A Graph Drawing Based Spatial Mapping Algorithm for Coarse-Grained Reconfigurable Architectures

Jonghee W. Yoon, Aviral Shrivastava, Sanghyun Park, Minwook Ahn, and Yunheung Paek

***Abstract*—Recently coarse-grained reconfigurable architectures (CGRAs) have drawn increasing attention due to their efficiency and flexibility. While many CGRAs have demonstrated impressive performance improvements, the effectiveness of CGRA platforms ultimately hinges on the compiler. Existing CGRA compilers do not model the details of the CGRA, and thus they are i) unable to map applications, even though a mapping exists, and ii) using too many processing elements (PEs) to map an application. In this paper, we model several CGRA details, e.g., irregular CGRA topologies, shared resources and routing PEs in our compiler and develop a graph drawing based approach, Split-Push Kernel Mapping (SPKM), for mapping applications onto CGRAs. On randomly generated graphs our technique can map on average 4.5× more applications than the previous approach, while generating mappings which have better qualities in terms of utilized CGRA resources. Utilizing fewer resources is directly translated into increased opportunities for novel power and performance optimization techniques. Our technique shows less power consumption in 71 cases and shorter execution cycles in 66 cases out of 100 synthetic applications, with minimum mapping time overhead. We observe similar results on a suite of benchmarks collected from Livermore loops, Mediabench, Multimedia, Wavelet and DSPStone benchmarks. SPKM is not a customized algorithm only for a specific CGRA template, and it is demonstrated by exploring various PE interconnection topologies and shared resource configurations with SPKM.**

***Index Terms*—Compiler, kernel mapping, reconfigurable architecture.**

I. Introduction

COARSE-GRAINED reconfigurable architectures (CGRAs) have emerged as a promising reconfigurable platform, by providing operation-level programmability, word-level datapaths, and area-efficient routing switches which is also very powerful. CGRAs are essentially a set of processing elements (PEs), such as arithmetic logic units (ALUs) or multipliers. The PEs are connected with each other by routing buses, and thus a PE can use the results of their neighboring PEs. CGRAs can fully exploit the parallelism in an application, and therefore they are extremely well suited for the applications that require very high throughput including multimedia, signal processing, networking, and other scientific applications. Several CGRA implementations such as MorphoSys [31], RSPA (Resource Sharing and Pipelining Architecture) [14], KressArray [12], etc. have been proposed, and a comprehensive summary of them can be found in [11].

The success of CGRAs critically hinges on how efficiently the applications are mapped onto them. For the efficient use of CGRAs, it is required to exploit the parallelism in the applications and minimize the number of computation resources since it is directly translated into either a reduced power consumption or an increased throughput. However, the problem of mapping an application onto a CGRA to minimize the number of resources has been shown to be NP-complete [13], and therefore several heuristics have been proposed. However, existing heuristics do not consider the details of the CGRA such as the following.

PE Interconnection: Most existing CGRA compilers assume that the PEs in the CGRA are connected only in a simple 2-D mesh structure, i.e., a PE is connected to it's neighbors only. However, in most existing CGRAs, each PE is connected to more PEs than just the neighbors. In Morphosys, a PE is connected to every other PEs through shared buses. A PE in RSPA also has the connections with the immediate neighbors and the next neighbors.

Shared Resources: Most CGRA compilers assume that all PEs are similar in the sense that an operation can be mapped to any PE. However, not all PEs in the current CGRAs contain multipliers in order to reduce the cost, power and complexity. Few multipliers and memory units are made available as shared resources in each row. For example, the PEs in each row of RSPA share two multipliers, two load units, and one store unit.

Routing PE: Most CGRA compilers cannot use a PE just for a routing. This implies that in a 4 × 4 CGRA which has a mesh interconnection, it is not possible to map application DAGs in which any node has more than 4 degrees. However, most CGRAs allow a PE to be used for routing only, which makes it possible to map DAGs with any degrees onto the CGRA.

Owing to the simplistic model of CGRA in the compilers, existing CGRA compilers are i) unable to map applications onto the CGRA, even though it is possible to map them, ii) using too many PEs in their solution. Thus, even if a mapping exists for an application, it consumes significant power on CGRAs.

The mapping techniques can be broadly categorized into the temporal mapping in which the functionality of PEs changes with time, and the spatial mapping in which an instruction in a

Manuscript received October 30, 2007; revised March 23, 2008. First published March 16, 2009; current version published October 21, 2009. This work was supported in part by a grant from Microsoft, the Korea Science and Engineering Foundation (KOSEF) NRL Program grant funded by the Korea government (MEST) (No. R0A-2008-000-20110-0), the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST) / Korea Science and Engineering Foundation (KOSEF) (R11-2008-007-01001-0), and the Human Resource Development Project for IT SoC Architect.

J. W. Yoon, S. Park, M. Ahn, and Y. Paek are with the School of Electrical Engineering and Computer Science, Seoul National University, Seoul 151-600, Korea (e-mail: jhyoon@optimizer.snu.ac.kr; shpark@optimizer.snu.ac.kr; mwahn@optimizer.snu.ac.kr; ypaek@snu.ac.kr).

A. Shrivastava is with the Department of Computer Science and Engineering, Arizona State University, Tempe, AZ 85281 USA (e-mail: Aviral.Shrivastava@asu.edu).

Digital Object Identifier 10.1109/TVLSI.2008.2001746

kernel loop corresponds to one PE and does not change during the execution of the kernel. Although many temporal mapping techniques [19], [22], [27], [33] can efficiently map loop structures using modulo scheduler [29], in this paper we focus on the static spatial mapping technique. The CGRAs which support the spatial mapping do not need the context memory [15], and therefore it has the advantages of smaller chip area, more power efficiency, and simpler hardware design as compared to the CGRAs with the temporal mapping.

In an attempt to develop better heuristics for mapping applications to CGRAs, we observe that the problem of application mapping is very similar to that of graph drawing. We formulate our problem as finding the mapping between two graphs; the input dependency graph and the CGRA interconnection graph, subject to several constraints like minimizations of bends, edge crossings, and area. The contributions of this paper are as follows.

- We formulate the application mapping problem onto CGRA considering the arbitrary PE interconnections, shared resource constraints, and routing PEs.
- We propose a graph drawing based approach to map applications onto CGRAs which can map 4.5× more randomly generated application DAGs than previous approach, while showing shorter execution cycles in 66 cases and less power consumption in 71 cases out of 100 synthetic applications, with minimal mapping time overhead.
- We demonstrate that SPKM is not a customized solution only for a specific CGRA. We explore various CGRA designs and show that SPKM still gives prominent results for any of the designs.

The rest of this paper is organized as follows. Section II and Section III formally define the problem of mapping applications onto CGRAs. Section IV describes CGRAs and the previously proposed heuristics for mapping applications onto CGRAs. In Section V formulates the application mapping problem with Integer Linear Programming (ILP) to find the optimal solution. Section VI gives an overview of our design flow and Section VII proposes our graph drawing based algorithm. We explain our experimental setup in Section VIII and Section IX demonstrates the effectiveness of our solution, followed by Section X in which we explore several CGRA design spaces and demonstrate that SPKM is a general solution for CGRA. We summarize and conclude this paper in Section XI.

II. Notation and Definitions

Since applications spend most of the execution time and the power in loops, we focus on mapping only loop kernels onto CGRAs. We map given loop kernels to CGRAs while minimizing the number of required resources. In this section, we define all the notations that are used throughout this paper for the better understanding of our problem formulation.

Loop kernel: The loop kernel can be represented as a Directed Acyclic Graph (DAG), $K = (V, E)$, where vertices in V are the operations in the loop, and an edge $e = (u, v) \in E$ exists for any two vertices, $u, v \in V$, iff the operation v is data dependent on the operation u.

CGRA: An $M \times N$ CGRA can be represented as another directed graph $C = (P, L)$, where the elements of P, p_{ij} (where $1 \leq i \leq M$, and $1 \leq j \leq N$) are the PEs of the CGRA. For any two elements $p, q \in Pq$, an interconnection link $l = (p, q) \in L$ exists iff PE q can use the result of PE p which is computed in the previous cycle.

Application Mapping: An application mapping is a function $\phi : K \rightarrow C$, which in turn implies two functions, $\phi_V : V \rightarrow P$, and $\phi_E : E \rightarrow 2^L$.

ϕ_V is an injective function that maps the operations to the PEs. This implies that each vertex $v \in V$ corresponds to a distinct PE $p \in P$, and some PEs may be unused.

ϕ_E is a multi-valued function that maps data dependence edges $e \in E$ of the application kernel to a subset of interconnection links $ll \in 2^L$. Thus, a dependence edge can be mapped onto a set of interconnection links on the CGRA, starting from $\phi_V(u)$, and ending at $\phi_V(v)$.

Path Existence Constraint: If $\phi_E(e = (u,v)) = ll \in 2^L$, then $\exists l_1 = (p_1, q_1) \in ll$ such that $p_1 = \phi_V(u)$, $\exists l_2 = (p_2, q_2) \in ll$ such that $q_2 = \phi_V(v)$, and $\exists l = (p', q') \in ll$, where $p' = q$, for $\forall l = (p, q) \in ll$ such that $q \neq \phi_V(v)$.

Simple Path Constraint: If $\phi_E(e = (u,v)) = ll \in 2^L$, then $\forall l_i, l_j \in ll$. If $l_i = (p, q)$, and $l_j = (r, s)$, then $q \neq s$. This implies that there are no loops in the path from $\phi_V(u)$ to $\phi_V(v)$, represented by ll.

Routing Order: With the path existence and the simple path constraints, a total order can be defined on the elements of ll. The order between two operations is represented by $\prec$. The total order for all the operations, which we call routing order, identifies a unique path from $\phi_V(u)$ to $\phi_V(v)$. The routing order is defined as:

1) $\forall l_i, l_j \in ll$, if $l_i = (p, q) \wedge \phi_V(u) = p$, then $l_i \prec l_j$
2) $\forall l_i, l_j \in ll$, if $l_j = (p, q) \wedge \phi_V(v) = q$, then $l_i \prec l_j$
3) $\forall l_i, l_j \in ll$, if $l_i = (p, q) \wedge l_j = (q, r)$, then $l_i \prec l_j$

The routing order implies that the first element $\phi_V(u)$ is the smallest element, and $\phi_V(v)$ is the largest element. When two active interconnection links share a PE, the interconnection link that uses the PE as the source is larger than the one that uses it as a destination. If we arrange the PE vertices in an increasing order, the path from $\phi_V(u)$ to $\phi_V(v)$ can be represented. The simple path constraint makes sure that if $ll \neq \phi$, then there exists a path from $\phi_V(u)$ to $\phi_V(v)$.

Routing PE: RPE for a data dependence edge $e \in E$ is a set of PEs which are used to transfer data between two interconnection links. Thus, for any $e = (u, v) \in E$, if $ll = \phi_E(e)$, then $RPE^e = \{q | \forall l = (p, q) \in ll, q \neq \phi_V(v) \wedge p \neq \phi_V(u)\}$. We also define $RPE = \bigcup_{e \in E} RPE^e$.

Uniqueness of Routing PE: A routing PE can be used to route only one value.

No Computation on Routing PE: No computations can be performed on a PE if it is used as a routing PE on a specific execution cycle.

Shared Resource Constraint: Most CGRAs have row-wise constraints since the rows usually share expensive resources such as multipliers, memory buses, etc. In RSPA for example, each row shares 2 multipliers, and therefore at most two multiply operations can be performed in each row. To specify the shared resource constraints, we define an attribute '*type*' of the shared resource for each vertex of the kernel graph K and the number of shared resources in a row as S_t for a specific shared resource t. Thus, if we define $V_i^t = \{v|\exists j, \phi_V(v) = p_{ij} \wedge v.type = t\}$, then $\forall i, |V_i^t| \leq S_t$.

Utilized CGRA Rows: We define UR as a set of CGRA rows that are utilized in mapping an application K onto the CGRA C. Thus, $UR = \{P_i|\exists j, p_{ij} \in Range(\phi_V) \vee p_{ij} \in RPE\}$.

III. Problem Formulation

In modern CGRAs, owing to the severe restrictions of the shared resource constraints, utilizing a smaller number of rows is a useful objective function. Utilizing fewer number of rows is directly translated into increased opportunities for novel power and performance optimization techniques. For example, the whole row of the unused PEs can be power gated to reduce the power consumption. In addition, it is possible to cleanly execute another loop iteration on the unused rows to improve throughput. Therefore, we formulate our problem as follows:

Given a kernel DAG $K = (V, E)$, and a CGRA $C = (P, L)$, find a mapping $\phi(K)$, with the objective of $min|UR|$ or $min|RPE|$, under i) path existence, ii) simple path, iii) uniqueness of routing PE, iv) no computation on routing PE, and v) shared resource constraints.

As compared to previous approaches, our problem formulation is novel in several aspects:

- We allow various types of orthogonal interconnections in CGRAs.
- We allow shared resource constraints which can be used to model resource-constrained operations, e.g., loads, stores, multiplications and divisions.
- We allow a data dependence edge to be mapped onto a set of interconnection links of CGRAs, and therefore allow PEs to be used as routing resources.

IV. Related Work

Various CGRAs have been proposed and summarized in [11]. MorphoSys [31] consists of 64 reconfigurable cells arranged in an 8 by 8 array coupled with a Tiny RISC processor. The cell array allows 16-bits SIMD (Single Instruction Multiple Data) model of computation. It is divided into four quadrants of 4 by 4 cells and each cell in a quadrant is fully connected with its nearest neighbors. Each quadrant is also connected with each other via buses. RSPA [14] is based on MorphoSys, but it has MIMD (Multiple Instruction Multiple Data) model of computation. Unlike MorphoSys, RSPA shares multipliers in order to reduce the chip size. The PEs in the same row share the data bus and thus only 2 load operations and 1 store operation are allowed for each of the row. RSPA has an interconnection network similar to MorphoSys. ADRES [21] is an architecture template which consists of a VLIW (Very Long Instruction Word) processor and a CGRA. It has an XML-based architecture description language to define the overall topology, operation set, resource allocation, timing, and an internal organization of each PE. The CGRA in ADRES has 8 by 8 functional units and Morphosys-like interconnection topology. Another CGRA is XPP (eXtreme Processing Platform) [3] which is a runtime reconfigurable architecture optimized for parallel data stream processing. It has 4 by 4 or 8 by 8 PE array and each PE performs typical DSP (Digital Signal Processing) operations such as multiplication, addition, shift, sort and comparison. KressArray [12] is the reconfigurable version of the super systolic array [30]. It is a mesh of rDPUs (reconfigurable Datapath Units) and an array of 128 DPUs can be realized. The DPUs are interconnected by the routing resources and the interconnection network can be fixed statically or dynamically. REMARC (Reconfigurable Multimedia Array Coprocessor) [24] is a reconfigurable accelerator tightly coupled to a MIPS RISC processor. It consists of an 8 by 8 array of nano processors containing an ALU, data RAM, register file, and special purpose registers. Each nano processor is connected with its nearest neighbors, but it shares broadcast buses in the same row or column. RAW (Reconfigurable Architecture Workstation) [34] from MIT consists of interconnected tiles and each tile contains instruction and data memories, an ALU, registers, configurable logic, and a programmable switch for routing. It has a 2-D mesh structure in which each microprocessor tile is connected with its immediate neighbors. There are many other CGRAs such as PipeRench [10], PADDI [4], MATRIX [23], RaPiD [7], etc., and the overview for those architectures can be found in [11].

Although CGRAs show an enormous computation power, the performance of CGRAs critically hinges on the mapping algorithm. Convergent scheduling for RAW [19] is proposed as a generic framework for instruction scheduling on spatial architectures. Its framework comprises a series of heuristics that address independent concerns like load balancing, communication minimization, etc. Convergent scheduling focuses on an instruction level parallelism and proposes a scheduling method for acyclic regions of code. Our mapping algorithm exploits not only instruction level parallelism but also loop level parallelism by unrolling. The mapping algorithm for Morphosys [33] also takes a data flow graph (DFG) as an input. In their algorithm, to minimize communication penalty, the DFG is partitioned into cliques and operations in each clique are allocated to the same resource. After the operations are allocated, the time slots for operations are assigned in the scheduling phase. Their algorithm is different from ours in that we generate pipeline schedules for the innermost loop bodies so that iterations can be issued successively, each node in the DFG can be mapped onto each PE in the target CGRA and the number of routing PEs can be minimized. Their execution model is also different from ours. Their algorithm is limited to SIMD execution model. This execution model is to endure some redundancies of fetching values from memory before it starts the computation. [22] and [27] propose similar modulo scheduling algorithms for mapping the DFGs onto CGRAs, which are based on a simulated annealing method.

The mapping algorithms proposed by [8], [16], and [18] use the one to one mapping approach for successively issuing iterations of the kernel loop. [18] focuses on enabling memory sharing between operations of different iterations placed on the same PE. Although they propose a compilation approach for a generic CGRA, their work is limited because it can not exploit different interconnect topologies as compared to ours. [16] finds a spanning tree in a DFG and route the interconnections corresponding to tree edges through abutment (connection between adjacent PEs). The remaining edges are routed through global interconnection. Basically, their strategy can map a DFG into only 2-D mesh, and does not consider the shared resources (multipliers) between DPUs. Although the liaison DPUs (routing PEs) are inserted for making connections between non-adjacent DPUs, their algorithm does not try to minimize the number of the liaison DPUs. The algorithm from [8] is guided by a depth-first traversal in both the architecture and the application graphs. Similar to our approach, their approach can exploit various interconnect topologies. However, their mapping algorithm assumes that a PE can perform data routing and operation simultaneously.

All the previous application mapping approaches assume a very simple model of the CGRA, and do not model the complexities in the CGRA designs like row constraints, shared resources, and memory interfaces, and irregular interconnections. Lee *et al.* [17] proposed a spatial mapping approach for a CGRA in which each row shares the memory interface. The work closest to ours is [1], in which authors consider both shared multipliers and memory interface, and propose a spatial mapping technique which is referred as *AHN* in this paper. However, they only consider the mesh-style CGRA interconnection, and more importantly, they did not take the minimization of routing PEs into consideration. Their channel PE is similar to routing PE but it is added only for connecting two column-wisely unreachable PEs. Thus, channel PE is not helpful for removing the diagonal edges or making it possible to map node with more degree than the degree of PE onto CGRA. In addition, they only consider binary tree as their input form even though DAG is also possible to be mapped onto CGRA using their approach. Using DAG as an input is the more general and commonly accepted form of application specification.

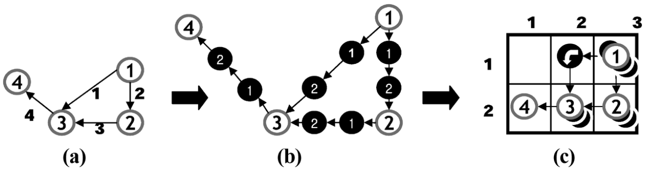
V. ILP Formulation

The problem of application mapping onto a CGRA has been proven to be NP-complete [13], even in the special case when the application is represented in a complete binary tree and the CGRA consists of a two dimensional grid with just the neighboring connections. Therefore, we formulate the problem with an Integer Linear Programming (ILP).

When an edge e is mapped to a subset of interconnection links ll such that $|ll| > 1$, it uses routing PEs. To accommodate the routing PEs, we add extra routing vertices on each edge of K. Since the exact number of routing PEs required for an edge $e \in E$ cannot be known until the mapping is complete, we insert the maximum possible number of routing vertices. The upper bound of the number of routing PEs is $|P - V|$. Therefore, we transform $K \rightarrow K' = (V', E')$ by inserting $|P - V|$ vertices on each edge $e \in E$. Fig. 1 shows the transformation of adding the candidates of routing vertices [dark vertices in Fig. 1(b)] on every edge in K. Let R^e be the set of the inserted black vertices onto $e \in E$, and R as $R = \bigcup_{\forall e \in E} R^e$. The problem is translated into mapping K' to C. Unlike normal vertices $v \in V$, the candidates of routing vertices, $r \in R$, can be mapped to the same PE $p \in P$ of the CGRA, including the PE on which a normal vertex $v \in V$ is mapped. Now we define our ILP on the transformed DAG K′.

Boolean Decision Variables:

- v_{ikl} is 1 if ith vertex $v_i \in V$ is mapped onto $p_{kl} \in P$.



$|V|=4, |E|=4, M=2, N=3, |R^e|=M*N - |V|=2,$
$v_{113}=1, v_{223}=1, v_{322}=1, v_{421}=1, \ldots$
$r_{1113}=1, r_{1222}=1, r_{2113}=1, r_{2213}=1, \ldots$
$q_{1113}=0, q_{1212}=1, q_{2113}=0, q_{2213}=0, q_{3123}=0, \ldots$

(d)

Fig. 1. Example of ILP formulation. (a) Kernel DAG K. (b) K′. (c) Mapping result. (d) Variables in ILP.

- r_{ijkl} is 1 if jth candidate of routing vertex $r_j \in R^{e_i}$ for $e_i \in E$ is mapped onto $p_{kl} \in P$.
- q_{ijkl} is 1 if jth routing vertex $r_j \in R^{e_i}$ is mapped onto $p_{kl} \in P - Range(\phi_V)$, which means q corresponds to using an actual routing PE.

In Fig. 1(d), r_{1113} is 1 because the first r for edge e_1 is placed on p_{13}, and q_{1212} becomes a routing PE because the second r for edge e_1 is placed on p_{12} where there is no operation v.

Objective Function: The objective function is to minimize $|UR|$ and minimize the number of routing PE under $min|UR|$ when we map K onto C. In order to satisfy both of them, we use weight functions $w_{UR}(k)$ and w_R. M and N are the numbers of rows and columns, respectively.

$$Minimize \left(\sum_i^{|V|} \sum_k^M \sum_l^N w_{UR}(k) \cdot v_{ikl} \right. \\ \left. + \sum_i^{|E|} \sum_j^{|R^e|} \sum_k^M \sum_l^N (w_{UR}(k) + w_R) \cdot q_{ijkl} \right) \quad (1)$$

where $\forall k \leq |UR|_{lbound}$, $w_{UR}(k) = 0$, $w_R = 1$, $\forall k' > |UR|_{lbound}$, $(|P| - |V|) \cdot w_R < w_{UR}(|UR|_{lbound})$, and $N \cdot w_{UR}(k') < w_{UR}(k'+1)$. $|UR|_{lbound}$ is lower bound of $|UR|$ in the CGRA, and it will be explained how to get $|UR|_{lbound}$ in the next section.

Constraints: For $1 \leq i \leq |V|$, $1 \leq l \leq |P| - |V|$, $1 \leq k \leq M$, $1 \leq l \leq N$,

$$\forall i, \sum_k^M \sum_l^N v_{ikl} \leq 1 \quad (2)$$

$$\forall i, j, \sum_k^M \sum_l^N r_{ijkl} = 1 \quad (3)$$

$$\forall k, l, \sum_i^{|V|} v_{ikl} + \sum_i^{|E|} \sum_j^{|R^e|} q_{ijkl} \leq 1 \quad (4)$$

$$\forall k, \sum_i^{|V|} \sum_l^N v_{ikl}^t \leq S_t \quad (5)$$

$$\forall e = (p, r_1) \in E'$$

$$\sum_{k_1}^M \sum_{l_1}^N \sum_{k_2}^M \sum_{l_2}^N Z_{k_1 l_1 k_2 l_2} \cdot v_{pk_1 l_1} \cdot r_{i1k_2 l_2} = 1 \quad (6)$$

$$\forall e\left(r_j, r_{(j+1)}\right) \in E',\ 1 \le j \le |R^e| - 1$$

$$\sum_{k_1}^{M}\sum_{l_1}^{N}\sum_{k_2}^{M}\sum_{l_2}^{N} Z_{k_1l_1k_2l_2} \cdot r_{ijk_1l_1} \cdot r_{i(j+1)k_2l_2} = 1 \qquad (7)$$

$$\forall e\left(r_{|R^e|}, q\right) \in E'$$

$$\sum_{k_1}^{M}\sum_{l_1}^{N}\sum_{k_2}^{M}\sum_{l_2}^{N} Z_{k_1l_1k_2l_2} \cdot r_{i|R^e|k_1l_1} \cdot v_{qk_2l_2} = 1 \qquad (8)$$

- Constraint 2 represents that each $v \in V$ is mapped onto exactly one PE $p \in P$.
- Constraint 3 represents that each $r \in R$ is mapped onto exactly one PE $p \in P$.
- Constraint 4 represents that each $p_{kl} \in P$ can have only one operation, v or q.
- In constraint 5, v_{ikl}^t is an element of set V_k^t. Constraint 5 represents that Each kth row of C can have at most S_t number of type t operations
- Constraint 6–8 construct the interconnection links between $p_{kl} \in P$. We use an adjacent matrix table Z which contains all the information about the connections between p_{kl}. $Z_{k_1l_1k_2l_2}$ is 1 if $p_{k_1l_1}$ is directly connected to $p_{k_2l_2}$ or if $k_1 = k_2$ and $l_1 = l_2$. Using Z, we represent all the possible connections when all $v, r \in V'$ are mapped onto $p_{kl} \in P$ using these three constraints.

Note that (6)–(8) are not linear. They contain products of boolean decision variables. Let a and b be boolean variables. The term $a \cdot b$ can be linearized by using an additional boolean variable t, and the following constraint:

$$t \ge a + b - 1, \quad t \le (a + b)/2.$$

VI. Design Flow

The entire design flow of our approach is described in Fig. 2. We manually extract the loop kernels from the applications, then the kernels are transformed into the predicated and unrolled code. The C compiler takes the rest of the code as an input, and performs compiler code generation. After the assembly code is generated, the additional code for controlling the loop kernel is inserted at the last step of the conventional compilation. It should be noted here that the effort in this paper is not to automate the entire design flow, but to solve the kernel mapping problem. Thus, in this section, we briefly introduce two preliminary steps, *predication* and *loop unrolling*, for loop kernels before describing the mapping algorithm.

A. Predication

Mainly due to lack of the program counter, the control logic in most existing CGRA templates (e.g., [5], [21], [26]) is not as versatile to fully support various conditional execution features as in ordinary microprocessors. Therefore, in order to execute a kernel loop, we transform all branches, such as if-statements, within the original loop into predicate statements [20].

B. Loop Unrolling

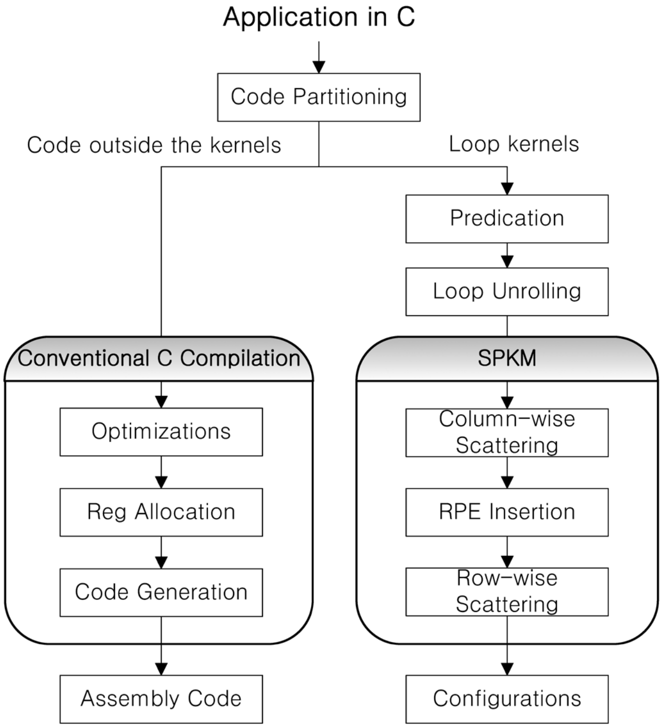


Fig. 2. Entire design flow.

Since we are mapping applications with the minimal number of utilized rows, there are more chances that we can map the unrolled loop kernels onto CGRAs. Thus, in the first step of our approach, we unroll the kernel loop and see if the unrolled loop can be mapped onto a given CGRA. The number of utilized rows, UR, is closely related to the unroll factor, UF. Therefore, we compute its lower bound, $|UR|_{lbound}$, which is required for the kernel unrolled by UF, as follows:

$$|UR|_{lbound} = max\left(\lceil (UF \cdot |V|)/|N| \rceil, \lceil (UF \cdot V^{load})/S_{load} \rceil, \lceil (UF \cdot V^{store})/S_{store} \rceil, \lceil (UF \cdot V^{mul})/S_{mul} \rceil\right)$$

in which N is the number of columns on CGRA, V^i represents the number of corresponding operation i, and S_t is the number of shared resources t in a row of CGRA. The parameters of max function represent the number of required rows for all vertices of an application, for the load and store operations, and for the multiply operations respectively. With this equation, we evaluate the unroll factor by increasing UF from 1 (no unroll) until it results in larger $|UR|_{lbound}$ than the number of rows of CGRA. Once UF is decided, the kernel loop unrolled by this UF value is mapped onto CGRA using our *SPKM* as will be described in the next section. In the example of Fig. 6, the unroll factor is '1' because if we unroll the kernel two times ($UF = 2$), the minimally required number of rows is computed as

$$max\left(\lceil (2 \cdot 10)/4 \rceil, \lceil (2 \cdot 3)/2 \rceil, \lceil (2 \cdot 1)/1 \rceil\right) = 5$$

which exceeds the number of rows of the CGRA.

VII. Our Approach: Split-Push Kernel Mapping (SPKM)

Our approach of mapping a kernel onto a CGRA is based on the split & push algorithm [2] which is popular in the area of

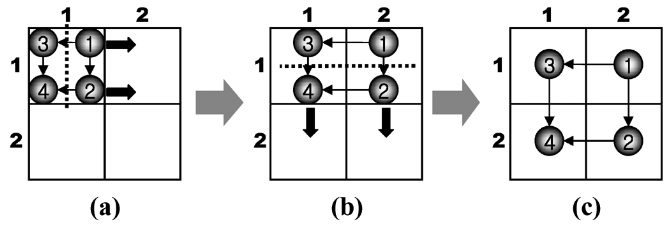


Fig. 3. Split & push approach.

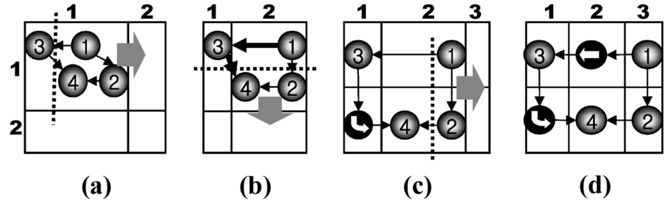


Fig. 4. Formation of fork.

graph drawing. Fig. 3 shows an example of mapping kernel K that has four operations onto a mesh-style CGRA C.

The algorithm starts with a *degenerate drawing* where all the nodes in a graph are located at the same coordinate (1,1), as shown in Fig. 3(a). In the first step, we locate each vertex $v \in K$ using *cuts*. A cut is a plane which *splits* the nodes into two groups and it is shown by a dotted line in Fig. 3. After the graph has been split, all v in one of the groups are *pushed* to new coordinate. Fig. 3(b) shows the result of *split & push* along the dotted line. This split & push is repeated until every v has distinct coordinates as shown in Fig. 3(c).

The crucial step in the split & push approach is finding appropriate cuts which generate the minimal number of coordinates. Consider another application of split & push applied to the same K in Fig. 4. The final graph requires much more vertices than the solution in Fig. 3 due to the routing vertices. This is because v_3 is separated from other nodes in the first stage of split & push algorithm. This separation produces a *fork*. A fork is adjacent edges cut by a split. Once there is a fork and the fork consists of n adjacent edges, $n-1$ bends (dummy nodes or routing vertices) are required as $n-1$ edges in the fork become slant, which is not allowed in mesh graph drawing. Forks can be avoided by finding a *matching-cut*. A matching-cut is defined as a set of edges which have no common node and whose removal makes the graph disconnected. The problem of finding a matching-cut in a graph is again an NP-complete problem [28].

In order to minimize the number of utilized rows and routing PEs in the mapping, we propose a novel mapping heuristic which is based on the split & push graph algorithm. Fig. 5 describes our mapping technique, *Split-Push Kernel Mapping*, or *SPKM*. Each step is thoroughly explained with an example of Fig. 6 in this section. We assume that in our CGRA C, all PEs are connected to their immediate and the next immediate neighbors. Thus, a PE is connected to at most 6 other PEs. We also assume that at most 2 load operations and one store operation can be scheduled in a row. K of Fig. 6(a) has 10 operations including 3 loads (gray nodes) and 1 store (dark grey node).

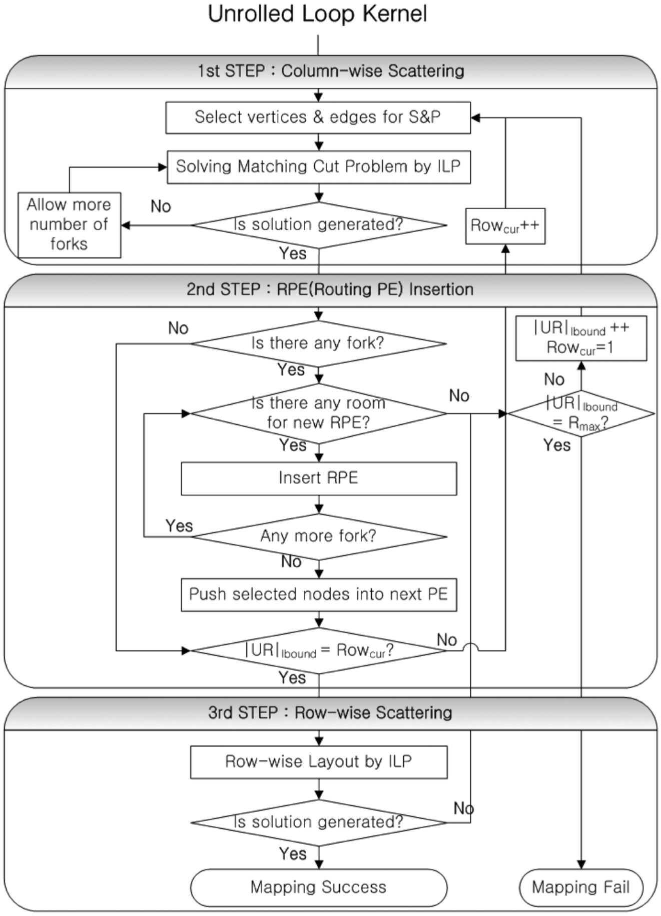


Fig. 5. SPKM: split and push kernel mapping.

A. Column-Wise Scattering

In the first step, we distribute vertices $v \in V$ to the minimum number of UR in the same column considering the minimal number of forks and shared operations like multiplication, load and store. We distribute all $v \in V$ to p_{k1}, $1 \leq k \leq |UR|_{lbound}$. All v located at p_{k1} are separated into two sets by cutting and the nodes in one of the sets are pushed into $p_{(k+1)1}$. For example, all the nodes in p_{11} of Fig. 6(a) are separated into two sets of nodes $\{v_4, v_8, v_{10}\}$ and $\{v_1, v_2, v_3, v_5, v_6, v_7, v_9\}$. The nodes in the set $\{v_1, v_2, v_3, v_5, v_6, v_7, v_9\}$ are pushed into p_{21} like in Fig. 6(c). Now the nodes $\{v_1, v_2, v_3, v_5, v_6, v_7, v_9\}$ are separated into two sets $\{v_1, v_3, v_9\}$ and $\{v_2, v_5, v_6, v_7\}$ again, and the nodes $\{v_2, v_5, v_6, v_7\}$ are pushed into p_{31}. The split & push is repeated until there are no *empty* p_{k1} in the $|UR|_{lbound}$. In each repetition, we try to find the matching cut to minimize the number of routing PEs.

ILP Solution of Matching Cut: Since matching cut problem is NP-complete, we solve it by formulating as an ILP. In the kth repetition, the graph K^k consisting of all the nodes in p_{k1} are split into two separated graphs, and one of them, K^{k+1} is pushed into $p_{(k+1)1}$. K^1 is the same as K. We find a matching cut in K^k satisfying following ILP.

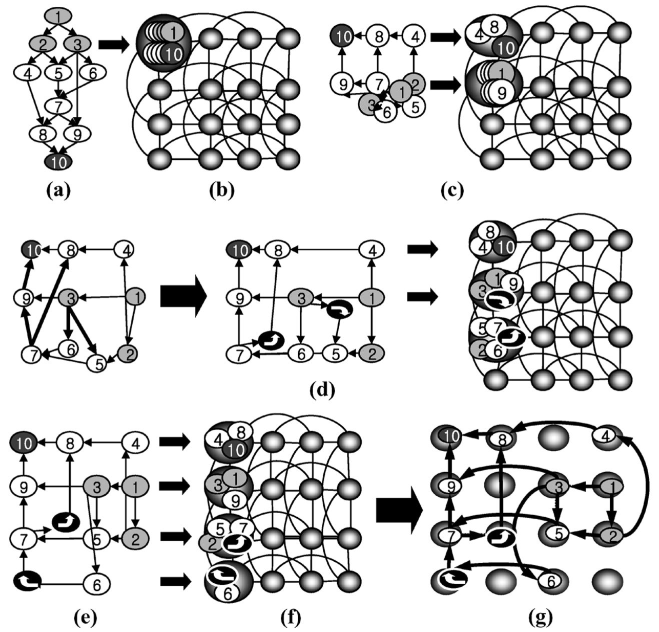


Fig. 6. Mapping process example.

Objective Function:

$$Minimize \left| \sum_{v \in V^k} v_{ik1} - \xi \right| \tag{9}$$

where v_{ik1} is 1 if the nodes are not pushed into $p_{(k+1)1}$, or 0 otherwise, and ξ is a constant restricting the number of nodes left in p_{k1}. As evenly distributing all nodes in K^1 to all p_{k1} within $|UR|_{lbound}$ gives more chances for getting better mapping by providing space in each row to add routing vertex, we set $\xi = \lfloor (N + |UR|_{lbound} - 1)/|UR|_{lbound} \rfloor$.

Constraints:

- The first constraint restricts the number of nodes left in p_{k1} due to shared resources like memory buses or heavy computation resources. For example, the node v_1 in Fig. 6(a) has one load primitive operation inside. So v_{111}^{load} is 1. In this CGRA, p_{kl} within one row shares two read buses, and therefore S_{load} is 2.
- To minimize the forks, we have another constraint for all v_i^m with multiple edges in K^k.

$$\sum_{v_i \in V^k} v_{ik1}^t \le S_t \tag{10}$$

$$\sum_{v_j \in adj\left(v_i^m\right)} \left(v_{jk1} + v_{ik1}^m\right) \le \zeta_1 \;\; or$$
$$\sum_{v_j \in adj\left(v_i^m\right)} \left(v_{jk1} + v_{ik1}^m\right) \ge 2 \cdot deg\left(v_{ik1}^m\right) - \zeta_2 \tag{11}$$

where $\mathrm{adj}(v_i^m)$ is the set of nodes adjacent to v_i^m and $deg(v_i^m)$ is the degree of v_i^m. ζ_1 and ζ_2 are used for determining how many forks are allowed.

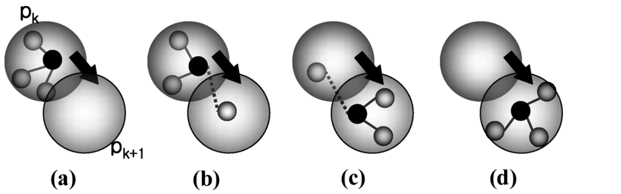


Fig. 7. Fork minimization algorithm.

Equation (11) is not linear due to 'or'. In order to linearize this, we change this equation to (12) and (13) using new constant η which is big enough.

$$\sum_{v_j \in adj\left(v_i^m\right)} \left(v_{jk1} + v_{ik1}^m\right) \le \zeta_1 + \eta \cdot v_{ik1}^m \tag{12}$$
$$\sum_{v_j \in adj\left(v_i^m\right)} \left(v_{jk1} + v_{ik1}^m\right) \ge 2 \cdot deg\left(v_{ik1^m}\right) - \zeta_2 - \eta \cdot \left(1 - v_{ik1}^m\right) \tag{13}$$

When a *multi-degree-node* v_{ik1}^m, black node in Fig. 7 and its adjacent nodes in K^k are determined to be pushed into $p_{(k+1)1}$, there are four ways to avoid forks. Fig. 7(a) and (b) show two possible cases where v_{ik1}^m is not pushed into $p_{(k+1)l}$ and $deg(v_{ik1}^m)$ or $deg(v_{ik1}^m) - 1$ adjacent nodes are not pushed either. Fig. 7(c) and (d) show the other cases where v_{ik1}^m is pushed into $p_{(k+1)l}$ and only 0 or 1 adjacent node is not pushed. Thus, if we do not want to allow forks, we set $\zeta_1 = \zeta_2 = 1$. If there is no matching cut in the kth repetition of split & push, we increase ζ_1 and ζ_2, allowing more forks until finding feasible solution in ILP. The leftmost kernel DAG K in Fig. 6(d) shows the result of column-wise scattering. Because $|UR|_{lbound}$ is 3, SPKM tries to map the kernel with three rows and it generates two forks during scattering.

B. Routing PE Insertion

At every step of column-wise scattering, routing vertices should be inserted on the edges of each fork generated in the first stage of SPKM to route data via indirectly connected PE. In this step, we generate the routing vertices and connect them with existing vertices. Since (v_7, v_8) is an edge of the first fork in Fig. 6(d), we need a routing PEs (black nodes) on it. A routing PE is also inserted into the edge (v_3, v_5) of the second fork. Sometimes we cannot map the kernel with a given $|UR|_{lbound}$, since the number of vertices in K^k exceeds the number of PEs in the kth row after routing PEs are inserted into K. For instance, the rightmost graph with routing PEs in Fig. 6(d) has five nodes at p_{31} due to the insertion of a routing PE. Because we have four PEs in a row, this is an invalid mapping. In this case, we go back to column-wise scattering and increase $|UR|_{lbound}$ by one. Fig. 6(f) shows the result of column-wise scattering with $|UR|_{lbound}$ of four. We also need two routing PEs. One is on the edge (v_7, v_8) and the other is on (v_6, v_7). After the insertion of routing PEs, it is still a valid result.

C. Row-Wise Scattering

In this last stage, we distribute all the nodes at p_{k1} to the nodes p_{kl} where $l \in [1, n]$. To avoid diagonal edge and edge crossing, all the nodes that have connections between different rows should be placed in the same column. For example, the nodes $\{v_3, v_5, v_6\}$ in Fig. 6(d) are located in different rows but they have connections to each other. If the node v_6 is located in the fourth column while other two nodes v_3 and v_5 are located in the third column, we need a routing PE between v_3 and v_6.

ILP Solution of Row-Wise Scattering: We solve row-wise scattering by formulating as an ILP. Before explaining the ILP, we define expanded loop kernel DAG, $K^e = (V^e, E^e)$ which includes added routing vertices and relevant edges. Fig. 6(e) and the middle of Fig. 6(d) show the example of K^e.

Objective Function:

$$Minimize \sum_{i}^{|V^e|} \sum_{c}^{C} c \cdot v_{ic} \tag{14}$$

where v_{ic} is 1 if ith vertex $v_i \in V^e$ is mapped on p_c at the row fixed in the column-wise scattering. This objective can maximize the number of unexecuted column and we can apply power gating techniques to the column.

Constraints:

$$\forall i \in V^e, \quad \sum_{c=1}^{C} v_{ic} = 1 \tag{15}$$

$$\sum_{\forall v_i \in V_k^e} v_{ic} \leq 1 \tag{16}$$

where V_k^e is a subset of V^e existing on the kth row.

$$\forall l \subset 2^{E^e}, \quad \sum_{e \in l} \sum_{c=1}^{C} |c \cdot v_{i_e c} - c \cdot v_{j_e c}| = 0 \tag{17}$$

where l is the longest path connected with only inter-row links.

$$\sum_{c_i=1}^{C} \sum_{c_j=1}^{C} Z_{c_i c_j} \cdot (x_{i c_i} \cdot x_{j c_i}) \leq 1 \tag{18}$$

where Z represents the predefined adjacent matrix table that has 1 if ith column and jth column are connected, otherwise 0.

- Each $v \in V^e$ is mapped onto exactly one PE $p \in P$, (15).
- Each $p_{kl} \in P$ can have only one operation, (16).
- all $v \in l$ are mapped onto same column, (17).
- we represent all the possible connections when all $v, r \in V^e$ are mapped onto $p_{kl} \in P$ using (18).

As a result of row-wise scattering, Fig. 6(g) shows the final mapping of the application shown in Fig. 6(a). SPKM is able to take complex PE interconnections, shared resources, and routing resources into account due to the following reasons. First, SPKM can take care of complex PE interconnections because we model the interconnection topology of CGRA with graph edges and use graph-drawing algorithm to map applications. Second, thanks to the insertion of routing PEs described in Section VII-B and our solution for matching cut problem, SPKM is able to consider and minimize routing PEs during mapping process. Finally, the constraints in the column-wise scattering step enable SPKM to take care of the shared resources.

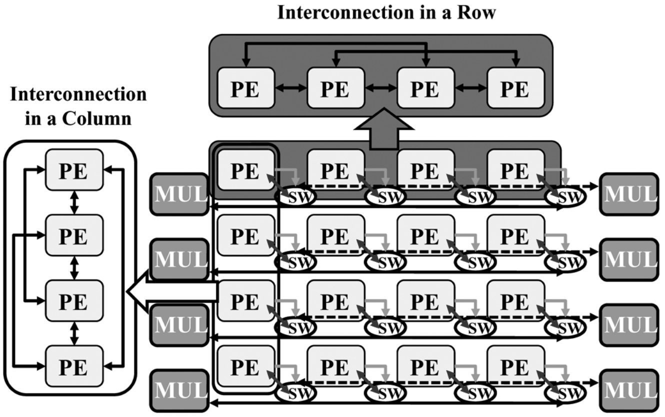


Fig. 8. Target architecture: RSPA.

VIII. Experimental Setup

We test SPKM on a CGRA called RSPA [14] which is shown in Fig. 8. RSPA consists of 16 PEs in which each PE is connected to four neighboring PEs, and also the four next neighboring PEs (PE interconnection). In addition, it has two shared multipliers in each row (shared resource), each row can perform two loads and one store (also shared resource), and it allows PEs to be used for routing (routing PE). However, when a PE is used for routing, it cannot perform any other operation. All of our experimental results shown in this paper are based on the comparison with another spatial mapping technique [1] (we refer as AHN in this paper), because it is the only spatial mapping technique that considers both the shared resources and the routing resources. They divided the mapping problem into three subproblems and formulated them with ILP. The effectiveness of their technique has been shown on the same architecture, RSPA, and compared with hand-optimized outputs for many loop kernels. However, as will be demonstrated in the experimental section, the mapping results of AHN are less efficient as compared to our technique because they scatter the nodes considering only the resource constraints, not the interconnections between the rows. And more importantly, they did not take the minimization of routing PEs into consideration. More details of the previous technique AHN can be found in [1]. In fact, we did not compare thoroughly our spatial mapping technique with other temporal mapping ones mainly because AHN [1] has already compared these two mapping strategies with examples in more detail.

To demonstrate the effectiveness of SPKM, we have devised a random kernel DAG generator. Our DAG generator randomly generates 100 DAGs for each value of node cardinalities from 5 to 16 nodes (1200 in total). AHN can also take DAGs as inputs since the features of RSPA allow AHN to map DAGs. Each DAG is generated according to the following steps. Once the number of nodes is given, the operations which are supported by PEs of RSPA are randomly assigned to each of the node. In this process, at least one load operation should be assigned to source vertex and one store operation to sink vertex respectively. Finally edges are inserted, satisfying that each node should not

TABLE I
BENCHMARK INFORMATION

Benchmark	Nodes	Load/Store	Mul
Livermore Loops			
Hydro	9	3/1	3
Inner	6	3/1	1
State	26	9/1	8
Iccg	10	5/1	2
DSPStone			
N complex update	16	6/2	4
Wavelet Benchmark			
RGBtoYUV transform	27	9/3	9
Mediabench			
Bdist2	27	10/1	1
Dist1	15	6/1	1
Multimedia benchmark			
Laplace	23	9/1	5
FFT	20	4/4	4
Prewitt	27	9/1	0

TABLE II
POWER PARAMETERS OF RSPA

Components	Power Consumption
Active PE	2.54293 mW
Routing PE	0.847321 mW
Idle PE	0.254293 mW
Configuration cache in a row	10.8867 mW
The rest part of RA (constant)	64.97043 mW

have more than two incoming edges. In addition, we also compare SPKM and AHN on a collection of benchmarks from Livermore loops, MultiMedia and DSPStone (they are summarized in Table I). In order to get the effective unroll factors from these algorithms, we use unrolled inputs with all possible unroll factors within the limitation of the resources like PEs, shared bus and multipliers. Then for each input we select the best mapping result which can unroll the kernel loop as many as possible and requires the least number of routing PEs. All experiments are done on Pentium4 3-GHz machine with 1-GB RAM. We use glpk4.8 [9] for solving our ILP formulations.

We have analyzed the power consumption of PE and configuration cache of the RSPA. The architecture has been synthesized using Design Compiler of Synopsys [32] with technology of DongbuAnam 0.18-μm [6]. We have used SRAM Macro Cell library for the frame buffer and configuration cache. ModelSim [25] and PrimePower [32] have been used for gate-level simulation and power estimation. The operation frequency was 100 MHz and V_{dd} was 1.8 V at 27 °C. The power of active PE, routing PE, and idle PE, and configuration caches in a row is described by Table II.

IX. EXPERIMENTS

In this section, we demonstrate that SPKM can not only map more applications but also generate better quality mappings. To demonstrate the effectiveness of our technique, we divided the experiments into four parts. First, we show that SPKM is able to map much more applications than the previous one, since SPKM considers the orthogonal interconnections effectively and exploits routing PEs. Second, we demonstrate that we can generate better quality mappings than the previous technique, in terms of the utilized CGRA rows. The less usage of CGRA rows results in the shorter execution time and the lower power consumption of mapped applications as will be discussed in the following subsection. SPKM can achieve those outperforming results with minimal mapping-time overhead, and it is described next. Finally, we show that SPKM is able to map real benchmarks collected from Livermore loops, MultiMedia and DSPStone as well.

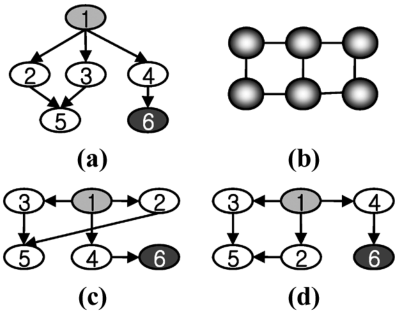


Fig. 9. Example of why SPKM can map more applications. (a) Example graph. (b) Example CGRA. (c) AHN's scattering. (d) SPKM's scattering.

A. SPKM Can Map More Applications

SPKM shows better ability of application mapping than the previous mapping technique, AHN, for two reasons. First, although AHN considers the shared resources, AHN cannot take the orthogonal PE interconnections into consideration when it scatters the nodes of application onto CGRA rows. For example in RSPA shown in Fig. 8, each row can execute no more than two multiplies, two loads, and one store. When AHN decides which nodes will be mapped on each row, AHN splits the graph according to the constraints of the number of shared resources. For instance, AHN splits the graph in Fig. 9(a) into $\{1, 2, 3\}$ and $\{4, 5, 6\}$ since this splitting satisfies the resource constraints that Fig. 9(b) has. However, AHN fails to map this graph because AHN is not aware of the orthogonal PE interconnections as shown in Fig. 9(c). On the other hand, SPKM considers PE interconnections also during it finds matching-cut in the graph, and is able to find mapping for this example graph [Fig. 9(d)]. The second and more important reason is that, since the previous technique does not take routing PEs into consideration, a node which has a large number of edges can not be mapped onto CGRAs. For example, if a node in an application graph has 2 incoming and 5 outgoing edges, then this node cannot be mapped on any PEs of RSPA because each PE on RSPA can have at most 6 interconnections (with 4 neighboring PEs and 2 next-neighboring PEs). Using a PE as a routing resource makes it possible to map any degree DAGs onto the CGRAs.

Fig. 10 plots how many applications out of 100 applications can be mapped by the three techniques for each value of node cardinality. The X-axis represents the number of nodes that each input application contains, and the Y-axis shows the number of valid mappings among 100 applications. Since ILP takes a lot of time for large input graphs, we stop the ILP solver after 24 hours. ILP cannot find a solution for graphs with 13 or more nodes, therefore there is no bar for ILP from 13 nodes in Fig. 10. The main observation from this graph is that SPKM can on average map 4.5× more applications than AHN. It is interesting

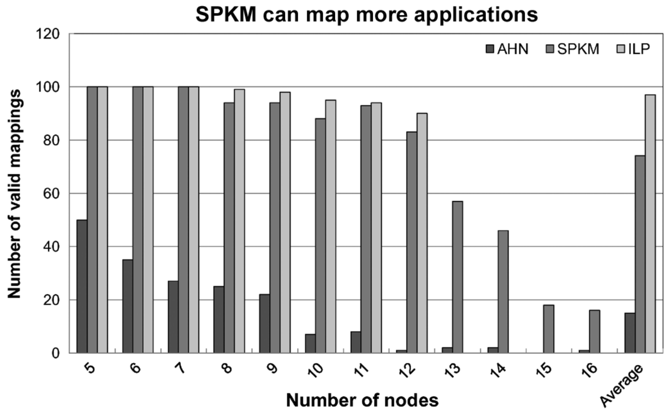


Fig. 10. SPKM can map 4.5× more applications than AHN.

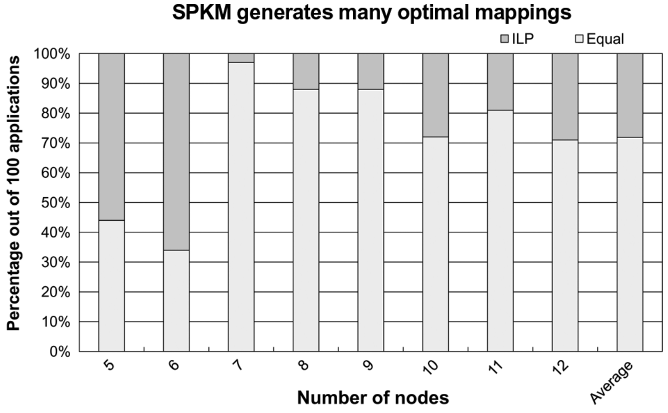


Fig. 12. SPKM can generate optimal mappings in 72% of the applications.

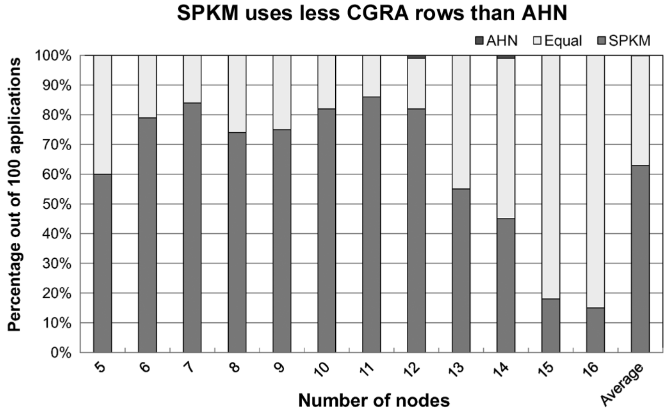


Fig. 11. SPKM uses fewer rows for mapping than AHN.

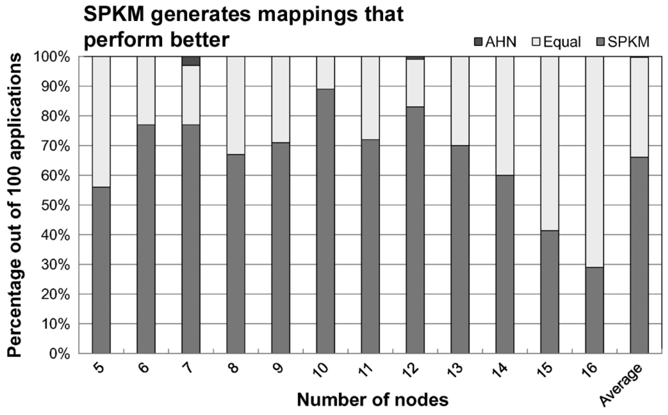


Fig. 13. SPKM generates mappings which have better performance.

to note that the graph shows that the mapping ability of SPKM over AHN increases with the increase in the number of nodes. This implies the effectiveness of our technique for the large applications.

B. SPKM Can Generate Better Quality Mappings

In addition to being able to map more applications than AHN, SPKM is also able to generate better quality mappings for the applications in terms of the number of utilized rows. Both algorithms compute the same lower bound of utilized rows, e.g., 3 in the example in Fig. 6(a). However, since AHN often fails to map applications within the lower bound of rows for the same reason explained in the previous subsection, AHN needs more CGRA rows to map applications. As will be shown in this subsection, using more CGRA resources impacts the performance and power consumption of mapped applications.

Fig. 11 plots the number of better, similar, or worse mappings generated by SPKM and AHN out of 100 applications for each node cardinality. The white bars represent the number of applications in which SPKM and AHN maps with the same number of rows. For example, for the 100 applications which have 12 nodes, in 82 cases, SPKM can generate mappings which have fewer rows than AHN; AHN generated a mapping with fewer rows in one case, and in the rest 17 cases, SPKM and AHN generate mappings with the same number of rows. On average, SPKM can generate better mappings than AHN for 62% of the applications, and the similar or better mappings for 99% of the applications. Fig. 12 plots in how many applications out of 100, ILP generates better mapping than SPKM. Note that this graph has data only until 12 nodes since ILP cannot map large applications in reasonable time. Also note that there are only two kinds of bars. This is because SPKM can never generate better mapping than ILP. On an average, SPKM is able to generate optimal mapping in 72% of the applications.

The fewer the utilized rows are, the more opportunities there are to boost up the performance by using more loop-unrolled kernels. Therefore, we can compare the execution time for each algorithm using unroll factor for each mapping. Moreover if we can use a smaller number of rows, there are more opportunities to reduce the power consumption of PE arrays by switching the unused PEs to low power mode. Fig. 13 plots the number of better, similar, or worse mappings generated by SPKM and AHN out of 100 applications for each node cardinality in terms of unroll factors. On average, SPKM can generate mappings which have better performance than AHN for 66% of the applications, and the same quality mappings for 33% of the applications. This implies that for 99% of the applications, SPKM is able to generate better, or at least the same quality mappings than AHN in terms of the execution time for the generated mappings. Fig. 14 plots in how many applications out of 100 applications, SPKM generates better mapping than AHN in terms of power consumption. Since SPKM is able to generate mappings which require fewer resources, the mapped applications can run

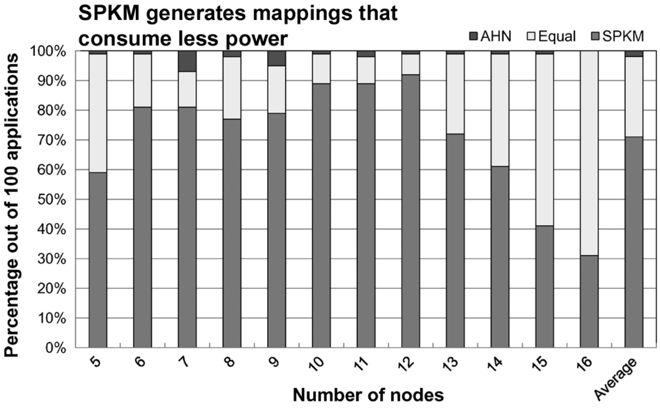


Fig. 14. SPKM generates mappings which consume less power.

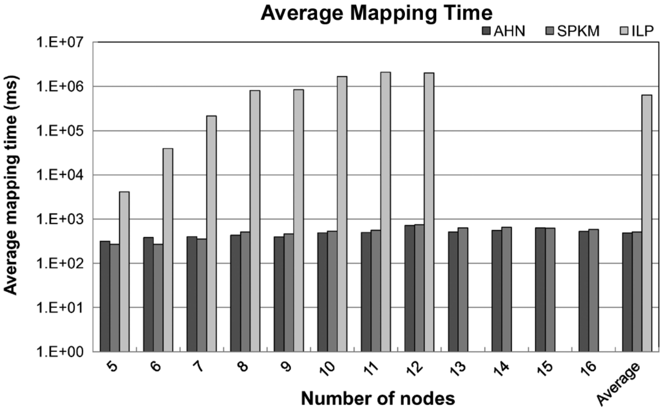


Fig. 15. SPKM has less than 5% mapping-time overhead.

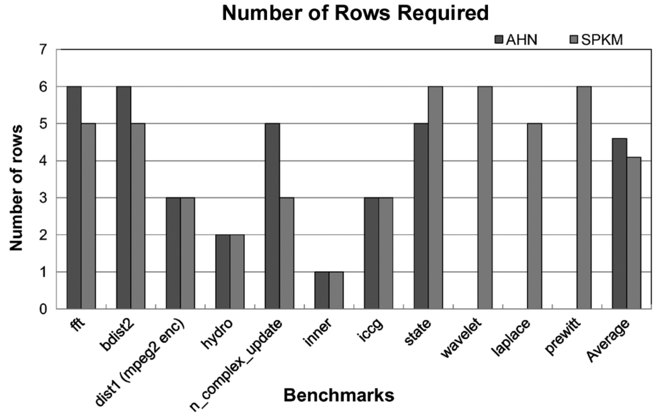


Fig. 16. Number of rows for real benchmarks.

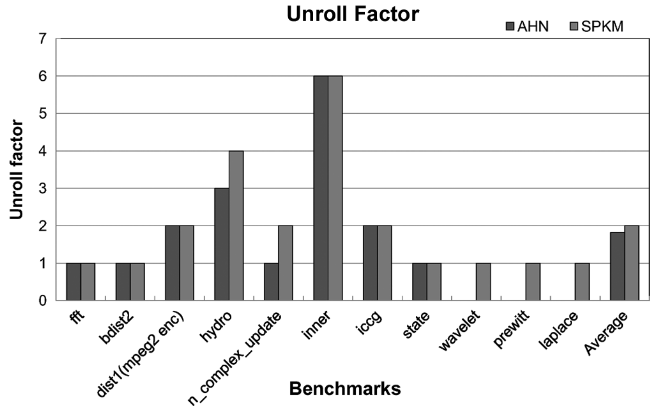


Fig. 17. Unroll factor for real benchmarks.

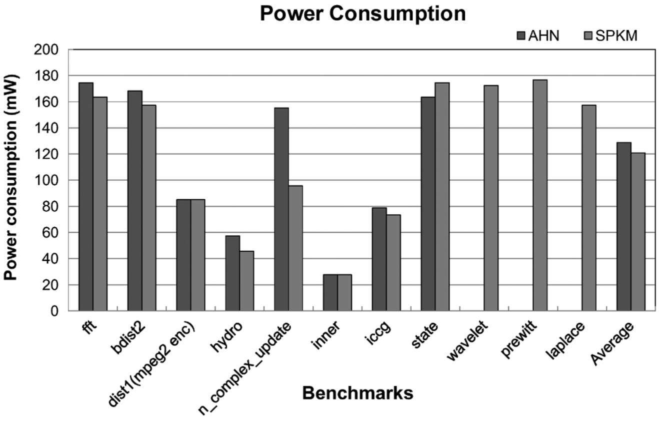


Fig. 18. Power consumption for real benchmarks.

with less power. Fig. 14 shows that on average SPKM can generate mappings that consume less power than AHN for 71% of the applications, and the same quality mappings for 27% of the applications, implying that for 98% of the applications, SPKM is able to generate better, or at least the same quality mappings than AHN.

C. SPKM Has Minimum Mapping-Time Overhead

SPKM generates effective mapping described above with minimal mapping time overhead. Fig. 15 shows the average mapping time for all the three algorithms. Note that the Y-axis is a logarithmic scale. On average, SPKM has only 5% overhead in mapping time as compared to AHN, and both are much less than the time taken by ILP.

D. SPKM on Real Applications

To demonstrate the effectiveness and usefulness of SPKM, we compare the number of rows of SPKM and AHN for a set of benchmarks collected from Livermore loops, multimedia, and DSPStone. Since these applications are large, we modified RSPA to have 6 × 4 PEs for the applications.

Fig. 16 shows the number of rows required for the mapping generated by SPKM and AHN. ILP is unable to find a mapping for any of these real applications in reasonable time. The first observation from this graph is that AHN is unable to map three of the real applications, demonstrating that SPKM can map more applications than AHN. The second observation is that the mappings generated by SPKM uses a smaller number of rows than AHN. SPKM uses fewer rows in three real applications and the same number of rows in four real applications out of eight that can be mapped by AHN, demonstrating the goodness of mapping.

As described above, the goodness of less utilized rows implies the improved performance and lower power consumption of the mapped application. Fig. 17 shows the unroll factor of SPKM and AHN which is directly related to the performance of the mapped applications. This graph shows that SPKM can map applications with larger, or at least the same unroll factors. Fig. 18

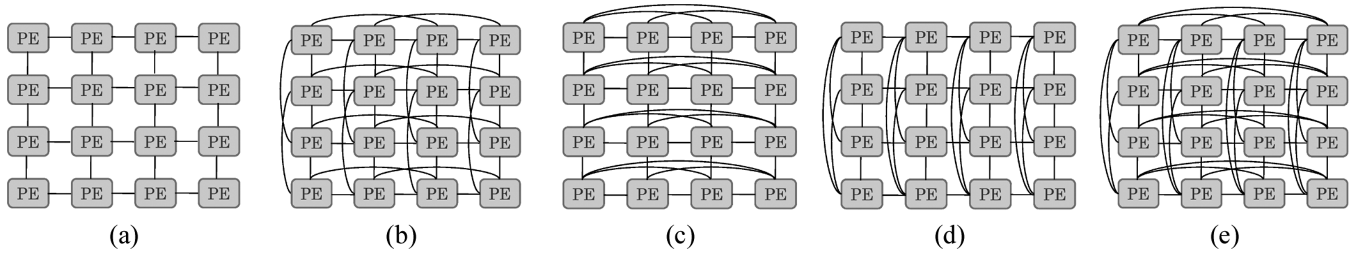


Fig. 19. Various PE interconnection topologies. (a) Mesh interconnection. (b) One-hop interconnection. (c) Two-hop row-only interconnection. (d) Two-hop column-only interconnection. (e) Two-hop (full) interconnection.

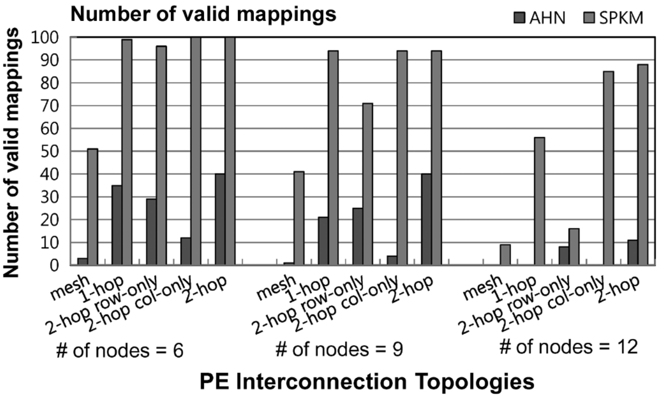


Fig. 20. SPKM can map more applications even if PE interconnection topology changes.

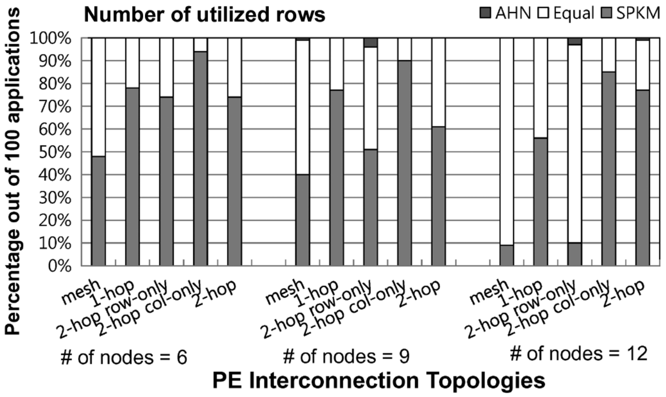


Fig. 21. SPKM also generates better quality mappings for any interconnection topologies.

depicts the power consumption in mW of the benchmarks. This graph shows that on average the applications mapped by SPKM can reduce the power consumption by 9.6%, demonstrating the goodness of mapping. Regarding the mapping time, SPKM uses just 2% more time than AHN for the benchmarks that AHN could map.

X. Design Space Exploration

Finally, we demonstrate that SPKM shows prominent results not only for RSPA but also for several CGRA designs which have various interconnection topologies and shared resource configurations.

A. *Exploration on Various PE Interconnections*

The PE interconnection topology is one of the most important parameters since it determines the schedulability of an application. Fig. 19 depicts five PE interconnections; the basic mesh structure in which PE has interconnections with only its neighboring PEs, one-hop interconnection which connects next neighboring PEs also, two irregular interconnections in which only rows or columns are fully connected, and two-hop Morphosys-like structure that PEs in each row and column are fully connected with each other.

In this experiment, we explore those interconnection topologies and show that SPKM can exploit the PE interconnections better than AHN and thus map more applications and generate better quality mappings for each of PE interconnections. Fig. 20 shows the number of valid mappings that both SPKM and AHN can generate, for each interconnection topologies and for the three values of node cardinality. The main observation from this graph is that SPKM maps much more applications than AHN for any PE interconnection topologies. Note that SPKM shows relatively low mapping capability on mesh and two-hop row-only interconnection since our algorithm starts with column-wise scattering in which it exploits the vertical interconnections first. Although it is generally known that it is easier to schedule kernels with rich PE interconnections, SPKM shows better schedulability if more column-wise interconnections appear on the CGRAs due to its algorithmic property.

Fig. 21 plots the number of better, similar, or worse mappings generated by SPKM and AHN out of 100 applications for various PE interconnection topologies. This graph shows that SPKM generates the mappings which use fewer, or at least the same number of rows than AHN. The metric of the utilized rows directly impacts the performance and power consumption of mapped applications as described in Section IX-B. To summarize, SPKM maps more applications and generates mappings which have better quality than AHN for various PE interconnection topologies.

B. *Exploration on Various Shared Resource Configuration*

RSPA shows the shared resource constraints in which each row shares two multipliers, two load units and one store unit. The shared resource constraints restrict the schedulable nodes of applications for each row. To demonstrate that SPKM is not customized only for the shared resources of RSPA, we vary and explore the shared resource configurations. Table III shows 4 configurations for the exploration. We start from 1M-1L-1S configuration in which each row shares 1 multiplier, 1 load unit and 1 store unit. In this configuration, no more than 1 operation

TABLE III
SHARED RESOURCE CONFIGURATION

Configuration	Description
1M-1L-1S	4 PEs in each row share 1 multiplier, 1 load, and 1 store unit.
2M-2L-1S	4 PEs in each row share 2 multipliers, 2 load, and 1 store unit.
4M-2L-2S	PEs in each row shares 2 load and 2 store units. Each PE has its own multiplier.
4M-4L-4S	Each PE has its own multiplier, load and store unit.

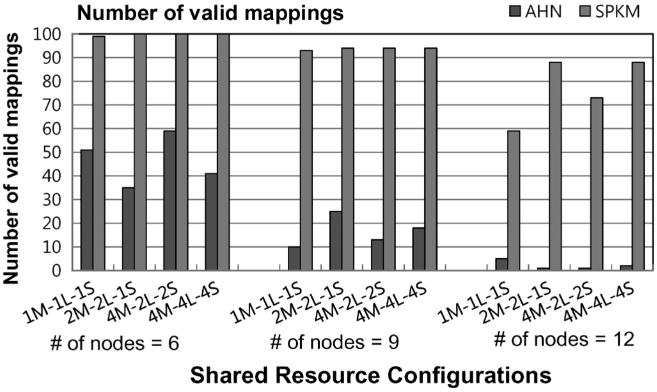


Fig. 22. SPKM can map more applications even if shared resource configuration changes.

that shares each of those resources can be mapped onto the same rows. In the 4M-4L-4S configuration, each PE has all the necessary resources and does not share the resources with other PEs. Thus, there are more opportunities to schedule the nodes, but it consumes too much power. In the exploration of the shared resource, we use the original PE interconnection topology of RSPA shown in Fig. 8.

Fig. 22 depicts the number of the valid mappings that both mapping techniques can generate, for each shared resource configurations and for the three node cardinalities. The similar observation that SPKM can map much more applications than AHN can be easily made from this graph. The interesting observation from this graph is that the shared resource constraints do not have significant impacts on the mapping capability of SPKM. This implies that SPKM maps application very effectively with any given resource constraints. However, even though AHN also considers the shared resource constraints, AHN does not *exploit* those constraints which means that AHN uses the information only to decide how many multiply, load and store operations can be mapped on the same CGRA row.

Fig. 23 plots the number of better, similar, or worse mappings generated by SPKM and AHN out of 100 applications for various shared resource configurations. This graph shows that for any resource constraints and for any node cardinalities, SPKM generates the mappings which use fewer utilized CGRA rows than AHN, thus generates mappings which shows better performance and lower power consumption.

XI. Summary and Future Work

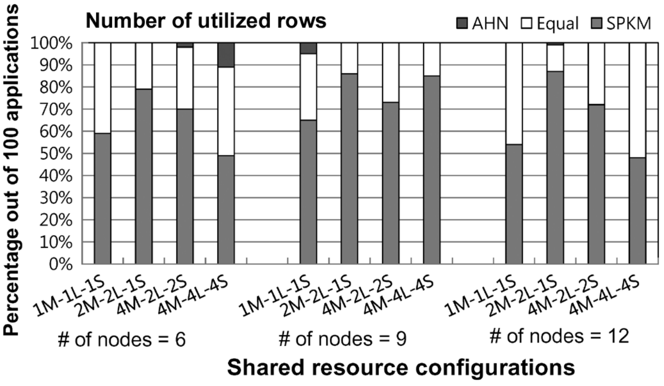


Fig. 23. SPKM also generates better quality mappings for any configurations of the shared resources.

While coarse-grained reconfigurable architectures (CGRAs) are emerging as attractive design platforms due to their efficiency as well as flexibility, efficient mapping of applications onto them still remains a challenge. Existing CGRA compilers assume a very simplistic architecture of the CGRA. In this paper, we propose a graph drawing based approach, *SPKM*, which takes several architectural details of CGRA into account and is therefore able to effectively and efficiently map applications onto CGRAs. Our experiments demonstrate that SPKM can map 4.5× more synthetic applications than the previous approach, and generate better results in 62% of cases in terms of utilized CGRA rows which is directly translated into less execution time and lower power consumption. Results on benchmarks from Livermore loops, MultiMedia and DSP-Stone also convey the same. A set of exploration experiments shows that SPKM gives these prominent results on various PE interconnection topologies and shared resource configurations, demonstrating the generality of SPKM for various CGRA designs.

Our future work is to extend the SPKM approach to include dynamic reconfigurability. We are currently working on the temporal mapping technique considering various architectural details of CGRAs. We are also in the process of automating the entire design flow, and extending the program representation to consider the control flow of the applications.

References

[1] M. Ahn, J. W. Yoon, Y. Paek, Y. Kim, M. Kiemb, and K. Choi, "A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures," in *Proc. DATE'06*, 2006, pp. 363–368.

[2] G. D. Battista, M. Patrignani, and F. Vargiu, "A splitpush approach to 3D orthogonal drawing," *Graph Drawing*, pp. 87–101, 1998.

[3] J. Becker and M. Vorbach, "Architecture, memory and interface technology integration of an industrial/academic configurable system-on-chip (CSOC)," in *Proc. ISVLSI'03*, 2003, p. 107.

[4] D. C. Chen, "Programmable arithmetic devices for high speed digital signal processing," Ph.D. thesis, Univ. California, Berkeley, CA, 1992.

[5] G. Dimitroulakos, M. D. Galanis, and C. E. Goutis, "Exploring the design space of an optimized compiler approach for mesh-like coarse-grained reconfigurable architectures," in *Proc. 20th IPDPS'06*, 2006, pp. 10–19.

[6] DongbuAnam Semiconductor. [Online]. Available: http://www.dsemi.com

[7] C. Ebeling, D. C. Cronquist, and P. Franklin, "Rapid—Reconfigurable pipelined datapath," in *Proc. FPL'96*, 1996, pp. 126–135.

[8] R. Ferreira, A. Garcia, T. Teixeira, and J. M. P. Cardoso, "A polynomial placement algorithm for data driven coarse-grained reconfigurable architectures," in *Proc. ISVLSI'07*, 2007, pp. 61–66.

[9] GNU Linear Programming Kit. [Online]. Available: http://www.gnu.org/software/glpk/

[10] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer, "Piperench: A co/processor for streaming multimedia acceleration," in *Proc. ISCA'99*, 1999, pp. 28–39.

[11] R. Hartenstein, "A decade of reconfigurable computing: A visionary retrospective," in *Proc. DATE'01*, 2001, pp. 642–649.

[12] R. W. Hartenstein and R. Kress, "A datapath synthesis system for the reconfigurable datapath architecture," in *Proc. ASP-DAC'95*, 1995, p. 77, CD-ROM.

[13] C. O. Shields, Jr., "Area efficient layouts of binary trees in grids," Ph.D thesis, Univ. Texas, Dallas, 2001.

[14] Y. Kim, M. Kiemb, C. Park, J. Jung, and K. Choi, "Resource sharing and pipelining in coarse-grained reconfigurable architecture for domainspecific optimization," in *Proc. DATE'05*, 2005, pp. 12–17.

[15] Y. Kim, I. Park, K. Choi, and Y. Paek, "Power-conscious configuration cache structure and code mapping for coarse-grained reconfigurable architecture," in *Proc. ISLPED'06*, 2006, pp. 310–315.

[16] Y.-T. Lai, H.-Y. Lai, and C.-N. Yeh, "Placement for the reconfigurable datapath architecture," in *Proc. ISCAS'05*, 2005, vol. 2, pp. 1875–1878.

[17] J. Lee, K. Choi, and N. Dutt, "Mapping loops on coarse-grain reconfigurable architectures using memory operation sharing," Ctr. for Embedded Computer Systems, Univ. California, Irvine, CA, TR 02-34, Sep. 2002.

[18] J. Lee, K. Choi, and N. Dutt, "Compilation approach for coarse-grained reconfigurable architectures," *IEEE Des. Test Comput.*, vol. 20, no. 1, pp. 26–33, Jan. 2003.

[19] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. P. Amarasinghe, "Space-time scheduling of instruction-level parallelism on a raw machine," *Architectural Support for Programming Languages and Operating Systems*, pp. 46–57, 1998.

[20] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proc. MICRO 25*, 1992, pp. 45–54.

[21] B. Mei, S. Vernalde, D. Verkest, and R. Lauwereins, "Design methodology for a tightly coupled VLIW/reconfigurable matrix architecture: A case study," in *Proc. DATE'04*, 2004, pp. 1224–1229.

[22] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarsegrained reconfigurable architectures using modulo scheduling," in *Proc. MICRO 31*, 2003, pp. 296–301.

[23] E. Mirsky and A. DeHon, "MATRIX: A reconfigurable computing architecture with configurable instruction distribution and deployable resources," in *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, 1996, pp. 157–166.

[24] T. Miyamori and K. Olukotun, "REMARC: Reconfigurable multimedia array coprocessor (abstract)," in *FPGA*, 1998, p. 261.

[25] ModelSimSE 6.5. Model Technology Corp. [Online]. Available: http://www.model.com/products/products_se.asp

[26] A. Paar, M. L. Anido, and N. Bagherzadeh, "A novel predication scheme for a SIMD system-on-chip," in *Proc. Euro-Par'02*, 2002, pp. 834–843.

[27] H. Park, K. Fan, M. Kudlur, and S. Mahlke, "Modulo graph embedding: Mapping applications onto coarse-grained reconfigurable architectures," in *Proc. CASES'06*, 2006, pp. 136–146.

[28] M. Patrignani and M. Pizzonia, "The complexity of the matching-cut problem," in *Proc. 27th Int. Workshop on Graph-Theoretic Concepts in Computer Science (WG'01)*, 2001, pp. 284–295.

[29] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proc. MICRO 27*, 1994, pp. 63–74.

[30] M. Schimmler, H.-W. Lang, and R. Maa, "The instruction systolic array—Implementation of a low-cost parallel architecture as add-on board for personal computers," in *Proc. Int. Conf. and Exhibition on High-Performance Computing and Networking (HPCN Europe 1994)*, 1994, vol. 2, pp. 487–488.

[31] H. Singh, M.-H. Lee, G. Lu, N. Bagherzadeh, F. J. Kurdahi, and E. M. C. Filho, "Morphosys: An integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Trans. Comput.*, vol. 49, no. 5, pp. 465–481, May 2000.

[32] Synopsys Corp. [Online]. Available: http://www.synopsys.com

[33] 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 *Proc. CASES'01*, 2001, pp. 116–125.

[34] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, "Baring it all to software: Raw machines," *Computer*, vol. 30, no. 9, pp. 86–93, 1997.

Jonghee W. Yoon received the B.S. degree in electrical engineering from Korea Advanced Institute of Science and Technology, Daejeon, Korea, in 2005. He is currently working towards the Ph.D. degree in the School of Electrical Engineering and Computer Science, Seoul National University, Seoul, Korea.

He was a visiting researcher at the Compiler Micro-architecture Laboratory of Arizona State University, Tempe, in 2007. His current research interests are embedded software, reconfigurable array processor, and MPSoC.

Mr. Yoon received a Best Paper Award at WASP 2007.

Aviral Shrivastava received the Bachelors degree in computer science and engineering from the Indian Institute of Technology, Delhi, and the Masters and Ph.D. degrees in information and computer science from the University of California, Irvine.

He is an Assistant Professor in the Department of Computer Science and Engineering, Arizona State University, Tempe, where he established and heads the Compiler and Microarchitecture Labs (CML). His research interests lie at the intersection of compilers and computer architecture, with a particular interest in embedded systems.

Dr. Shrivastava is a lifetime member of the ACM, and serves on organizing and program committees of several premier embedded system conferences, including CODES+ISSS, CASES and LCTES.

Sanghyun Park received the B.S. degree in electrical engineering from Seoul National University, Seoul, Korea, in 2004, where he is currently working towards the Ph.D. degree in the School of Electrical Engineering and Computer Science.

He was a visiting researcher at the ACES group of the University of California, Irvine, in 2005 and at the CML group of Arizona State University in 2007. His research interests are in the compiler framework for multiprocessor system-on-chip, compiler-driven embedded system design, coarse-grained reconfigurable architecture, and design methodologies for low power.

Mr. Park received a Best Paper Award at LCTES 2006 and WASP 2007.

Minwook Ahn received the Bachelors degree in electrical engineering from Seoul National University, Seoul, Korea. He is working towards the Ph.D. degree in the Department of Electrical Engineering and Computer Science, Seoul National University, where he has studied in the Software Optimizations and Restructuring Labs (SO&R). His research interests lie in software optimizations, compilers and computer architecture, especially for embedded systems.

Yunheung Paek received the B.S. and M.S. degrees in computer engineering from Seoul National University, Seoul, Korea, and the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign (UIUC).

In the spring of 2003, he joined Seoul National University as an Associate Professor in the School of Electrical Engineering. Before he came to Seoul National University, he had been in the Department of Electrical Engineering at Korea Advanced Institute of Science and Technology (KAIST) as an Associate Professor for one year and an Assistant Professor for two and a half years. He has served on the program committees and organizing committees for numerous conferences, and also worked for associate editors or reviewers for various journals and transactions. His current research interests are embedded software, embedded system development tools, retargetable compiler and MPSoC.