Loop Kernel Pipelining Mapping onto Coarse-Grained Reconfigurable Architecture for Data-Intensive Applications

Dawei Wang
College of Computer Science, University of Defense Technology, Changsha, P.R.China
Email: daweiwang@nudt.edu.cn

Sikun Li and Yong Dou
College of Computer Science, University of Defense Technology, Changsha, P.R.China
Email: lisikun@263.net.cn, yongdou@163.net

***Abstract*—Coarse-grained reconfigurable architectures (CGRA) provide flexible and efficient solution for data-intensive applications. Loop kernels of these applications always consume much execution time of the whole program. However, mapping loop kernels onto CGRA is still hard to meet performance/cost constraints. This paper proposes a novel approach for automatically mapping loop kernels onto CGRA with loop self-pipelining to optimize data-intensive applications. The problem formulation is shown first. Then we present the resource sharing and pipelining of lspCGRA, together with its template standard. Further, a loop kernel pipelining mapping is proposed. The conclusions show that our approach gains less resource occupation by 16.3% and more throughputs by 169.1% than previous advanced SPKM.**

***Index Terms*—Reconfigurable computing, loop self-pipelining, data-intensive application, design decision**

I. INTRODUCTION

Reconfigurable computing has the efficiency of custom computation and flexibility of common computation, so it can accelerate most kinds of applications [1] [2]. CGRA can provide more speedup and less energy cost. However, although CGRA can provide high performance and flexibility, it still remains a hard problem to meeting the severe constraints on performance and cost while developing some field specific applications.

CGRA provide efficient and convenient solutions to speed-up data intensive applications (DIA), such as image compression, pattern recognition and digital signal processing. Currently various CGRAs have been developed for mapping applications onto them. Some of CGRAs use special hardware accelerator to execute critical loops, which lose more flexibility [3] [4]. Most approaches use some pipelining techniques in high level compiling, which gain less speedup with complex compiling labor [5] [6].

It needs high throughput and parallel to handle DIA, while it is inclined to errors and time-consuming to map DIA onto CGRA manually. Besides, there are many loop kernels which consume much system resources. It's necessary to speed up and optimize these critical loop kernels. Due to the insufficient work of mapping loop kernels onto CGRAs, this paper proposes a novel loop kernel pipelining mapping onto coarse-grained reconfigurable architectures. We developed a loop self-pipelining coarse-grained reconfigurable architecture (lspCGRA) [16]. The lspCGRA uses fix instruction multiple dataflow (FIMD), which can support loop self-pipelining and gain high computation throughput. Besides, we show the details about the share resource and pipelining features of lspCGRA templates. A novel loop kernel pipelining mapping (LKPM) approach is presented together. The contributions of this paper are shown as follows:

- We formulate the loop kernel pipelining mapping problem onto CGRA. Rather than existing approaches, ours support simultaneously mapping of both computation and storage. The throughput can be greatly improved by some strategies, such as loop unrolling & pipelining, conflict avoidance and data reuse.
- The performance of our mapping approach is close to that of manual mapping and optimizing. Besides, LKPM shows less resource occupation by 16.3% and higher throughput by 169.1% than advanced SPKM [5].

The rest of the paper is organized as follows: Section II overviews the previous work. Section III defines the problem formulation of automatically mapping loop kernels onto lspCGRA. In Section IV we describe lspCGRA template. In Section V, we propose field specific application driven mapping and loop kernel pipelining mapping approach. Section VI shows the experiments to prove the efficiency of our approach. Finally, in Section VII we present the conclusions.

II. RELATED WORK

In recent years many optimization techniques have been brought into hardware architectures for exploiting parallelism. Besides, reconfigurable processors are popular to speed up field specific applications with fine flexibility and efficiency. CGRA provide well solutions to develop the parallel of data-intensive applications [9] [10]. However, many loop kernels of

the applications consume much percent of execution time. Thus developing loop level parallel becomes very important for execution of DIA application programs [3] [11].

The performances of CGRA rely on the architecture templates and the mapping strategy. XPP use vector C compiler namely XPP-VC to support automatic loop unrolling, which use parallel dataflow to handle critical inner loop and outer loop simultaneously [9]. Mophosys compiler can analyze SA-C program and optimize image applications [10]. However, both [9] and [10] can deal with only simple loops. Lee [12] proposes automatically mapping loop onto dynamic reconfigurable ALU array (DRAA). The array uses common template to configure a broad range of CGRA. However, the CGRA compilers become very complex due to its numerous architectural features and their inter-dependency. Rong [13] use single soft pipelining to process multi-dimension loops, which reduces the problem of n-dimensional software pipelining into a simpler problem of one-dimensional software pipelining, but the approaches only support the IA-64 architecture. Besides, both [12] and [13] do not consider the optimization of mapping DIA onto their architectures.

On the area of architecture template, Kim [14] is close to ours, which consider the resource pipelining of CGRA to handle some field specific optimization. However, it does not further consider loop pipelining and the field of DIA.

On the area of mapping strategy, SPKM [5] and Spatial [15] can efficiently map most applications onto CGRA automatically, which can gain close results to manual mapping and achieve higher resource utilization ratio. However, when they handle DIA, few strategies are used, such as data reuse, storage bottleneck avoidance.

Besides, existing mapping approaches consider only simple CGRA models but not more complex models such as lspCGRA to process DIA. While our approach use lspCGRA, which can support loop self-pipelining and automatic loop iteration. High level expression loop sentences can be mapped onto PEs directly to improve the resource utilization ratio and throughput efficiently.

III. Problem Formulation For Mapping Loop Kernel onto lspCGRA

DIA needs high data parallel, regular and throughput. When dealing with DIA, we find that some critical blocks of application programs, such as loops, consume many system resources. So it is necessary to identify and extract them for speeding up and optimizing.

When mapping DIA onto lspCGRA, we use program information aided control-dataflow task graph (PIA-CDTG) to represent the function and behavior of applications, use virtual instruction data flow graph (vi-DFG) to represent critical loop kernels, and use reconfigurable architecture graph (RAG) to represent lspCGRA template. The problem formulation for mapping vi-DFG onto RAG is shown as follows:

Definition 1 (PIA-CDTG). PIA-CDTG = (T, E, I, O, *Info*) consists of node T, edge E, input/output of nodes and program profiling information. T is the node set. $E \subseteq (O \times I)$ is the edge set. Info includes dataflow/control flow, computation/storage distribution, the dependency and relevant of nodes.

PIA-CDTG model defines architecture-independent information about field specific application program. We can get the information by program profiling and speed up the critical blocks.

Definition 2 (vi-DFG). vi-DFG = (T, E) consists of node T and edge E, which is represented by virtual instruction dataflow graph. T describes virtual instruction instance defined by Kahn process. $E \subseteq (T \times T)$ describes the channels that connect process element nodes, such as loop start/end, data relevant, data copy and write control.

vi-DFG describes the dataflow execution semantic of loop kernels, which does not consider the implement of architecture.

Definition 3 (RAG). $M \times N$ lspCGRA can be represented by RAG = (PE, C). PE_{ij} of lspCGRA consist of computation PE (cPE) and memory PE (mPE), $1 \le i \le M$, $1 \le j \le N$. C describe the data relevant dependency. For $p,q \in$ PE, $c=(p, q) \in C$, if only q needs the data from p.

In RAG model, computation and storage are modeled separately. mPE handles the memory access while cPE handles computation only. When data are delivered between PEs, there are two kinds of dependency: the dependency from mPE to cPE and the dependency from mPE to mPE.

When mapping DIA onto lspCGRA, we use program profiling to get critical loop blocks from PIA-CDTG. Then the critical loop kernels described by vi-DFG are mapping onto lspCGRA. The characteristics of lspCGRA are shown as follows, which are different from existing CGRA models.

Process Element. Process Element (PE) handles the execution of loop kernels, which consists of mPE and cPE.

Local Memory. Local Memory (LM) can buffer the data of loop kernels, which can be shared only by neighboring mPEs due to the separation of computation and storage. mPE read data from LM and write data to it directly. Besides, smart buffer (SB) that consist of registers is used to buffer data too.

Route Network. In route network, mPE is connected with LM directly and mPE is connected with cPE by 2D torus. When loop is executing, the interconnections between mPE and CPE are fixed. Process element array (PEA) of the same line are connected with the same branch of bus. The bus uses scattered structure to reduce transfer delay. The configure bus delivers the configuration information and load configuration enable. The start bus controls the execution of PEA.

According to the above descriptions, the problem formulation of mapping loop kernel onto lspCGRA is defined as follows:

Definition 3 (LKPM Problem). Given a critical loop kernel block V = (T, E) of PIA-CDTG for a data-intensive application and the RAG of lspCGRA R = (PE, C), find a mapping M: V-> R, with the objective of minUPE and maxTHO, under the meeting of resource constraints, such as computation/storage/communication resource. Among them, UPE is the occupation ratio of PE and THO is the throughput of loop pipelining. When the resource is sufficient, we prefer to optimize the objective maxTHO, while when the resource is insufficient, we prefer to optimize the objective minUPE.

The problem mainly consists of objective function and constraints. The objective function is defined as follows:

$$\text{Objective Function} = \{\text{UPE}, \text{THO}\}; \quad (1)$$

For M×N PE array, there are n×(m-1) cPE and n mPE. If a mapping M use s cPE and t mPE, $s \le m$, $t \le n$, then:

$$\text{UPE} = (s+t)/(m \times n); \quad (2)$$

The size of LM and SB is not considered here. We only consider the UPE of PAE, which indirectly represents the area cost of mapping M. That is, the fewer the UPE is, the fewer the area cost is, vise versa.

$$\text{THO} = \text{OP}/\sigma; \quad (3)$$

Wherein, OP represents the operation number of loop kernels and σ represents the execution clock cycle of loop kernels. The unit of THO is operation per second. The value of THO indirectly show the operation handling ability of mapping *M*. Because our approach uses pipelining parallel for loop kernels, the ideal THO is to complete a loop iteration per clock cycle. That is, the max THO #OP is equal #cPE. The actual THO is influenced by some factors, such as the start cost of pipelining, the delay of memory access.

The constraint set that influence the two objectives are:

$$\text{Constaint Parameters} = \{\text{mPE}_c, \text{cPE}_c, \text{LM}_c, \text{LBus}_c\}; \quad (4)$$

These constraint parameters are defined as follows:

mPE constraint (mPE_c). Given a *M*×*N* PEA, the number of mPE that cPE use is not more than the total number of mPE. Let m_{ij} be the configuration to mPE_{ij} in PEA, then mPE_c:

$$\sum_{j}^{n} m_{ij} \le \#of\ mPE = n; \quad (5)$$

cPE constraint (cPE_c). Given a *M*×*N* PEA, the number of cPE is not more than *m*×(*n*-1). The total number cPE used in each line is not more than *n*. Let c_{ij} be the configuration to cPE_{ij} in PEA, then cPE_c:

$$\sum_{j}^{n} c_{ij} \le \#of\ cPE\ in\ ith\ row = n; \quad (6)$$

Local Memory constraint (LM_c). LM can be shared by only neighboring two mPEs. Let l_{ij} be the atomic access operation to LM, then LM_c:

$$l_{ij} + l_{i(j+1)} \le \#of\ \ LM\ resources\ in\ ith\ row,\ jth\ column; \quad (7)$$

Line bus constraint (LBus_c). The PEs of PEA in the same line are connected on the same bus branch. Let c_{ij} be the configuration to cPE_{ij} in PEA, b_{ij} be the atomic bus access operation, then LBus_c:

$$\sum_{j}^{n} b_{ij} \le \#of\ memory\ bus\ resources\ in\ ith\ row; \quad (8)$$

IV. ARCHITECTURE TEMPLATE OF LSPCGRA

A. Resource Sharing and Loop Self-Pipelining of lspCGRA

There are many share resources in lspCGRA, such as cPE, mPE, LM, data memory and route network. The PEs in the same line share a bus branch and the neighboring two mPEs share a LM, as shown in Figure 1.

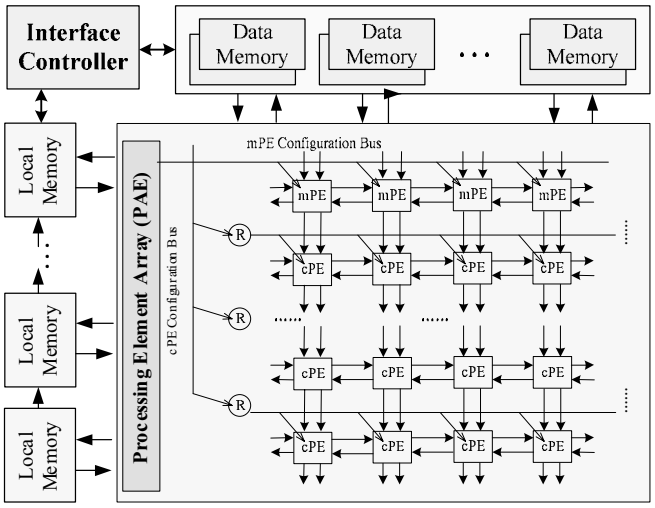

Figure 1. The PAE structure and share resource of lspCGRA

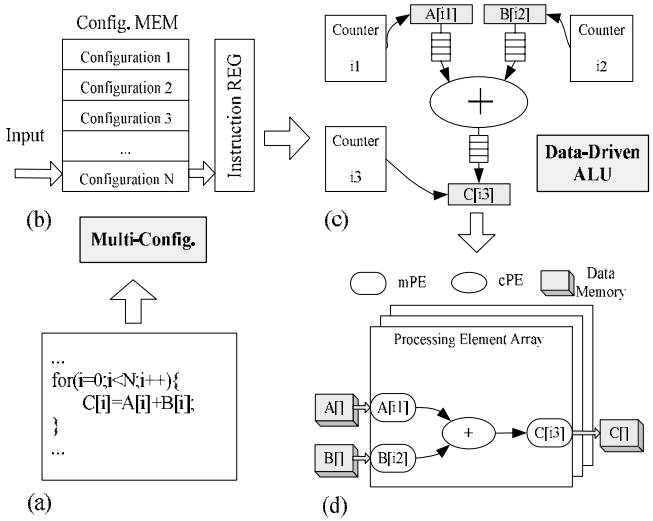

Figure 2. The mechanism of loop self-pipelining

The lspCGRA use resource pipelining for loop kernels, as shown in Figure 2. The loop self-pipelining uses the advantages of data driven architecture, which transfers the control to loop for memory operation. Thus it can control the data flow by the memory operation. The whole loop is executing in this way. mPE provides the data of dataflow. The former two mPE control the input of dataflow while the latter mPE control the output of dataflow. When the PEA is full filled with dataflow, computation result will be output per cycle. That is, the loop will handle the iteration once per cycle.

This mechanism needs not extra synchronization control the dataflow, but it can get high throughput.

Besides, in order to support loop pipelining for DIA, lspCGRA use guess mechanism to improve the ability for the judgment of condition and data reuse to improve the utilization of memory resource.

B. Template Standard of lspCGRA

CGRA have many implement details for field specific applications, such as array topology, communication protocol and reconfigurable strategy. In order to map DIA onto lspCGRA efficiently, we customize DIA-lspCGRA template, which has many parameters to configure a broad range of DIA and lspCGRA.

DIA-lspCGRA template consists mainly of Header, DOMAIN_DIA and ARCH_lspCGRA, as shown in Figure 3.

DOMAIN_DIA describe the program profiling information, loop model and in-loop data info of DIA. Data-info provide some data relevant information of vi-DFG, such as data reuse, cross window, which can support mapping and optimizing well. For example, utilizing the bandwidth of LM to schedule storage resources, or using data relevant for data reuse when handling median filter and edge detection. The data deliver delay is hided by overlap execution of computation and storage operation. When handling FFT, read/write operation can be rapidly converted between double-port bank。

ARCH-lspCGRA describes the pipelining loop, constraint model, resource model and performance model of lspCGRA. The pipelined loop inputs the configuration information to configure memory, which determines the operation of PEs and the data exchange of route network. Constraint model consists of mPE_c, cPE_c, LM_c and $LBus_c$. Resource model describes the configuration of lspCGRA resource, such as PEs, LM, SB and route network. Performance model consists of FPGA synthesis results of lspCGRA MPE, CPE and route network, such as the logic elements, storage and clock frequency of one MPE.

```
class Template{                          //DIA-lspCGRAs template
    Header{                              //general description
        name DIA-lspCGRAs_template;
        description "xxx";
        path "X:/XX/XX...";
        author "xxx xxx";
        version "X.X.X";
        date "XXXX-XX-XX";
    }
    DOMAIN_DIA{                  //field domain: data-intensive app
        PIA-CDTG;                //program profiling info
        vi-DFG;                  //loop kernel model
        Data-Info;               //in-loop data info
    }
    ARCH_lspCGRAs{               //architecture - lspCGRAs
        Loop_Pipeline;           //loop pipelining
        Constraint_model;        //constraint model
        Resource_model;          //resource model
        Performance_model;       //performance model
    }
}
```

Figure 3. The standard of DIA-lspCGRA template

V. MAPPING AND OPTIMIZATION

A. Field Specific Application Driven Mapping Flow

We use field specific application driven mapping flow to map PIA-CDTG onto lspCGRA. The flow uses application profiling to extract critical loop blocks. The PIA-CDTG is mapped onto lspCGRA by partial partition and design space exploration. Besides, the critical loop kernels is pipelining mapping onto PEA of lspCGRA, as shown in Figure 4.

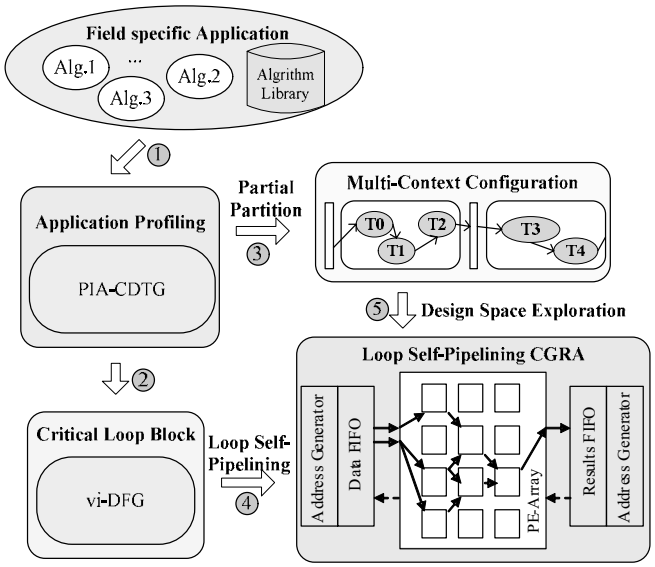

Figure 4. Field specific application driven mapping flow

The flow consists of five parts mainly:

1). Application analysis and profiling

We analyze some application features including control flow/dataflow, computation/storage distribution, critical code blocks and relevant information. And then, the PIA-CDTG feature model is generated based on analysis results [19]. The field specific application program is transformed into intermediate code for profiling and extracting. The analysis results are saved in application profiling information library and system model library for design reuse.

2). Finding critical loop blocks

Based on application analysis and profiling, we find the critical loop blocks according to the computation/storage need of applications.

First, we partition the source code of application program into code block and build TFG (task flow graph) by taking code block as the node of TFG. Generally, there are three grains of TFG: instruction level, block level, and procedure level. We use block level here, which further includes blocks, loops and functions. The loop or function can be compound, which means there may be some child blocks in them.

Then, we statistics the features of each basic block according to the analysis results, as shown in Table I.

Finally, according to the statistics results, we recognize some blocks as critical blocks if the execution time percentage and storage capacity is higher than a threshold value.

For the critical loop blocks, we transfer control flow into data flow using if-conversion and standardize loop kernels:

1). Let the begin value, end value and step value of loop be a constant.

2). Smooth away "break" and "continue" sentence.

3). Eliminate indirect operation of get the address.

4). Rename the register and clear some relevant of read after write.

Then build vi-DFG of critical loop block according to the virtual instruction defined by KAHN for loop self-pipelining.

TABLE I. STATISTICS DATA OF BASIC BLOCKS

Basic block	Function	Loop	Block
Name	√	√	√
Iteration times	×	√	×
Call times	√	√	√
Instruction number	√	√	√
Instruction type number	√	√	√
Execute frequency	√	√	√
Execute time	√	√	√
Execute time percentage	√	√	√
Storage requirement	√	√	√
Storage percentage	√	√	√

3). Partial Partitioning

The spatial mapping is limited by the size of PEA and the structure of route network. In order to mapping multiple applications onto lspCGRA simultaneously, we use partial partitioning to split applications into child blocks and map them onto PEA in turn. Partial partitioning always is limited by the size of configuration memory and the resource of lspCGRA.

Some information from program profiling can guide to partial partition, such as percentage of execution time, storage requirement of active variable, storage requirement of objective code, data communication among nodes, parallel ability of nodes and the precedence of nodes. The nodes are clustered according to the relevant to reduce the data exchange and communication expense.

Besides, the resources of lspCGRA impact on the grain of critical blocks. If the resources are plenty, the grain of critical blocks can be bigger. If the resources are limited, the grain of critical blocks can be smaller and when the critical block is multi-dimension loop, the loop unrolling is needed for mapping smaller blocks onto lspCGRA.

4).Loop self-pipelining

For each vi-DFG, we implement it onto lspCGRA. The "LEAP" (Loop engine array processor) architecture use pipeline execution to map loop onto PE array and the loop program is discomposed to multi-parts which implemented by some PEs:

1). Loop finite state machine is corresponding to the control conditions of loop.

2). Process element array is corresponding to the loop body.

3). Loop finite state machine control read/write data from memory.

4). Process element array handle the execution of the input data from memory.

The pipeline execution of loop depends on the mPEs and cPEs. mPEs provide flexible storage scheduling and cPEs provide powerful computation capacity. The index value of loop control variable is changed step by step in mPE and the data are read from LM according to the suffix of loop. These data are put to cPE for computing and the results are saved to the address that mPE specifies. Thus the lspCGRA can achieve high throughput.

5).Design space exploration

When mapping field specific application, design space exploration is necessary to trade-off multiple performance objects. We explore the parameters of lspCGRA to optimize field specific application mapping:

- The number of mPE and cPE,
- The number and size of LM,
- The size of smart buffer,
- The bandwidth of input and output data,
- The stage and partition of pipeline

The exploration can exploit the suitability of lspCGRA for specific applications. The critical blocks are recognized for special optimization, which can reduce the design space and improve the quality of mapping.

A. Loop Kernel Pipelining Mapping

We use LKPM to map loop kernels onto lspCGRA, as shown in Figure 5.

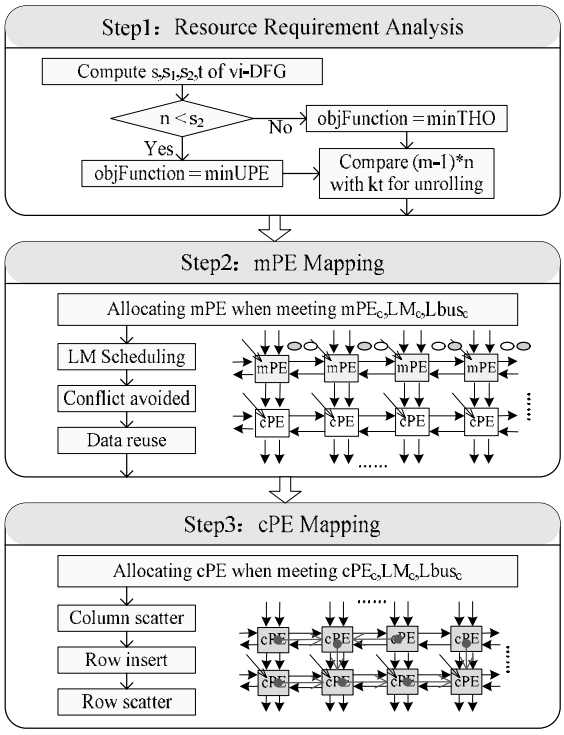

Figure 5. Loop-Kernel Pipelining Mapping

The LKPM consists of three parts:

1). Resource requirement analysis

Analyze the operation of loop kernel and find s PE nodes for storage operation, in which there are s_1 input data for operation, s_2 input data after relevant analysis, and t other operations. There are n MPE and $(m\text{-}1)\times n$ cPE in lspCGRA, $t < (m\text{-}1)\times n$.

We set the objective function as follows:

$$objFunction = \begin{cases} \min UPE, & n < s_2 \\ \max THO, & (m-1)\times n,\ n \geq s_2 \end{cases}; \quad (9)$$

When the objective function is maxTHO, if (m-1)×n > kt, we unroll multi-dimension loop k times.

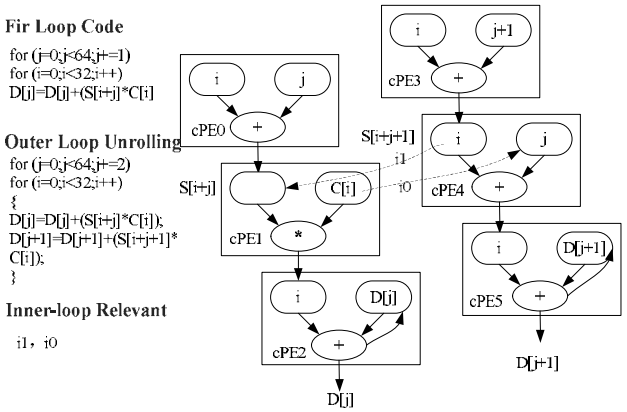

Figure 6. the dataflow of loop unrolling and pipelining

■ Loop unrolling and pipelining

In order to improve throughput, we unroll outer loop if the resource is enough, as shown in Figure 6.

In the Fir loop code, the inner loop consists of a multiply and an add operation, which is pipelining mapped onto lspCGRA. If the variable of outer loop j is unrolled, the inner loop will execute two multiply and add operations at each cycle.

2).mPE mapping

The lspCGRA implement the separation of computation and storage. Thus when mapping loop kernels, input data are allocated onto mPE first, in which we must reduce the delay of data transfer to cPE.

In this step, we schedule the LM between neighboring mPEs, to ensure that the pipeline is not conflicted. Besides, the computation and data transfer are executed at the same time to hide the delay of data transfer.

The two strategies are used in the step:

■ Confliction avoidance

When mapping the data of loop kernel onto storage resource, the conflict of data access must be avoided. There are two main data access confliction: parallel confliction and self confliction [18].

For the code in a loop, such as Fir loop:

tmp1 =S[i+j]×C[i];

If the data of S and C are put in the same storage resource, such as FIFO queue of cPE, the structure relevant will be brought, which is also called parallel confliction.

For the code in a loop, such as convolution function:

y=X[i]×X[k-i];

If the data of X are put in the same storage resource, structure relevant will be brought, which is also called self confliction. For the data which are parallel conflicted and self conflicted, they should be allocated onto different storage resources.

■ Data reuse

For most DIA, such as stream media, cross window applications, there are many data reuse. When mapping them, inter-loop reuse and inner-loop reuse are exploited to reduce the time for reading the input data.

We put the data in registers for inner-loop reuse, such as:

```
for(j=0;j<m;j=j+1)
    B[j]=A[j]+A[j+1]+C;
```

We put the data of current cross window in LM for inter-loop reuse, such as:

```
for(i=0;i<n;i=i+1)
  for(j=0;j<m;j=j+1)
      B[i][j] =A[i][j] +A[i+1][j] +C;
```

3).cPE mapping

We map cPEs use split-push, just like SPKM, but there are some difference [5]:

■ cPEs interconnection

In SPKM, PEs can be connected with their first, as well as their second horizontal or vertical neighbors. In LKPM, cPEs can be connected with their first horizontal or vertical neighbors only.

■ mPE scheduling

In SPKM, there are only 2 LD and 1 ST operations in a line. While in LKPM, there is no such restriction, but the constraints of mPEc and LMc must be met.

■ optimization objection of ILP matching

In SPKM, when doing row-wise scattering, the objective function of ILP (integer linear program) match is defined as |UR|min. In our LKPM, we define it |UPE|min or |THO|max according to objFunction.

VI. EXPERIMENTAL RESULTS

We design some embedded media algorithms on lspCGRA named “LEAP” developed by our research group. “LEAP” use loop self-pipelining, which map the control conditions of loop onto mPEs and the body of loop onto cPEs. WGM [17] has manually mapped some DIA algorithms, such as FFT, Sobel Edge Detection, Median Filter and Matrix Multiply. In this paper, we map these algorithms onto lspCGRA automatically, the performance results of which are shown in Table II.

We tested some DIA algorithms mapped onto lspCGRA. Table II shows the details about the mapping, such as the number of cPE and mPE, throughput THO, the percentage of computation active degree AP, execution cycles ECycle and pipeline start delay #Delay. The data in the parentheses are manual and others are automatically.

From Table II we find that in the MM algorithm of the last line, UPE and THO cannot achieve optimal concurrently. The

more cPE occupied, the higher THO is, for ideal highest throughput is equal to #cPE. In our approach, we use design space exploration to trade-off the multi-objective of simultaneously mapped tasks. Besides, the fifth column of AP represents cPE execution time percentage of all execution time. We carefully handle the overlap of computation and storage, while AP still can show that memory access delay is the important factor to influence the whole THO. We use the strategy of confliction avoidance and data reuse to reduce the time of access memory while meeting resource constraints.

In order to explain the efficiency of LKPM, we select existing advanced SPKM to compare the performance. Four algorithms in Table II are selected as benchmarks. The mapping resource occupied ratio UPE and throughput THO are shown in Figure 7 and Figure 8. Only the mapping of cPE is considered in SPKM, so the memory access delay is high, causing that the THO is low. On the other hand, there are only 2 LD and 1 ST in a line of LKPM, so the storage bandwidth becomes bottleneck and the cPE lines are probably increased Thus the UPE of SPKM is lower than LKPM. Because lspCGRA uses double-port memory for LM, when mapping FFT, the read/write operation can be rapidly switched between double-port memory, which can save the storage resource and improve UPE. There are many data relevant and data reuse in the algorithm of ED and MF, our approach can reduce the line of cPE, reduce the time of memory access and improve the throughput. So their mapping results are higher than SPKM. The average UPE of LKPM is 16.3% less than that of SPKM and the average THO of LKPM is 169.1% more than that of SPKM in totality.

We randomly generate 40 RAGs of loop kernels to compare the mapping effective of LKPM with that of SPKM. The experiments are operated with the nodes of RAGs from 10 to 16. All the RAGs are divided into 4 groups. There are 10 DAGs in each group with different application type: data reuse between loops, data reuse in loop, data access with confliction and no data relevant. The mapping effective is measured by UPE, as shown in Figure 9. From the experiment results we can find that LKPM can generate 31 better mapping which has lower UPE than SPKM and LKPM can generate 6 same quality mappings with SPKM. This implies that for the 75.6% RAGs, LKPM is able to generate better quality mappings than SKPM in terms of UPE for the generated mappings.

VII. CONCLUSIONS

CGRA provide flexible and efficient solutions to develop complex applications while mapping data-intensive applications onto CGRA is still a hard work. In order to reduce the complexity of mapping and exploit CGRA for embedded media applications, this paper proposes a novel loop kernel automatically pipelining mapping approach, which support loop self-pipelining and achieve high throughput for DIA.

The LKPM problem formulation is presented and the resource sharing and pipelining of lspCGRA template are shown. The LKPM approach with its experimental results is shown together. The performance of our mapping approach is close to that of manual mapping. Besides, ours approach shows less resource occupation by 16.3% and higher throughput by 169.1% than advanced SPKM.

ACKNOWLEDGMENT

This work is sponsored by the National Science Foundation of China under the grant NO.2006CB303000 and NO.90707003.

TABLE II. MAPPING RESULTS OF TYPICAL DIA ONTO LSPCGRA

Benchmark Algorithms	UPE		THO (OP/cycle)	AP/%	ECycle/K	#LDelay
	mPE	cPE				
512 point FFT	4	10(10)	3.9(4.0)	39.3(40.3)	26.2(25.6)	34(34)
Edge Detection (320x240)	7	16(16)	6.3(6.6)	39.6(41.5)	227(217)	54(54)
Median Filter (320x240)	7	30(30)	12.4(12.7)	41.3(42.3)	225(220)	54(54)
Matrix Multiply (64x64)	10	30(16)	12.5(6.7)	40.7(41.6)	42.4(79.1)	45(24)

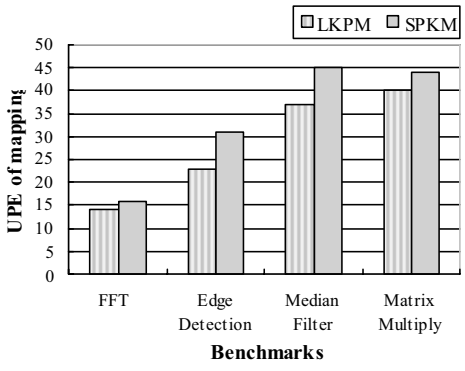

Figure 7. UPE of algorithm mapping

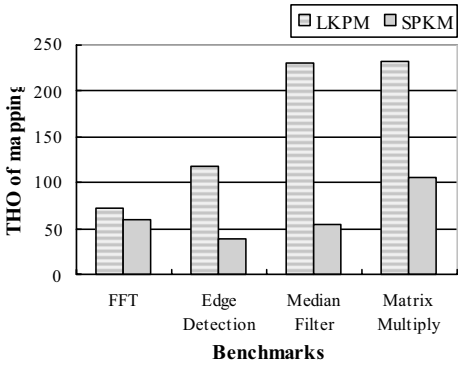

Figure 8. THO of algorithm mapping

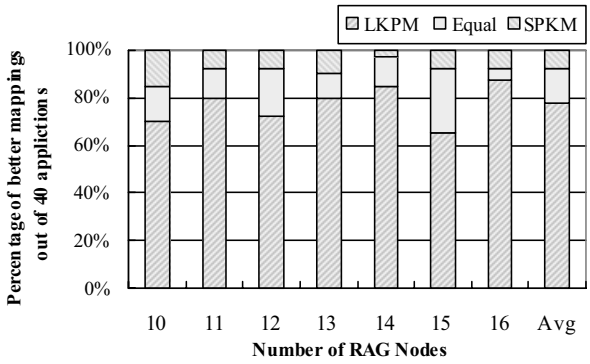

Figure 9. Percentage of better mappings of LKPM and SPKM

REFERENCES

[1] Compton Katherine,Hauck Scott, "Reconfigurable Computing: A Survey of Systems and Software," *ACM Computing Surveys*, 2002, 34(2), pp. 171–210.

[2] R. Hartenstein, "A Decade of Reconfigurable Computing: a.Visionary Retrospective," *DATE2001*, Munich, Germany, March 12-15, 2001, pp. 642–649.

[3] Aviral Shrivastava, Eugene Earlie, Nikil Dutt, and Alex Nicolau, "PBExplore: A Framework for Compiler-in-the-Loop Exploration of Partial Bypassing in Embedded Processors," *DATE 2005*, Washington, pp. 1264–1269.

[4] Kevin Fan, Manjunath Kudlur, Hyunchul Park, Scott A. Mahlke, "Increasing hardware efficiency with multifunction loop accelerators," *CODES+ISSS 2006*, pp. 276–281.

[5] Jonghee W. Yoon, Aviral Shrivastava, Sanghyun Park, Minwook Ahn, Reiley Jeyapaul, Yunheung Paek, "SPKM : A novel graph drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures," *ASP-DAC 2008*, pp. 776–782.

[6] J. Lee, K. Choi et al., "Mapping Loops on Coarse-Grain Reconfigurable Architectures Using Memory Operation Sharing," *TR 02-34*, Center for Embedded Computer Systems, University of California, Irvine, 2002.

[7] Yong Dou, Xicheng Lu: LEAP, "A Data Driven Loop Engine on Array Processor," *APPT 2003*, pp. 12–22.

[8] Arun Kejariwal, Alexander V. Veidenbaum, Alexandru Nicolau, Milind Girkar, Xinmin Tian, Hideki Saito, "Challenges in exploitation of loop parallelism in embedded applications," *CODES+ISSS 2006*, pp.173–180.

[9] Volker Baumgarten, G. Ehlers, F. May, Armin Nuckel, Martin Vorbach, and Markus Weinhardt, "PACT XPP – A Self-Reconfigurable Data Processing Architecture," *The Journal of Supercomputing*, 2003, 26(2), pp. 167–184.

[10] H Singh, M Lee, G Lu, F J Kurdahi, N Bagherzadeh, E Filho, R Maestre, "Morhposys: case study of a reconfigurable computing system targeting multimedia applications," *DAC'00*, Los Angeles, California, 2000, pp. 573–578.

[11] Qubo Hu, Arnout Vandecappelle, Martin Palkovic, etc, "Hierarchical memory size estimation for loop fusion and loop shifting in data-dominated applications." *ASP-DAC 2006*, pp. 606–611.

[12] Jong-eun Lee, Kiyoung Choi, Nikil D. Dutt, "An algorithm for mapping loops onto coarse-grained reconfigurable architectures," *LCTES 2003*, pp. 183–188.

[13] Hongbo Rong, Zhizhong Tang, Ramaswamy Govindarajan, Alban Douillet, Guang R. Gao, "Single-dimension software pipelining for multidimensional loops," *TACO*, 4(1), 2007.

[14] Yoonjin Kim, Mary Kiemb, Chulsoo Park, Jinyong Jung, Kiyoung Choi, "Resource Sharing and Pipelining in Coarse-Grained Reconfigurable Architecture for Domain-Specific Optimization," *DATE 2005*, pp. 12–17.

[15] Minwook Ahn, Jonghee W. Yoon, Yunheung Paek, Yoonjin Kim, Mary Kiemb, Kiyoung Choi, "A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures," *DATE 2006*, pp. 363–368.

[16] Yong Dou, Guiming Wu, Jinhui Xu and Xingming Zhou, "A Coarse-grained Reconfigurable Computing Architecture with Loop Self-Pipelining," *Science in China*, 38(4), 2008, pp. 579–591.

[17] Guiming Wu, "*Research on Coarse-grain Reconfigurable Architecture Base on Loop Pipelining*," A Dissertation Submitted for the Master's Degree, National University of Defense Technology, Changsha, 2006.

[18] T. S. Rajesh Kumar, C. P. Ravikumar, R. Govindarajan, "MODLEX: A Multi Objective Data Layout EXploration Framework for Embedded Systems-on-Chip," *ASP-DAC 2007*, pp. 492–497.

[19] Peng Zhao, Sikun Li, Dawei Wang, and Ming Yan, "Application-driven System-on-Chip System Model Extraction Approach," *The 12th International Conference on CSCW in Design*, Xi'an, 2008.

Dawei Wang was born in Donggang of P.R.China in 1980. He is now Ph.D.candidate in College of Computer Science from National University of Defense Technology at Changsha. The major field of his interest has been SoC system design method, and electronic design automation, etc.

Since 1998 he was 10 years with the College of computer Science from National University of Defense Technology. As a Ph.D.candidate, he worked at the university over 6 years in all. His research is recently oriented into application specific and coarse-grained reconfigurable architectures.

Sikun Li was born in Qingdao of P.R.China in 1941. He obtained Bachelor's degree in Military Engineering Institute of the PLA, Haerbin, in 1965. The major field of his interest has been SoC design methodology and advanced VLSI design method, and electronic design automation, etc.

Since 1984 he was 24 years with the the College of computer Science from National University of Defense Technology at Changsha. As a Full Professor he taught and made researches in the field of CAD, VLSI and SoC. His research is recently oriented into SoC system level design and embedded media design.

Prof. Sikun Li is the manager of CAD/CG committee of China Computer Federation, committees for international conference CAD/CG, CSCWD.

Yong Dou was born in Jilin of P.R.China in 1966. He obtained Ph.D. degree in Military Engineering Institute of the PLA, Haerbin, in 1995. The major field of his interest has been computer architecture and compiler optimization, and reconfigurable computing, etc.

Since 1995 he was 13 years with the the College of computer Science from National University of Defense Technology at Changsha. As a Full Professor he taught and made researches in the field of parallel architecture, reconfigurable array processor design. His research is recently oriented into FPGA pipeline scheduling and loop-self pipelining onto reconfigurable array.

Prof. Yong Dou is the architecture professional committee of China Computer Federation, committees for international conference APPT.