Combining Data Reuse With Data-Level Parallelization for FPGA Targeted Hardware Compilation: a Geometric Programming Framework

Qiang Liu, George A. Constantinides, *Senior Member, IEEE,* Konstantinos Masselos, *Member, IEEE,* and Peter Y. K. Cheung, *Senior Member, IEEE,*

***Abstract*—A nonlinear optimization framework is proposed in this paper to automate exploration of the design space consisting of data reuse (buffering) decisions and loop-level parallelization, in the context of FPGA-targeted hardware compilation.**

Buffering frequently accessed data in on-chip memories can reduce off-chip memory accesses and open avenues for parallelization. However, the exploitation of both data reuse and parallelization is limited by the memory resources available on-chip. As a result, considering these two problems separately, *e.g.* first exploring data reuse and then exploring data-level parallelization, based on the data reuse options determined in the first step, may not yield the performance-optimal designs for limited on-chip memory resources. We consider both problems at the same time, exposing the dependence between the two. We show that this combined problem can be formulated as a nonlinear program, and further show that efficient solution techniques exist for this problem, based on recent advances in optimization of so-called *geometric programming* problems.

Results from applying this framework to several real benchmarks implemented on a Xilinx device demonstrate that given different constraints on on-chip memory utilization, the corresponding performance-optimal designs are automatically determined by the framework. We have also implemented designs determined by a two-stage optimization method that first explores data reuse and then explores parallelization on the same platform, and by comparison the performance-optimal designs proposed by our framework are faster than the designs determined by the two-stage method by up to 5.7 times.

***Index Terms*—Data-level parallelization, data reuse, FPGA hardware compilation, geometric programming, optimization.**

I. INTRODUCTION

As modern FPGAs' size, capabilities and speed increase, 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 [1]. Meanwhile, the complex applications and plurality of hardware resources increase the complexity of FPGA-based system design. As a result, how to efficiently exploit the flexibility provided by heterogeneous reconfigurable resources on FPGAs to achieve an optimal design while shrinking the design cycle has become a serious issue faced by designers. 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.

An 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* [2], has been investigated in depth. In our previous work, a systematic approach for data reuse exploration in applications involving arrays accessed in loop nests has been proposed [3], [4], [5], where the architecture exploiting a scratch-pad memory to load and store reused data is shown in Fig. 1 (b).

Loop nests are the main source of potential parallelism, and loop-level parallelization has been widely used for improving performance [6]. However, the performance improvement is in fact limited by the number of parallel data accesses to fetch the operands. There are a number of embedded RAM blocks on modern FPGAs, often with two independent read/write ports. Therefore, buffering data in on-chip RAMs can increase memory access bandwidth and open avenues for parallelism.

In this paper, we present an approach that buffers potentially reused data on-chip and duplicates the data into different dual-port memory banks embedded in modern FPGAs to reduce off-chip memory accesses and increase loop-level parallelism. The target hardware structure is shown in Fig. 1 (c). Each PU executes a set of loop iterations and all PUs run in parallel. Because different sets of loop iterations may access the same data (data reuse), every dual-port RAM bank holds a copy of buffered data, and is accessed by two PUs through its two ports. In this work, registers are not used as on-chip data reuse buffers, although the proposed framework could be combined with register-oriented work [7]. Therefore, in this paper, the on-chip memory cost is measured in units of atomic on-chip embedded RAM blocks.

To our knowledge, there exist only few works exploiting both data reuse and loop-level parallelization [3], [8], [9]. However, there is no prior design flow combining data reuse exploration and loop-level parallelization within 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 [4]. We consider the code to have been pre-

Q. Liu, G. A. Constantinides and P. Y. K. Cheung are with the Department of Electrical and Electronic Engineering, Imperial College, London SW7 2BT, U.K. e-mail: {qiang.liu2, g.constantinides, p.cheung}@imperial.ac.uk.

K. Masselos is with university of Peloponnese, Tripolis, Greece. e-mail: k.masselos@imperial.ac.uk.

processed by a dependence analysis tool such as [10] and each loop to have been marked as parallelizable or sequential. The parallelization decision is to decide an appropriate strip-mining for each loop level in the loop nest [11]. Performing these two tasks separately may not lead to performance-optimal designs. 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 inefficiently large on-chip memory requirements to hold all the operands, and large run-time penalties for loading data, many of which may not be reused, from off-chip into on-chip memories. A more sensible approach is to first make data reuse design decisions to minimize off-chip accesses and secondly improve the parallelism of the resulting code [2]. However, once the decision is made to fix the loop level where on-chip buffers are inserted, 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 using a geometric programming framework [12]. The overall optimization takes place while respecting an on-chip RAM utilization constraint and the dependence between the two problems. The main contributions of this paper are thus:

- recognition of the link between data reuse and loop-level parallelization and optimization of both problems within a single step,
- an integer geometric programming formulation of the exploration of data reuse and data-level parallelization for performance optimization under an on-chip memory constraint, revealing a computationally tractable lower-bounding procedure, and thus allowing solution through branch and bound,
- the application of the proposed framework to several signal and video processing kernels, resulting in performance improvements up to 5.7 times compared with a two-stage method that first explores data reuse and then performs loop parallelization.

The rest of the paper is organized as follows. Section II describes related work. Section III presents a motivational example and Section IV precisely states the targeted problem. Section V formulates the problem in a naïve way, and Section VI shows that via a change of variables and re-ordering, the problem can be re-formulated as an integer geometric program. Section VII presents results from applying our proposed framework to several real benchmarks. Section VIII concludes the paper and suggests future work.

II. Background

The two areas of optimizing memory architectures for data reuse and techniques for automated parallelization have been extensively investigated over the last decade.

A number of approaches for optimizing memory configurations have been proposed for embedded systems. A systematic methodology for data reuse exploration is proposed in [2], where a cost function of power and area of the memory system is evaluated, in order to decide promising data reuse options and the corresponding memory hierarchy. In [13], [14]

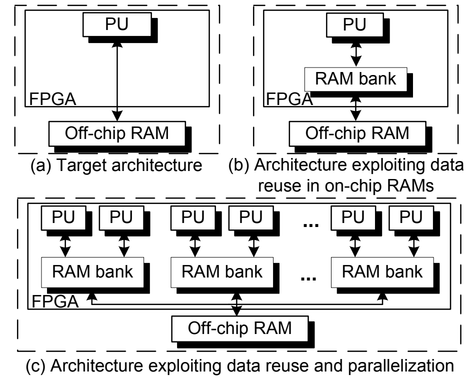

Fig. 1. Target platform.

and [15], approaches for exploiting data reuse in scratch-pad memory (SPM) have been presented. In [13] large arrays are divided into *data blocks* and computations that access the same data block are scheduled as close as possible in time slots using a greedy heuristic to maximize data reuse with minimum on-chip memory requirements. Approaches in [14] and [15] determine which data should be transferred into SPM, and when and where in a code these transfers happen to improve the performance of the code, based on memory access cost models. Research into buffering reused data in FPGA on-chip RAMs and registers has been carried out in [16], [7], [8] and [5]. In [16] applications speed up through pipelining with high data throughput, which is obtained by storing reused data in shift registers and shift on-chip RAMs. In [7] and [8], arrays more beneficial to minimize the memory access time are stored in either registers or on-chip RAMs if register is not available. The work in [5] formulates the problem of data reuse exploration aimed at low power as the Multi-Choice Knapsack Problem.

Improvement of the parallelism of programs has been a hot topic in the computing community. Loop transformations have been used in [6] to enable loop parallelization for programs with perfectly nested loops. Lefebvre *et al.* [17] propose an approach for parallelizing static control programs, which introduces memory reuse to the single assignment transformation of the programs in order to reduce memory requirements for parallelization. Gupta *et al.* [18] identify two objectives, in conflict with each other, distribution of data on as many processors as possible and reduction in the communication among processors, in data distributions over multiple processors. They propose a technique for performing data alignment and distribution to determine data partitions suitable to the distributed memory architecture. There is also significant work from the systolic array community on mapping sequential programs onto systolic arrays with unlimited and limited processors [19]. A systematic description of hardware architectures and software models of multiprocessor systems has been published in [20].

However, exploration of both data reuse and parallelization has been discussed in only a few previous works. Eckhardt *et al.* [21] recursively apply the locally sequential globally parallel and the locally parallel globally sequential partitioning

schemes to mapping an algorithm onto a processor array with a two-level memory hierarchy. Data reuse is considered during the algorithm partitions in order to reduce accesses to high memory levels. However, the purpose of this work is to model the target architecture in order to partition the algorithm and to improve memory utilization, rather than a system design exploration. A greedy algorithm is proposed in [22] for mapping array references resident in computation critical paths onto FPGA on-chip memory resources to provide parallel data accesses for instruction-level parallelism. The algorithm only produces a locally optimal solution. Data access patterns of arrays are considered and an array with reused data is mapped on-chip as a whole and duplicated into different memory banks, while in our approach reused data of an array could be partially buffered and duplicated on-chip and on-chip buffers are updated at run-time to improve efficiency of memory usage. In [9] loop transformations, such as loop unrolling, fusion and strip-mining, are performed before the exploitation of data reuse. The work [3] focuses on a systematic approach for the derivation of data reuse options and describes an empirical experiment that demonstrates the potential for data reuse and parallelization. However, no systematic formulation is proposed for this combined optimization. In [23] and [24], 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.

Previous research has not formulated the problem of exploring data reuse and loop parallelization at the same time. The motivation of this work is to investigate this, in order to improve system performance under an on-chip memory utilization constraint in FPGA-based platforms. Specifically, in the proposed framework, this problem is formulated as an INLP problem exhibiting a convex relaxation and existing solvers for NLP problems are applied to solve it. As a result, this exploration problem is automated and system designs with optimal performance are determined at compile time.

III. Motivational Example

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' [25] or Handel-C's 'replicated par' [26]. 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. In this work, we obtain high performance by buffering frequently accessed data on chip in scratch-pad memories to reduce off-chip memory access, and by replicating these data in distinct dual-port memory banks to allow multiple loop iterations to execute in parallel.

To illustrate these two related design issues, an example, matrix-matrix multiplication (MAT), is shown in Fig. 2. The original code in Fig. 2 (a) consists of three regularly nested loops. The matrices A, B, and C are stored in off-chip

```
Do i=0, N-1
  Do j=0, N-1
    s=0;
    Do m=0, N-1
      s=s+A[i][m] x B[m][j];
    Enddo;
    C[i][j]=s;
  Enddo;
Enddo;
```

(a) Original code

```
Load(RLB0s, B);
Do i=0, N-1
  Load(RLA1s, A);
  Doall j1=1, kj
    Do j2=⌈N/kj⌉(j1-1) to min(N-1,⌈N/kj⌉j1-1)
      sj1=0;
      Do m=0, N-1
        sj1=sj1+RLA1⌈j1/2⌉[m] x RLB0⌈j1/2⌉[m][j2];
      Enddo;
      Store(sj1, C);
    Enddo;
  Enddoall;
Enddo;
```

(b) Loop j could be parallelized

```
Load(RLA0s, A);
Load(RLB0s, B);
Doall i1=1, ki
  Doall j1=1, kj
    Do i2=⌈N/ki⌉(i1-1) to min(N-1,⌈N/ki⌉i1-1)
      Do j2=⌈N/kj⌉(j1-1) to min(N-1,⌈N/kj⌉j1-1)
        si1j1=0;
        Do m=0, N-1
          si1j1=si1j1 + RLA0⌈i1/2⌉⌈j1/2⌉[i2][m] x RLB0⌈i1/2⌉⌈j1/2⌉[m][j2];
        Enddo;
        Store(si1j1, C);
      Enddo;
    Enddo;
  Enddoall;
Enddoall;
```

(c) Two loops i and j could be parallelized

Fig. 2. Matrix-matrix multiplication example.

memory. This code exhibits data reuse in accesses to the arrays A and B; for example, for the same iterations of the loops i and k, different iterations of loop j read the same array element $A[i][k]$. Also, this code presents potential parallelism in loop i or j, that can be revealed by [10]. 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.

Following the approach in [3] to exploit data reuse, a *data reuse array*, stored in on-chip SPM, is introduced to buffer elements of an array which is stored in off-chip memory and frequently accessed in a loop nest. Before those elements are used they are first loaded into the data reuse array in the on-chip memory from the original array. Such a data reuse array may be introduced at any level of the loop nest, forming different data reuse options for the original array and giving rise to different tradeoffs in on-chip memory size versus off-chip access count [4]. To avoid redundant data copies, only beneficial data reuse options, in which the number of off-chip accesses to load the data reuse arrays is smaller than the number of on-chip accesses to them, are considered.

With respect to these rules, the array A has two beneficial data reuse options shown in Fig. 2 (b) and (c): loading the data reuse array RLA between loops i and j, or outside loop i, respectively. Assuming the matrices are 64×64 with 8-bit entries, both options obtain a 64-fold reduction in off-chip memory accesses, whereas they differ in on-chip RAM requirements, needing one and two 18 kbits RAM blocks, respectively, on our target platform with a Xilinx XC2v8000 FPGA. Similarly, the array B owns one beneficial data reuse option, in which the data reuse array RLB is loaded outside loop i, with 64 times reduction in off-chip memory accesses and a memory requirement of two RAM blocks.

In prior work, the goal of data reuse has been to select appropriate loop-levels to insert data reuse arrays for all array

references, in order to minimize off-chip memory accesses with the minimum on-chip memory utilization [3]. Following this approach, if there are more than two on-chip RAM blocks available, an optimal choice would be to select the first data reuse option for A and the single data reuse option for B as shown in Fig. 2 (b).

Since the data reuse arrays are stored on-chip, we use the dual-port nature of the embedded RAMs and replicate the data in different RAM banks to increase the parallel accesses to the data. This is also illustrated in Fig. 2 (b), where loop j is strip-mined and then partially parallelized, utilizing k_j distinct parallel processing units in the FPGA hardware. As a result, $\lceil k_j/2 \rceil$ copies of a row of matrix A and $\lceil k_j/2 \rceil$ copies of matrix B are held on-chip in arrays $RLA1_{\lceil j_1/2 \rceil}$ and $RLB0_{\lceil j_1/2 \rceil}$ mapped onto dual-port RAM banks. The parameter k_j thus allows a tuning of the tradeoff between on-chip memory and compute resources and execution time. Note that for this selection of data-reuse options, loop i cannot be parallelized because parallel access to the array A is not available, given single port is available for off-chip memory access. Had the alternative option, shown in Fig. 2 (c), been chosen, we would have the option to strip-mine and parallelize loop i as well.

Therefore, exploiting data-reuse to buffer reused data on-chip and duplicating the data over multiple memory banks make data-level parallelization possible, resulting in performance improvements while the number of off-chip memory accesses is reduced. However, if data-reuse decision is made prior to exploring parallelization, then the potential parallelism existing in the original code may not be completely explored. Proceeding with the example, if we first explore data reuse, then the data reuse option shown in Fig. 2 (b) will be chosen as discussed above. Consequently, the opportunity of exploring both parallelizable loops i and j is lost. In other words, the optimal tradeoff between on-chip resources and execution time may not be carried out. This observation leads the conclusion that parallelism issues should be considered when making data-reuse decisions.

The issue is further complicated by the impact of *dynamic single assignment form* [27] on memory utilization. Notice that the temporary variable s in Fig. 2 (a) storing intermediate computation results has been modified in Fig. 2 (b) and (c) so that each parallel processor writes to a distinct on-chip memory location through independent ports, avoiding contention. Final results are then output to off-chip memories sequentially. For the MAT example, in Fig. 2 (b), for 64×64 matrices with 8-bit entries, the on-chip memory requirement after data-level parallelization is $4 \times \lceil k_j/2 \rceil$ RAM blocks, on the target platform with a Xilinx XC2v8000 FPGA. Similarly, the on-chip memory requirement in Fig. 2 (c) is $5 \times \lceil k_i k_j/2 \rceil$ RAM blocks.

Therefore, greater parallelism requires more on-chip memory resources, because 1) reused data need to be replicated into different on-chip memory banks to provide the parallel accesses required; and 2) temporary variables need to be expanded to ensure that different statements write to different memory cells.

In the following section, we will generalize the discussion above, before proposing a methodology to make such decisions automatically.

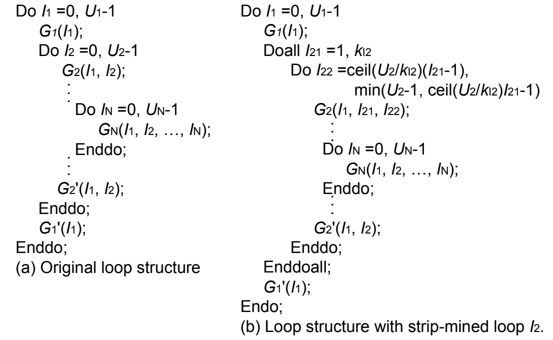

Fig. 3. Target loop structure.

IV. Problem Statement

In this paper, we target a N-level regularly nested loops $(I_1, I_2, \ldots, I_N)$ with R references to off-chip arrays, where I_1 corresponds to the outer-most loop, I_N corresponds to the inner-most loop, $G_j(I_1, I_2, \ldots, I_j)$ and $G'_j(I_1, I_2, \ldots, I_j)$ are groups of sequential assignment statements inside loop I_j but outside loop I_{j+1}, as shown in Fig 3 (a). Without loss of generality, in line with Handel-C, we assume that each assignment takes one clock cycle. When a data reuse array is introduced at loop j for a reference which is accessed within loop j, a code for loading buffered data from off-chip memory into the on-chip array is inserted between loops $j-1$ and j within group G_{j-1} and executes sequentially with other statements. Inside the code, the off-chip memory accesses are pipelined and a datum is loaded into the data reuse array in one cycle after few initiation cycles. The data reuse array is then accessed within loop j in stead of the original reference. 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. The iterations of other untouched loops still run sequentially. For example, Fig. 3 (b) shows the transformed loop structure when loop I_2 is strip-mined. In this situation, the inner loops $(I_{22}, I_3, \ldots, I_N)$ form a sequential segment, and for a fixed iteration of loop I_1 and all iterations of loop I_{21} the corresponding k_{I_2} segments execute in parallel.

Given N nested loops and each loop l with k_l parallel partitions, the on-chip memory requirement for exploiting data reuse and loop-level parallelization is $\lceil (\prod_{l=1}^{N} k_l)/2 \rceil (B_{temp} + B_{reuse})$, where B_{temp} is the number of on-chip RAM blocks required by expanding temporary variables, B_{reuse} is the number of on-chip RAM blocks required by buffering a single copy of the reused data of all array references, and the divisor 2 is due to dual-port RAM. As the number of partitions increases, the on-chip memory requirement increases quickly. Therefore, we can adjust the number of partitions $\{k_l\}$ to fit the design into the target platform with limited on-chip memory.

Complete enumeration of the design space of data reuse and data-level parallelization is expensive for applications with

multiple loops and array references. Given N-level regularly nested loops surrounding R array references, each array reference could have N data reuse options: to insert a data reuse array before the entire nested loop structure or inside any one of the N loops except the inner-most loop, and thus there are N^R data reuse options in total. Moreover, there could be $\prod_{l=1}^{N} L_l$ data-level parallelization options under each data reuse option, where L_l is the number of iterations of the loop l. As a result, the design space maximally consists of $N^R \prod_{l=1}^{N} L_l$ design options, which increases exponentially with the number of array references R and the number of loop levels N. Therefore, we want to formulate this problem in a manner that allows for an automatic and quick determination of an optimal design at compile time.

V. Problem formulation

We formulate this problem as an Integer Non-Linear Programming (INLP) problem and will show in the next section that the INLP problem can be transformed to an integer geometric program, which has a convex relaxation [12], allowing efficient solution techniques.

For ease of description, we formulate the problem in this paper for the case where a set of R references $A_1, A_2, \ldots, A_R$ to arrays is present inside the inner-most loop of a N-level loop nest (the proposed framework can be extended to allow array references to exist at any loop level). As only data reuse options in which the number of off-chip accesses is smaller the number of on-chip accesses are considered, reference A_i could have a total of E_i ($0 \leq E_i \leq N$) beneficial data reuse options $OP_{i1}, OP_{i2}, \ldots, OP_{iE_i}$. E_i equal to zero means that there is no reason to buffer data on-chip. 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 ($1 \leq l \leq N$) can be partitioned into k_l ($1 \leq k_l \leq L_l$) pieces. 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 I. Based on these notations, the problem of exploration of data reuse and data-level parallelization is defined in equation (1)–(6), described in detail below.

$$\min : \sum_{s=1}^{S} \prod_{l=1}^{W_s} \lceil \frac{L_l}{k_l} \rceil + \sum_{i=1}^{R} \sum_{j=1}^{E_i} \rho_{ij} C_{ij} \tag{1}$$

subject to

$$\lceil \frac{1}{2} \prod_{l=1}^{N} k_l \rceil B_{temp} + \lceil \frac{1}{2} \prod_{l=1}^{N} k_l \rceil \sum_{i=1}^{R} \sum_{j=1}^{E_i} \rho_{ij} B_{ij} \leq B \tag{2}$$

$$\sum_{j=1}^{E_i} \rho_{ij} = 1, 1 \leq i \leq R \tag{3}$$

$$\rho_{ij} \in \{0,1\}, 1 \leq j \leq E_i, 1 \leq i \leq R \tag{4}$$

$$k_l \geq 1, 1 \leq l \leq N \tag{5}$$

$$k_l - (L_l - 1) \textstyle\sum_{j=1}^{l} \rho_{ij} \leq 1$$
$$1 \leq l \leq N, 1 \leq j \leq E_i, i \in Q_l \tag{6}$$

TABLE I
A list of notations for variables (v) and parameters (p).

Notation	Description	
ρ_{ij}	binary data reuse variables	v
k_l	# partitions of loop l	v
v_l	# iterations in one partition of loop l	v
d	# duplications of reused data	v
S	# statements in a program	p
W_s	loop level of statement s in a loop nest	p
N	# loops	p
R	# array references	p
Q_l	the set of indices of array references within loop l	p
E_i	# data reuse options of array reference i	p
L_l	# iterations of loop l	p
B_{temp}	# on-chip RAM blocks for storing temporary variables	p
B	# on-chip RAM blocks available	p
B_{ij}	# on-chip RAM blocks for the data reuse array of option j of reference i	p
C_{ij}	# loading cycles of the data reuse array of option j of reference i	p

In this formulation, all capitals are known parameters at compile time, and all variables are integers. The objective function (1) and the constraint function (2) are not linear due to the *ceil* functions and the *products*, resulting in an INLP (Integer Non-Linear Programming) problem. The INLP 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 the additional time required to load data into on-chip buffers, given each statement takes one clock cycle. In the first part, $W_s = 1$ means that statement s is located between the first loop and the second loop and $W_s = N$ means it is located inside the innermost loop. The *ceil* function here guarantees that all iterations of the loop l are executed after parallelization. In the second part, the data reuse variables ρ_{ij} are binary variables, which is guaranteed by (4). ρ_{ij} taking value one means the data reuse option OP_{ij} is selected for the reference A_i. Equality (3) ensures that exact 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 and the second addend expresses the number of on-chip RAM blocks taken by reused data of all array references. The *ceil* function indicates the number of times the on-chip buffers are duplicated. 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 segments accommodate all PUs with data. 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), where $Q_l \subseteq \{1, 2, \ldots, R\}$ is a subset of array reference indices such that if $i \in Q_l$ then array reference A_i is inside loop l, show the link between data reuse variables ρ_{ij} and loop partition variables k_l. The essence of this constraint is that *a 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*. For the example in Fig. 2 (b), loop j can be

strip-mined and be executed in parallel, while loop i cannot. It is this observation that is exploited within this framework to remove redundant design options combing data reuse and data-level parallelization.

The design exploration of data reuse options and data-level parallelization is thus formulated as an INLP problem, by means of at most RN data reuse variables $\{\rho_{ij}\}$ and N loop partition variables $\{k_l\}$. In this manner, $N(R+1)$ variables are used to explore the design space with $N^R \prod_{l=1}^{N} L_l$ options.

The enumeration of the design space can be avoided if there is an efficient way to solve this INLP problem. In the next section, it will be shown that how this formulation is transformed into a geometric program.

VI. Geometric programming transformation

The problem of exploring data reuse and data-level parallelization to achieve designs with optimal performance under an on-chip memory constraint has been formulated in a naïve way as an INLP problem. However, there are no effective methods to solve a general NLP problem because they may have several locally optimal solutions [12]. Recently, the geometric program has achieved much attention [12]. The geometric program is the following optimization problem:

$$\begin{array}{ll} \text{min :} & f_0(x) \\ \text{subject to} & f_i(x) \le 1, \quad i = 1, \dots, m \\ & h_i(x) = 1, \quad i = 1, \dots, p \end{array}$$

where the objective function and inequality constraint functions are all in *posynomial* form, while the equality constraint functions are *monomial*. A monomial $h_i(x)$ is a function $h_i(x) = c x_1^{a_1} x_2^{a_2} \dots x_n^{a_n}$ with $x \in \mathbb{R}^n$, $x > 0$, $c > 0$ and $a_i \in \mathbb{R}$, and a posynomial $f_i(x)$ is a sum of monomials. The reason for posynomial requirement is that posynomial functions can be transformed into convex functions, whereas this is not the case for general polynomials. By replacing variables $x_i = e^{y_i}$ and taking the logarithm of the objective function and constraint functions, the geometric program can be transformed to a convex form. The importance of this observation is that unlike general NLPs, convex NLPs have efficient solution algorithms with guaranteed convergence to a global minimum [12].

The INLP given in Section V can be transformed into an integer geometric program. We first remove the *ceil* functions from the original problem by introducing two constraints with auxiliary integer variables v_l and d as shown below, and variables v_l and d take the least integers satisfying the constraints. After that, we substitute variables $\rho'_{ij} = \rho_{ij} + 1$ for the variables ρ_{ij} and perform expression transformations by means of logarithm and exponential to reveal the geometric programming characteristic of the original problem. Finally, the INLP in Section V is transformed to the following:

$$\text{min :} \sum_{s=1}^{S} \prod_{l=1}^{W_s} v_l + \sum_{i=1}^{R} \sum_{j=1}^{E_i} (\rho'_{ij} - 1) C_{ij} \tag{7}$$

subject to

$$dB_{temp} + d \sum_{i=1}^{R} \prod_{j=1}^{E_i} {\rho'_{ij}}^{\log_2 B_{ij}} \le B \tag{8}$$

$$\sum_{j=1}^{E_i} (\rho'_{ij} - 1) = 1, 1 \le i \le R \tag{9}$$

$$\rho'_{ij} \in \{1, 2\}, 1 \le j \le E_i, 1 \le i \le R \tag{10}$$

$$k_l^{-1} \le 1, 1 \le l \le N \tag{11}$$

$$\begin{array}{c} k_l \prod_{j=1}^{l} {\rho'_{ij}}^{-\log_2 L_l} \le 1 \\ 1 \le l \le N, 1 \le j \le E_i, i \in Q_l \end{array} \tag{12}$$

$$L_l k_l^{-1} v_l^{-1} \le 1, 1 \le l \le N \tag{13}$$

$$\frac{1}{2} d^{-1} \prod_{l=1}^{N} k_l \le 1 \tag{14}$$

Now, we can see that the relaxation of this problem, obtained by allowing k_l to be real values, and replacing (10) by $1 \le \rho_{ij} \le 2$, is exactly a geometric program. Note that the transformation between the original formulation in Section V and the convex geometric programming form just involves variable substitution and expression reorganization, rather then any approximation of the problem. As a result, the two problems are the equivalent. Thus, the existing methods for solving convex INLP problems can be applied to obtain the optimal solution to the problem.

A branch and bound algorithm used in [28] is applied to the framework to solve problem (7)–(14), using the geometric programming relaxation as a lower bounding procedure. The algorithm first solves the relaxation of the INLP problem, and if there exists an integral solution to this relaxed problem then the algorithm stops and the integral solution is the optimal solution. Otherwise, solving the relaxed problem provides a lower bound on the optimal solution and a tree search over the integer variables of the original problem starts. The efficiency of the branch and bound algorithm can be evaluated by the gap between the initial lower bound at the root node and the optimal solution and the number of search tree nodes. The larger gap and the more search nodes mean more time to obtain the optimal solution. We apply the proposed framework to several benchmarks in the next section and use this algorithm to obtain the optimal solutions. It shall be shown that our geometric programming bounding procedure generates a small gap.

VII. 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 the FPGA on-chip memory constraint, we have applied the framework to three kernels: full search motion estimation (FSME) [29], matrix-matrix multiplication of two 64×64 matrices (MAT64) and the Sobel edge detection algorithm (Sobel) [30].

On the target platform Celoxica RC300 used for our experiments, on-chip memory takes one cycle and off-chip memory

TABLE II
THE DETAILS OF THREE KERNELS.

Kernel	k_l	Ref.	Option	B_{ij}	C_{ij}
FSME	$1 \le k_1 \le 36$	current	OP_{11}	13	25344
	$1 \le k_2 \le 44$		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	$1 \le k_1 \le 64$	A	OP_{11}	2	4096
	$1 \le k_2 \le 64$		OP_{12}	1	4096
	$k_3 = 1$	B	OP_{21}	2	4096
Sobel	$1 \le k_1 \le 144$	image	OP_{11}	13	25344
	$1 \le k_2 \le 176$		OP_{12}	1	76032
	$k_3 = 1$		OP_{13}	1	228096
	$k_4 = 1$	mask	OP_{21}	1	18

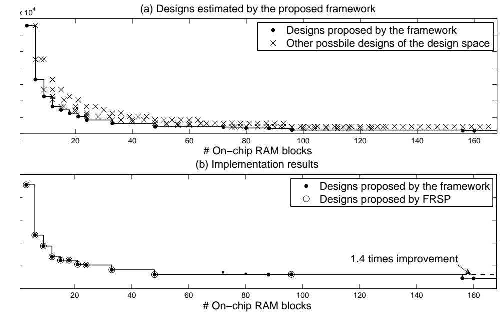

Fig. 4. Experimental results of MAT64. (a) Design Pareto frontier proposed by the framework. (b) Implementation of designs proposed by the framework and the FRSP approach on an FPGA.

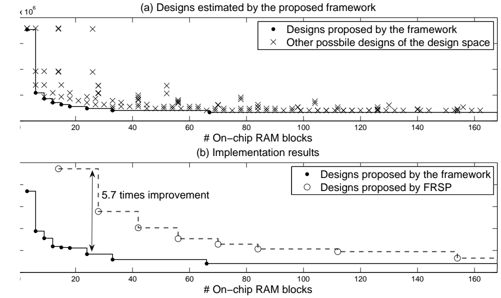

Fig. 5. Experimental results of FSME. (a) Design Pareto frontier proposed by the framework. (b) Implementation of designs proposed by the framework and the FRSP approach on an FPGA.

takes two cycles. We pipeline off-chip memory accesses to obtain a throughput of one access per cycle. Reused elements of each array reference in the kernels are buffered in on-chip RAMs and are duplicated in different banks. 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.

The benchmarks shown in Table II have been selected for their regularly rectangularly nested loops and multiple arrays. FSME is a classical algorithm of motion estimation in video processing [29]. It has six regularly nested loops, which is representative of the deepest loop nest seen in practice, and two array references, corresponding to the *current* and *previous* video frames. We consider three beneficial data reuse options for each of two array references, and in total there are 9 different data reuse designs, as shown in Table II. Matrix-matrix multiplication is involved in many computations and usually locates in the critical paths of the corresponding hardware circuits [31]. It has two array references multiplied in a 3-level loop nest, and there are 2 different data reuse designs. The Sobel edge detection algorithm is a classic operator in image processing [30]. It includes four loops, an *image* array with three beneficial data reuse options and a 3×3 *mask* array having one beneficial data reuse option. Similarly, there are 3 data reuse designs for the Sobel kernel. Moreover, the outermost two loops of these three kernels are parallelizable, and all parallelization options k_l are also presented in Table II. The number of on-chip RAM blocks B_{ij} and time for loading reuse data C_{ij}, listed in Table II, required by a data reuse option of each array reference are determined by our previous work [4].

Given all input parameters, the problem formulated in Section VI can be solved by YALMIP [28], which is a MATLAB toolbox for solving optimization problems. All designs given by YALMIP have been implemented in Handel-C [26] and mapped onto the Xilinx XC2v8000 FPGA with 168 on-chip RAM blocks to verify the proposed framework, as shown in Figs. 4, 5 and 6. 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 Figs. 4, 5 and 6, 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. 4 (a), the proposed design using fewer than 6 on-chip RAM blocks is $(OP_{12}, OP_{21}, k_1 = 1, k_2 = 2, k_3 = 1)$, 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. 4 (a), 5 (a) and 6 (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 proposed framework. It is shown that when the on-chip RAM constraint is tight, the optimization framework does a potentially good job at selecting high speed solutions.

The actual execution times, after synthesis, placement and routing effects are accounted for, are plotted in Figs. 4 (b), 5 (b) and 6 (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

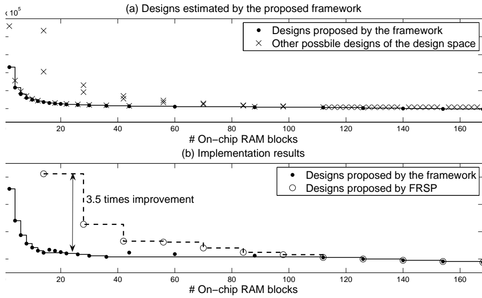

Fig. 6. Experimental results of Sobel. (a) Design Pareto frontier proposed by the framework. (b) Implementation of designs proposed by the framework and the FRSP approach on an FPGA.

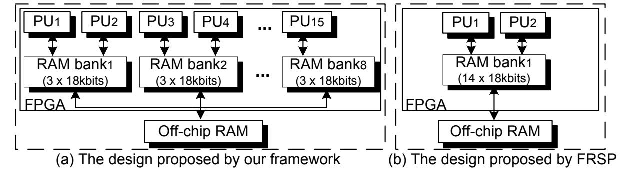

Fig. 7. The system architectures of the designs of FSME proposed by the framework and the FRSP method. (a) The design with 15 parallel processing units and 24 RAM blocks in 8 banks. (b) The design with 2 parallel processing units and 14 RAM blocks in 1 bank.

TABLE III
THE AVERAGE PERFORMANCE OF THE FRAMEWORK AND THE FRSP APPROACH.

	The framework			FRSP
Kernel	Time (s)	# nodes	Gap_rootnode (%)	Time (s)
FSME	6.0	61	12.2	1.1
MAT64	5.0	96	25.3	0.8
Sobel	10.6	136	13.5	1.4

in Figs. 4 (b) and 6 (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 by following the method that first explores data reuse and then data-level parallelization [2] (we denote this method as FRSP here) have been implemented as well. These designs are plotted in Figs. 4 (b), 5 (b) and 6 (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, 5.7 and 3.5 times, respectively, have been achieved by using the proposed framework for three benchmarks. 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 almost the same performance-optimal designs as those our framework proposes, because the different data reuse options of MAT64 have similar effects on the on-chip memory requirement and the execution time, as can be seen in Table II. In Fig. 7 the system architectures of the performance-optimal design of FSME, when there are 27 RAM blocks available on-chip, proposed by the framework and the FRSP method are shown respectively. The design in Fig. 7 (a) operates at the speed 5.7 times faster than the design in Fig. 7 (b) by trading off the off-chip memory access (two times more in former) and parallelism under the same memory constraint. Note that in Figs. 4 (b), 5 (b) and 6 (b) 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 no on-chip memory constraint. In other words, the proposed optimization framework is particularly valuable in the presence of tight on-chip memory constraints.

The number of slices, which are configured as control logics and registers, used by the designs of the kernels also increases as the degree of parallelism increases. If slice logic is scarce, a further constraint could be added to the framework. However, for most reasonable design tradeoffs in Figs. 4, 5 and 6, the slice utilization well below the RAM utilization, as a proportion of device size. Thus we have not included such a constraint in the present compilation framework.

The average execution time of the proposed framework and the FRSP method to obtain a performance-optimized design under different on-chip memory constraints for each benchmark are shown in Table III. On average, for three benchmarks, an optimal design under an on-chip memory constraint is generated by the framework within 11 seconds. This quick exploration is guaranteed by the quality of the lower bounds provided by the geometric programming relaxation, with average differences within 25.3% over the optimal solutions at the root node, resulting in an efficient branch-and-bound tree search of fewer than 150 nodes for each benchmark. The FRSP method is faster because the data reuse variables and loop parallelization variables are determined in two separate stages. However, the optimal solutions can not be guaranteed. Given small problems, brute-force which enumerates all candidate solutions is an alternative approach to find the optimal solutions to the problems. Nevertheless, this approach is not scalable as the problem size increases. We illustrate this through the matrix-matrix multiplication in Fig. 8. As the order of matrices increases, the time spent on the full enumeration increases exponentially, and exceeds the time required by the proposed framework.

VIII. CONCLUSIONS

A geometric programming framework for improving the system performance by combining data reuse with data-level parallelization in FPGA-based platforms has been presented in this paper. We originally formulate the problem as an INLP, reveal its geometric programming characteristic, 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,

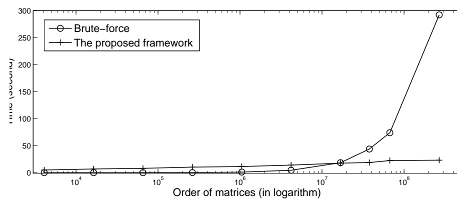

Fig. 8. Comparison of the average performance of the proposed framework and the brute-force method to obtain an optimal design for matrix-matrix multiplication under on-chip memory constraints.

in combination with the novel bounding procedure based on a convex geometric program, the exploration of the design space is efficiently automated and can be applied to high level hardware synthesis process. The framework has been applied to three signal and video processing kernels, 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 5.7 times have been achieved by the framework compared with the two-stage method.

In this framework, the reused elements of an array reference are duplicated for all parallel segments. For some memory access patterns this may result in unnecessary duplication of unaccessed data. The advantage of this simple data partition method is that there are no additional overheads on logic controls for the data partition. The disadvantage is that the data duplication may cause redundant requirement of on-chip memories, resulting in suboptimal designs. Therefore, in the future, we will extend the framework with an optimized data partition method to copy reused data only to those segments where they are accessed. Moreover, though the proposed framework is to optimize system performance, it can be extended to energy by multiplying the objective function (7) with a monomial power model similar to one proposed in [3]. The resultant problem formulation will be still an INLP with geometric programming relaxation. In the future, we may also combine this work with register-oriented work [7] and add a further constraint on register utilization to the framework.

REFERENCES

[1] T. Todman, G. Constantinides, S. Wilton, P. Cheung, W. Luk, and O. Mencer, "Reconfigurable computing: Architectures and design methods," *Proc. IEE Comput. Digital Techniques*, vol. 152, pp. 193–207, Mar. 2005.

[2] F. Catthoor, E. de Greef, and S. Suytack, *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Norwell, MA, USA: Kluwer Academic Publishers, 1998.

[3] Q. Liu, K. Masselos, and G. A. Constantinides, "Data reuse exploration for FPGA based platforms applied to the full search motion estimation algorithm," in *Proc. FPL '06*, Madrid, Spain, Aug. 2006, pp. 389–394.

[4] Q. Liu, G. A. Constantinides, K. Masselos, and P. Y. K. Cheung, "Automatic on-chip memory minimization for data reuse," in *Proc. FCCM '07*, CA, USA, Apr. 2007, pp. 389–394.

[5] ——, "Data reuse exploration under area constraints for low power reconfigurable systems," in *Proc. WASP '07*, 2007.

[6] U. K. Banerjee, *Loop Parallelization*, Norwell, MA, USA, 1994.

[7] N. Baradaran and P. C. Diniz, "A register allocation algorithm in the presence of scalar replacement for fine-grain configurable architectures," in *Proc. DATE '05*, USA, 2005, pp. 6–11.

[8] N. Baradaran, J. Park, and P. C. Diniz, "Compiler reuse analysis for the mapping of data in FPGAs with RAM blocks," in *Proc. FPT '04*, 2004, pp. 145–152.

[9] Z. Guo, B. Buyukkurt, and W. Najjar, "Input data reuse in compiling window operations onto reconfigurable hardware," *SIGPLAN Not.*, vol. 39, no. 7, pp. 249–256, 2004.

[10] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "SUIF: an infrastructure for research on parallelizing and optimizing compilers," *SIGPLAN Not.*, vol. 29, pp. 31–37, 1994.

[11] M. Wolfe, "More iteration space tiling," in *Proc. ACM/IEEE conference on Supercomputing*, NY, USA, 1989, pp. 655–664.

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

[13] M. Kandemir, G. Chen, and F. Li, "Maximizing data reuse for minimizing memory space requirements and execution cycles," in *Proc. ASP-DAC '06*, Piscataway, NJ, USA, 2006, pp. 808–813.

[14] M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh, "A compiler-based approach for dynamically managing scratch-pad memories in embedded systems," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst*, vol. 23, pp. 243–260, Feb. 2004.

[15] S. Udayakumaran and R. Barua, "Compiler-decided dynamic memory allocation for scratch-pad based embedded systems," in *Proc. CASES '03*, NY, USA, 2003, pp. 276–286.

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

[17] V. Lefebvre and P. Feautrier, "Automatic storage management for parallel programs," *Parallel Comput.*, vol. 24, pp. 649–671, 1998.

[18] M. Gupta and P. Banerjee, "Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers," *IEEE Trans. Parallel Distrib. Syst.*, pp. 179–193, 1992.

[19] G. M. Megson and X. Chen, *Automatic Parallelization for A Class of Regular Computations*. Singapore: World Scientific Publishing Co Pte Ltd, 1997.

[20] H. El-Rewini and M. Abd-El-Barr, *Advanced Computer Architecture and Parallel Processing (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2005.

[21] U. Eckhardt and R. Merker, "Hierarchical algorithm partitioning at system level for an improved utilization of memory structures," *IEEE Trans. Comput-Aided Des. Integr. Circuits Syst.*, vol. 18, pp. 14–24, Jan. 1999.

[22] N. Baradaran and P. C. Diniz, "Memory parallelism using custom array mapping to heterogeneous storage structures," in *Proc. FPL '06*, Madrid, Spain, Aug. 2006, pp. 383–388.

[23] I. Issenin, E. Brockmeyer, B. Durinck, and N. Dutt, "Multiprocessor system-on-chip data reuse analysis for exploring customized memory hierarchies," in *Proc. DAC '06*, USA, 2006, pp. 49–52.

[24] M. Dasygenis, N. Kroupis, K. Tatas, A. Argyriou, D. Soudris, and A. Thanailakis, "Power and performance exploration of embedded systems executing multimedia kernels," in *Proc. IEE Comput. Digital Techniques*, 2002, pp. 164–172.

[25] Cray computer systems, *CFT77 Reference Manual*. Cray Research, Publication SR 0018A, Minnesota, 1987.

[26] http://www.celoxica.com, "Handel-C language reference manual," accessed Aug. 2006.

[27] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, and F. Catthoor, "A practical dynamic single assignment transformation," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, p. 40, Sept. 2007.

[28] J. Lofberg, "Yalmip : A toolbox for modeling and optimization in MATLAB," Taipei, Taiwan, 2004.

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

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

[31] J. D. Hall, N. A. Carr, and J. C. Hart, "Cache and bandwidth aware matrix multiplication on the GPU," in *UIUC Technical Report UIUCDCS-R-2003-2328*, 2003.