Graph-Based Approaches for Application Mapping onto CGRAs

An Overview

Mirjana Stojilović, Dejan Vujičić
Mihailo Pupin Institute
University of Belgrade
Belgrade, Serbia
mirjana.stojilovic, dejan.vujicic@pupin.rs

Lazar Saranovac
University of Belgrade
School of Electrical Engineering
Belgrade, Serbia
laza@el.etf.rs

Abstract—**Coarse grain reconfigurable arrays (CGRAs) offer more computational power for word-level computing than field programmable gate arrays (FPGAs). This is especially important for highly computationally intensive multimedia applications. However, CGRAs have not yet been fully adopted due to the existence of many different architectures and lack of universal scheduling approach. In recent years, graph based approaches to application mapping onto CGRAs have attracted the attention of researchers. This paper provides an overview of the main contributions in this field so far and suggests new approaches for future development of graph-based mapping solutions.**

Keywords-graph layout; coarse grain reconfigurable arrays; compiler; application mapping; routing network.

I. INTRODUCTION

The increasing functionality and need for adaptability in the multimedia handheld devices imposes the use of highly flexible programmable architectures. Field programmable gate arrays (FPGAs) are appropriate for applications involving complex logic and bit manipulations, but multimedia applications are better suited for arrays with coarser granularity — coarse grain reconfigurable arrays (CGRAs). Another benefit of the coarser granularity is that the size of configuration bitstream for programming CGRAs is much smaller than the size of the configuration bitstream for FPGAs. Therefore, reconfiguration of CGRAs can be done faster, even during runtime. CGRAs typically consist of a 2-D reconfigurable array of functional units (FUs) and a host processor. The computationally intensive kernels of the applications are mapped to the array, while the remaining code is executed by the processor. Although various CGRAs have been proposed [1], not many of them have been adopted, mainly due to lack of mapping tools. Hence, good compilers are of a significant importance for providing efficient application mapping onto CGRAs and enabling a better use of the available computational power.

Compilation for CGRAs has traditionally been focused on two issues [2]-[10]: (i) placing operations (arithmetic, logic, multiplication, and load/store) of a loop kernel onto the FUs, and (ii) assuring the flow of data (routing) among operations using the existing interconnection resources. Then the loop is transformed into a pipeline on a CGRA, completing one iteration every cycle or every *II* cycles, where *II* is the initiation interval of the pipeline [5]).

A common approach to present compute-intensive application kernels is in the form of a dataflow graph (DFG), in which nodes represent operations and edges the flow of data between them. Therefore, mapping an application to a regular CGRA structure can be thought of as finding a suitable transformation between a DFG and an array of interconnected FUs. In Fig. 1 we show an example DFG, and a possible mapping of that DFG to a 4×4 CGRA. It is assumed that functional units are composed of an arithmetic and logic unit (ALU), multiplexers at the ALU inputs, and register files (RFs), and that they can communicate only with immediate neighboring FUs. Clearly, there are many possible ways this mapping can be done.

In recent years, some promising research results in the domain of application mapping, which relied on using a graph-based approach, were published. It appears that using graph-based approaches is a promising and not yet fully explored research direction. In this paper we give an overview of the main contributions in this area with two motivations: first motivation is to compare the existing approaches, and the second is to get new ideas for the future work on the very topic.

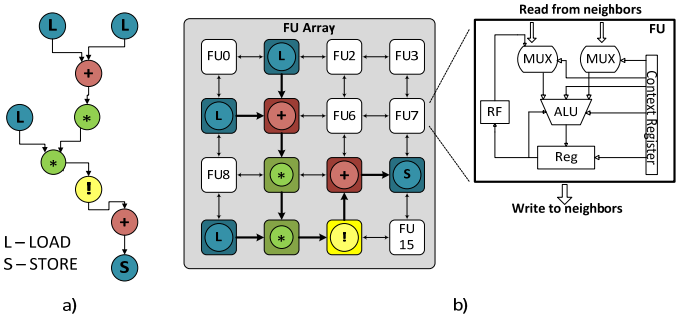

Figure 1. a) An example of the application kernel. b) One possible mapping of the kernel in a) onto a 4×4 CGRA. It is assumed that FUs are identical, composed of an ALU, multiplexors at the ALU inputs, and register files. Additionally, only connections between four neighboring FUs are provided.

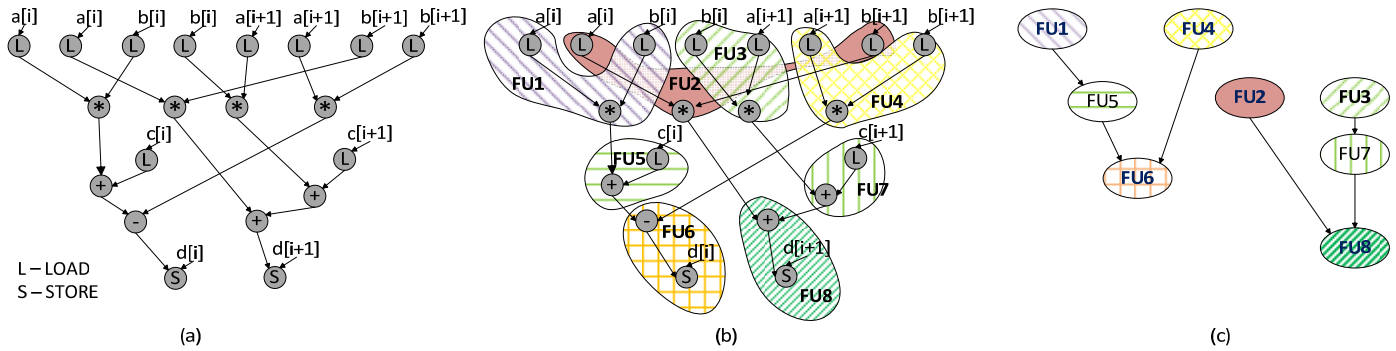

Figure 2. a) The kernel tree of the *complex update* application from DSPStone benchmarks [23]. b) Kernel tree after covering. c) Configuration tree.

The paper is organized in several sections, covering from the oldest to the newest selected work on the application mapping to CGRAs. In Section II we discuss the spatial mapping algorithm published by Ahn et al [2], which uses Sugiyama method [12] for drawing layered graphs to find the node placement. In Section III we present a mapping algorithm by Yoon et al. [3] based on *Split & Push* algorithm used in the graph drawing area [13]. Then, we give an overview of the edge-centric approach by Park et al. [4], in which mapping is guided by the *affinity* cost function, directly related to the proximity of operations in the DFG. Finally, the most recently published *graph minor approach* by Chen et al. [14] is discussed, followed by conclusions.

II. Spatial Mapping Algorithm for HCGRAs

Ahn et al. [2] analyzed the problem of automatically mapping applications onto Multiple Instruction Multiple Data (MIMD) heterogeneous CGRAs (HCGRAs). In HCGRAs each functional unit can be configured separately, as shown in Fig. 1b. The overall performance depends mainly on the application mapping, which should exploit the parallelism embedded in an application and the computational resources of the hardware simultaneously. In the earlier works [7]–[9], [11], several attempts of mapping applications to CGRA were made, but with some differences and limitations. Kim et al. performed manual mapping [11], Mei et al. provided no support for sharing of common resources among FUs [8], Lee et al. assumed homogeneous FUs [9], while Venkatarani et al. used single instruction multiple data (SIMD) CGRA [7].

Application mapping can be roughly classified into two categories: *temporal* and *spatial* mapping. In the temporal mapping, necessary configurations are all stored in the configuration cache and the configuration of each FU is dynamically changed with time. In spatial mapping, each FU has a fixed configuration, and the data to be processed are routed through FUs. The temporal mapping may reduce the number of FUs required for mapping in comparison with the spatial mapping. In spatial mapping, the mapping is limited by the topology and size of the reconfigurable array, but it has no configuration overhead and thus reduces the configuration storage. Therefore, the spatial mapping strategy is in some cases more effective for embedded applications. Ahn et al. [2] proposed a novel algorithm for spatial mapping that uses methods for drawing hierarchical graphs.

They first analyzed the application code to detect performance-critical parts, usually loop kernels. These kernels are then represented in a tree form, called the kernel tree. In this tree, each node is an atomic operation, such as addition or multiplication, while an edge represents data dependence between operators. An example of the application code extracted from *complex update* application from DSPStone benchmarks [24] is as follows:

```
for (i = 0; i < N; i += 2;) {
   temp1 = c[i] + a[i] * b[i];
   d[i] = temp1 – a[i+1] * b[i+1];
   temp2 = c[i+1] + a[i+1] * b[i];
   d[i+1] = temp2 + a[i] * b[i+1];
}
```

Fig. 2a shows the kernel tree extracted from this code.

Ahn et al. [2] assumed that FUs are composed of one ALU, preceded by multiplexors at its every input and followed by a shifter and pipelining register. By changing the configuration of the FU, various combinations of operations can be executed on it, e.g., loading operators on both inputs followed by multiplication, performing ALU operation followed by shifting, or loading only one operator and shifting. Several kernel operations can be mapped on one FU, if the corresponding configuration exists. Using the set of possible configurations, Ahn et al. transform the kernel tree to a configuration tree, in which each node represents a configuration for each FU, possibly covering, and thus executing, more than one operation of the kernel tree.

They divided the algorithm for spatial mapping into three phases: covering, partitioning, and layout. The covering problem is essentially analogous to instruction selection problem [15]. To solve it, they implemented a compiler that takes as input the kernel code, computes the covers for nodes in the kernel tree as shown in Fig. 2b, and produces as output the configuration tree in Fig. 2c. Several nodes in the kernel tree can be replaced by one node in the configuration tree if there is a configuration of FU in the CGRA that supports their execution on a single FU. In the partitioning phase, they partition the nodes of the configuration tree into different clusters, each scheduled later to each column of the PE array. When partitioning, CGRA architectural constraints, such as bus sharing, multiplier sharing and array size, are respected. Later, in the laying-out phase, these partitions are input to their integer linear programming (ILP) solver for finding vertical

assignment of nodes in the CGRA. The algorithm always tries to assign two partitions with heavy data traffic as close as possible, which usually leads to minimized number of FUs that are used as route-through only. After vertical assignment, they use the Sugiyama method [12] for drawing layered graphs to find the node positions that (i) minimize edge crossings and (ii) keep adjacent the node pairs that exchange data.

A. *Performance Evaluation*

In the experimental evaluation, a 6×6 RSPA (resource sharing and pipelining architecture) was used. It had 36 FUs and pair-wise connections and cross connections inserted between not-neighboring FUs. Two multipliers were attached to each row as shared computation resources. Ahn et al. used as input a set of representative kernels from embedded applications. The effectiveness of their spatial mapping algorithm was compared with hand-optimized spatial mappings. The results showed that their algorithm does not always produce optimal solutions, although in many cases its performance is comparable to those obtained with hand optimizations.

III. SPLIT & PUSH KERNEL MAPPING ALGORITHM

Owing to the simplistic model of the CGRA architecture in the compilers, they either (i) cannot guarantee to find a feasible application mapping, even though it might exist, or (ii) use too many FUs in the mapping solution. This motivated Yoon et al. [3] to formulate the application mapping problem onto CGRA considering routing FUs, shared resource constraints, and complex FU interconnection schemes. The focus of their work was in solving the problem of mapping a kernel of a given loop to a CGRA, while simultaneously minimizing the number of resources required. They proposed a graph-drawing based approach called split-push kernel mapping (SPKM).

The SPKM is a heuristic approach for solving the problem of mapping a kernel onto a large-scale CGRA, which cannot be achieved with ILP due to the exponential time complexity. It is based on the *Split & Push* algorithm used in the graph-drawing area [13]. In Fig. 3 we show an example of mapping a four-operation kernel graph onto a 2×2 CGRA. The algorithm starts with all nodes of a kernel graph located at the same coordinate (Fig. 4a). Then, it uses *cuts* to split vertices into two distinct groups. A cut is a plane orthogonal to one of the axes (shown by dotted lines). After the node separation, the vertices in one of the two groups are pushed to a new coordinate. Fig. 3b shows the result of Split & Push along the horizontal dotted line. This Split & Push step is repeated until every vertex has distinct coordinate, as shown in Fig. 3c.

The most important step in the split & push approach is finding a suitable cut. Here is another example of split & push to the same kernel in Fig. 3a. The first cut separates the node number 3 from other nodes. Consequently, the final graph will require two more FUs than the graph in Fig. 3c. This kind of separation produces a *fork*—adjacent edges cut by a split. In general, forks can be avoided by finding a matching-cut: a set of edges which have no common node and whose removal makes the graph disconnected. However, this is an NP complete problem [16].

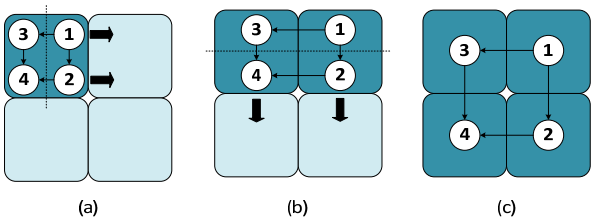

Figure 3. Split & Push approach. Vertices are iteratively separated into two groups and pushed to new coordinates.

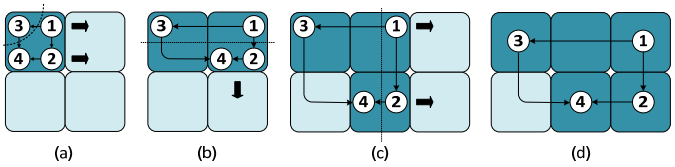

Figure 4. Formation of fork.

To minimize the number of utilized rows in the mapping, Yoon et al. proposed a three-stage heuristic. The first stage is a column-wise scattering, in which vertices are distributed to the minimum number of utilized rows in the same column, considering the minimum number of forks and shared operations like multiplication, load, and store. On each edge of each fork generated in this stage, they need to insert routing vertices to route data via indirectly connected FUs. The second stage is routing FU insertion, in which routing FUs are generated and connected with existing vertices. The third stage is a row-wise scattering, in which they try to avoid diagonal edges and edge crossings by placing the nodes that have connections between different rows in the same column.

A. *Performance Evaluation*

The performance of the SPKM algorithm was tested on RSPA (resource sharing and pipelining architecture) CGRA [2], mentioned in the previous section. This CGRA was modified to have a 6×4 structure for fair comparison with the approach in the reference work [2]. They compared the SPKM with the approach in [2] on a set of kernel graphs from benchmarks such as Livermore loops, MultiMedia and DSPStone, and on a set of randomly generated kernel graphs. The results showed that SPKM can on average map 4.5 times more applications than the approach in [2]. SPKM is also able to generate mappings requiring smaller number of rows than the approach in [2] in 62% of the applications. Additionally, SPKM has only 5% overhead in mapping time, while both approaches are significantly faster than ILP.

IV. EDGE-CENTRIC MODULO SCHEDULING

Traditional schedulers [17], [18] address the scheduling task in a node-centric manner by focusing on assigning DFG nodes to FUs. Park et al. [4] argue that selecting intelligent paths from producing to consuming FUs that do not block other operand paths is crucial for achieving higher throughput schedules. Hence, they propose an edge-centric modification of *modulo scheduling*, in which the scheduler focuses primarily on routing, while placement is a by-product of the routing process. Modulo scheduling is a software pipelining technique that exposes parallelism by overlapping successive loop

iterations to find a valid schedule that will minimize the interval between successive iterations (initiation interval, or II) [5]. It is discussed in more details in Section V-A.

In an edge-centric approach, the scheduler does not place operations up front. Instead, it selects an edge from the already placed producers or consumers of the operator and attempts routing that edge. To do that, the router searches for an empty slot capable of executing the target operation. In a node-centric approach, the router would instead route towards a placed operation. In the edge-centric approach, once a compatible slot is found, the target operation is placed in it and the scheduler continues routing DFG edges to remaining nodes. The router is guided by two criteria:

- minimizing the number of used routing resources,
- avoiding routing failure.

Minimization of the number of routing resources used is achieved by assigning a statically determined fixed cost to the routing resources, forcing the router to look for a path minimizing the total cost. Park et al. [4] propose using *affinity cost* [18]. The affinity cost of a pair of operations reflects their proximity in the DFG. Hence, the affinity cost forces the router to place operators near their producers and consumers whenever possible, and thus reduce the number of used routing resources. To avoid routing failure, they associate occupancy probability to each scheduling slot, to discover which resources are likely to be used by other edges in the future. These probabilities are calculated for (i) expensive and (ii) placed operations. Expensive operations are those that only a subset of FUs can execute, such as memory access operations and multiplication. The occupancy probability for these FUs equals the number of unscheduled expensive operations divided by the number of remaining compatible FUs. For operations already placed, the probabilities are determined based on the number of possible options for routing values to producers/consumers.

In addition to the routing cost, they introduce priorities among edges, based on their importance for the scheduler. The most important are recurrence edges, followed by simple edges (outgoing edges with only one consumer) and high-fanout edges.

A. *System Flow*

The system flow of the edge-centric modulo scheduling is shown in Fig. 5. There are three preprocessing steps, focused on analyzing the input DFG, simplifying its structure, and determining the edge priorities. Then, the scheduling begins. For each target operation, the scheduler determines whether there are any placed producers or consumers. If not, the target operation is placed in a scheduling slot with minimum cost. For an operation that has producers or consumers placed, the scheduler selects an edge to route. Then, a routing cost for the scheduled edge is calculated for each available slot, based on affinity and probability costs. Then, the router looks for a path from the source to the target operation. If the target is already placed, the route direction is toward the slot that contains the target operation. Otherwise, it will find a slot that can execute the target operation. When a slot is found, the scheduler checks if there are other edges connected to the target that need to be placed. When all edges are successfully routed, the scheduler moves to the next operation in priority order. In the case of scheduling failure, the scheduler increases the initiation interval II and repeats scheduling.

B. *Performance Evaluation*

To evaluate the performance of the edge-centric approach, loops from typical media applications were taken (H.264 decoder, 3D graphics, AAC decoder, MP3 decoder). These loops were mapped onto various CGRA configurations. The results showed that the edge-centric approach improved performance by 25% over the traditional modulo scheduling and achieved 85-98% of the performance compared to the state-of-the-art simulated technique DRESC [8] with compilation time reduced by 18×. However, the drawback of this approach is that its performance strongly depends on the characteristics of DFG structures and the underlying CGRA architectures.

V. GRAPH MINOR APPROACH

Chen et al. [14] noticed that none of the previous approaches attempted to share routes corresponding to different graph edges having the same source node. Additionally, most of the existing techniques did not even model explicit routing through register files. Most approaches implicitly assumed the availability of a sufficient number of registers and interconnections between them as well as functional units, even though the register files consume significant amount of area and substantially impact CGRA performance [17]. Sutter et al. perform explicit routing through register files [19], but their register rotation approach uses a post-pass register allocation and assumes sufficient capacity of the register file during scheduling.

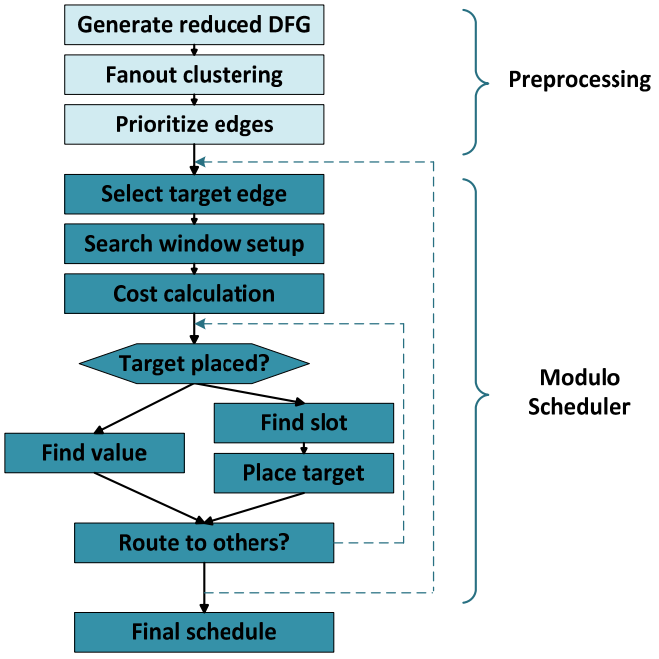

Figure 5. System flow for edge-centric modulo scheduling.

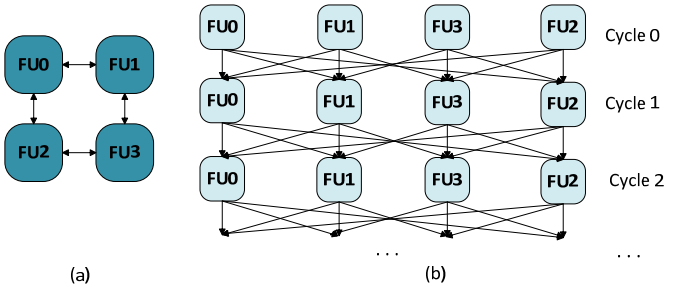

Figure 6. a) A simple 4×4 CGRA. b) The corresponding modulo routing resource graph (MRRG).

Chen et al. [14] introduced another approach that integrates register allocation with scheduling through explicit modeling of the register files and their connectivity with the functional units. Essentially, they transformed a CGRA mapping problem with route sharing into a *graph minor problem* and proposed a framework based on graph mapping for solving this problem. Experimental validation of their approach has shown that this approach with route sharing in both registers and routing functional units often leads to better schedule. Additionally, it may reduce the register pressure compared to approaches that adapt VLIW compilation technologies, such as register spilling [20] and register rotation [19], which are out of the scope of this paper.

A. *Modulo Routing Resource Graph*

To model the register files and their connectivity, Chen et al. [14] introduced some modifications to modulo routing resource graph (MRRG). Initially, MRRG was proposed by Mei et al. [17] as a resource-management graph that captures interconnections among FUs and register files. Park et al. [18] introduced a slightly modified version of MRRG, which is used by Chen et al. in [13]. According to them, MRRG is a directed graph whose nodes represent an FU or a register file, while edges represent the connectivity between nodes in a time-space view. For example, Fig. 6b shows the MRRG corresponding to the CGRA shown in Fig. 6a. The resources of the CGRA are replicated every cycle along the time axis and the edges always point forward in time. During modulo scheduling, when a node $v = (n,t)$ (where n refers to an FU or an RF and t is the cycle) in the MRRG becomes occupied, then all the nodes $v' = (n, t + k \times II)$, where $k > 0$ and II corresponds to the initiation interval, are also marked occupied. The goal of CGRA modulo scheduler is to generate II different configurations for the CGRA, where each configuration corresponds to a particular cycle.

B. *Restricted Graph Minor*

Chen et al. use the *graph minor* [21] based formulation of the application mapping problem on CGRA with route sharing. An undirected graph G' is called a minor of the graph G if G' is isomorphic to a graph that can be obtained by edge contractions on a subgraph of G . An edge contraction is an operation that removes an edge from a graph while simultaneously merging together the two vertices it used to connect. Since the definition of the graph minor is restricted to undirected graphs, they extended it by defining the edge contraction operation for directed graphs.

Finally, the algorithm for CGRA with route sharing by Chen et al. is as follows. First they compute the minimum possible II , similarly to the traditional modulo resource scheduling approaches. Then, they create the modulo routing resource graph G corresponding to the CGRA architecture and the minimum II . Further on, they look for a subgraph G' in G , such that the application dataflow graph G'' is a restricted minor of G'. If such graph exists and can be found, then the DFG G'' can be mapped to the CGRA with the initiation interval II . Otherwise, they increment II by one, create the MRRG corresponding to the new value of II , and look again for a subgraph in the new MRRG such that the DFG can be a minor. They repeat this process until a MRRG with sufficiently large II is generated and that the DFG can satisfy the graph minor test. This DFG is then called a restricted minor of the MRRG.

C. *Performance Evaluation*

To evaluate the performance of their algorithm, they used a set of kernels from standard benchmarking suites and three different register file configurations: one with no register files, one with local shared register files, and one with a central shared register file. The target CGRA architecture was a 4×4 array with possibly heterogeneous units that can be found in ADRES [22], MorphoSys, and other known CGRAs. They measured the achieved II for different CGRA configurations (different number of memory units, different register file configurations). The results have shown that adding registers may not necessarily improve II , contrary to the conclusions published by Kwok et al. [23], who recommended a global register file with a large number of registers. The reason was that the algorithm that performs register allocation as a post-processing step may end up with a schedule using a large number of registers, while sharing routes helps reducing register file pressure and thus achieves a valid schedule using smaller number of registers.

Compared to DRESC, their algorithm yields improvement in the compilation time more than an order of magnitude, along with increased average resource utilization (62% compared to 54% for DRESC).

VI. FUTURE WORK

In all previous works, authors have assumed sparse connectivity among FUs in the array. Most usually, FUs were connected only to the neighboring four FUs (left, right, top, bottom) or neighboring eight FUs (diagonal FUs included). However, the current state of development of semiconductor technology is allowing for integration of hundreds of operators and complex interconnection networks on a single die. Thus, even highly flexible routing networks, such as those found in FPGAs [25], could be implemented in CGRAs. The only difference would be the usage of a 16b/32b buses to route data, instead of a individual wires. One possibility would be to introduce FPGA-like routing channels between consecutive rows/columns of the array. Moreover, these routing channels

could use different lengths of busses, spanning only the part of the array row/column.

If the array has a large number of FUs, an application DFG could be mapped so that the edges and node positions resemble as much as possible to those suggested by state-of-the-art graph-visualizing tools, such as Gephi, H3Viewer, Otter, Ineato, Dotty, and others. We believe that these tools and CGRA mapping approaches share important common goals — reducing the edge length between neighboring nodes and reducing the number of edge crossings, and should thus be combined into a new graph-based approach for application mapping onto CGRAs.

VII. CONCLUSION

This paper gives an overview of the recently published research papers about improving application mapping onto CGRAs by using a graph-based approach. We discuss the spatial mapping algorithm [2] that uses Sugiyama method [12] for drawing layered graphs, a mapping algorithm based on Split & Push algorithm for graph drawing [3], edge-centric mapping approach [4] guided by the proximity of nodes in DFG, and the graph minor approach [14]. Some of these approaches offer significantly better performance results compared to traditional approaches. Hence, the proper analysis of DFG topology seems to be of importance for successful node mapping and edge routing in CGRAs. Additionally, we provide some ideas for further development of graph-based mapping solutions.

ACKNOWLEDGMENT

This work has been supported in part by the Ministry of Education, Science and Technological Development of the Republic of Serbia under grant TR 32043.

REFERENCES

[1] Y. Kim, "Reconfigurable multi-array architecture for low-power and high-speed embedded systems," Journal of Semiconductor Technology and Science, vol. 11, no. 3, pp. 207–220, 2011.

[2] M. Ahn, J. Yoon, Y. Paek, Y. Kim, M. Kiemb, and K. Choi, "A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures," in Proceedings of the Conference on Design, Automation, and Test in Europe, pp. 363-368, 2006.

[3] J. Yoon, A. Shrivastava, S. Park, M. Ahn, R. Jeyapaul, and Y. Paek, "SPKM: A novel graph drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures," Asia and South Pacific Design Automation Conference, pp. 776-782, 2008.

[4] H. Park, K. Fan, S. Mahkle, T. Oh, H. Kim, and H. Kim, "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures," in Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, pp. 166-176, 2008.

[5] B. R. Rau, "Iterative modulo scheduling: an algorithm for software pipelining loops," in Proceedings of the 27th Annual International Symposium on Microarchitecture, pp. 63–74, 1994.

[6] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe, "Space-time scheduling of instruction-level parallelism on a raw machine," in Proceedings of the Eigth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 46-57, 1998.

[7] G. Venkataramani, W. Najjar, F. Kurdahi, N. Bagherzadeh, and W. Bohm, "A compiler framework for mapping applications to a coarse-grained reconfigurable computer architecture," in Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, pp. 116-125, 2001.

[8] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "DRESC: A retargetable compiler for coarse-grained reconfigurable architectures," in Proceedings of IEEE International Conference on Field Programmable Technology, pp. 166-173, 2002.

[9] J. Lee, K. Choi, and N. Dutt, "An algorithm for mapping loops onto coarse-grained reconfigurable architectures," in Proceedings of ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems, pp. 183-188, 2003.

[10] J. Lee, K. Choi, and N. Dutt, "Compilation approach for coarse-grained reconfigurable architectures," IEEE Design & Test of Computers, vol. 20, issue 1, pp. 26–33, 2003.

[11] Y. Kim, M. Kiemb, C. Park, J. Jung, and K. Choi, "Resource sharing and pipelining in coarse grained reconfigurable architecture for domain-specific optimization," in Proceedings of the Conference on Design, Automation, and Test in Europe, vol. 1, pp. 12-17, 2005.

[12] K. Sugiyama, S. Tagawa, and M. Toda, "Methods for visual understanding of hierarchical system structures," IEEE Transactions on Systems, Man and Cybernetics, vol. 11, issue 2, pp. 109–125, 1981.

[13] G. D. Battista, M. Patrignani, and F. Vargiu, "A split&push approach to 3D orthogonal drawing," in Proceedings of the 6th International Symposium on Graph Drawing, pp. 87-101, 1998.

[14] L. Chen, T. Mitra, "Graph minor approach for application mapping on CGRAs," IEEE International Conference on Field Programmable Technology (FPT), pp. 285–292, 2012.

[15] K. Atasu, L. Pozzi, and P. Ienne, "Automatic application-specific instruction-set extensions under microarchitectural constraints," in Proceedings of the Conference on Design, Automation, and Test in Europe, pp. 256–261, 2003.

[16] M. Patrignani and M. Pizzonia, "The complexity of the matching-cut problem," in Proceedings of the 27th International Workshop on Graph-Theoretic Concepts in Computer Science, pp. 284–295, 2001.

[17] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," in Proceedings of the Conference on Design, Automation, and Test in Europe, vol. 1, pp. 296–301, 2003.

[18] H. Park, K. Fan, M. Kudlur, and S. Mahlke, "Modulo graph embedding: mapping applications onto coarse-grained reconfigurable architectures," in Proceedings of the 2006 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, pp. 136–146, 2006.

[19] B. De Sutter, P. Coene, T. Vander, and B. Mei, "Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays," in Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '08), pp. 151-160, 2008.

[20] G. Dimitroulakos, M. Galanis, and C. Goutis, "Exploring the design space of an optimized compiler approach for mesh-like coarse-grained reconfigurable architectures," in Proceedings of the 20th International Conference on Parallel and Distributed Processing (IPDPS '06), pp. 113-123, 2006.

[21] N. Robertson and P. D. Seymour, "Graph minors," Journal of Combinatorial Theory, series B, vol. 77, issue 1, pp. 162-210, 1999.

[22] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," Field-Programmable Logic and Applications, vol. 2778, pp. 61-70, 2003.

[23] Z. Kwok and S.J.E. Wilton, "Register file architecture optimization in a coarse-grained reconfigurable architecture," in Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 35-44, 2005.

[24] V. Živojnović, J. Martinez, C. Schläger, and H. Meyr, "DSPSTONE: A DSP-oriented benchmarking methodology," in Proceedings of the International Conference on Signal Processing and Technology, 1994.

[25] A. Ye and J. Rose, "Using bus-based connections to improve field programmable gate-array density for implementing datapath circuits," IEEE Transactions on VLSI Systems, vol. 14, no. 5, pp. 462–473, 2006.