An Optimal Algorithm for Enumerating Connected Convex Subgraphs in Acyclic Digraphs

Chenglong Xiao*,Shanshan Wang*,Wanjun Liu†,Xinlin Wang†, Emmanuel Casseau‡
*Shantou University, China
{chenglong.xiao,celine.shanshan.wang}@gmail.com
†Liaoning Technical University, China
{liuwanjun,wangxinlin}@lntu.edu.cn
‡Univ Rennes, Inria, France
{emmanuel.casseau}@irisa.fr

Abstract

Reconfigurable computing system is emerging as an important computing system for satisfying the present and future computing demands in performance and flexibility. Extensible processor is a representative implementation of reconfigurable computing. In this context, custom instruction enumeration problem is one of the most computationally difficult problems involved in custom instruction synthesis for extensible processors. Custom instruction enumeration problem is essentially enumerating connected convex subgraphs from a given application graph. In this paper, we propose a provable optimal algorithm for enumerating connected convex subgraphs in acyclic digraphs in the sense of time complexity. The running time of the proposed algorithm is theoretically proved to be $O(\sum_{C\in CC(G)}(|V(C)|+|E(C)|))$, where $CC(G)$ denotes the set of connected convex subgraphs in directed acyclic graph G, $|V(C)|$ and $|E(C)|$ denote the number of vertices and the number of edges in subgraph C respectively. Experimental results show that the proposed algorithm is more efficient than the state-of-the-art algorithms in terms of runtime.

Index Terms

Reconfigurable Computing, Extensible processors, custom instruction, custom instruction enumeration, directed acyclic graph, connected convex subgraphs.

I. Introduction and Related Works

Extensible processor is one of the most important implementations of reconfigurable computing. Extensible processors are becoming more and more popular as they can provide an excellent computational acceleration over regular embedded processors while ensuring the flexibility [1]. Extensible processors improve the performance of time critical applications by replacing the computation intensive codes with a set of custom instructions. These custom instructions are implemented on custom function units. The flexibility is ensured through the custom instruction set that can be tuned to run the other applications in the same domain.

Manual identification of custom instructions from a given application code is an intractable and time-consuming job. Thus, an automatic and efficient compilation flow for identifying custom instructions is necessary [2]. In general, the automatic compilation flow involves two computationally difficult problems: custom instruction enumeration and custom instruction selection. As a custom instruction can be graphically represented by a subgraph, the custom instruction enumeration problem is essentially a subgraph enumeration problem. In this paper, we focus on the subgraph enumeration problem.

Given an application code, a typical compilation flow transforms the source code to a control data-flow graph (CDFG). The subgraph enumeration step tries to enumerate all the subgraphs satisfying micro-architectural constraints from data-flow graphs (DFG) inside CDFG. In the custom instruction synthesis, the traditional approaches enumerate convex subgraphs (A subgraph *S* is convex if there is no directed path between vertices of *S* which contains a vertex not in *S*.) under the maximum number of input and output constraints imposed by the available register file ports [3]–[5]. Previous studies show that the subgraphs with more input/output operands than input/output constraints may bring higher speed-up. Therefore, a class of works focusing on enumerating maximal convex subgraphs have been proposed [6]–[8].

In recent years, some approaches aim at enumerating all possible connected convex subgraphs (cc-subgraphs) as candidate custom instructions [9], [10]. For every connected acyclic digraph G with n vertices, the maximal number of connected convex subgraphs (cc-subgraph) in G has been proved to be $2^n+n+1-d_n$, where $d_n = 2\cdot 2^{n/2}$ for every even n and $d_n = 3\cdot 2^{(n-1)/2}$ for every odd n [11]. Since the number of possible subgraphs is potentially exponential, an efficient and optimal algorithm for enumerating connected convex subgraphs from DFG is required.

In [9], Balister et al. present a top-down algorithm for enumerating cc-subgraphs. The algorithm produces the cc-subgraphs by adding the 'largest' out-neighbor or 'smallest' in-neighbor and the vertices required to maintain convexity. The algorithm makes $2\cdot|CC(G)|-n$ recursive calls. The time complexity of the algorithm is $O(n\cdot|CC(G)|)$. The authors of [10] describe a more efficient algorithm that requires only $|CC(G)|$ recursive calls to enumerate all the cc-subgraphs. The algorithm generates the cc-subgraphs in a bottom-up manner by adding topologically sorted neighbor vertex to smaller subgraphs. Experimental

results reveal that the algorithm can achieve on average 3.29 times speedup over the algorithm proposed in [9]. Although the algorithm makes fewer recursive calls and fewer computations inside each call, the time complexity of the algorithm is still $O(n \cdot |CC(G)|)$.

In [12], the authors have raised an open question:"Is there an $O(\sum_{C \in CC(G)} |V(C)|)$-time algorithm for generating all connected convex sets of a connected acyclic digraph G?". In this paper, we propose an optimal algorithm for enumerating all connected convex subgraphs from a DFG. This paper makes the following two contributions:

• The proposed algorithm is a provable optimal algorithm with time complexity of $O(\sum_{C \in CC(G)}(|V(C)| + |E(C)|))$. On the theoretical side, this work could be an answer to the open question raised in [12] .

• Experiments results reveal that the proposed algorithm outperforms the state-of-the-art algorithms. On the practical side, as custom instruction enumeration problem is one of the most computationally difficult problems involved in custom instruction synthesis, the proposed algorithm can be used to accelerate the process of custom instruction synthesis for extensible processors.

The rest of the paper is organized as follows. In Section II, the notations and terminologies are given. Section III presents the proposed algorithm. The proofs of the correctness and the complexity of the proposed algorithm are given in Section IV. Section V evaluates and compares the runtime of the proposed algorithms. Finally, conclusions are presented in Section VI.

II. Notation and Terminology

A data-flow graph (DFG) is a directed acyclic graph (DAG) $G = (V, E)$, where $V = \{v_1, ..., v_n\}$ is the set of vertices and $E = \{e_1, ..., e_m\} \subseteq V \times V$ is the set of edges. Each vertex in DFG represents a primitive operation and each edge corresponds to the data flow dependency between two primitive operations. For a DAG G, $V(G)$,$E(G)$ and $CC(G)$ denote the set of vertices, the set of edges and the set of connected convex subgraphs in G respectively.

Problem $\mathcal{P}$: Given a DFG $G = (V, E)$, enumerate all the subgraphs that satisfy the following constraints as custom instruction candidates.

• Convexity: S is convex;

• Connectivity: S is connected .

The convexity constraint is applied to ensure that the identified custom instruction can be executed atomically. For example, in Fig. 1, the subgraph $\{v_1, v_3, v_4\}$ is a convex subgraph, while the subgraph $\{v_1, v_4, v_7\}$ is not. A subgraph *S* is connected if there is an undirected path between any pair of vertices in *S*. In Fig. 1, the subgraph $\{v_1, v_3\}$ is a connected subgraph, while the subgraph $\{v_3, v_4, v_6\}$ is a disjoint subgraph. A candidate subgraph may contain one or more disjoint components. Enumerating disjoint subgraphs with multiple components may generate very large custom instructions that have very little chance of reuse in an application or across applications [13]. Similar to the work presented in [13], we only enumerate connected convex subgraphs as custom instruction candidates in this paper.

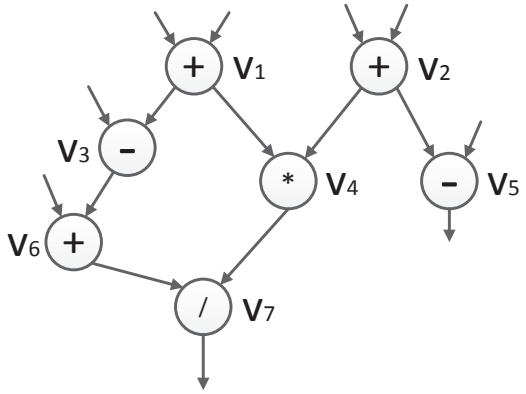

Fig. 1. The example data-flow graph

If uv is an edge of G, we say that u is a direct predecessor of v and v is a direct successor of u. The set of direct predecessors of v is denoted by $IPred(v)$ and the set of direct successors of u is denoted by $ISucc(u)$. For a set X of vertices of G, its direct predecessor (resp. direct successor) is $IPred(X) = \bigcup_{x \in X} IPred(x) \setminus X$ (resp. $ISucc(X) = \bigcup_{x \in X} ISucc(x) \setminus X$). For a set $X \subseteq V(G)$, the subgraph induced by X is denoted by $G[X]$.

A vertex u in a connected DAG is called an articulation point (or cut vertex) 'iff' removing u and the edges connecting it disconnects the graph. As an example, removing vertex v_2 and the edges connecting to v_2 from the data-flow graph in Fig. 1 results in a disjoint subgraph $\{v_1, v_3, v_4, v_5, v_6, v_7\}$. Hence, vertex v_2 is an articulation point. The articulation points in a connected DAG $G = (V, E)$ can be found in time $O(|V| + |E|)$ [14].

A vertex u in a DAG G is called a source (sink) vertex if it has no direct predecessors (direct successors) in G. In Fig. 1, vertex v_1 and vertex v_2 are source vertices. Vertex v_5 and vertex v_7 are sink vertices.

Algorithm 1: Connected Convex Subgraph Enumeration Algorithm

Input: A data-flow graph $G = (V, E)$, F: A set of must contained vertices
Output: CC A set of enumerated connected convex subgraphs

```
   /* enumerate all cc-subgraphs in a top-down manner                    */
1  Procedure TD(C, F)
2  CC = CC ∪ C;
3  for each vertex u ∈ C do
4      if visited[u] == false then
5          call DFS(u, visited, seq, low, parent, A);
6  Y = (V\F)\A;
7  for each vertex s ∈ Y with |IPred(s)| == 0 or |ISucc(s)| = 0 do
8      C' = C\{s};
9      call TD(C', F);
10     F = F ∪ {s};
   /* find all the cut vertices in C                                     */
11 Procedure DFS(u, visited, seq, low, parent, A)
12 visited[u] = true;
13 seq[u] = low[u] = ++counter;
14 children = 0;
15 for each adjacent vertex v of u do
16     if visited[v] == false then
17         children++;
18         parent[v] = u;
19         call DFS(v, visited, seq, low, parent, A);
20         low[u] = Math.min(low[u], low[v]);
21         if parent[u] == NIL&&children > 1 then
22             A = A ∪ {u};
23         if parent[u] ≠ NIL&&low[v] >= seq[u] then
24             A = A ∪ {u};
25     else if v ≠ parent[u] then
26         low[u] = Math.min(low[u], seq[v]);
```

Lemma 1. *Let G be a DAG, let X be a connected convex subgraph of G, and let $s \in G$ be a source or sink of X and s is not an articulation point of X. Then $X\backslash\{s\}$ is a connected convex subgraph of G.*

Proof. Firstly, as s is not an articulation point of X, removing s and the edges connecting s in X will not result in a disjoint subgraph. Therefore, $X\backslash\{s\}$ is a connected subgraph. Then, we prove that $X\backslash\{s\}$ is a convex subgraph of G. Suppose that $X\backslash\{s\}$ is non-convex , then there is a directed path v, u, w where $v, w \in X\backslash\{s\}$ and $u \notin X\backslash\{s\}$. Since X is convex, we can easily know that $u = s$. As u has at least one direct predecessor and one direct successor in X, this contradicts with the assumption that s is a source or sink of X. □

III. Proposed Algorithm

In this section, we propose a provable optimal algorithm for enumerating all the connected convex subgraphs from data-flow graph. We assume the original data-flow graph is a connected DAG, if not we can simply deal with each piece separately.

The pseudo code of the proposed algorithm is presented in Algorithm 1 (we denote it by $\mathcal{TD}$). The algorithm accepts a data-flow graph G as input and generates all the cc-subgraphs. The algorithm works in a top-down manner. By Lemma 1 we can see that if X is a cc-subgraph and s is a source or a sink of X and s is not a cut vertex, then $X\backslash\{s\}$ is a cc-subgraph. Based on this observation, we find all the cc-subgraphs by recursively deleting a vertex that is not a cut vertex and is a source or a sink of the current graph.

The algorithm outputs a cc-subgraph in each recursive call (line 2). Inside each recursive call, we first find all the cut vertices of the current graph by calling a depth-first search method (lines 3-5 and lines 11-26), then we consider all sources and all sinks of the current graph that are not in F and are not cut vertices. Before calling the depth-first search method, an array $visited[]$ is used to mark the vertex is visited or not, and the initial value of the elements in the array is false. A set A is used to record the cut vertices found in C. For each source or sink s that is not in F, we call $TD(C\backslash\{s\}, F)$ and then add s to F as a must contained vertex (lines 7-10). In such a way, for each sink or source $s \in X$ the algorithm generates all cc-subgraphs without s and all cc-subgraphs containing s.

In order to find all the cut vertices of a graph, we adopt a depth-first search (DFS) method proposed in [14]. The pseudo code of the depth-first search method is shown in lines 11-26. Although the depth-first search method is originally proposed to find the cut vertices from undirected graphs, in this work, we can use it directly to find the cut vertices from directed graphs by ignoring the direction of each edge. The method traverses each vertex in depth-first order. In the DFS tree, a vertex u is a

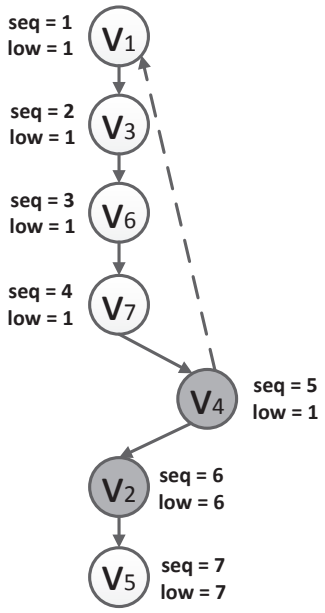

Fig. 2. Depth first traversal of the DFG presented in Fig. 1. v_2 and v_4 are cut vertices.

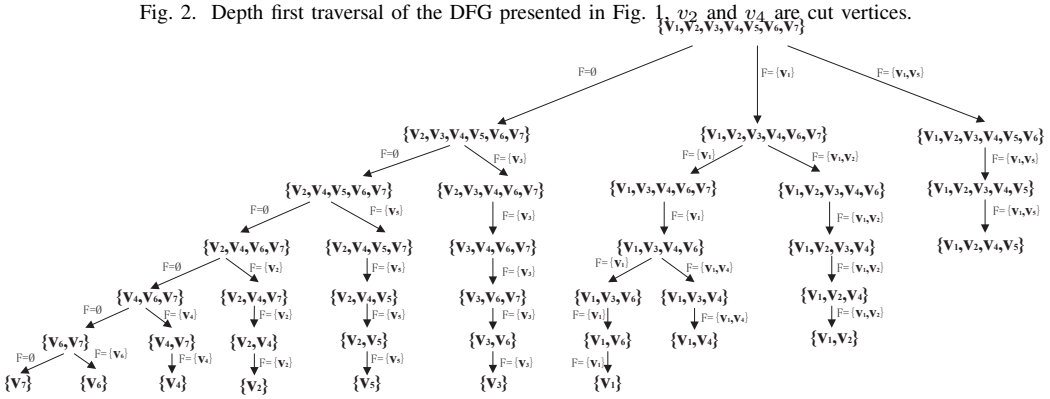

Fig. 3. Search tree of the proposed algorithm for enumerating cc-subgraphs from the DFG in Fig. 1.

parent of another vertex v, if v is discovered by u (line 18). If a vertex w is adjacent to u and w is already visited, then we say there is a back edge from u to w. A vertex u is a cut vertex if one of the following two conditions is true.

- u is a root of the DFS tree and it has at least two children;
- u is not a root of the DFS tree and it has a child v such that no vertex in subtree rooted with v has a back edge to one of the ancestors of u in the DFS tree (lines 23-24).

In the DFS traversal, for each vertex, we count the number of its children (line 17). If the visited vertex u is a root and has more than one child, then it is a cut vertex (lines 21-22). For the second condition, we maintain an array $seq[]$ to store visited sequence of vertices (line 13). For every vertex u, we need to find out the earliest visited vertex that can be reached from subtree rooted with u through a directed path. Thus, we use $low[u]$ to store the seq value of the earliest visited vertex that can be reached from subtree rooted with vertex u. For every vertex u, $low[u]$ is defined as follows.

$$low[u]=\left\{ \begin{array}{ll} min(low[u], low[v]), & if\ u\ is\ a\ parent\ of\ v \\ min(low[u], seq[v]), & if\ (u,v)\ is\ a\ back\ edge\ and \\ & v\ is\ not\ a\ parent\ of\ u \end{array} \right\} \quad (1)$$

As an example, Fig. 2 shows the DFS traversal of the DFG presented in Fig. 1. In Fig.2, each vertex is associated with two values (seq and low). v_2 and v_4 are two cut vertices as their seq value is less than the low value of their children.

Fig. 3 presents the search tree of the proposed algorithm on the data-flow graph presented in Fig. 1. In Fig. 3, we can see that the algorithm makes $CC(G) = 37$ recursive calls in total. Each recursive call outputs a connected convex subgraph.

IV. Correctness and complexity of the algorithm

In this section, we first present the proof of the correctness of the proposed algorithm $\mathcal{TD}$.

Lemma 2. *Algorithm $\mathcal{TD}$ correctly outputs all cc-subgraphs of G, where G is a connected directed acyclic graph.*

TABLE I
COMPARISON OF SUBGRAPH ENUMERATION ALGORITHMS (IN MILLISECOND)

Benchmark	NV	NE	NS	$\mathcal{A}$(CT)	$\mathcal{BS}$(CT)	$\mathcal{TD}$(CT)
DOTPRODUCT	20	19	4082	16	5	3
MP3	43	66	181,533,673	775,440	221,554	107,377
IDCT	29	38	139,121	517	171	89
MESA	37	65	7,554,499	31,308	8,027	3,713
FFT	32	37	30,597	118	36	19
ARF	25	37	84,401	283	104	57
IIR	40	56	23,195,414	102,331	28,725	12,231

Proof. Firstly we show that each set C output by $\mathcal{TD}$ is a connected convex subgraph. According to the algorithm, each subgraph is generated by deleting a vertex that is not a cut vertex and is a source or a sink of the current cc-subgraph. By Lemma 1, we can easily know that each enumerated subgraph is a cc-subgraph.

Secondly we prove that if $C \neq \emptyset$ is a cc-subgraph of G, then C is output exactly once by $\mathcal{TD}$. We first claim that for every connected convex subgraph H with $C \subset H \subseteq G$, there exists a source or sink $s \in H \setminus C$ of the digraph H and s is not a cut vertex. If the claim is true, by Lemma 1, we can obtain C by repeatedly deleting such sources or sinks.

To prove this claim, we will first show that for every connected convex subgraph H with $C \subset H \subseteq G$, there exists a source or sink $s \in H \setminus C$ of the digraph H. As $H \setminus C$ is an acyclic digraph, $H \setminus C$ has at least one source and one sink. If there exists no arc from a vertex of C to a vertex of $H \setminus C$ then any source of $H \setminus C$ is a source of H. If there is an arc from a vertex u to a vertex v of $H \setminus C$, considering the longest directed path $v_1 v_2 \cdots v_k$ in $H \setminus C$ leaving v, we can observe that v_k is a sink of $H \setminus C$ and there is no arc from v_k to any vertex of C otherwise C is a non-convex subgraph. Hence, v_k is a sink of H.

Then, we prove that for every connected convex subgraph H with $C \subset H \subseteq G$, there exists at least a source or sink $s \in H \setminus C$ of the digraph H and s is not a cut vertex. Suppose that every source and sink $s \in H \setminus C$ of the digraph H is a cut vertex. We may assume $S = \{s_1, s_2, ..., s_t\}$ is the set of such sources and sinks. Consider H and a cut vertex s_1, which can divide H into at least two subgraphs SG_1 and $SG_2 (SG_1 \cap SG_2 = s_1)$. We know that C is only included in one of these subgraphs. Let assume $C \subset SG_1$. If $SG_2 \cap \{s_2, ..., s_t\} = \emptyset$, since SG_2 has at least one sink and one source, then there exists at least one sink or source v_s of SG_2 and $v_s \neq s_1$, which has no arc connecting to other subgraphs. Thus, $v_s \in H \setminus C$ is a source or sink of H and $v_s \notin S$. We arrive at a contradiction. If SG_2 contains vertex(s) from $\{s_2, ..., s_t\}$, then we can choose one cut vertex s_i to divide SG_2 into several subgraphs. Then, we choose one subgraph $SG_{2i} \subset SG_2$ and $s_1 \notin SG_{2i}$. We repeat previous steps for SG_{2i}. Then, we will finally get a subgraph with at least a sink or source v'_s, which has no arc connecting to the other subgraphs generated by division. Therefore $v'_s \in H \setminus C$ is a source or sink of H and $v'_s \notin S$, a contradiction.

Next it suffices to show that there is a unique execution path outputting a cc-subgraph C. Suppose that there are at least two execution paths outputting C. Let further assume the subgraphs yielding by the two execution paths are C_1 and C_2 respectively, where $C_1 = C_2$. Consider the search tree of the algorithm, we track the reverse direction of the two execution paths, the two execution paths should be converged at a point. Starting from this point, one of the two execution paths enumerates the subgraph(s) excluding a vertex s, while the other execution path enumerates the subgraph(s) containing s. Therefore, $C_1 \neq C_2$, we arrive at a contradiction. □

Then, we prove the running time of algorithm $\mathcal{TD}$.

Lemma 3. *The running time of algorithm $\mathcal{TD}$ is $O(\sum_{C \in CC(G)}(|V(C)| + |E(C)|))$, where $|V(C)|$ and $|E(C)|$ denote the number of vertices and the number of edges in subgraph C respectively.*

Proof. According to the call tree of the algorithm, we can see that each recursive call outputs a cc-subgraph at line 2. So, the number of recursive calls is the same as the number of cc-subgraphs. Thus we have $CC(G)$ recurve calls. Then, we will show that the running time of $TD(C, F)$ without recursive calls is $O(|V(C)| + |E(C)|)$. Inside each recursive call, a depth-first traversal of graph C is made to find all the cut vertices in C at lines 3-5. Each vertex and each edge of C is visited once in the process of the depth-first traversal. It is easy to know that the depth-first traversal requires $O(|V(C)| + |E(C)|)$ time. The time to compute Y at line 6 is at most $O(|V(C)|)$. Then, all sources and sinks can be determined in $O(|V(C)|)$ time at line 7. Therefore, each call of $TD(C, F)$ without recursive calls runs in time $O(|V(C)| + |E(C)|)$ in total. Finally, the total running time of the algorithm is $O(\sum_{C \in CC(G)}(|V(C)| + |E(C)|))$. □

It should be noted that the open question raised in [12] is:"Is there an $O(\sum_{C \in CC(G)} |V(C)|$-time algorithm for generating all connected convex sets of a connected acyclic digraph G?". As we know, both the vertices and edges should be visited when a directed graph is traversed. Hence, we can claim that the proposed algorithm with running time of $O(\sum_{C \in CC(G)}(|V(C)| + |E(C)|))$ is an optimal algorithm for enumerating connected convex subgraphs in acyclic digraphs.

Lemma 4. *The space complexity of algorithm $\mathcal{TD}$ is $O(n^2)$, where n is the number of vertices in the DFG G.*

Proof. It is clear that the space complexity of algorithm $\mathcal{TD}$ is dominated by the space storing the graph. Therefore, the space complexity of algorithm $\mathcal{TD}$ is $O(n^2)$. □

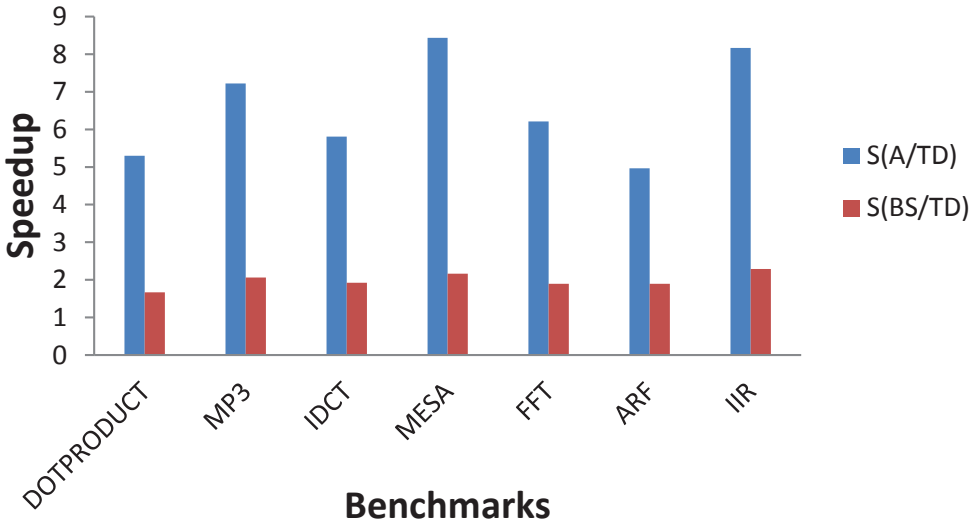

Fig. 4. Speedup achieved by the proposed algorithm over the state-of-the-art algorithms

V. Experimental Results

We compare the efficiency of our algorithm (denoted by $\mathcal{TD}$) with the state-of-the-art connected convex subgraph enumeration algorithm of [9] (denoted by $\mathcal{A}$) and the algorithm proposed in [10] (denoted by $\mathcal{BS}$). Seven benchmarks with rich arithmetic/logical operators are used in our experiments. These benchmarks are taken from Mibench [15] and Mediabench [16] and fall into domains including telecommunication, multimedia, etc. The three enumeration algorithms are implemented in Java under the compilation framework of GeCoS [17][1]. The input to the enumeration algorithms is a DFG representing a frequently executed basic block of each benchmark that has to be accelerated in practice using an extensible processor for example. We have run all the experiments on a PC with a i3-3240 processor running at 3.4 GHz with 4GB memory.

Table I presents the characteristics of the benchmarks and the runtime results of the three enumeration algorithms on the benchmarks. The first column of the table gives the name of each benchmark. The number of vertices and the number of edges in the DFG are denoted by NV and NE respectively. The number of enumerated connected convex subgraphs is presented in column NS. The runtime results of the three enumeration algorithms are given in column $\mathcal{A}$(CT), $\mathcal{BS}$(CT) and $\mathcal{TD}$(CT) respectively. The runtime unit is millisecond.

As expected, the three enumeration algorithms produce the same set of connected convex subgraphs for each benchmark. According to the runtime results, we can find that the $\mathcal{TD}$ algorithm outperforms the two state-of-the-art algorithms. The speedup achieved by the proposed algorithm over the $\mathcal{A}$ algorithm and $\mathcal{BS}$ algorithm on the seven benchmarks is given in Fig. 4. The proposed algorithm is on average 6.6 times faster than the $\mathcal{A}$ algorithm and 2.0 times faster than the $\mathcal{BS}$ algorithm.

It should be noted that the space complexities of the three algorithms are identical. The time complexities of $\mathcal{BS}$ and $\mathcal{A}$ are $O(n \cdot |CC(G)|)$. The time complexity of $\mathcal{TD}$ is $O(\sum_{C \in CC(G)}(|V(C)| + |E(C)|))$. The $\mathcal{BS}$ algorithm enumerates the cc-subgraphs in a bottom-up manner. The cc-subgraphs are produced by absorbing neighbor vertex to smaller subgraphs. However, the proposed algorithm enumerates the cc-subgraph in a top-down manner. Each cc-subgraph is formed by deleting a source or sink that is not an articulation point. The proposed algorithm may be accelerated using parallel techniques such as multithread technique or cloud computing. The $\mathcal{TD}$ algorithm can be easily adapted to the parallel framework proposed in [10]. Since the partitioning strategy proposed in [10] is independent of any specific algorithm running in each compute node, we can simply replace the sequential algorithm running in compute node by the $\mathcal{TD}$ algorithm.

VI. Conclusion

This paper proposes an optimal algorithm for enumerating connected convex subgraphs from a given application graph in a top-down manner. The proposed algorithm is proved to be an $O(\sum_{C \in CC(G)}(|V(C)| + |E(C)|))$-time algorithm, which is the first optimal algorithm for enumerating connected convex subgraphs in acyclic digraphs in the sense of time complexity.

References

[1] Ammendola R, Biagioni A, Frezza O, et al. ASIP acceleration for virtual-to-physical address translation on RDMA-enabled FPGA-based network interfaces. Future Generation Computer Systems,2015, 53:109-118.

[2] G. Zacharopoulos, L. Ferretti, E. Giaquinta, G. Ansaloni and L. Pozzi. RegionSeeker: Automatically Identifying and Selecting Accelerators from Application Source Code. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2019, 38(4):741-754.

[3] L. Pozzi, K. Atasu and P. Ienne. Exact and approximate algorithms for the extension of embedded processor instruction sets. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,2006, 25(7):1209-1229.

[1]https://gitlab.inria.fr/gecos

[4] C. Xiao, and E. Casseau. Exact custom instruction enumeration for extensible processors. Integration, the VLSI Journal,2012, 45(3):263-270.
[5] J. Reddington and K. Atasu. Complexity of Computing Convex Subgraphs in Custom Instruction Synthesis. IEEE Transactions on Very Large Scale Integration (VLSI) Systems,2012,20(12):2337-2341.
[6] T. Li, Z. Sun, W. Jigang, and X. Lu. Fast enumeration of maximal valid subgraphs for custom-instruction identification. CASES 2009, pp.29-36.
[7] K. Atasu, W.Luk,O.Mencer, C.Ozturan, G.Dundar. FISH: Fast Instruction SyntHesis for Custom Processors. IEEE Transactions on Very Large Scale Integration (VLSI) Systems,2012,20(1):52-65.
[8] E. Giaquinta, A. Mishra and L. Pozzi. Maximum Convex Subgraphs Under I/O Constraint for Automatic Identification of Custom Instructions. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2015, 34(3):483-494.
[9] P. Balister, S. Gerke, G. Gutin, A. Jojnstone, J. Reddington, E. Scott, A. Soleimanfallah and A. Yeo. Algorithms for generating convex sets in acyclic digraphs. Journal of Discrete Algorithms,2009,7(4):509-518.
[10] S. Wang, C. Xiao and W. Liu. A Faster Algorithm for Enumerating Connected Convex Subgraphs in Acyclic Digraphs. IEEE Embedded Systems Letters,2017, 9(1):9-12.
[11] G. Gutin, and A. Yeo. On the number of connected convex subgraphs of a connected acyclic digraph. Journal of Discrete Algorithms,2009, 157(7):1660-1662.
[12] P. Balister, S. Gerke and G. Gutin. Convex Sets in Acyclic Digraphs. Order-a Journal on the Theory of Ordered Sets & Its Applications, 2009, 26(1):95-100.
[13] P. Yu and T. Mitra. Scalable custom instructions identification for instruction-set extensible processors. ACM CASES, 2004, pp. 69-78.
[14] R. Tarjan. Depth-first search and linear graph algorithms. SIAM journal on computing,1972, 1(2):146-160.
[15] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, R.B. Brown. Mibench: a free, commercially representative embedded benchmark suite, in: Annual Workshop on Workload Characterization, 2001, pp. 3-14.
[16] C. Lee, M. Potkonjak, W.H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems, in: MICRO, 1997, pp. 330-335.
[17] A. Floch, T. Yuki, A.E. Moussawi, A. Morvan, et al. GeCoS: A framework for prototyping custom hardware design flows, IEEE SCAM, 2013, pp. 100-105