COMBINING DATA REUSE EXPLOITATION WITH DATA-LEVEL PARALLELIZATION FOR FPGA TARGETED HARDWARE COMPILATION: A GEOMETRIC PROGRAMMING FRAMEWORK

Qiang Liu[*]*, George A. Constantinides*[*]*, Konstantinos Masselos*[†] *and Peter Y.K. Cheung*[*]

[*]Imperial College, London SW7 2BT, U.K.
[†]University of Peloponnese, Tripolis, Greece
{qiang.liu2, g.constantinides, k.masselos, p.cheung}@imperial.ac.uk

ABSTRACT

A *geometric programming* framework is proposed in this paper to automate exploration of the design space consisting of data reuse (buffering) exploitation and loop-level parallelization, in the context of FPGA-targeted hardware compilation. We expose the dependence between data reuse and data-level parallelization and explore both problems under the on-chip memory constraint for performance-optimal designs within a single optimization step. Results from applying this framework to several real benchmarks demonstrate that given different constraints on on-chip memory utilization, the corresponding performance-optimal designs are automatically determined by the framework, and performance improvements up to 4.7 times have been achieved compared with the method that first explores data reuse and then performs parallelization.

1. INTRODUCTION

FPGA-based reconfigurable systems have been applied to an extensive range of applications, such as digital signal processing, video and voice processing, and high performance computing. However, to efficiently exploit the flexibility provided by heterogeneous reconfigurable resources on FPGAs in order to achieve an optimal design is still a difficult problem. This paper introduces an optimization framework to aid designers in exploration of the data reuse and data-level parallelization design space at compile time with the objective of maximizing system performance while meeting constraints on on-chip memory utilization.

Our target FPGA-based reconfigurable system is shown in Fig. 1 (a). External RAMs are accessed by an FPGA as main memories. It is well known that data transfers between external memories and the processing unit (PU) are often the bottleneck when trying to use reconfigurable logic as a hardware accelerator. As a result, the use of on-chip RAMs to buffer repeatedly accessed data, known as *data reuse* [1], has been investigated in depth. In our previous work, a systematic approach for data reuse exploration has been proposed [2, 3], exploiting a scratch-pad memory (SPM) to load and store reused data, as shown in Fig. 1 (b). Data that tend to be frequently accessed are first loaded into the SPM and then accessed by the PU. In this manner, the number of off-chip memory accesses is significantly reduced. A systematic methodology for data reuse exploration is proposed in [1] for ASICs. In [4], an approach for exploiting data reuse in scratch-pad memory (SPM) has been presented. Research into buffering reused data in FPGA on-chip RAMs and registers has been carried out in [5], [6] and [7]. In [7], the proposed compiler exploits the data reuse characteristic of window operations and performs loop transformations to pipeline the inner loop iterations.

Improvement of the parallelism of programs has been a hot topic in the computing community. Lefebvre *et al*. [8] propose an approach for parallelizing static control programs with optimized memory requirements. Gupta *et al*. [9] identify the constraints in data distributions over multiple processors and address a technique for determining suitable data partitions. Loop nests are the main source of potential parallelism, and loop-level parallelization has been widely used for improving performance [10]. However, no matter what the data dependence structure of the code, the performance improvement is in fact limited by the number of parallel data accesses to fetch the operands. Buffering data in on-chip RAMs opens avenues for parallelism. There are a number of embedded RAM blocks on modern FPGAs, often with two independent read/write ports. If data are duplicated or distributed into different memory banks, then further parallelization can be achieved, as shown in Fig. 1 (c).

To our knowledge, there is no prior work on combining data reuse exploration and loop-level parallelization within a single optimization step. Eckhardt *et al*. [11] apply the *co-partitioning* scheme to mapping an algorithm onto a processor array with a two-level memory hierarchy. Data reuse is considered during the algorithm partitions. However, the main purpose of the work is to model the target architecture to fit the algorithm, rather than to develop an application-specific architecture. In [12] and [13], authors experiment

with the effects of different data reuse transformations and memory system architectures on the system performance and power consumption. The results prove the necessity of the exploration of data reuse and data-level parallelization. However, if these two tasks are performed separately, then performance-optimal designs may not be obtained. If parallelization decisions are made first without regard for memory bandwidth, then a memory subsystem needs to be designed around those parallelization decisions, typically resulting in infeasibly large on-chip memory requirements to hold all the operands, and large run-time penalties for loading data from off-chip. A more sensible approach may be to first make data reuse design decisions to maximally improve data locality and secondly improve the parallelism of the resulting code [1]. However, once the decision is made to fix the memory subsystem design, sometimes only limited parallelism can be extracted from the remaining code.

Thus, in this paper, we address the combined problem as a single optimization step. In the context of this paper, the data-reuse decision is to decide at which levels of a loop nest to insert new on-chip arrays to buffer reused data for each array reference, in line with [3]. We consider the code to have been pre-processed by a dependence analysis tool such as SUIF [14] and each loop to have been marked as parallelizable or sequential. In the context of this paper, the parallelization decision is to decide an appropriate strip-mining [10] for each loop level in the loop nest, allowing a parameterized degree of parallelism. The overall optimization takes place while respecting an on-chip RAM utilization constraint, and the dependence between the two problems is incorporated into the formulation. We show that a geometric programming framework [15] can address the combined problem. The main contributions of this paper are thus:

- recognition of the dependence between data reuse and data-level parallelization and exploration of both problems within a single optimization step,
- an Integer Geometric Programming (IGP) formulation of the combined data reuse and data-level parallelization problem for performance optimization under an on-chip memory constraint, revealing a computationally tractable lower-bounding procedure, and thus allowing solution through branch and bound, and
- the application of the proposed framework to several signal and video processing kernels, resulting in performance improvements up to 4.7 times compared with the method that first explores data reuse and then data-level parallelization.

The rest of the paper is organized as follows. Section 2 describes the targeted problem. Section 3 formulates the problem as an IGP. Section 4 presents results from applying our proposed framework to several real benchmarks. Section 5 concludes the paper and suggests future work.

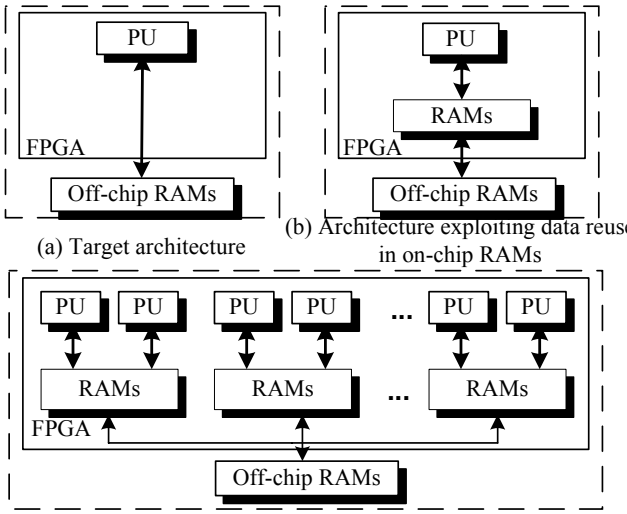

Fig. 1. Target platform.

2. PROBLEM STATEMENT

The general problem under consideration is how to design a high performance FPGA-based processor from imperative code annotated with potential loop-level parallelism using constructs such as Cray Fortran's 'doall' or Handel-C's 'replicated par'. In the target platform, the central concern is that the off-chip RAMs only have few access ports. Without loss of generality, this paper assumes for simplicity that one port is available for off-chip memory accesses.

To simplify the problem, in this paper, we focus on N-level perfectly nested loops $(I_1, I_2, \ldots, I_N)$, where I_1 is the outer-most loop and I_N is the inner-most loop, with multiple arrays stored in off-chip memory inside the inner-most loop, as shown in Fig. 2 (a). The problem formulation is easily extended to the imperfectly nested loops, but the details are omitted in this paper due to the space limitation.

A code consisting of a loop nest as shown in Fig. 2 (a) usually exhibits data reuse in accesses to the array references and also presents potential parallelism in some loops, which can be revealed by existing data dependence analysis tools. However, despite the apparent parallelism, the code can only be executed in parallel in practice if an appropriate memory subsystem is developed, otherwise the bandwidth to feed the datapath will not be available.

Data reuse mainly targets access locality improvement, but can also benefit parallelization. Following the approach described in [2], a *data reuse array*, stored in on-chip memory, is introduced to buffer reused elements of an array stored in off-chip memory. Before the elements are used they are first loaded into the data reuse array in the on-chip memory from the original array. Such a scratch-pad memory may be

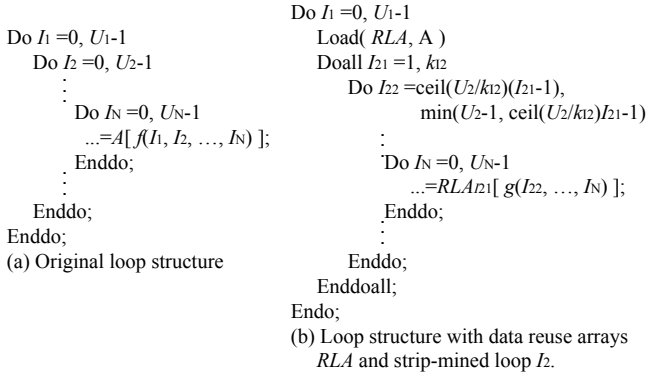

Fig. 2. Target loop structure.

introduced at any level of the loop nest, giving rise to different tradeoffs in on-chip memory size versus off-chip access count [3]. Since the data-reuse arrays are stored on-chip, we may use the dual-port nature of the embedded RAMs and even replicate the data across different RAMs in order to increase the number of ports available to allow multiple loop iterations to execute in parallel. For example, in Fig. 2 (b), several data reuse arrays, RLA, are introduced inside loop I_1 but outside loop I_2 to buffer reused elements of the array A. After that, the potentially parallelizable loops of $(I_2, I_3, \ldots, I_N)$ could be strip-mined. When a loop is strip-mined for parallelism, the loop that originally executes sequentially is divided into two loops, a $Doall$ loop that executes in parallel and a new Do loop running sequentially, with the latter inside the former. For example, Fig. 2 (b) shows the transformed loop structure when loop I_2 is strip-mined and then partially parallelised, utilizing k_{I_2} distinct parallel processing units in the FPGA hardware. As a result, k_{I_2} copies of the data are held on-chip in arrays $RLA_{I_{21}}$ ($\lceil k_{I_2}/2 \rceil$ for dual-port RAMs). This parameter k_{I_2} thus allows a tuning of the tradeoff between on-chip memory and compute resources and execution time. It is clear therefore that exploiting data reuse makes data-level parallelization possible.

However, the choice of data-reuse option impacts on the possible parallelism that can be extracted from the code, because different data reuse options involve loading reused data from off-chip memory in different loop levels, for example some parallelizable loops contain off-chip memory accesses in their loop bodies and cannot be parallelized in some options. Proceeding with the example in Fig. 2 (b), loop I_1 cannot be parallelized whether it is parallelizable or not, because loading arrays RLA from off-chip memory happens inside it. Therefore, if the data-reuse decision is made prior to exploring data-level parallelization, then the potential parallelism existing in the original code may not be completely exploited. This observation leads the conclusion that parallelism issues should be considered when making data-reuse decisions.

3. PROBLEM FORMULATION

To automatically solve the problem of exploring data reuse and data-level parallelization to achieve the designs with optimal performance under an on-chip memory constraint, we formulate it as an IGP program, which has a convex relaxation [15], allowing efficient solution techniques.

A set of R references $A_{i\ (1\le i\le R)}$ to arrays are accessed inside the loop nest. Reference A_i owns a total of E_i beneficial data reuse options[1] $OP_{ij\ (1\le j\le E_i)}$ and option OP_{ij} occupies B_{ij} blocks of on-chip RAM and needs C_{ij} cycles for loading reused data from off-chip memories. Loop l $(1 \le l \le N)$ can be partitioned into k_l $(1 \le k_l \le L_l)$ parallel segments. The k_l variables corresponding to those loops not parallelizable in the original program are set to one. All notations used in this paper are listed in Table 1. Based on these notations, the problem of exploration of data reuse and data-level parallelization is defined in equation (1)–(8), which will be described in detail below.

$$\min : S\prod_{l=1}^{N} v_l + \sum_{i=1}^{R}\sum_{j=1}^{E_i}(\rho_{ij}-1)C_{ij} \tag{1}$$

subject to

$$dB_{temp} + d\sum_{i=1}^{R}\prod_{j=1}^{E_i}\rho_{ij}^{\ log_2 B_{ij}} \le B \tag{2}$$

$$\sum_{j=1}^{E_i}(\rho_{ij}-1) \le 1, 1 \le i \le R \tag{3}$$

$$\rho_{ij} \in \{1,2\}, 1 \le j \le E_i, 1 \le i \le R \tag{4}$$

$$k_l^{-1} \le 1, 1 \le l \le N \tag{5}$$

$$k_l \prod_{j=1}^{l} \rho_{ij}^{\ -log_2 L_l} \le 1$$
$$1 \le l \le N, 1 \le j \le E_i, 1 \le i \le R \tag{6}$$

$$L_l k_l^{-1} v_l^{-1} \le 1, 1 \le l \le N \tag{7}$$

$$\frac{1}{2}d^{-1}\prod_{l=1}^{N} k_l \le 1 \tag{8}$$

In this formulation, all capitals are known parameters at compile time, and all variables (lower case) are integers. The objective function minimizes the number of execution cycles of a program in the expression (1), which is composed of two parts: the number of cycles taken by the parallel execution of the original program and by loading data into on-chip buffers. The integral data reuse variables ρ_{ij}

[1] A beneficial data reuse option is a loop level to place an on-chip reuse array where the number of off-chip accesses for loading the reuse array is smaller than the number of accesses to the data by the processor, and thus buffering data on-chip is beneficial [1].

Table 1. A list of notations used in this paper.

Notation	Description	
ρ_{ij}	binary data reuse variables	variables
k_l	# partitions of loop l	
v_l	# iterations in one partition of loop l	
d	# duplications of reused data	
S	# execution cycles of the inner-most loop body	parameters
N	# loops	
R	# array references	
E_i	# data reuse options of array reference i	
L_l	# iterations of loop l	
B_{temp}	# on-chip RAM blocks for storing temporary variables	
B	# on-chip RAM blocks available	
B_{ij}	# on-chip RAM blocks for the data reuse array of option j of reference i	
C_{ij}	# loading cycles of the data reuse array of option j of reference i	

can only take value one or two, which is guaranteed by (4). ρ_{ij} taking value two means the data reuse option OP_{ij} is selected for the reference A_i, value one means it is not. Inequality (3) ensures that at most one data reuse option is chosen for each reference.

Inequality (2) defines the most important constraint on the on-chip memory resources. B on the right hand side of the inequality is the number of available blocks of on-chip RAM. On the left hand side, the first addend is the on-chip memory required by expanding temporary variables, which store intermediate computation results, and in the second term of (2), the term multiplying d expresses the on-chip RAM blocks taken by each copy of reused data of all array references. Note that each dual-port on-chip memory bank is shared by two PUs, as shown in Fig. 1 (c). Hence, half as many data duplications as the total number of parallelized PUs of the code accommodate all PUs with data, which is inequality (8). This constraint also implicitly defines the on-chip memory port constraint, because the number of memory ports required is the double of the number of on-chip RAM blocks required.

Inequalities (5) and (6) give the constraints on the number of partitions of each loop, k_l. Inequalities (6) show the link between data reuse variables ρ_{ij} and loop partition variables k_l. The essence of this constraint is that *a parallelizable loop can only be parallelized if the array references contained within its loop body have been buffered in data reuse arrays prior to the loop execution*. This observation is exploited within this framework to remove redundant design options combing data reuse and data-level parallelization.

The relaxation of problem (1)–(8), obtained by allowing k_l to be real valued solutions and replacing (4) by $1 \leq \rho_{ij} \leq 2$, is exactly a convex non-linear programming (NLP) problem, known as a *geometric program*, and recent advances in optimization of convex NLPs provide efficient solution algorithms with guaranteed convergence to global minimization [15]. A branch and bound algorithm used in [16] is applied to the framework to solve the problem, using the geometric programming relaxation as a bounding procedure.

Table 2. The details of three kernels.

Kernel	k_l	Reference	Reuse options	B_{ij}	C_{ij}
FSME	$1 \leq k_1 \leq 36$	current	OP_{11}	13	25344
	$1 \leq k_2 \leq 44$		OP_{12}	1	25344
	$k_3 = 1$		OP_{13}	1	25344
	$k_4 = 1$	previous	OP_{21}	13	25344
	$k_5 = 1$		OP_{22}	2	76032
	$k_6 = 1$		OP_{23}	1	228096
MAT64	$1 \leq k_1 \leq 64$	A	OP_{11}	2	4096
	$1 \leq k_2 \leq 64$		OP_{12}	1	4096
	$k_3 = 1$	B	OP_{21}	2	4096
Sobel	$1 \leq k_1 \leq 144$	image	OP_{11}	13	25344
	$1 \leq k_2 \leq 176$		OP_{12}	1	76032
	$k_3 = 1$		OP_{13}	1	228096
	$k_4 = 1$	mask	OP_{21}	1	18

4. EXPERIMENTAL RESULTS

For demonstration of the ability of the proposed framework to determine the performance-optimal designs within the data reuse and data-level parallelization design space under an FPGA on-chip memory constraint, we have applied the framework to three kernels: full search motion estimation (FSME) [17], matrix-matrix multiplication of two 64×64 matrices (MAT64) and Sobel edge detection algorithm (Sobel) [18].

In our experiments, the target platform is shown in Fig. 1 (c). Reused elements of each array reference in the kernels are buffered in on-chip RAMs and are duplicated in different banks to provide a number of parallel memory accesses. For a temporary variable, if it is an array, then the array is expanded in on-chip RAM blocks; if it is a scalar, then the variable is expanded in registers. The luminance component of QCIF image (144×176 pixels) is the typical frame size used in FSME and Sobel kernels.

The details of these benchmarks are shown in Table 2, including parallelization options k_l of each loop, array references, beneficial data reuse options, the number of on-chip RAM blocks B_{ij} and loading time of data reuse arrays C_{ij} required by a data reuse option of each array reference. B_{ij} and C_{ij} are determined by our previous work in [3].

Given all input parameters, the problem formulated in Section 3 can be solved by YALMIP [16], which is a MATLAB toolbox for solving optimization problems. All designs given by YALMIP have been implemented in Handel-C and mapped onto the Xilinx XC2v8000 FPGA with 168 on-chip RAM blocks to verify the proposed framework, as shown in Fig. 3, 4 and 5. In this section, we use $(OP_{1j}, OP_{2j}, \ldots, OP_{Rj}, k_1, k_2, \ldots, k_N)$ to denote a design with data reuse options OP_{ij} and parallelization options k_l.

In subfigures (a) of Fig. 3, 4 and 5, for every amount of on-chip RAM blocks between 0 and 168, the designs with the optimal performance estimated by the proposed framework are shown and are connected using bold lines to form the performance-optimal Pareto frontier. For example, in Fig. 3 (a), the proposed design using fewer than 6 on-chip RAM blocks is $(OP_{12}, OP_{21}, k_1 = 1, k_2 = 2, k_3 =$

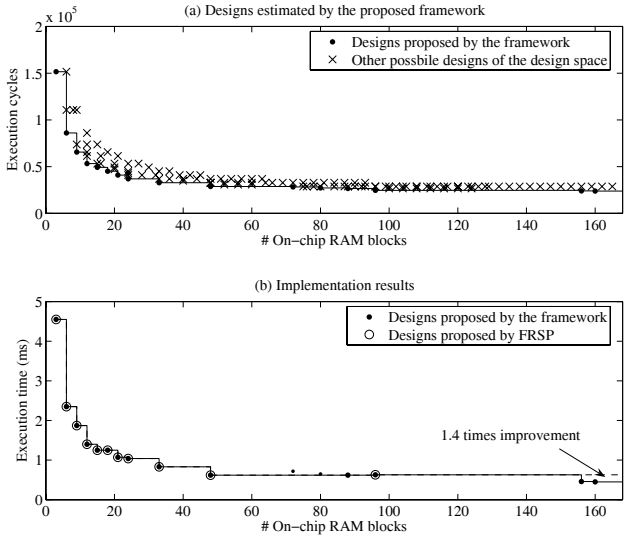

Fig. 3. Experimental results of MAT64. (a) Performance-optimal design Pareto frontier proposed by the framework. (b) Implementation of designs proposed by the framework and the FRSP approach on an FPGA.

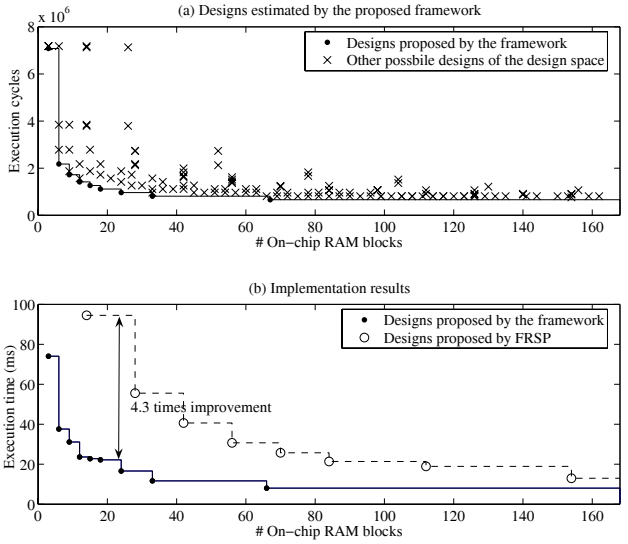

Fig. 4. Experimental results of FSME. (a) Performance-optimal design Pareto frontier proposed by the framework. (b) Implementation of designs proposed by the framework and the FRSP approach on an FPGA.

1), the leftmost one; for an on-chip RAM consisting of 80 blocks it is $(OP_{11}, OP_{21}, k_1 = 6, k_2 = 6, k_3 = 1)$; and if the number of on-chip RAM blocks is fewer than 3, then the proposed design is the sequential code $(k_1 = 1, k_2 = 1, k_3 = 1)$ without data reuse and data-level parallelization. It can be seen in Figs. 3 (a), 4 (a) and 5 (a) that the number of execution cycles decreases as the number of on-chip RAM blocks increases, because the degree of parallelism increases. To demonstrate the advantage of the optimization framework, some other possible designs randomly sampled from the space of feasible solutions in the design space of each benchmark, *i.e.* other combinations of data reuse options OP_{ij} and parallelization options k_l, are also plotted in these figures. These designs are all above the performance-optimal Pareto frontier and have been automatically rejected by the framework. It is shown that when the on-chip RAM constraint is tight, the optimization framework does a particularly good job at selecting high speed solutions.

The actual execution times, after synthesis, placement and routing effects are accounted for, are plotted in Figs. 3 (b), 4 (b) and 5 (b). In these figures, the designs proposed by the framework are shown in dots and the corresponding performance-optimal design Pareto frontiers are drawn using bold lines. Clearly, there are the similar descending trends of the frontiers in (a) and (b) over the number of on-chip RAM blocks for three kernels. There exist a few exceptions in Figs. 3 (b) and 5 (b), where the performance of some designs, shown in dots above the Pareto frontier, become worse as the on-chip memory increases. This is because on-chip RAMs of the Virtex-II FPGA we have used are located in columns across the chip and as the number of required RAMs increases the delay of accessing data from the RAMs, which are physically far from the datapath, is increased, degrading the clock frequency. However, for most cases, the proposed framework estimates the relative merits of different designs and indicates the optimal designs for each kernel in the context of different on-chip memory constraints.

In addition, the designs obtained using the approach in [1], which first explores data reuse under the same range of memory constraints to maximize data locality and then maximizes data-level parallelization based on the decision made in the first phase and the remaining memory resources (we denote this method as FRSP here), have been implemented as well. These designs are plotted in Figs. 3 (b), 4 (b) and 5 (b) in circles and form performance Pareto frontier in dashed lines. By comparing the performance-optimal Pareto frontiers obtained by our framework and the FRSP method for each kernel, we can see that the performance improvement up to 1.4, 4.3 and 4.7 times, respectively, have been achieved by using the proposed framework. These show the advantage of the proposed framework that explores data reuse and data-level parallelization at the same time. For the MAT64 case, the FRSP yields designs which are almost the same as those our framework proposes, because the regularity of the algorithm means that different data reuse options have similar effects on the on-chip memory requirement and performance, as can be seen in Table 2.

However, for the FSME and the Sobel algorithms, where the two problems are significantly less separable, our approach can produce far superior designs. Also note that in

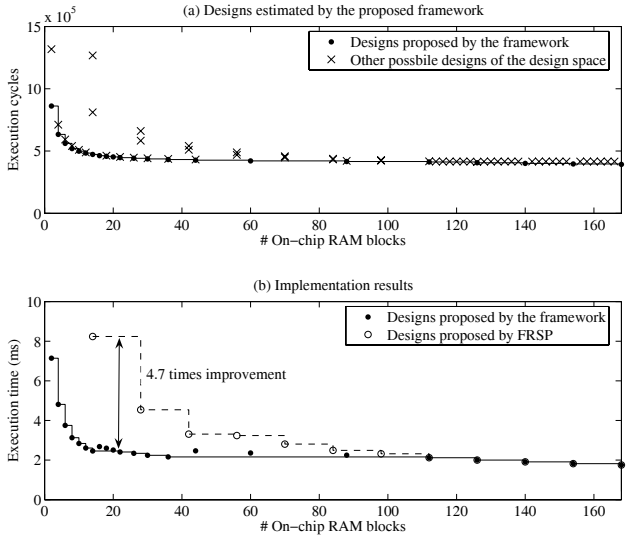

Fig. 5. Experimental results of Sobel. (a) Performance-optimal design Pareto frontier proposed by the framework. (b) Implementation of designs proposed by the framework and the FRSP approach on an FPGA.

these figures as the number of RAMs available on-chip increases the performance-optimal Pareto frontiers obtained by both approaches converge. This is because data reuse and data-level parallelization problems become decoupled when there is not an on-chip memory constraint. In other words, the proposed optimization framework is particularly valuable in the presence of tight on-chip memory constraints.

As the degree of parallelism increases, the number of slices used by the designs of the kernels also increases. We have not included such a constraint in the present compilation framework, because on-chip memory is always exhausted before slice logic and the proportion of occupied slices to device size is below 20% in most cases. Moreover, all optimal designs in this paper were generated under 10 seconds using the proposed framework.

5. CONCLUSIONS

A geometric programming (GP) framework for improving the system performance by combining data reuse exploitation with data-level parallelization in FPGA-based platforms has been presented in this paper. We formulate the problem of exploiting data reuse and data-level parallelization subject to the on-chip memory constraint as an Integer GP problem, and apply an existing solver to solve it. A limited number of variables are used to formulate the problem, fewer than 10 variables for each of the benchmarks used in this paper. Thus, the exploration of the design space is efficiently automated. The framework has been applied to three real benchmarks, and the results demonstrate that the proposed framework has the ability to determine the performance-optimal design, among all possible designs under the on-chip memory constraint. Performance improvements up to 4.7 times have been achieved compared with the FRSP method.

In this framework, the reused elements of an array reference are duplicated for all parallel processors. In the future, we will extend the framework with an optimized data partitioning method to buffer reused data locally for only those processors which access them.

References

[1] F. Catthoor *et al.*, *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Norwell, MA, USA: Kluwer Academic Publishers, 1998.

[2] Q. Liu *et al.*, "Data reuse exploration for FPGA based platforms applied to the full search motion estimation algorithm," in *Int. Conf. FPL*, 2006, pp. 389–394.

[3] ——, "Automatic on-chip memory minimization for data reuse," in *Int. Conf. FCCM*, 2007, pp. 389–394.

[4] M. Kandemir *et al.*, "A compiler-based approach for dynamically managing scratch-pad memories in embedded systems," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Syst.*, vol. 23, no. 2, pp. 243–260, Feb. 2004.

[5] M. Weinhardt and W. Luk, "Memory access optimization for reconfigurable systems," in *IEE Proc. Computers and Digital Techniques*, 2001, pp. 105–112.

[6] N. Baradaran *et al.*, "Compiler reuse analysis for the mapping of data in FPGAs with RAM blocks," in *IEEE Int. Conf. FPT*, 2004, pp. 145–152.

[7] Z. Guo *et al.*, "Input data reuse in compiling window operations onto reconfigurable hardware," *SIGPLAN Not.*, vol. 39, no. 7, pp. 249–256, 2004.

[8] V. Lefebvre and P. Feautrier, "Automatic storage management for parallel programs," *Parallel Comput.*, vol. 24, no. 3-4, pp. 649–671, 1998.

[9] M. Gupta and P. Banerjee, "Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 3, no. 2, pp. 179–193, 1992.

[10] U. K. Banerjee, *Loop Parallelization*. Norwell, MA, USA: Kluwer Academic Publishers, 1994.

[11] U. Eckhardt and R. Merker, "Hierarchical algorithm partitioning at system level for an improved utilization of memory structures," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Syst.*, vol. 18, no. 1, pp. 14–24, 1999.

[12] I. Issenin *et al.*, "Multiprocessor system-on-chip data reuse analysis for exploring customized memory hierarchies," in *Proc. Conf. DAC*, 2006, pp. 49–52.

[13] M. Dasygenis *et al.*, "Power and performance exploration of embedded systems executing multimedia kernels," in *IEE Proc. Computers and Digital Techniques*, 2002, pp. 164–172.

[14] R. P. Wilson *et al.*, "SUIF: an infrastructure for research on parallelizing and optimizing compilers," *SIGPLAN Not.*, vol. 29, no. 12, pp. 31–37, 1994.

[15] S. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge University Press, 2004.

[16] J. Lfberg, "Yalmip : A toolbox for modeling and optimization in MATLAB," in *Proc. Conf. CACSD*, 2004.

[17] V. Bhaskaran and K. Konstantinides, *Image and Video Compression Standards: Algorithms and Architectures*. Norwell, MA, USA: Kluwer Academic Publishers, 1997.

[18] http://www.pages.drexel.edu/∼weg22/edge.html, accessed Aug. 2006.