BitColor: Accelerating Large-Scale Graph Coloring on FPGA with Parallel Bit-Wise Engines

Haishuang Fan
fanhaishuang20z@ict.ac.cn
State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences; University of Chinese Academy of Sciences
Beijing, China

Ming Li
lim@yusur.tech
YUSUR Technology Co., Ltd.
Beijing, China

Jingya Wu
wujingya@ict.ac.cn
State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China

Wenyan Lu
luwenyan@ict.ac.cn
State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China

Xiaowei Li
lxw@ict.ac.cn
State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China

Guihai Yan
yan@ict.ac.cn
State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China

ABSTRACT

The graph coloring algorithm plays a crucial role in many applications such as social network analysis. However, since the minimal graph coloring problem is NP-complete, which is increasingly computationally and memory-intensive as the number of vertices in the graph grows rapidly. Despite numerous FPGA-based works proposed to accelerate large-scale graph processing algorithms, such as Single Source Shortest Path, and various coloring algorithms, such as linear programming algorithms, efficiently implementing the greedy coloring algorithm for large-scale graphs on FPGA still remains highly challenging due to several reasons: ① The coloring algorithm requires color state traversal to determine the final color after traversing neighbor vertices. The time complexity of color traversal is equal to the neighbor vertices traversal, which is inefficient. ② Neighbor vertices traversal requires extensive random memory accesses on vertex color data. ③ Coloring different vertices in parallel is difficult due to potential color update conflicts between adjacent vertices.

In this paper, we present BitColor to accelerate large-scale graph coloring on FPGA. Firstly, BitColor uses a bit-wise processing engine to complete fast color state determination in one cycle instead of multiple traversal cycles. Secondly, to mitigate heavy random accesses, BitColor adopts a customized high-degree vertex cache (HVC) and a graph prune strategy to reduce off-chip memory access and improve throughput for reading and updating vertex colors. Finally, BitColor employs a data conflict table and a multi-port HVC to enable simultaneous coloring of vertices. The experimental results demonstrate that BitColor achieves 54.9× performance speedup over CPU and 2.71× speedup over GPU.

CCS CONCEPTS

• **Hardware** → **Hardware-software codesign**; **Hardware accelerators**.

KEYWORDS

Graph accelerator, Graph coloring, FPGA

ACM Reference Format:
Haishuang Fan, Ming Li, Jingya Wu, Wenyan Lu, Xiaowei Li, and Guihai Yan. 2023. BitColor: Accelerating Large-Scale Graph Coloring on FPGA with Parallel Bit-Wise Engines. In *52nd International Conference on Parallel Processing (ICPP 2023), August 07–10, 2023, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3605573.3605623

1 INTRODUCTION

The graph coloring assigns colors to vertices of a graph such that no two adjacent vertices share the same color. It is an essential algorithm in many applications, such as pattern matching [10], resource allocation [13, 14], traffic scheduling [2] and social network analysis [21]. However, the problem of coloring a graph with minimum number is an NP-hard problem. With the rapid growth in graph vertices, there is an exponential demand of computing complexity and memory bandwidth to process the coloring algorithm [9].

Due to the considerable deceleration in CPU performance scaling and the substantial power consumption of GPUs, there has been growing interest in expediting large-scale graph processing using FPGAs [1, 6, 7]. Many studies have achieved significant performance and energy efficiency improvements over CPUs and GPUs on some graph algorithms, such as Single Source Shortest Path (SSSP) [8] and k-nearest neighbor (KNN) [19]. However, few works have focused on accelerating large-scale graph coloring on FPGAs.

ICPP 2023, August 07–10, 2023, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0843-5/23/08.
https://doi.org/10.1145/3605573.3605623

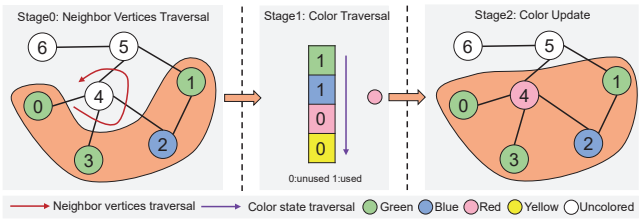


Figure 1: An illustrative example of graph coloring

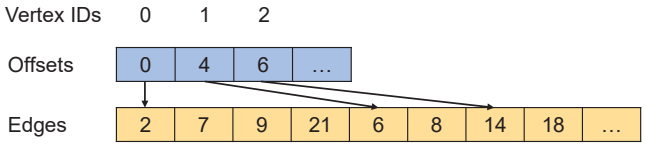


Figure 2: An example of an unweighted graph stored in the compressed sparse row (CSR) format.

Existing works [23, 25] have only accelerated small-scale graphs but shown limited performance gain on large-scale graphs.

There are several algorithms that can reduce the time complexity of coloring, including greedy algorithms [5], dynamic programming algorithms [11], and linear programming algorithms [20]. However, as the number of graph vertices increases, it becomes increasingly challenging for CPU-only graph coloring algorithms to meet the demand for fast graph coloring. Through analyzing commonly used greedy-method algorithms, we have identified three issues that contribute to performance inefficiency. Firstly, the color process of one vertex can be divided into three stages as shown in Fig 1: traversing neighbor vertices to collect the color information of neighbors, traversing color states to determine the final color and updating the vertex color. However, the time complexity of color traversal is equal to that of neighbor vertices traversal, which is inefficient. Secondly, the algorithm checks the color state of neighbor vertices. These neighbors are not successive and this results in extensive random memory accesses on vertices color data. Finally, it is difficult to color different vertices in parallel due to potential conflicts between two adjacent vertices when updating colors.

Three observations have been made about these issues. Firstly, a single color state saved in an array is represented by a single bit and color state traversal aims to checking for the first usable bit. In practice, the first bit among successive bits can be located through AND and NOT bit-wise operations in one cycle. Secondly, vertices in a graph can be divided into high-degree vertices (HDVs) and low-degree vertices (LDVs). The color data of HDVs are more likely to be accessed by their neighbors. If HDVs are cached on-chip, the loaded color data will have a higher probability of being reused than LDVs, thus reducing expensive off-chip data access. Additionally, HDVs have more neighbors than LDVs and require more computations. However, since the vertices can be reordered in descending order of in-degree [12], when coloring HDVs, LDVs will not yet have been colored and their color data check can be skipped to save computation and memory access. Thirdly, conflicts between the color data of two adjacent vertices (vertex 1 and vertex 2) only exist along one edge that connects them. Normally, vertex 2 must be colored after vertex 1 to ensure the correction or vice versa. If we move vertex 1 to the last position in the neighbor vertices traversal stage of vertex 2 coloring and let vertex 2 complete the traversal of neighbor vertices other than vertex 1, it can wait for the color result of vertex 1. When vertex 1 result is ready, vertex 2 collects the final adjacent vertex color and obtains the result through color state traversal. Through this conflict check, adjacent vertices can be colored in parallel.

This paper proposes an FPGA-based accelerator with parallel bit-wise processing engines (BWPEs) for large-scale graph coloring, called BitColor, based on above three observations. To speed up the color state determination, each BWPE uses a bit-wise operation to determine the color. To optimize the data loading between off-chip memory and on-chip buffers for each BWPE, BitColor utilizes a high-degree vertex cache and a graph prune strategy. Finally, BitColor adds a data conflict table and a corresponding scheme to enable the parallel coloring of different BWPEs. As there are data read conflicts among different BWPEs, BitColor uses a customised multi-port cache.

The main contributions of this work are summarized as follows:

- By analyzing the greedy algorithm of the graph coloring problem, we identify the performance bottleneck. We observe that the inefficiency of the algorithm stems from long color state determination, irregular color data access, and lack of parallelisms.
- We propose four techniques to optimize the graph algorithm: high-degree vertices caching, bit-wise color state determining, DRAM read merging, and uncolored neighbor vertices pruning.
- We design a FPGA-based accelerator, BitColor. To enable parallel vertex coloring, we add a data conflict table and a multi-port high-degree cache to achieve high-performance graph coloring.
- We implement BitColor on a Xilinx Alveo U200 accelerator card. The experimental results on various real-world graph datasets demonstrate that BitColor achieves 54.9× performance speedup over CPU and 2.71× over GPU.

2 BACKGROUND

2.1 Graph Format

The graph in this paper is represented in the standard compressed sparse row (CSR) format [28] as shown in Figure 2. CSR uses two arrays to represent a graph: the edges array and the offsets array. The values stored in the edges array represent the destination index. The vertex ID in the offset array implicitly stores the source index. The i-th entry in the offsets array is the starting index (s_e) of i-th vertex and the ending index (d_e) of i-1th vertex in the edge array.

2.2 Graph Coloring Algorithm

Graph coloring problem involves labeling vertices such that no two adjacent vertices have the same color. The problem of coloring a graph with the smallest number of colors is a fundamental NP-hard problem which influences how efficiently some application problems can be solved, such as and social network analysis [21]. The smallest number of colors needed is called the chromatic number

of a graph. As it takes exponential time to compute the chromatic number, there are various heuristic techniques to solve this problems [9], such as greedy algorithms [5] ,dynamic programming [11], and linear programming algorithms [20]. Among them, the greedy method is the most widely used algorithm. In this paper, we mainly focus on the greedy method and present a hardware method to solve the large-scale graph coloring problem.

2.3 Greedy Coloring Algorithm

The basic greedy coloring algorithm iteratively colors on active vertices with the color that has not been assigned on any of its neighbours, shown in Algorithm 1. First, the algorithm initializes the color array for each vertex color state. Then the graph is colored by iterating all vertices. The iteration is divided into three stage: neighbor vertices traversal, color traversal and color update. Since initial of the color array is negligible, we only focus on the three subsequent stages.

Stage 0 - Neighbor Vertices Traversal. The state of each color is attached with a flag initially set to 0 (unused). Once a color is used by a neighbor, its flag is updated to 1 (used). As shown in the Stage0 in Figure 1, vertex 4 has three colored neighbors and one uncolored neighbors. Vertex 0 and vertex 3 are colored by green, and vertex 2 is colored by blue, while vertex 5 is uncolored. After traversing these neighbors, the color state flag array (1,1,0,0) is gotten.

Stage 1 - Color Traversal. The color traversal starts after all neighbors have been accessed. It traverses the color flag array and locates the first unused color as the final color result. After the color result of current vertex is found, the color flag array is cleared to 0 for coloring of next vertex. The color traversal of Figure 1 shows the result color is red.

Stage 2 - Color Update. After the color result is computed, the color of current vertex will be updated. As shown in the color update stage in Figure 1, vertex 4 is updated to red for subsequent vertices coloring.

Algorithm 1 Basic Greedy Coloring Algorithm

Require:
A graph G=(V,E)
Ensure:

```
color_array[VERTEX_NUMBER] = {0}
color_flag[COLOR_NUMBER] = {0}
for each vertex_id in [0,VERTEX_NUMBER) do
  % Stage0 Neighbor Vertices Traversal
  s_e = offset[vertex_id]
  d_e = offset[vertex_id+1]
  for each vertex_des in edge[s_e, d_e − 1] do
    color_neighbor = color_array[vertex_des]
    color_flag[color_neighbor] = 1
  end for
  % Stage1 Color Traversal
  for each color_result in [0,COLOR_NUMBER) do
    if color_flag[color_result] == 0 then
      break
    end if
  end for
  for each color_id in [0,COLOR_NUMBER) do
    color_flag[color_id] = 0;
  end for
  % Stage2 Color Update
  color_array[vertex_id] = color_result
end for
```

2.4 Other Graph Coloring Algorithms Other Than The Greedy Algorithm

We have so far focused on the greedy algorithm, but other coloring algorithms, such as Maximal Independent Set (MIS) [4] and Backtracking (BT) [3] provide different tradeoffs. We highlight three key advantages of the greedy algorithm in BitColor:

- **Time Complexity.** Greedy has lower time complexity ($O(n)$) than Maximal Independent Set (MIS) ($O(5.3^n)$) and Backtracking (BT) ($O(1.3^n)$) algorithms [9]. Here, n represents the number of vertices. However, the greedy algorithm intrinsically lacks parallelism due to potential color update conflicts. We tackle this limitation by employing a conflict detection table and multi-port cache on FPGA.
- **Space Complexity.** The greedy algorithm takes lower space complexity and memory pressure. It relies solely on local information within each iteration. In contrast, MIS-based and BT-based algorithms require additional storage for maximal independent sets and backtracking information. This imposes greater storage pressure on FPGAs especially with limited on-chip resources.
- **Generality.** Our methods in BitColor are applicable to other algorithms facing similar challenges of memory access efficiency and multi-port read conflicts. By utilizing our high-degree cache strategy and multi-port cache, we can greatly enhance FPGA performance for these algorithms as well.

3 MOTIVATION

3.1 Performance Bottleneck Analysis

We conduct quantitative characterizations for the basic greedy coloring algorithm with c-based codes in Intel Xeon Silver CPU 4114. The details of the system configuration and datasets used in this evaluation are given in Setion 5.1. Figure 3(a) shows that the three stages take up 39.24%, 46.53%, and 14.23% of the total execution time in the basic greedy algorithm. The results suggest that the algorithm is inefficient due to the following problems.

3.1.1 Inefficient Color State Traversal. During the actual execution of the algorithm, the most time-consuming stage is the Stage1 Color Traversal, which is inefficient and wastes the computational resources. We analyze this using the example in Figure 1. Suppose the current vertex 4 has neighbors 0, 3, 2, and 5. When coloring vertex 4, it needs four cycles to traverse four neighbors in Stage0, three cycles to find the color result in Stage1, four cycles to clear the color flag array in Stage1, and one cycle to update the vertex 4 color in the color array. As a result, Stage1 takes up seven cycles, which is more than four cycles of Stage0 and one cycle of Stage2. This is contrary to intuition that more time should be took up in neighbors traversal instead of managing color state and Stage0 should be the

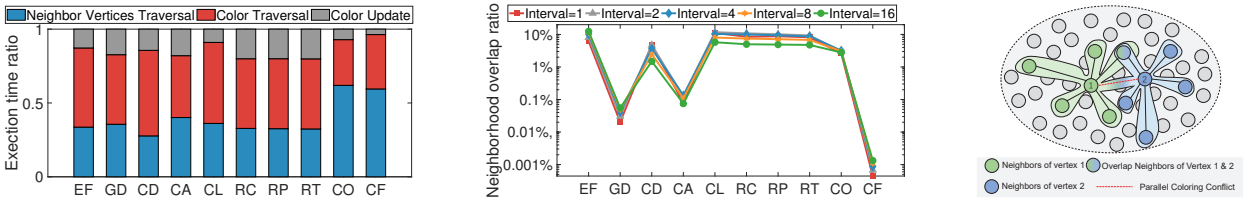


Figure 3: Performance bottleneck analysis of the basic greedy coloring algorithm, include (a)Execution time breakdown of the three stages in the basic greedy coloring algorithm, (b)The average neighborhood of statistical vertices in the order of vertex indices in the graph under different iteration intervals, (c) An example of illustrating the random neighbors of one vertex, the little overlap neighborhoods ratio of the vertex 1 and vertex 2, and parallel coloring conflict

most time-consuming. This abnormal appearance requires us to rethink of a new way to maintain the color state of neighbors.

3.1.2 Irregular Memory Access. The memory (color array) access of graph coloring is irregular, which come from two aspects: random neighbors of one vertex and very little neighborhood overlap ratio.

- First, since most graphs are irregular [6, 26], different neighbors of one vertex are in random distribution. Figure 3(c) shows that the neighbors of vertex 1 are far apart in space, which causes irregular color array access in Stage0. What's more, as the algorithm colors each vertex in order of vertex indices, some neighbors of one vertex may be not colored and this causes useless computation. The example in Figure 1 can illustrate this. The vertex 5 has not been colored when coloring vertex 4, checking the color state of vertex 5 causes redundant computation and memory accesses.
- Secondly, neighbors of two vertices in order are barely overlapped, resulting in little reuse of the color array data. Assume that vertex 1 and vertex 2 in graph are processed sequentially. Figure 3(c) shows that there are nine neighbors of two vertices but there is only one overlap neighbor. In Figure 3(b), we measure the average neighborhood overlap ratio of vertices in order of the vertex indices under different iteration intervals. The iteration interval is the statistical length, for example, when the interval is four and current vertex is vertex 10, neighbors of vertex 9, 8, 7, and 6 will be collect to calculate the overlap ratio. The neighborhood overlap ratio is defined as the number of common neighbors of statistical vertices divided by the number of neighbors of statistical vertices. We can find that most overlap ratios are not more than 10%, ratios of EF and CA are less than 0.1%, ratios of CF are even about 0.001%, and the average value is only 4.96%. Therefore, there is seldom data reuse of the color array due to the low neighborhood overlap ratio.

Since the color data are located in the memory, the random access of neighbors and low data reuse ratio will degrade the algorithm performance when the on-chip cache capacity is limited and the cache strategy is not reasonable. Although the typical U200 FPGA only has 7.947MB ($1766 \times 36Kb$) Internal BRAM, the large graph such CF has 65.6M vertices. Therefore, it is challenging to cache all the color data of the vertices on-chip, which necessitates the design of an efficient and customized cache strategy.

3.1.3 Unfriendly Parallelization. The above two bottlenecks limits the performance when coloring one vertex, but the main reason for limiting the coloring performance of large-scale vertices is actually unfriendly parallelism. This is because the algorithm needs considering the color state of neighbors in stages, which causes potential color update conflicts. We illustrate this using the example in the Figure 3(c). Assume that vertex 1 and vertex 2 are adjacent and ordered. Vertex 1 is colored before vertex 2 in serial coloring, and vertex 2 depends on the color state of vertex 1 when coloring. If vertices 1 and 2 are colored simultaneously, the coloring may fail due to color conflicts between them caused by the delayed update of vertex 1. The parallel difficulty caused by color conflicts between adjacent vertices severely hinders the performance of coloring algorithms.

3.2 Our Observations

Through deep analysis of three stages of the greedy coloring algorithm, we have three main observations.

3.2.1 ***Observation 1****: a single bit in an array represents the single color state saved, and color state traversal aims to check for the first available bit. In practice, one can locate the first bit among successive bits through AND and NOT bit-wise operations in one cycle.*

To solve the inefficient color state traversal problem, we use a bit-level resource pool approach for graph coloring, which uses memory space to uniformly manage the color situation. This approach requires a memory space complexity of O(n), but it only requires a constant time complexity of O(1) to complete the final coloring. The specific method is as follows:

1) Bit-wise color representation. We use a binary string to represent the color state, where a bit indicates whether a color is available or not, and the position of the bit corresponds to different colors. For example, in the sample in Figure 1, we could use four bits to represent four colors: $4'b0001$ for green, $4'b0010$ for blue, $4'b0100$ for red, and $4'b1000$ for yellow.

2) Bit-OR operation to obtain color state in Stage0. We load the colors of the adjacent vertices and color the current vertex, then perform a bit OR operation on the binary strings to obtain the coloring state of the neighbors of the current vertex to be colored:

$$Color\_state = a1 \mid a2 \mid \ldots \mid an$$

In Figure 1, we could obtain the color state of vertex 4 after traversing all neighbors as

$$Color\_state_1 = a_0 \mid a_2 \mid a_3 \mid a_5$$

$$= 4'b0001 \mid 4'b0010 \mid 4'b0001 \mid 4'b0000 = 4'b0011.$$

Algorithm 2 Bit-Wise Greedy Coloring Algorithm

Require:
A graph G=(V,E)
Ensure:
color_array[VERTEX_NUMBER] = {0}
color_state = 0
for each vertex_id in [0,VERTEX_NUMBER) **do**
  % Stage0 Neighbor Vertices Traversal
  s_e = offset[vertex_id]
  d_e = offset[vertex_id+1]
  for each vertex_des in edge[s_e, $d_e - 1$] **do**
    color_neighbor = color_array[vertex_des]
    $color\_state = color\_state \mid color\_neighbor$
  end for
  % Stage1 Color Traversal
  if color_state == 0 **then**
    color_result = 1
  else
    $color\_result = (\sim color\_state)\ \&\ (color\_state + 1'b1)$
  end if
  $color\_state = 0$
  % Stage2 Color Update
  color_array[vertex_id] = color_result
end for

3) Bit-AND and NOT operation to get color result in Stage1. After traversing all adjacent vertices to obtain the final state, we assign the highest-order colors to the colored vertices through the AND and NOT bit-wise operations to satisfy the greedy strategy:

$$Color\_result = (\sim Color\_state)\&(Color\_state + 1'b1)$$

As a result, the color result of vertex 4 in Figure 1 is:

$$Color_1 = (\sim 4'b0011)\&(4'b0011 + 1'b1) = 4'b0100\,(red).$$

Algorithm 2 shows the revised bit-wise greedy coloring algorithm. It is obvious that the bit-wise operation only takes up one cycle in Stage1. What's more, this bit-wise operation is more resource and timing friendly than the comparator in FPGA implementation.

4) Color-Bits compression to save memory resources. The bit representation method used here itself consumes memory resources. For example, the typical 1024 colors only need 10 bits, but now it requires 1024 bits, which is 100 times the original size. We solve this through color bit compression as shown in Table 1. Since color number and color bits are one-to-one correspondences, we can build a look-up-table (1024-entries) to complete the compression and decompression. When a color number of one vertex are loaded to process, we convert the number into bits through looking up the Num2Bit (decompression) conversion table (BRAM) to perform bit-wise operation. When the color result is sent to update the final result, we converts the color bits into the color number. Since the compression is a logarithmic operation, the look-up-table for compression is too large to implement on-chip and the for-loop based logarithmic operation is time-consuming. We proposed a Bit Color Compression Compression Scheme shown in Figure 4 based the observation that there is only one 1-bit in the Color Bits. In this scheme, we can get the compression result in three cycles through three cascaded multiplexers.

Number	Bits
0	00000 (0)
1	00001 (1)
2	00010 (2)
3	00100 (4)
4	01000 (8)
5	10000 (16)
…	…
n	2^{n-1}

Table 1: Color conversion

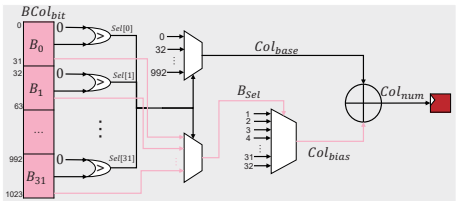


Figure 4: Color bits compression scheme

3.2.2 ***Observation 2:*** *vertices in a graph can be divided into high-degree and low-degree vertices. The color data of high-degree vertices is more likely to be accessed by their neighbors.*

The greedy algorithm processes the vertices in order and edges of each vertex are accessed sequentially. Vertices can be easily dived into two categories according to the number of edges or degree: high-degree vertices (HDVs) and low-degree vertices (LDVs). This is can be achieved using the degree-based grouping (DBG) algorithm [12], which is a common technique to balance graph partitions [1, 6]. DBG reorders and renames vertices in the descending order of in-degree. As a result, smaller vertex index has higher degree. HDVs and LDVs have different characteristics in both computation and memory access:

- **High-Degree Vertices** have a large of adjacent vertices and the access to the color data array is frequent in Stage0, resulting in long processing time. When coloring other vertices, color data of HDVs is frequently accessed.

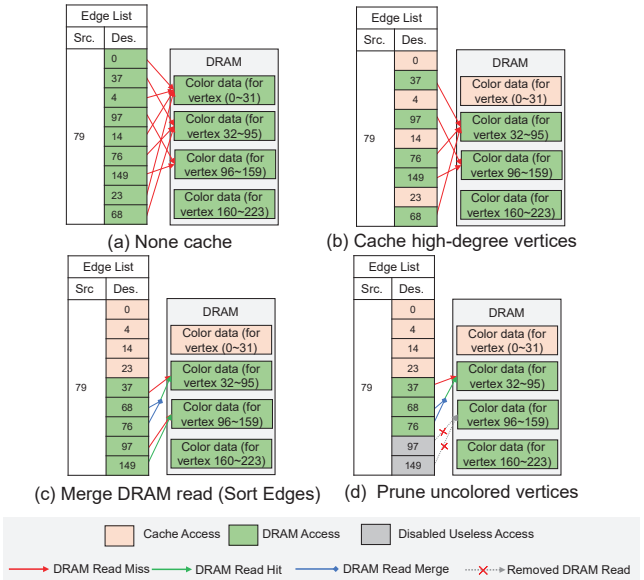


Figure 5: DRAM access in different optimization strategies

- **Low-Degree Vertices** have fewer adjacent vertices, access color data infrequently, and have shorter processing time. When coloring other vertices, LDVs have fewer access.

Based on the vertices partition, we propose three strategies to optimize irregular memory access as shown in Figure 5:

1) Cache high-Degree vertices. Since the color data of HDVs is hot data in memory, we propose to cache the color data of HDVs on-chip and store color data of LDVs in off-chip DRAM. Figure 5(b) shows our cache strategy that can decrease the memory access time of HDVs efficiently. Since the random neighbors of one vertex and low neighbors overlap ratio result in poor temporal locality as shown in Figure 3(c) and frequent DRAM read miss Figure 5(a), spatial locality is only opportunity to optimize the memory access. Therefore, a direct HDV cache strategy is simple but efficient.

2) Merge DRAM read of one vertex to optimize DRAM access of low-degree vertex data through sorting edges. The access speed of color data for LDVs in off-chip DRAM is much slower than that for HDVs in the on-chip cache, which may limit the performance. To address this issue, we propose to sort the edges of each vertex in ascending order of the destination vertex index (neighbor) in the preprocessing stage. This way, we can cluster the LDVs together and make their DRAM access a sequential read operation, which can improve DRAM hit rate. Moreover, since the DRAM data width is 512 bits and the color data is 16 bits, one DRAM access can load 32 color data of consecutive vertices. By sorting the edges, we can merge some DRAM accesses into one burst read, which improves the DRAM access efficiency. Figure 5(c) shows the example after sorting edge of one vertex.

3) Prune uncolored vertices. As uncolored neighbors do not affect the correctness of coloring results, some useless computation and DRAM accesses can be pruned. Because the algorithm colors vertices in order of vertex indices, vertices with bigger indices than current vertex must be not colored. Therefore, we can identify uncolored vertices through comparing indices. Using the example in Figure 5(d), vertex 97, which is bigger than current vertex 79, has not been colored and its computation and DRAM access can be removed. What's more, as edges are sorted in ascending order, once an uncolored vertex is found then its following vertices, such as vertex 149 in Figure 5(d), are also uncolored and can be pruned directly.

3.2.3 ***Observation 3:*** *color update conflicts between two adjacent vertices only exist along one edge that connects them. Through conflict detection and computation scheduling, we can color vertices in parallel.*

In Figure 3(c), we can move vertex 1 to the end of the adjacency list of vertex 2. After vertex 2 has traversed all neighbors except vertex 1, it waits for the color result of vertex 1 until vertex 1 finalizes coloring. This technique allows us to defer the computation of conflict vertices and enable parallel coloring of adjacent vertices. However, there are two main challenges for implementing this scheme. The first is how to efficiently detect and resolve data conflicts, as the current vertex may not be aware of conflicts and updates from other vertices. The second is how to ensure multiple coloring tasks of different vertices can read the cache at the same time, since the cache instantiated using BRAM has only two read/write ports in FPGA. To address these challenges, we design **a data conflict table and a multi-port cache to enable parallel coloring** in our accelerator, which we will describe in Section 4.

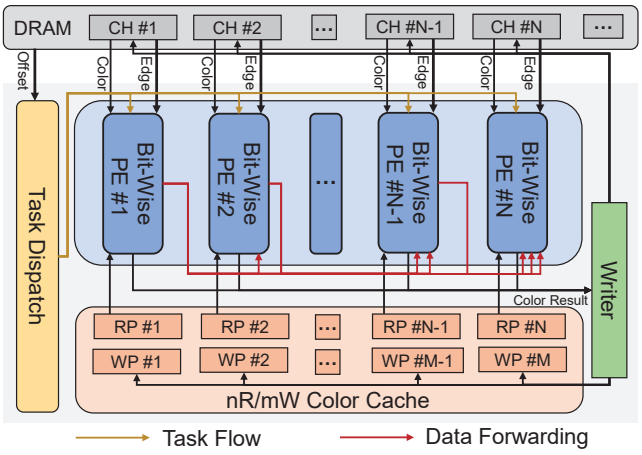


Figure 6: Architecture overview of BitColor

4 ARCHITECTURE

4.1 Architecture Overview

Based on above observations, we present BitColor, an FPGA based accelerator that can improve the efficiency of large-scale graph coloring. Figure 6 shows the architecture overview of BitColor. It is composed of a Task Dispatch Unit, parallel Bit-Wise processing engines (BWPEs), a multi-port Color Cache with N read ports (RPs) and M write ports (WPs), and the Writer modules.

Task Dispatch Unit assigns different vertices to parallel BWPEs. It sends the source vertex index, the destination vertices (edges) range consists of the start address and the end address, and vertices of all BWPEs to configure the data conflict table for parallel coloring. Each BWPE connects to a separate logical DRAM channel (CH) and one Color Cache RP. N parallel BWPEs process multiple source vertices and edges per cycle. The input of a BWPE is destination vertices indices of edges (neighbors). The BWPE fetches color data of neighbors from the DRAM or Cache according to its degree, completes vertex coloring, and forwards its color result to subsequent BWPEs to resolve the data conflict. The Writer module receives color results from BWPEs and updates colors of source vertices to DRAM channels or cache according to degrees. All accesses to DRAM are in granularity of a block (with 512 bits) for high memory efficiency.

In this section, we first introduce the architecture of the BWPE in subsection 4.2, which performs the bit-wise greedy algorithm. Second, we present the two key modules to enable parallel operations of BWPEs: the data conflict scheme based on the data conflict table (DCT) and the multi-port cache for HDVs respectively in subsection 4.3 and subsection 4.4. Then we introduce the color loader for LDVs to implement DRAM read merging to improve the DRAM access efficiency in subsection 4.5. Finally, we describe the degree-aware task scheduling in subsection 4.6.

4.2 Bit-wise Processing Engine

Figure 7 depicts the architecture of the the bit-wise processing engine. Before coloring, the Task Dispatch Unit initializes parameters of the BWPE, including the source vertex (v_{src}), the start address (s_e) and end address (d_e) of edges for loading edges, and the vertices of other BWPEs (v_1, v_2, v_3) to configure data conflict table (DCT).

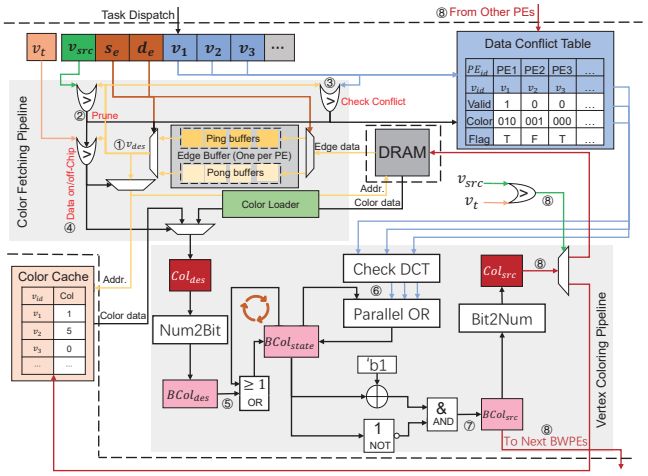


Figure 7: The architecture of the bit-wise processing engine

The logic of the BWPE is divided into two small pipelines: the color fetching pipeline, which minimizes the number of memory requests sent to DRAM, and vertex coloring pipeline, which completes the bit-wise coloring operation.

As shown in Figure 7, the data flow of the BWPE is as follows. In Step ①, BWPE loads destination vertices (v_{des}) of edges data from an edge ping-pong buffer. The ping-pong buffer enables the BWPE to fetch edges from DRAM to one buffer and load destination vertex (v_{des}) for processing from the other buffer, thus improving effective memory bandwidth. We allocate both ping and pong buffers for each BWPE for parallel processing. In Step ②, the color fetching pipeline compares the v_{des} with v_{src}: it will prune the uncolored vertices and move to the next edge if v_{des} is greater than v_{src}. Otherwise, it proceeds to the following steps. In Step ③, v_{des} is sent to check data conflict by comparing it with vertices in DCT. The vertex will be moved to the Step ⑥ and skip Step ④ & ⑤ if there is a conflict. Otherwise, v_{des} will be sent to check the color state. In Step ④, BWPE compares v_{des} with v_t, which is the vertex threshold to indicate whether the color data of v_{des} (Col_{des}) is cached on-chip or not. If v_{des} is less than v_t, Col_{des} will be fetched from the color cache directly. Otherwise, the Color Loader will request it from off-chip DRAM.

The vertex coloring pipeline processes the vertex color data from the color fetching pipeline. In Step ⑤, it decompresses Col_{des} into bits ($BCol_{des}$) through looking up the Num2Bit table as shown in Table 1 and performs Bit-OR operation to update the color state ($BCol_{state}$). Step ①~ ⑤ repeat until all neighbors of current vertex are done. In Step ⑥, the pipeline checks the data conflict table to complete the conflict neighbors found in Step ③. Unlike Step ⑤, it performs parallel OR on all color data and obtains the $BCol_{state}$ result in one cycle. Then $BCol_{state}$ is sent to perform the Bit-AND and NOT operation to get the color result of v_{src} ($BCol_{src}$) in Step ⑦. Finally, the pipeline compresses $BCol_{src}$ into Col_{src} through the proposed compression scheme as shown in Figure 4 and stores it into on-chip cache or off-chip DRAM in Step ⑧. Moreover, $BCol_{src}$ is also forwarded to other BWPEs to update the data conflict table.

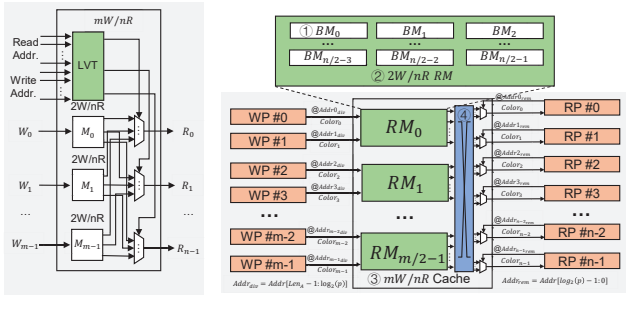
4.3 Data Conflict Detection Scheme

The data conflict detection scheme in BitColor is added to detect data conflicts among BWPEs. We introduce a data conflict table into BWPEs, as shown in Figure 7. The table has a total of five rows. The first row is the PE index (PE_{id}). We stipulate that when two PEs conflict, the PE with smaller index completes the coloring first, such as PE1 before PE2. The second row is the vertex index (v_{id}) that is being colored in PE_{id}. It is used to check if there is a conflict between v_{des} with v_{id}. The third row is the completion valid bit that indicates whether v_{id} has completed coloring, 1 means completion, 0 indicates incomplete, and the initial value is 0. The fourth row is the coloring result (bits) of v_{id}, which is used for the current PE coloring, with an initial value of 0. The fifth row is the conflict flag that indicates whether two BWPEs conflict. When there is a conflict, it is True (T), when there is no conflict, it is False (F), and initially it is F.

The conflict detection scheme mainly works in three steps in Figure 7 to ensure the correctness of coloring. The first is Step ③. When two vertices collide, the valid flag is set to T, the coloring check of the vertex is performed after all non conflicting vertices have completed coloring. If there is no conflict, the flag is set to F. The second is in Step ⑥. After completing the previous color check, BWPEs read the data conflict table to check for vertices that have not been calculated when all conflict vertices (Flag is T) have complete coloring (Valid is T). In order to improve performance, the entry of the conflict detection table is implemented using registers instead of BRAM, as the BRAM port restricts the parallelism of reading and calculation, so that the Bit-OR can be completed in one cycle. The third is in step ⑧. When the BWPE completes the coloring, the valid of PE_{src} in the data conflict table of other BWPEs is set to T to notify current BWPE's completion of the calculation, and the color row is also updated. This direct data update is to avoid accessing data from the cache and DRAM again.

4.4 Multi-Port Cache for High-degree Vertices

During processing, source vertices are processed simultaneously by BWPEs to obtain color data of neighboring vertices from the color data cache. To support parallel execution of BWPEs and resolve read conflicts, the cache must support multiple read ports. However, current FPGAs only provide block BRAMs with two ports (2W2R). Traditional approaches for constructing multi-ports memory, such



(a) LVT-based cache (b) Proposed multi-port cache based on address bit-selection

Figure 8: The architecture of different multi-port caches

as replication, banking, and multipumping, can be inefficient in terms of area and frequency. To address these issues, LaForest et al. [15] proposed a Live Value Table (LVT) based design that combines BRAMs into multi-ported memory, as shown Figure 8(a). LVT enables indirect steering of reads and selects data from the bank that holds the most recent or "live" write value for the input address. However, the LVT based method has some issues. Firstly, the number of items stored in the increased LVT table for addresses is as large as the original memory, which increases area overhead. Secondly, the read operation needs to check LVT, which corresponds to one LVT read operation and increases read latency. Lastly, this method directly replicates $m \times n$ copies of the memory bank with the original size of D, resulting in a final size of $m \times n \times D/4$. Due to the fact that not every position of the bank is written when writing BRAM, there is significant area redundancy caused by spatial redundancy.

In BitColor, we have discovered an effective solution to the aforementioned problems through replacing LVT by the bit-wise divisor and remainder operations. If each BWPE has a fixed color vertex index during parallel execution on HDVs, it will exhibits a certain pattern when writing to the cache. This pattern is as follows:

Assuming there are P BWPEs executing in parallel, the ID-th BWPE coloring HDVs are ID, ID+P, ID+2P, ... , ID+tP. Therefore, their write addresses to the cache are discrete and also ID, ID+P, ID+2P, ..., ID+tP.

We can ensure this pattern during task scheduling which will be discussed in Section 4.6. Based on the above pattern, we construct the proposed multi-port cache based on address bit-selection shown in Figure 8(b) as follows:

- In Step ①, we set the size of the basic 2W/2R BRAM (BM) is $2D/P$ instead of D. For example, if the parallelism(P) is 8, the BM size is D/4 and only stores the color results output by two neighboring BWPEs, such as #0&#1, #2&#3.
- In Step ②, we build a 2WnR BRAM (RM) by replicating n/2 copies of BM, which stores the same content. For example, when n=8, we copy BM 4 times, and the RM has eight read ports (RPs).
- In Step ③, we copy RM again in m/2 copies to build the mW/nR cache. These write ports (WPs) are assigned to different BWPEs and each BWPE writes color data of different vertices, so the content saved in different RMs is different. At the same time, the write address is converted to the actual address through dividing by P:

  $$Addr_{div} = Addr/P = Addr_{write}[Len_A - 1 : log_2(P)].$$

  This bit-selection based divisor operation avoids writing the LVT to save the RM number.
- In Step ④, multiplexers are added to enable n read operations. The $Addr_{div}$ of the i-th RP read address is sent to m RMs, m output colors are sent to the i-th multiplexer, and the result color is chosen by taking the remainder of the read address:

  $$Addr_{rem} = Addr\%P = Addr_{read}[log_2(P) - 1 : 0].$$

  This bit-selection divisor and remainder operation can accurately determine the address and the RM that stores the color.

In summary, our proposed bit-selection operation based address allocation method eliminates the area and delay overhead of LVT and the spatial redundancy. The total BRAM size is $m \times n \times D/(2 \times P)$, which is only 2/P of the LVT based method, As the parallelism increases, the effect of the proposed cache becomes more and more obvious. To be specific, m=n=P in our design, the BRAM size is $P \times D/2 (P > 1)$.

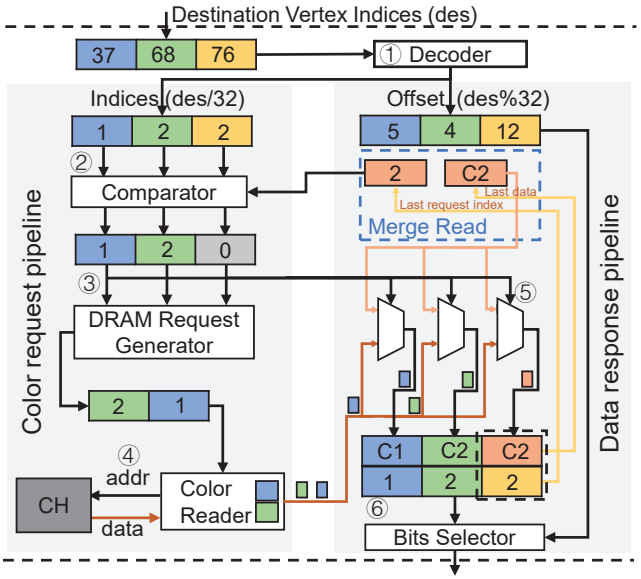


Figure 9: The architecture of the Color Loader

4.5 Color Loader for Low-degree Vertices

Figure 9 illustrates the architecture and data flow of the Color Loader. It takes a set of destination vertex indices from Step ④ in Figure 7 as input and produces a set of destination vertex colors as output. Since the indices are in ascending order as discussed in Section 3.2.2, we cache the last requested index and data to enable read merge operations and improve the DRAM access efficiency. The logic is divided into two small pipelines: the color request pipeline and the data response pipeline.

The data flow of the Color Loader is illustrated in Figure 9. In Step ①, the decoder computes the block indices in DRAM and the offset of destination (des) vertices in the block. Since the color is 16-bits (only 10 bits are used to represent 1024 colors) and a block is 512 bits, the index is calculated as $des/32$, and the offset is calculated as $des\%32$. In Step ②, the color request pipeline compares the indices with the last request index and sets it as zero if they are equal, otherwise, it outputs the index. In Step ③, the DRAM Request Generator module generates requests with non-zero indices. The index marked zero will skip the DRAM access to Step ⑤. In the given example, the third vertex 76 saves one DRAM access. In Step ④, the Color Reader fetches a color data block (512 bits) according to the indices from DRAM. Then it processes the color data block in the data response pipeline. In Step ⑤, it uses the comparison result from Step ② to get the color data. If the comparison index is zero, it reuses the last request block data, otherwise it outputs color data from the Color Reader. In Step ⑥, the Bits Selector decodes the vertex color data based on its offsets in the data block and outputs them. Finally, it updates the last request index and its data block to enable DRAM Read Merge.

Table 2: Preprocessing time with one CPU thread in the millisecond (ms)

Graphs	EF	GD	CD	CA	CL	RC	RP	RT	CO	CF
Graph reordering	0.59	10.63	92.28	135.75	1881.59	534.73	282.13	318.18	2007.18	80657.54
Coloring	78.83	303.31	1071.77	900.07	26282.60	4275.03	3333.81	3894.42	87184.40	757495.12

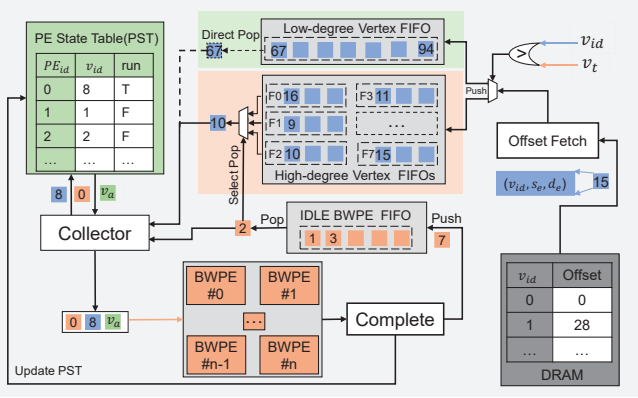


Figure 10: The architecture of Task Dispatcher Unit

4.6 Degree-Aware Task Scheduling

Effective resource scheduling is crucial for achieving the full potential of parallel BWPEs in BitColor, and we employ a degree-aware method for task allocation and workload balance. To this end, we introduce a task dispatcher unit in this section.

Our task scheduling strategy includes two allocation strategies based on the degree threshold (v_t) partition: high-degree vertex allocation and low-degree vertex allocation. The high-degree allocation strategy involves selecting an v_{id} according to the ID of BWPEs. The IDs of vertices allocated to BWPEs must match the pattern described in Section 4.4 to minimize the cost of the multi-port cache. Furthermore, since we reorder the graph in descending order of in-degree and the degree of vertices in BWPEs is close, the computational workload of parallel BWPEs is balanced. On the other hand, the low-degree allocation strategy is a direct allocation without considering IDs. Since the results of low-degree vertices are saved to DRAM and do not take up the write port of the cache, this first-come-first-serve mode eases workload imbalance caused by longer DRAM access of some LDVs.

Figure 10 depicts the architecture of the task dispatcher unit which comprises two main structures: high-degree vertex (HDV) and low-degree vertex (LDV) FIFOs , a PE state table (PST). The HDV and LDV FIFOs are crucial to implementing our degree-aware scheduling. The HDV FIFOs consists of sub-FIFOs bound with the BWPE ID (PE_{id}) and the LDV FIFO is not bound with any PE_{id}. The PE State Table (PST) stores the PE state, including the vertex ID (v_{id}) under processing and the running flag (T is BUSY, F is IDLE), and is used to update the data conflict table in the BWPE. The Offset Fetch module fetches the s_e and d_e of v_{id} from the offset array in DRAM and pushes it to FIFOs according to its degree. During coloring, the IDLE BWPE FIFO pops a PE_{id} if it is not empty. If the HDV FIFOs are not empty, it will pop a vertex from sub-FIFOs according to PE_{id}. Otherwise, the LDV FIFO directly pops a vertex. The collector receives v_{id} and PE_{id}, fetches entries in PST and updates the v_{id} entry and the running flag of PE_{id} in PST. The it dispatches the task to the BWPE PE_{id}. When a BWPE finishes coloring, the Complete Unit pushes its PE_{id} into the IDLE BWPE FIFO and sets its running flag in PST to F.

Table 3: The graph datasets.

Graphs	Nodes	Edges	Type	Categories
ego-Facebook (EF) [18]	4.1K	88.2K	UnDirected	Social network
gemsec-Deezer_HR (GD) [24]	54.5K	498.2K	UnDirected	Social network
com-DBLP (CD) [29]	317.0K	1.0M	UnDirected	Collaboration network
com-Amazon (CA) [29]	335.8K	925K	UnDirected	Product network
com-LiveJournal (CL) [29]	3.9M	34.7M	UnDirected	Social network
roadNet-CA (RC) [17]	1.9M	5.5M	UnDirected	Road networks
roadNet-PA (RP) [17]	1.1M	3.1M	UnDirected	Road networks
roadNet-TX (RT) [17]	1.3M	3.8M	UnDirected	Road networks
com-Orkut (CO) [29]	3.0M	117.1M	UnDirected	Social network
com-Friendster (CF) [29]	65.6M	1806.1M	UnDirected	Social network

5 EVALUATION

We first evaluate the efficiency of the proposed optimization techniques in the single BWPE and the parallel BWPEs. We then compare BitColor with GPU and CPU solutions. Finally, we report the resource utilization.

5.1 Experimental Setup

5.1.1 Hardware platform. We implement BitColor on the Xilinx Alveo U200 accelerator card equipped with a 64GB DDR using Xilinx Vitis 2022.2. The U200 card provides 63.576Mb on-chip BRAM, 892K LUTs and 2.364M Registers. In the evaluation, the maximum color number is set to 1024 and the single cache is 1MB (512K vertices color data). We host the FPGA on a server with two 10-core Intel Xeon Silver 4114 @ 2.02GHZ, 128 GB of DDR4-2133 RAM.

5.1.2 DataSets. Table 3 shows the details of the graph datasets from SNAP Datasets [16] that we use in our experiments, including five social networks, three road networks, one collaboration network, and one product network. We perform degree-based grouping reordering algorithm on these datasets. Table 2 shows the preprocessing time on one CPU thread. Compared with the coloring, the graph reordering cost is small.

5.1.3 Baseline. We compare BitColor with the state-of-the-art parallel graph coloring algorithm [22] based on Gunrock [27] in the GPU Platform and the basic greedy coloring algorithm in CPU. All evaluations are done on a Xilinx Alveo U200, an Intel Xeon Silver 4114, and an NVIDIA Titan V GPU with 12GB on-board memory.

5.2 Efficiency of the Proposed Optimization Techniques

The optimization techniques in BitColor consist of two components: non-parallel optimization techniques in a single BWPE and parallel optimization techniques in multiple BWPEs.

5.2.1 Optimization techniques in a single BWPE. We investigate the performance improvement of each optimization technique in a

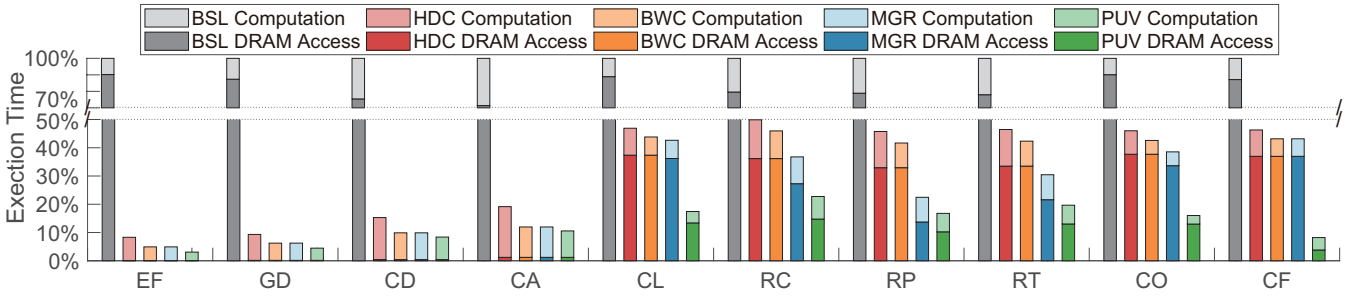


Figure 11: Performance of the single BWPE under different optimization techniques. (BSL: Baseline, HDC: High-degree Vertices Cache, BWC: Bit-Wise Coloring, MGR: Merge DRAM Read, PUV: Prune Uncolored Vertices)

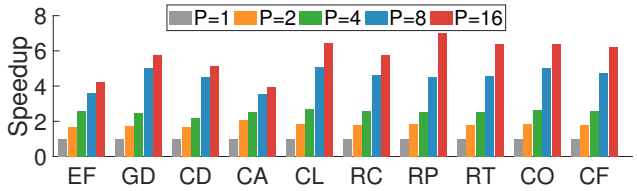


Figure 12: Performance speedup of BitColor with varying the parallelism

single BWPE. We use the performance without optimization techniques as the baseline (BSL) and measure the performance in terms of computation and DRAM access for each technique. All results are normalized to the BSL and are shown in Figure 11.

By using the high-degree cache (HDC), BitColor reduces a large amount of off-chip memory accesses. For small graph such as CD, almost all of DRAM accesses are eliminated. For large graphs, DRAM access time is reduced by about 55%. This shows the spatial locality exists in graph coloring. With the bit-wise coloring (BWC), BitColor further reduces the computation time by 45%. The effect of BWC is more obvious after the access time decreases, especially in small graphs. Through adding the Color Loader module to enable the Merge DRAM Read (MGR), we avoid redundant DRAM read operations. Though its effect is not significant in some graphs such as CF, it can reduce more than 10% DRAM access in the RC, RP and RT after HDC. Finally, after pruning uncolored vertices (PUV), both DRAM access and computation are reduced significantly. This shows that there are large number of useless computation in graph coloring. As a result, the BWPE in Bitcolor achieves 88.63% DRAM access reduction, 66.89% computation, and 82.91% total execution time reduction compared with BSL.

5.2.2 Optimization techniques in parallel BWPEs. Figure 12 illustrates the BitColor performance speedup with varying the parallelism. With the BRAM resource limitation, the maximum parallelism is set to 16. Figure 12 also demonstrates that 16 BWPEs in parallel can achieve a speedup 3.92× ∼ 7.01× over a single BWPE. The trends show that BitColor scales well but the speedup is lower than the parallelism, which is because of the data conflict among parallel vertices.

5.2.3 Accuracy (Color Number). Table 4 shows the results of color number with and without edge sorting. The number of colors is reduced by 9.3% on average in a single step iteration, thus improving the coloring efficiency. For example, the number of colors for CO graph decreases from 116 to 87. This is because the sorting scheme makes the global coloring information more obvious.

Table 4: Color number result

Graphs	EF	GD	CD	CA	CL	RC	RP	RT	CO	CF
BSL	86	21	334	114	10	116	156	5	5	5
Sorting	76	17	328	114	7	87	129	5	5	5

5.3 Comparison with CPU and GPU

We compare BitColor with a parallel graph coloring algorithm implementation in GPU [22] based on Gunrock [27], which is the state-of-the-art open-source graph analytic library, and a baseline greedy algorithm implementation in CPU. Figure 13 shows the performance results on ten datasets.

The performance speedup of BitColor to CPU is 30× ∼ 97× and the average is 54.9×. Due to the detailed optimization in terms of memory access and computation and the customized parallel hardware modules, BitColor has a significant performance gain compared to CPU.

The improvement of BitColor compared with Gunrock is 1.63× ∼ 6.69× and the average improvement is 2.71×. Compared with the BWC and PUV optimization techniques, Gunrock is lack of in-depth algorithm optimization. Even though GPU can utilize even more than 30 cuda cores, the cache size is too small to handle to the irregular memory access. On the other hand, it is difficult to parallelize these cuda cores to do computation. By exploiting the HDC strategy, customizing the data conflict detection hardware module, and employ the MB-level multi-port cache, BitColor enhances the effective of the parallel graph-coloring algorithm.

To measure the throughput,we use MCV/S (Million Colored Vertices per Second) as a standard metric. The average throughput of

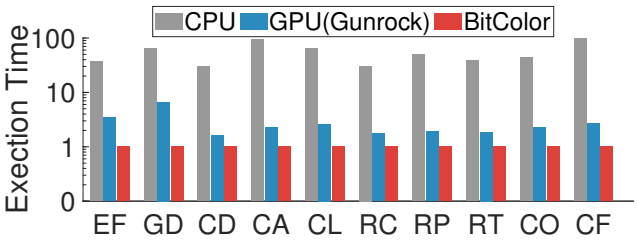


Figure 13: Performance speedup over CPU and GPU

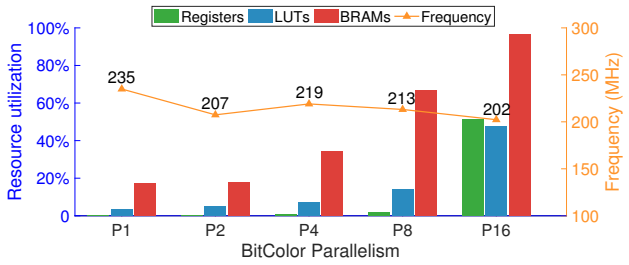


Figure 14: Resource utilization and frequency of BitColor implementations with varying the parallelism

CPU, GPU, and BitColor is 0.88MCV/S, 15.3MCV/S, and 41.6MCV/S, respectively.

For energy efficiency, we use the KCV/J (Kilo Colored Vertices per Joule) as a standard metric. Experiments show that the energy efficiency of CPU, GPU, and BitColor is 12 KCV/J, 19 KCV/J, and 156 KCV/J respectively. Compared with CPU and GPU, BitColor achieves 13x and 8.2x improvement in energy efficiency respectively.

5.4 Resource Utilization

Figure 14 shows resource utilization on U200 and frequency of BitColor implementation with varying the parallelism. As the parallelism increases, register, LUT and BRAM consumption grow nearly linearly before the parallelism is eight (P8). When the parallelism is sixteen (P16), the resource increases exponentially. As a result, BitColor utilizes around 51.09% of Registers, 47.79% LUTs, and 96.72% of BRAMs. Moreover, the frequency is always more than 200MHz, benefiting from the bit-wise color operation and the address bit-selection based multi-port cache.

6 CONCLUSION

In this paper, we present BitColor, an FPGA-based accelerator for large-scale graph coloring with parallel bit-wise engines. BitColor adopts a customized high-degree vertex cache and a graph prune strategy to reduce off-chip memory access and eliminate useless computation. Moreover, BitColor employs a data conflict table and a customized multi-port cache to enhance the parallelism. The experimental results demonstrate that BitColor achieves 54.9× performance speedup over CPU and 2.71× speedup over GPU.

ACKNOWLEDGMENTS

This work is supported in part by National Natural Science Foundation of China (NSFC) under grant No.(62002340, 62090020, 61872336), Youth Innovation Promotion Association CAS under grant No.Y201923, and the Strategic Priority Research Program of the Chinese Academy of Sciences under grant No.XDB44030100. Part of this work is supported by the internship program of YUSUR Technology Co., Ltd. The corresponding author is Guihai Yan (yan@ict.ac.cn).

REFERENCES

[1] Mikhail Asiatici and Paolo Ienne. 2021. Large-scale graph processing on FPGAs with caches for thousands of simultaneous misses. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 609–622.
[2] Nicolas Barnier and Pascal Brisset. 2004. Graph coloring for air traffic flow management. *Annals of operations research* 130 (2004), 163–178.
[3] Richard Beigel and David Eppstein. 2005. 3-coloring in time O (1.3289 n). *Journal of Algorithms* 54, 2 (2005), 168–204.
[4] Hans L Bodlaender and Dieter Kratsch. 2006. An exact algorithm for graph coloring with polynomial memory. *UU-CS* 2006 (2006).
[5] Daniel Brélaz. 1979. New methods to color the vertices of a graph. *Commun. ACM* 22, 4 (1979), 251–256.
[6] Xinyu Chen, Yao Chen, Feng Cheng, Hongshi Tan, Bingsheng He, and Weng-Fai Wong. 2022. ReGraph: Scaling Graph Processing on HBM-enabled FPGAs with Heterogeneous Pipelines. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1342–1358.
[7] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2021. ThunderGP: HLS-based graph processing framework on FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 69–80.
[8] Yuze Chi, Licheng Guo, and Jason Cong. 2022. Accelerating SSSP for power-law graphs. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 190–200.
[9] Alane Marie de Lima and Renato Carmo. 2018. Exact algorithms for the graph coloring problem. *Revista de Informática Teórica e Aplicada* 25, 4 (2018), 57–73.
[10] Riccardo Dondi, Guillaume Fertin, and Stéphane Vialette. 2011. Complexity issues in vertex-colored graph pattern matching. *Journal of Discrete Algorithms* 9, 1 (2011), 82–99.
[11] David Eppstein. 2002. Small maximal independent sets and faster exact graph coloring. *J. Graph Algorithms Appl* 7, 2 (2002), 131–140.
[12] Priyank Faldu, Jeff Diamond, and Boris Grot. 2019. A closer look at lightweight graph reordering. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 1–13.
[13] Gert Goossens, Johan Van Praet, Dirk Lanneer, Werner Geurts, Augusli Kifli, Clifford Liem, and Pierre G Paulin. 1997. Embedded software in real-time signal processing systems: Design technologies. *Proc. IEEE* 85, 3 (1997), 436–454.
[14] Jianding Guo. 2018. *Theoretical research on graph coloring: Application to resource allocation in device-to-device 4G radio system (LTE)*. Ph. D. Dissertation. Bourgogne Franche-Comté.
[15] Charles Eric LaForest and J Gregory Steffan. 2010. Efficient multi-ported memories for FPGAs. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*. 41–50.
[16] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[17] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. 2009. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics* 6, 1 (2009), 29–123.
[18] Jure Leskovec and Julian Mcauley. 2012. Learning to discover social circles in ego networks. *Advances in neural information processing systems* 25 (2012).
[19] Chaoqiang Liu, Haifeng Liu, Long Zheng, Yu Huang, Xiangyu Ye, Xiaofei Liao, and Hai Jin. 2023. FNNG: A High-Performance FPGA-based Accelerator for K-Nearest Neighbor Graph Construction. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 67–77.
[20] Anuj Mehrotra and Michael A Trick. 1996. A column generation approach for graph coloring. *informs Journal on Computing* 8, 4 (1996), 344–354.
[21] Mohamed Atef Mosa, Alaa Hamouda, and Mahmoud Marei. 2017. Graph coloring and ACO based summarization for social networks. *Expert Systems with Applications* 74 (2017), 115–126.
[22] Muhammad Osama, Minh Truong, Carl Yang, Aydın Buluç, and John Owens. 2019. Graph coloring on the GPU. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 231–240.
[23] LM Pochet, ML Linderman, SL Drager, and RL Kohler. 2002. Field-programmable gate-array-based graph coloring accelerator. *Journal of spacecraft and rockets* 39, 4 (2002), 474–480.
[24] Benedek Rozemberczki, Ryan Davies, Rik Sarkar, and Charles Sutton. 2019. GEMSEC: Graph Embedding with Self Clustering. In *Proceedings of the 2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2019*. ACM, 65–72.
[25] Valery Sklyarov, Iouliia Skliarova, and Bruno Pimentel. 2007. FPGA-Based Implementation of Graph Colouring Algorithms. *Autonomous Robots and Agents* (2007), 225–231.
[26] Jiawen Sun, Hans Vandierendonck, and Dimitrios S Nikolopoulos. 2017. GraphGrind: Addressing load imbalance of graph partitioning. In *Proceedings of the International Conference on Supercomputing*. 1–10.
[27] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T Riffel, et al. 2017. Gunrock: GPU graph analytics. *ACM Transactions on Parallel Computing (TOPC)* 4, 1 (2017), 1–49.
[28] Brian Wheatman and Helen Xu. 2018. Packed compressed sparse row: A dynamic graph representation. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–7.
[29] Jaewon Yang and Jure Leskovec. 2012. Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*. 1–8.