Efficient Mapping onto Coarse-Grained Reconfigurable Architectures using Graph Drawing based Algorithm

Jonghee Yoon
Seoul National University, Korea
jhyoon@optimizer.snu.ac.kr

Aviral Shrivastava
Arizona State University
Aviral.Shrivastava@asu.edu

Minwook Ahn
Seoul National University, Korea
mwahn@optimizer.snu.ac.kr

Sanghyun Park
Seoul National University, Korea
shpark@optimizer.snu.ac.kr

Doosan Cho
Seoul National University, Korea
dscho@optimizer.snu.ac.kr

Yunheung Paek
Seoul National University, Korea
ypaek@ee.snu.ac.kr

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 architecture, due to which they are, i) unable to map applications, even though a mapping exists, and ii) use too many PEs to map an application. In this paper, we model several CGRA details in our compiler and develop a graph mapping based approach (SPKM) for mapping applications onto CGRAs. On randomly generated graphs our technique can map on average 3.6X more applications than the previous approaches, while showing shorter execution cycle 66% times and less power consumption 71%, with less mapping time. We observe similar results on a suite of benchmarks collected from Livermore Loops, Multimedia and DSPStone benchmarks.

1. Introduction

Coarse Grain Reconfigurable Arrays (CGRAs) have emerged as a promising reconfigurable platform, by providing operation level programmability, word level datapaths, and powerful and very area-efficient datapath routing switches. CGRAs are essentially a set of processing elements (PEs), such as ALUs, or multipliers. The PEs are connected so that they can use the result of its neighboring PEs. It is the PE function selection and data routing that provides CGRAs with the reconfigurability. CGRAs can fully exploit the parallelism in an application, and therefore, they are extremely well suited for applications that need high throughput including multimedia, signal processing, networking, and other scientific applications. Several CGRA implementations like MorphoSys, RSPA, KressArray etc. have been proposed. Comprehensive summary of CGRAs can be found in [6].

However, the success of CGRAs critically hinges on the efficient mapping of applications onto it so as to exploit the parallelism in the application and utilize minimum computation resources of the CGRA. Minimizing the number of computation resources in CGRAs is an extremely important goal, as it directly translates into either reduced power consumption, or increased throughput. The problem of mapping an application onto a CGRA to minimize the number of resources used has been shown to be NP-complete, and therefore several heuristics have been proposed. However existing heuristics do not consider the details of the CGRA architecture. In particular,

PE Interconnection Most existing CGRA compilers assume that the PEs in the CGRA are connected only in a simple 2-D mesh fashion, 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 neighbor. In Morphosys, a PE is connected to every other PEs through shared buses, in RSPA, each PE is connected to the immediate neighbors and the next neighbors also.

Shared Resources Most CGRA compilers assume that all PEs are similar in the sense, that an operation can be mapped to any PE. However, modern CGRAs, in order to reduce the cost, power and complexity, do not include the multiplier in each CGRA. Few multipliers are made available as shared resources in each row. For example, RSPA has 2 shared multipliers in each row.

Routing PE Most CGRA compilers cannot use a PE just for routing. This implies that in a 4x4 CGRA, it is not possible to map application DAGs in which any node has more than 4 degree. However, most CGRAs allow a PE to be used for routing only, which makes it possible to map any degree

DAGs onto the CGRA.

Owing to the simplistic model of CGRA architecture in the compiler, existing CGRA compilers are i) unable to map an application on the CGRA, even though it is possible to map them. ii) uses too many PEs in the solution. Thus even if the mapping exists, it consumes more power on CGRAs.

In an attempt to develop better heuristics for mapping of applications to CGRAs, we observe the similarity of application mapping to the problem of graph drawing. We translate our problem into the problem of 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:

- 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 3.6X more randomly generated application DAGs than previous approach, while showing shorter execution cycle 66% times and less power consumption 71% times, with less mapping time.

The rest of the paper is organized as follows: Section 2 and Section 3 formally define the problem of mapping applications onto CGRAs. Section 4 describes CGRAs and the previously proposed heuristics for mapping applications onto CGRAs. In Section 5, we describe our graph drawing based solution, and Section 7 demonstrates the effectiveness of our solution. We summarize and conclude in Section 8.

2. Notation and Definitions

Since applications spend most of their time and power in loops, in this paper, we focus on mapping loops to CGRAs. Significant power and performance trade-offs can be made by unrolling the loop. Therefore, the first step in mapping applications onto CGRAs is to first unroll the loops to meet the power and performance requirements. The focus in this paper is solving the problem of mapping the kernel of a given loop to a CGRA while minimizing the number of resources required.

Loop kernel The loop kernel can be represented as a Directed Acyclic Graph (DAG), $K = (V, E)$, where the set of vertices V are the operations in the loop, and for any two vertices, $u, v \in V$, $e = (u, v) \in E$ iff the operation corresponding to 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 $p, q \in P$, $l = (p, q) \in L$ iff PE q can use the result that PE p computed in the previous cycle.

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

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

ϕ_E is a multi-valued function that maps data dependency 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 $p_1 = \phi_V(u)$, and $\exists l_2 = (p_2, q_2) \in ll$ such that $q_2 = \phi_V(v)$, and $\forall l = (p, q) \in ll$ such that $q \neq \phi_V(v)$ then $\exists l = (p', q') \in ll$, such that $p' = q$.

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 $\phi_V(v)$, described by ll.

Routing Order Under the path existence and the simple path constraint, a total order can be defined on the elements of ll. This total order, which we call routing order, and identified by $\prec$, identifies a unique path from $\phi_V(u)$ to $\phi_V(v)$. The total 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 one 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 increasing order, as defined by $\prec$, they describe the path from $\phi_V(u)$ to $\phi_V(v)$. 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 the 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 routing PE.

Shared Resource Constraint Most CGRAs have row-wise constraints, that arise from the fact that the rows share expensive resources, e.g. multipliers, memory buses etc. For example, there can be only two multiply operations in each row. To specify such constraints, we define an attribute “type” to each vertex of the kernel graph K and the number of shared resources, type t, within a row in C as

S_t. Thus, $\forall v^t \in \{v \in V | v.type = t\}$ and $p^t_{ij} = \phi_V(v^t)$, $\sum_j |p^t_{ij}| \le S_t$.

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

3. Problem Formulation

Although the previously considered objective for the problem has been to minimize the number of PEs used. However, in practice, owing to the severe restrictions of the shared resource constraints, utilizing less number of rows is the most useful objective function. Utilizing lesser number of rows directly translates into increased opportunities for novel power and performance optimization techniques. For example, to reduce the power consumption, a whole row of PEs may be power gated. In addition it might be possible to cleanly execute another application on the remaining 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 arbitrary interconnection links in the CGRA.
- We allow shared resource constraints which can be used to model resource constraint operations, e.g., loads, stores, multiplications and division.
- We allow a data dependence edge to be mapped onto a set of interconnection links of the CGRA, and therefore allowing the use of PEs as routing resources.

4. Related Work

Various CGRAs have been proposed as summarized in [6]. MorphoSys [15] consists of an 8×8 array of reconfigurable cells coupled with a Tiny RISC processor. The cell array performs 16-bits SIMD-style operations and each cell has its own multiplier. RSPA [7] is based on MorphoSys, but it is not SIMD but MIMD-style. Furthermore, RSPA shares multiplier in order to reduce the chip size. REMARC [11] is 8×8 array of nano processors containing an ALU, data RAM, register file, and special purpose registers. ADRES [9] has an XML-based architecture description language to define the overall topology, operation set, resource allocation, timing, and an internal organization of each PE. Another CGRA is XPP [3]. It has 4×4 or 8×8 PE array and each PE has no multiplier.

The performance of CGRA critically hinges on the mapping algorithm. For MorphoSys, a compiler framework [17] to analyze SA-C programs, perform optimizations, and map the application was proposed. XPP has mapping algorithm described in [5]. However, their approach was evaluated only for simple loops. Another approach is DRESC [10] for ADRES architecture template. They exploit loop level parallelism by adopting modulo scheduling. This approach takes very long time for mapping and mapping results shows low utilization of PEs. In order to improve utilization, similar approach [13] using affinity graph was proposed.

However, 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.

[8] proposes a spatial mapping approach, which considers memory interface sharing of rows. 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 called AHN in this paper. However, they only consider 2-D mesh PE interconnections, in which a PE is connected to neighboring PEs only, and do not use routing PEs. In addition, their inputs are limited to trees and not DAGs, which is the more general and commonly accepted form of application specification. Since all trees are planar, their technique has not been tested for non-planar graphs. Note that non-planar graphs cannot be mapped onto 2-D mesh, without using a PE only for routing.

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

Our approach of mapping a kernel onto a CGRA is based on the split & push algorithm [2] in graph drawing. Figure 1 shows an example of mapping 4-operation kernel K graph onto a mesh CGRA C.

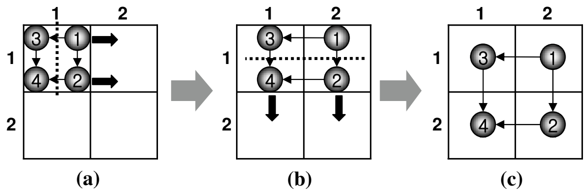


Figure 1. Split & Push Approach

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 Figure 1(a). In the first step, we locate each $v \in K$ using *cuts*. A cut is a plane orthogonal to one of the axis. It is shown by a dotted line in Figure 1. The cut *splits* all $v \in K$ into two groups. All v in one of the groups are *pushed* to new coordinate. Figure 1(b) shows the result of *split & push* along the dotted line. This split & push is repeated

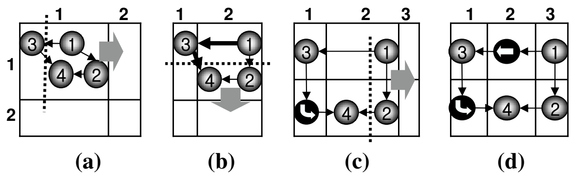
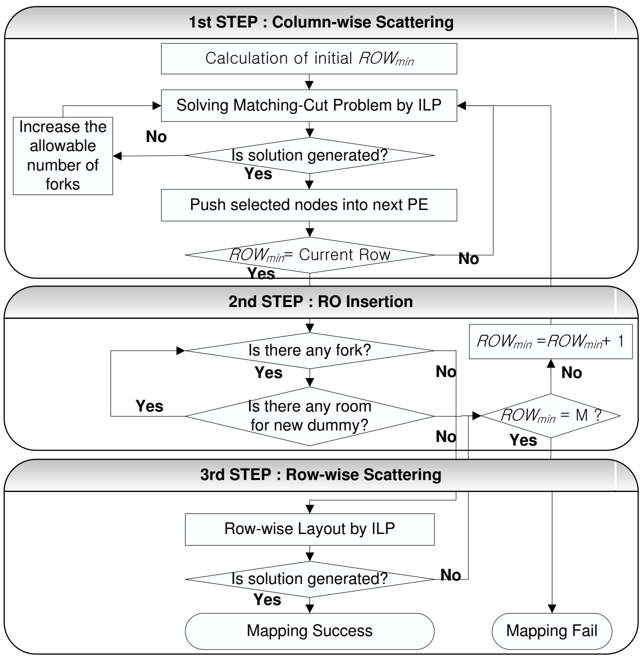


Figure 2. Formation of fork



1st STEP : Column-wise Scattering
Calculation of initial ROW_{min}
Solving Matching-Cut Problem by ILP
Increase the allowable number of forks
No
Is solution generated?
Yes
Push selected nodes into next PE
ROW_{min}= Current Row
No
Yes
2nd STEP : RO Insertion
Is there any fork?
Yes
No
Is there any room for new dummy?
Yes
No
ROW_{min} =ROW_{min}+ 1
No
ROW_{min} = M ?
Yes
3rd STEP : Row-wise Scattering
Row-wise Layout by ILP
Is solution generated?
No
Yes
Mapping Success
Mapping Fail

Figure 3. SPKM : Split and Push Kernel Mapping

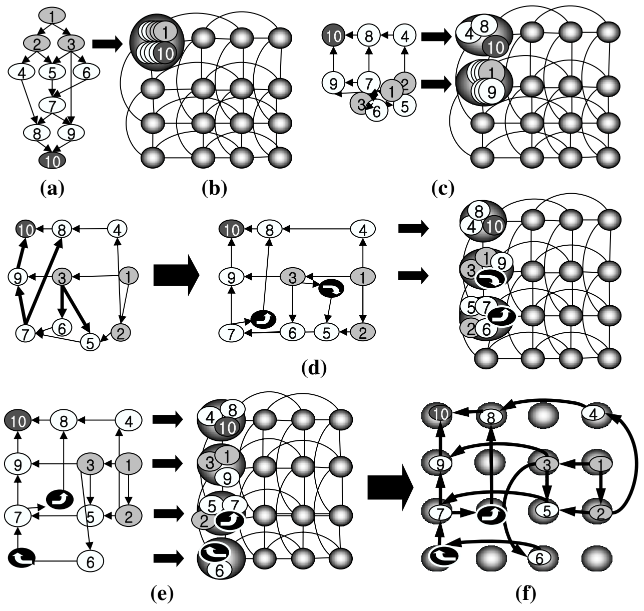


Figure 4. Mapping process example

until every v has distinct coordinate like the drawing shown in Figure 1(c).

The crucial step in the split & push approach is finding a suitable cut. Consider the application of split & push applied to the same K in Figure 2. The final graph requires much more PEs than the solution in Figure 1. 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 both cut by a split. Once there is a fork and the fork consists of n adjacent edges, $n-1$ bends (or dummy nodes, or routing vertices(PEs)) 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 [14].

In order to minimize the number of utilized rows in the mapping, we propose a three stage heuristic based on the split & push approach. Figure 3 describes our mapping technique, *Split-Push Kernel Mapping*, or *SPKM*. Following three subsections explain SPKM by mapping the kernel graph $K=(V,E)$ shown in Figure 4(a), onto a 4×4 mesh CGRA $C=(P,L)$, shown in Figure 4(b).

We assume that in C, all $p\in P$ are connected to their first, as well as their second horizontal or vertical neighbors. Thus a PE is connected to at most 6 other PEs. We also assume that in a row, at most 2 load operations and one store operation can be scheduled. In K of Figure 4(a), we have ten operations including three loads (gray nodes) and one store (dark grey node).

5.1 Column-wise Scattering

This step distributes vertices v in V to minimum number of UR in the same column considering minimum number of forks and shared operations like multiply and load/store. First we compute the lower bound on $|UR|$ in the CGRA, as in $|UR|_{min}=max(\lceil |V|/|N|\rceil,\lceil L/L_r\rceil,\lceil S/S_r\rceil)=max(\lceil 10/4\rceil,\lceil 3/2\rceil,\lceil 1/1\rceil)=3$. We try to map K onto three rows of C. We distribute all v in K to p_{k1}, $1\le k\le |UR|_{min}$. All v located at p_{k1} are separated into two sets and the nodes in one of the sets are pushed into $p_{(k+1)1}$. For example, all the nodes in p_{11} of Figure 4(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 the Figure 4(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 $|UR|_{min}$. In each repetition, a matching cut is found 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 k^{th} repetition, the graph K^k consisting of all the nodes in p_{k1} are split into two disconnected 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.

Objective Function

$$d=|\sum_{v\in V^k} v_{ik1}-\xi| \quad (1)$$

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^k to all p_{k1}

within $|UR|_{min}$ gives better chance for getting an optimal mapping, we set $\xi = \lfloor (N + |UR|_{min} - 1)/|UR|_{min} \rfloor$.

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 Figure 4(a) has one load primitive operation inside. So s_{111} is 1. In our CGRA, p_{kl} within one row share two read buses, S is 2.
- To minimize the forks, we have another constraint for all v_i^m with multiple edges in K^k.

$$\sum_{v_i \in V^k} v_{ik1}^t \leq S_t \tag{2}$$

$$\sum_{v_j \in adj(v_i^m)} (v_{jk1} + v_{ik1}^m) \leq \zeta_1 \ \ or$$

$$\sum_{v_j \in adj(v_i^m)} (v_{jk1} + v_{ik1}^m) \geq 2 \cdot deg(v_{ik1^m}) - \zeta_2 \tag{3}$$

where 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.

Equation 11 is not linear due to "or". In order to linearize this, we change this equation to Equation 4 and 5 using new constant η which is big enough.

$$\sum_{v_j \in adj(v_i^m)} (v_{jk1} + v_{ik1}^m) \leq \zeta_1, + \eta \cdot v_{ik1}^m \tag{4}$$

$$\sum_{v_j \in adj(v_i^m)} (v_{jk1} + v_{ik1}^m) \geq 2 \cdot deg(v_{ik1^m}) - \zeta_2 - \eta \cdot (1 - v_{ik1}^m) \tag{5}$$

When a *muti-degree-node* v_i^m, black node in Figure 5 and its adjacent nodes in K^k are determined to be pushed into $p_{(k+1)l}$, there are four ways to avoid generating forks. Figure 5(a) and (b) show two possible cases where v_i^m is not pushed into $p_{(k+1)l}$ and $deg(v_i^m)$ or $deg(v_i^m)$-1 adjacent nodes are not pushed either. Figure 5(c) and (d) show the other cases where v_i^m is pushed into $p_{(k+1)l}$ and only 0 or 1 adjacent node is not pushed. Thus, if we want to allow no forks, we set ζ_1 = ζ_2 = 1. If there is no matching cut in the k^{th} repetition of split & push, we increase ζ_1 and ζ_2, allowing more forks.

The left kernel DAG K in Figure 4(d) shows the result of column-wise scattering. Because $|UR|_{min}$ is three, it attempts a mapping within three rows. Fortunately, it has a valid mapping in three rows within C in the end, instead it allows two forks.

5.2 Routing PE Insertion

After finishing column-wise scattering above, routing vertices should be inserted on each edge of each fork generated in the first stage of SPKM to route data via indirectly connected PE. In this step, we generate needed routing vertices and connect them with existing vertices. As (v_7, v_8) is

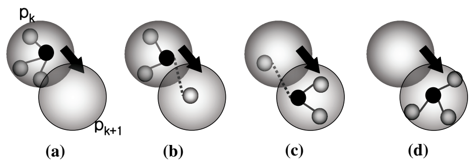


Figure 5. Fork minimization algorithm

an edge of the first fork in Figure 4(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 there are no available PEs after routing PEs are inserted into K. For instance, the right K with routing PEs in Figure 4(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 with increasing $|UR|_{min}$ by one. Figure 4(e) shows the result of column-wise scattering with $|UR|_{min}$ 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.

5.3 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 as well as 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 Figure 4(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.

6. Experimental Setup

We test SPKM on a CGRA called RSPA [7] which is shown in Figure 6. RSPA consists of 16 PEs in which each PE is connected to 4 neighboring PEs, and also the 4 next neighboring PEs (PE interconnection). In addition, it has 2 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 operation. It should be noted here that we modify RSPA to have

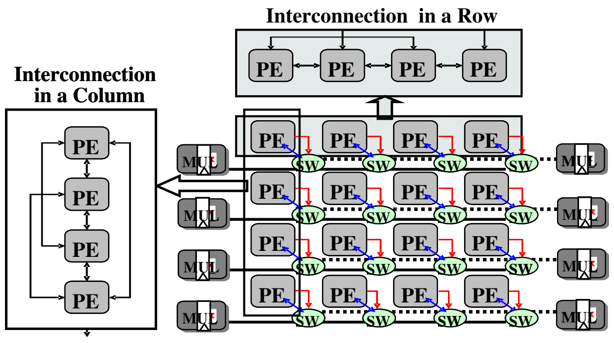


Figure 6. Target architecture : RSPA

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

Table 1. Power Parameters

6x4 structure for the fair comparison with the previous approach AHN, which is briefly described in Section 4. More details of the technique can be found in [1].

For quantitative estimation of the effectiveness of SPKM, we have devised a random kernel DAG generator. Our DAG generator randomly generated 100 DAGs are for each value of node cardinality from 5 to 16 nodes (1200 in total). AHN can also take DAGs as inputs since the features of baseline architecture RSPA allow AHN to map DAGs also. Each DAG is generated according to following steps. First the number of nodes is set. For each node, the operations possible in each PE of RSPA are randomly assigned. At least one load operation should be assigned to leaf vertices and one store operation to root vertices respectably. Finally edges are inserted, satisfying that each node should not 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. In order to get 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 the most number of times and requires the least number of RPEs. All experiments are done on Pentium4 3GHz machines with 1GB RAM. We use glpk4.8 for solving our ILP formulations.

We have analyzed the power consumption of PE and configuration cache of the reconfigurable architecture. The architecture has been synthesized using Design Compiler of Synopsys [16] with technology of DongbuAnam 0.18 μm [4]. We have used SRAM Macro Cell library for the frame buffer and configuration cache. ModelSim [12] and PrimePower [16] have been used for gate-level simulation and power estimation. The operation frequency was 100MHz and Vdd 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 1.

7. 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 experiment into four parts. Firstly, we show that SPKM is able to map much more applications than the previous one. Secondly, we show that we can generate better quality mappings than the previous technique, in terms of the execution time and the power consumption of mapped applications. SPKM, can achieve those outperforming results with minimal mapping-time overhead, and it is described next. And finally, we show that SPKM is able to map real benchmarks as well. All of our experimental result is based on the comparison with another spatial mapping technique, AHN [1], because it is the only spatial mapping technique for CGRA.

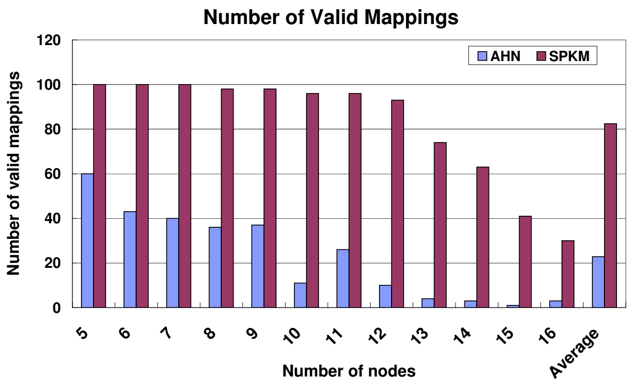


Figure 7. Number of applications mapped on CGRA validly

7.1 SPKM can map more applications

Figure 7 plots how many applications out of 100 applications can be mapped by the two 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. The main observation from this graph is that SPKM can on average map 3.6X more applications than AHN. It is interesting to note that the graph shows that the map-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.

7.2 SPKM can generate better mappings

In addition to being able to map more application than AHN, SPKM is also able to generate better mappings for the applications, in terms of execution time and power consumption. As described in Section 3, the objective of our technique is to

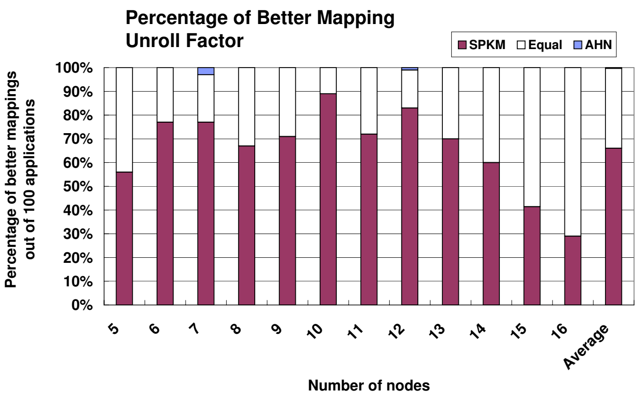


Figure 8. Percentage of better mapping for SPKM and AHN in terms of unroll factor

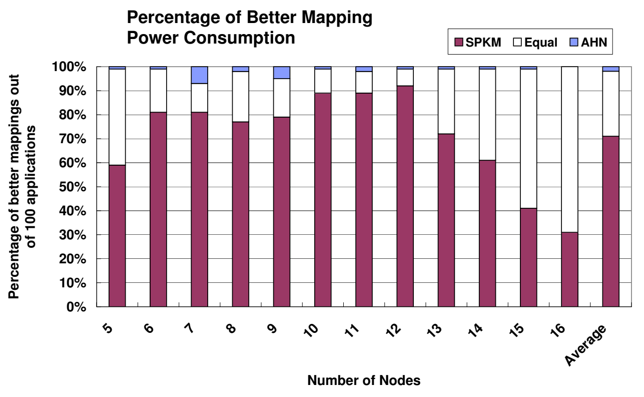


Figure 9. Percentage of better mapping for SPKM and AHN in terms of power consumption

minimize the utilized CGRA rows. The less the utilized rows are, the more opportunities there are to boost up the performance by using more loop-unrolled kernels. Therefore using unroll factor for each mapping, we can compare the execution time for each algorithm. Moreover, if we can use less number of rows, there are more opportunities to reduce the power consumption of PE arrays by switching the unused PEs to low power mode. With the sense of better mapping as need of less rows, Figure 8 plots how many applications out of 100, better, similar, or worse mapping is generated by SPKM and AHN, in terms of unroll factors. The white bars represent the number of applications in which SPKM and AHN maps with the same unroll factors. For example, for the 100 applications which have 12 nodes, SPKM can generate 83 mappings which have more unroll factors than AHN. Among the rest of the applications, SPKM and AHN generate the same quality mapping for 17 applications, and AHN has only one better mapping. On average, SPKM can generate better mappings than AHN for 66% of the applications, and the same quality mappings for 33% of the applications. This implies that for the 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.

Figure 9 plots in how many applications out of 100 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 with less power. Figure 9 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 the 98% of the applications, SPKM is able to generate better, or at least the same quality mappings than AHN.

7.3 SPKM has less mapping time

SPKM generates effective mapping described above with less mapping time. Figure 10 shows the average mapping time for SPKM and AHN. This execution time is measured with not-unrolled input kernels. On average, SPKM has 8% less mapping time as compared to AHN.

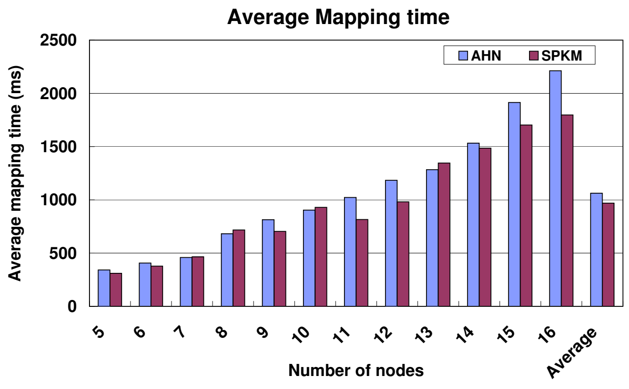


Figure 10. Average mapping time

7.4 Real Benchmarks

To demonstrate the effectiveness and usefulness of SPKM, we compare the number of rows and the mapping time of SPKM and AHN for a set of benchmarks collected from Livermore loops, multimedia, and DSPStone.

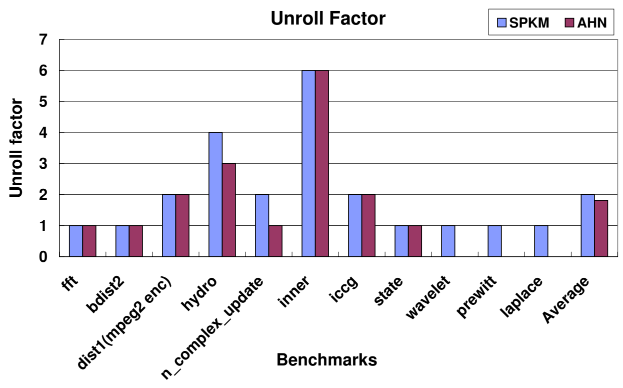


Figure 11. Unroll factor for real benchmarks

Figure 11 shows the unroll factor of SPKM and AHN which is directly related to the performance of the mapped applications. The first observation from this graph is that AHN is unable to map three of the applications, demonstrating that SPKM can map more applications than AHN. The second observation is that the mappings generated by SPKM

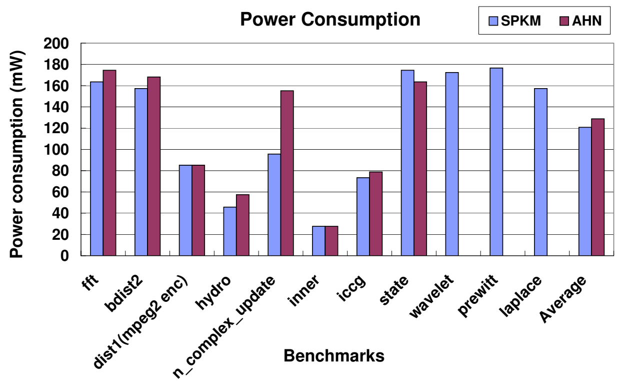


Figure 12. Power Consumption for real benchmarks

can map applications with larger, or at least the same unroll factors. Figure 12 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. For the benchmarks that AHN could map, SPKM uses 0.8% less time for mapping than AHN.

8. Summary

While coarse-grained reconfigurable architectures (CGRAs) is 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, due to which they are unable and ineffective. In this paper, we propose a graph drawing based based approach, *SPKM*, which takes in to account several architectural details of CGRA and is therefore able to effectively and efficiently map applications onto CGRAs. Our experiment demonstrate that *SPKM* can map 3.6X more synthetic applications than previous approach, and generates better results in 66% of cases in terms of execution time and power consumption, with less (8%) mapping-time. Results on benchmarks from Livermore loops, MultiMedia and DSPStone also convey the same. Our future work is to extend the *SPKM* approach to include dynamic reconfigurability.

9. Acknowledgement

This work was partially funded by Intel Corporation, UC Micro (03-028), SRC (Contract 2003-HJ-1111), and NSF (Grants CCR-0203813 and CCR-0205712), MIC (Ministry of Information and Communication), Korea, under the ITRC (Inofrmation Technology Research Center) support program supervised by the IITA (Institute of Information Technology Assessment) (IITA-2006-C1090-0603-0020), "System IC 2010" project of Korea Ministry of Science and Technology and Ministry of Commerce, Industry and Energy, ETRI SoC Industry Promotion Center, Human Resource Development Project for IT SoC Architect, and Nano IP/SoC Promotion Group of Seoul R&BD Program in 2006.

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 *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 363–368, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[2] G. D. Battista, M. Patrignani, and F. Vargiu. A splitpush approach to 3d orthogonal drawing. In *Graph Drawing*, pages 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 *ISVLSI '03: Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'03)*, page 107, Washington, DC, USA, 2003. IEEE Computer Society.

[4] DongbuAnam Semiconductor. *http://www.dsemi.com*.

[5] F. Hannig, H. Dutta, and J. Teich. Mapping of regular nested loop programs to coarse-grained reconfigurable arrays - constraints and methodology. In *IPDPS 2004: In Proceedings of the 18th International Parallel and Distributed Processing Symposium*, Washington, DC, USA, 2004. IEE Computer Society.

[6] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pages 642–649, Piscataway, NJ, USA, 2001. IEEE Press.

[7] 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 *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 12–17, Washington, DC, USA, 2005. IEEE Computer Society.

[8] J. Lee, K. Choi, and N. Dutt. Mapping loops on coarse-grain reconfigurable architectures using memory operation sharing, 2002.

[9] B. Mei, S. Vernalde, D. Verkest, and R. Lauwereins. Design methodology for a tightly coupled vliw/reconfigurable matrix architecture: A case study. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 21224, Washington, DC, USA, 2004. IEEE Computer Society.

[10] B. Mei, S. Vernalde, D. Verkest, H. Man, and R. Lauwereins. Dresc: A retargetable compiler for coarse-grained reconfigurable architectures, in international conference on field programmable technology, 2002., 2002.

[11] T. Miyamori and K. Olukotun. REMARC: Reconfigurable multimedia array coprocessor (abstract). In *FPGA*, page 261, 1998.

[12] Model Technology Corp. *http://www.model.com*.

[13] H. Park, K. Fan, M. Kudlur, and S. Mahlke. Modulo graph embedding: mapping applications onto coarse-grained reconfigurable architectures. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 136–146, New York, NY, USA, 2006. ACM Press.

[14] M. Patrignani and M. Pizzonia. The complexity of the matching-cut problem. In *WG '01: Proceedings of the 27th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 284–295, London, UK, 2001. Springer-Verlag.

[15] 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.*, 49(5):465–481, 2000.

[16] Synopsys Corp. *http://www.synopsys.com*.

[17] 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 *CASES '01: Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 116–125, New York, NY, USA, 2001. ACM Press.