platforms. We propose two GPU-friendly pattern routing algorithms and a heterogeneous task graph scheduler. The experimental results highlight the importance of GPU-accelerated kernel algorithms and the task scheduler for inter-net ordering in routing. An adequate fuse of them can assist in reducing design cycles and improve the solution quality at the same time. In the future, we plan to extend the task graph scheduler to other critical stages and exploit the power of GPU acceleration in the VLSI flow.
References


