doi:10.1093/comjnl/bxh000

Compiling C-like Languages to FPGA Hardware: Some Novel Approaches Targeting Data Memory Organisation*

Qiang Liu[1], George A. Constantinides[1], Konstantinos Masselos[2], Peter Y.K. Cheung[1]

[1]*Department of Electrical and Electronic Engineering, Imperial College London, U.K.,*
[2]*Department of Computer Science and Technology, University of Peloponnese, Greece*
Email: {qiang.liu2, g.constantinides, k.masselos, p.cheung}@imperial.ac.uk

This paper describes our approaches to raise the level of abstraction at which hardware suitable for accelerating computationally-intensive applications can be specified. Field-Programmable Gate Arrays (FPGAs) are becoming adopted as a computational platform by the high-performance computing community, but there are challenges to extract maximum performance from these devices. Unlike other approaches, our focus is on data memory organisation and input-output bandwidth considerations, which are the typical stumbling block of existing hardware compilation schemes. We describe our approaches, which are based on formal optimization techniques, and present some results showing the advantage of exposing the interaction between data memory system design and parallelism extraction to the compiler.

1. INTRODUCTION

A field-programmable gate array (FPGA) is a silicon chip containing an array of programmable logic cells and programmable routing resources, enabling the logic cells to be interconnected. This programmability allows users to configure an FPGA to perform different functions after the FPGA is manufactured. As a result of increasing device densities, together with the introduction of dedicated components, such as RAM blocks, hard multipliers, DSP blocks and microprocessors, FPGAs have now become viable computational platforms for massively parallel numerical computation. Compared to general-purpose processors, FPGAs have been shown to achieve higher speed and lower energy consumption, due to ability to specialise the hardware architecture to the algorithm being executed [1]. As a result, FPGA-based computing systems have been applied to an extensive range of applications, such as digital signal processing, video and voice processing, and high performance computing.

However, a significant issue from the designers' perspective is productivity, *i.e.* how a designer or a design team can implement complex applications on FPGAs by efficiently exploiting their resource densities, without designing the hardware architecture by hand using a hardware description language. Several C-like languages with extensions of hardware concepts, such as parallelism and timing have been developed [1, 2]. These high level languages allow designers to describe hardware circuits without knowing underlying hardware details, and obtain register transfer level (RTL) descriptions of the circuits directly from the high level descriptions by using hardware compiler, skipping the transformation made manually from behavioral descriptions to HDL descriptions in traditional HDL-based design flow. Therefore, the C-to-gates design flow facilitates algorithm implementation, debug, simulation and design exploration, and thus significantly shortens the design cycle.

Several optimization technologies have been applied to the compilation stage (data/control flow analysis) and target at facilitating the synthesis step [3, 4, 5], such as loop transformations, vectorization, code motion, sub-expression elimination, dynamic renaming and combining of variables. The method [6], which exploits FPGAs' fine-grain characteristic and automatically allocates different bit widths for different variables, is used to customize numerical representation and

*A preliminary version of this paper was presented at the BCS08 Visions of Computer Science Conference, held on September 22-24, 2008.

reduce area and power. Other technologies, such as constant propagation and dead-code removal, are widely used. Further work focuses on optimizing the synthesis step to harness the parallelism in FPGAs. Techniques, including dynamic scheduling, speculative execution, control branch balancing, pipelining and retiming, have been used in existing hardware compilers and frameworks [3, 4, 7, 8, 9]. Stream-C [10] targets multiple FPGA parallel processing and optimizes the communication between the parallelized thread.

These works mainly focus on making efficient use of computational resources in FPGAs to improve the performance of applications, by customising the datapath to the application. However, such parallel datapaths can only be exploited fully if a suitable memory subsystem is in place to keep the datapath fed with data. It is therefore also highly desirable to customise the memory subsystem to the application. The efficient use of limited on-chip memory to design memory subsystem around applications has not been given great attention during hardware compilation/synthesis. Because the computation-intensive applications usually involve significant data transfer and storage, off-chip memory access could consume a large proportion of the system power consumption and slow down the performance. Therefore, in this paper, we refocus on efficient exploitation of the memory resources and bandwidth provided by FPGAs.

Specifically, in our work, for data dominated applications containing regularly nested loops and operating on large amounts of data stored in off-chip memory with predictable memory access patterns, FPGA on-chip RAMs are configured as scratch-pad memory to buffer frequently used data. This technique is known as *data reuse* [11], and can reduce off-chip memory accesses. From the code transformation point of view, given a code with N-level loops and R references to off-chip arrays accessed inside the loops, *data reuse arrays*, mapped onto on-chip RAMs and used to store a subset of elements of the large off-chip arrays, could be introduced for each of R array references and inserted in any loop level of the loop nest. Introducing data reuse arrays at different loop levels forms a number of different data reuse options for each array reference, each with different off-chip memory access count and different on-chip RAM size requirement. For example, in Fig. 1 (a) we show a code fragment from the full search motion estimation algorithm, where the data of array *Previous*, stored in the off-chip memory, are repeatedly accessed in the 6-level nested loop. Six possible data reuse options (OP_1–OP_6), where a data reuse array *RLp* is introduced for *Previous* at six different loop levels, are shown in Fig. 1 (b). Given QCIF frame size (N=144, M=176) and B=P=4, implementing the first data reuse option OP_1 can reduce the number of off-chip memory accesses by 81 times compared with the original design, but requires 25344 words of on-chip RAM. When data reuse option OP_2 or OP_3 is chosen, less reduction in off-chip memory accesses is achieved, 27 times and 9 times respectively, but smaller on-chip RAM size is required, 2112 words and 144 words, respectively. Therefore, which data reuse option is chosen for each array reference needs to be determined under an on-chip RAM constraint. One of our concerns is how to minimize the on-chip memory overhead caused by exploiting data reuse. In this work an approach is also proposed to obtain indexing functions ($f_p(x, y, i, j, k, l)$) with minimum on-chip memory address space for introduced data reuse arrays and codes (Load()) for loading data into the data reuse arrays as shown in Fig. 1 (b).

Moreover, there are a number of RAM blocks embedded on modern FPGAs, often with two independent read/write ports. Therefore, buffering data in on-chip RAMs also increases the memory bandwidth and opens avenues for parallelism. We duplicate data into different memory banks, and then exploit the parallelization of all loops in the code, which can significantly harness parallelism available in FPGAs. For example in Fig. 1 (c), loop y is strip-mined by k_y after data are distributed in $\lceil k_y/2 \rceil$ banks of on-chip dual-port RAM. We consider the code to have been pre-processed by a dependence analysis tool such as [12] and each loop to have been marked as parallelizable or sequential. The proposed approach decides how much of that *potential* parallelism to realize, balancing this decision against its implications for memory subsystem design. Identifying the inter-dependence between data reuse and loop parallelization, we can optimize data locality and parallelism within a single step. However, the number of potential options in the design space combing data reuse and loop parallelization grows exponentially with the number of array references and loops of a code.

Therefore, the proposed approaches in this paper are to automate the exploration of the design space considering data reuse and loop parallelization at the same time *to determine at which levels of a loop nest to insert minimum-sized on-chip buffers for each array reference*, and *to determine an appropriate strip-mining for each loop level in the loop nest.* Thus, the main contributions of this paper are to identify:

- an approach to identify data reuse options, in each of which data reuse arrays are inserted into distinct places of the code of an application [13],
- an algorithm to determine the minimum sized on-chip buffer required for an identified data reuse option, based on 'modular address mappings', and compute the 'right inverse' of such mappings to efficiently effect the transfer of off-chip data onto on-chip buffers [14],
- recognition of the dependence between data reuse and loop-level parallelization and an integer geometric programming framework for concurrent

```
Do x = 0, N/B-1
 Do y = 0, M/B-1
  Do i = 0, 2P
   Do j = 0, 2P
    Do k = 0, B-1
     Do l = 0, B-1
      ... = Previous[(B*x+i+k-P)*M+B*y+j+l-P];
```

(a) A piece of original code

```
/*OP1: Load(RLp, Previous);*/
Do x = 0, N/B-1
 /*OP2: Load(RLp, Previous);*/
 Do y = 0, M/B-1
  /*OP3: Load(RLp, Previous);*/
  Do i = 0, 2P
   /*OP4: Load(RLp, Previous);*/
   Do j = 0, 2P
    /*OP5: Load(RLp, Previous);*/
    Do k = 0, B-1
     /*OP6: Load(RLp, Previous);*/
     Do l = 0, B-1
      ... = RLp[fp(x,y,i,j,k,l)];
```

(b) Code with possible data reuse options (OP_1~OP_6)

```
Do x = 0, N/B-1
 Load(RLp, Previous);
 Doall y1 = 10, ky-1
  Do y2=⌈M/(B*ky)⌉ y1 to min(M/B-1, ⌈M/(B*ky)⌉( y1+1)-1)
   Do i = 0, 2P
    Do j = 0, 2P
     Do k = 0, B-1
      Do l = 0, B-1
       ... = RLp⌊y1/2⌋ [fp(y,i,j,k,l)];
```

(c) Code with data reuse option OP_2 and loop y strip-mined by k_y

```
Do x = 0, N/B-1
 Do y = 0, M/B-1
  Load(RLp, Previous);
  Do i = 0, 2P
   Do j = 0, 2P
    Do k = 0, B-1
     Do l = 0, B-1
      ... = RLp[(i+k-P)*M+j+l-P];
```

(d) Code with data reuse option OP_3 and a directly generated indexing function for *RLp*

FIGURE 1. Example: full search motion estimation algorithm.

decision of the best data reuse and loop parallelization options under an on-chip memory constraint [15], and

- the application of the proposed approaches to a typical video processing kernel, resulting in the reduction in on-chip memory requirement up to 30 times, performance improvement up to 33 times compared with the general-purpose processor and up to 5.7 times compared with a method that optimises data reuse and loop parallelization separately for an FPGA.

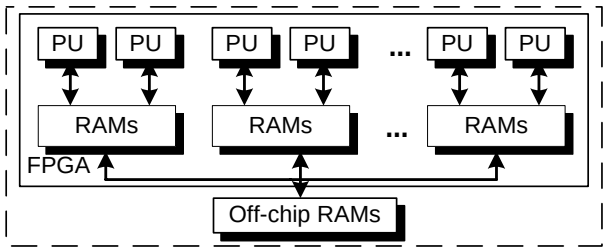

FIGURE 2. Target FGPA-based platform. PU stands for processor unit.

2. THE DATA REUSE APPROACH

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'.

To simplify the problem, in this paper, we focus on affine 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 R references $A_{i\ (1\leq i\leq R)}$ to arrays stored in off-chip memory inside the inner-most loop. The approach is easily extended to the imperfectly nested loops, but the details are omitted in this paper due to the space limitation. The target FPGA-based reconfigurable platform is shown in Fig. 2, including two memory levels: off-chip RAM and on-chip RAM. Without loss of generality, this paper assumes for simplicity that one port is available for off-chip memory accesses.

For each reference A_i accessed inside the nested loop according to an affine index expression given by the function $f_{A_i}(I_1,\ldots,I_N)$, a data reuse array [13], RLA_{i_j}, stored in on-chip RAMs, can be introduced outside loop I_j but inside loop I_{j-1} to buffer all data of A_i accessed in the inner loop nest $(I_j, \ldots, I_N)$. Before those data are used they are first loaded into the data reuse array from off-chip memory. RLA_{i_j} indexed by an address function $f_{A_{i_j}}(I_j, \ldots, I_N)$ then replaces A_i inside the loops with iterators $I_j, \ldots, I_N$. Introduction of the data reuse array at different level of the loop nest forms different data reuse options for reference A_i. Therefore, there are N data reuse options for reference A_i. To avoid redundant data transfers, only beneficial data reuse options, in which the number of on-chip memory accesses to the data reuse arrays is larger than the number of off-chip memory accesses to them, are considered. Reference A_i owns a total of E_i $(0 \leq E_i \leq N)$ beneficial data reuse options OP_{ij} $(1 \leq j \leq E_i)$ and option OP_{ij} will occupy B_{ij} blocks of on-chip RAM and need C_{ij} cycles for loading reused data from off-chip memories. E_i equal to zero means there is no reason to buffer data on-chip.

Similar options are available for all array references in a given program. As a result, there are up to $(N+1)^R$

data reuse options for a program, where the plus one means the option for each array reference in which no on-chip buffer is introduced for the reference.

3. EVALUATING DATA REUSE OPTIONS

From the section above we can see that for each beneficial data reuse option of reference A_i, a data reuse array RLA_{i_j} is accessed inside the inner loops with an index function $f_{A_{i_j}}(x)$, where $x = (I_j, \ldots, I_N)^T$ is an inner loop index vector. In this section, an algorithm [14] is presented to obtain a modular mapping as the index function for a data reuse array with minimum memory requirement B_{ij}, and an inverse of this modular mapping to load the data reuse array in C_{ij} cycles.

The index function $f_{A_{i_j}}(x)$ can be obtained in a simple way, where the outer loop index vector $u = (I_1, \ldots, I_{j-1})^T$ is eliminated from the indexing function of reference A_i, $f_{A_i}(u, x) = Qu + Fx + c$, where Q and F are coefficient matrices. In some cases such a *directly generated* address function requires memory space much larger than the number of distinct buffered data. For example, in Fig. 1 (d), data reuse option OP_3 is chosen for array *Previous* and a directly generated address function is shown. Assuming QCIF frame size for the code, the memory space addressed by this function is 1416 words, while the number of distinct data needing to be buffered is 144 words.

We thus propose an approach, starting from the function $f_{A_{i_j}}(x) = Fx + c$, to derive a new indexing function, which can reduce the required memory space. We consider the class of *modular mappings* for the potential indexing functions $g_{A_{i_j}}(x)$, *i.e.* $g_{A_{i_j}}(x) = Gx \bmod s$, where G is an integral matrix and s is a modulus vector.

The algorithm for obtaining a modular mapping for each data reuse array is shown in Algorithm 1. The input to the algorithm is the iteration space $\mathcal{IS}_j = \{(I_j, \ldots, I_N)^T \mid LB_i \leq I_i \leq UB_i, I_i \in \mathbb{Z}, j \leq i \leq N\}$ of the inner loop nest, where data reuse array RLA_{i_j} is accessed, and the coefficient matrix F appearing in the directly generated mapping $f_{A_{i_j}}(x) = Fx + c$. The output is a new modular mapping $Gx \bmod s$, on-chip memory requirement Mem_{On}, loading time C_{ij}, and a loading code for RLA_j. The idea is to incorporate memory reuse [16] into data reuse exploration, in order to minimize the memory space required by exploiting data reuse. Further details of Algorithm 1 are given in [14]. However, for completeness, we summarize the algorithm in the rest of this section.

To guarantee the correctness of a code after replacing the directly generated address mapping $f_{A_{i_j}}(x)$ with the modular mapping $g_{A_{i_j}}(x)$, two conditions need to be satisfied: 1) preserve existing data reuse, *i.e.* if $f_{A_{i_j}}(x_1) = f_{A_{i_j}}(x_2)$ then $g_{A_{i_j}}(x_1) = g_{A_{i_j}}(x_2)$; and 2) at least one of two iteration vectors, which are mapped on the same memory element by $g_{A_{i_j}}(x)$ but not by $f_{A_{i_j}}(x)$, is outside the iteration space, *i.e.* if $g_{A_{i_j}}(x_1) = g_{A_{i_j}}(x_2)$ and $f_{A_{i_j}}(x_1) \neq f_{A_{i_j}}(x_2)$ then $x_1 \notin \mathcal{IS}_j$ or $x_2 \notin \mathcal{IS}_j$. Because $f_{A_{i_j}}(x)$ is affine, $f_{A_{i_j}}(x_1) - f_{A_{i_j}}(x_2) = 0 \Leftrightarrow f_{A_{i_j}}(x_1 - x_2) = 0$, and hence algorithm works on the iteration difference set D as shown in Algorithm 1[1].

Therefore, the first step is to generate a polytope, which encloses the set D, by Minkowski sum. To preserve data reuse, one needs to first characterize the existing data reuse ($Fx = 0 \wedge x \in D$). The second step obtains a set of basis vectors of the kernel of F by computing its Hermite normal form, and then the third step obtains a *data reuse basis* by Fourier-Motzkin elimination of redundant vectors from the set of basis vectors obtained in step 2. We then combine the data reuse basis with another set of basis vectors (called *memory reducing basis*), which guarantees that the new mapping never maps two iteration vectors that *do not* re-use the same data element to the same array element. Together, these form the basis of a lattice, which is for the kernel of the required modular mapping.

We find the memory reducing basis in the subspace orthogonal to the data reuse set P as shown in steps 4-7 of Algorithm 1. This makes sure that the memory reducing basis is independent of the data reuse basis. Moreover, the second condition which a valid $g_{A_{i_j}}(x)$ has to satisfy can be represented as $(\ker(g_{A_{i_j}}(x)) \setminus \ker(F)) \cap D = \{0\}$ in terms of iteration difference. The memory reducing basis is just used to generate a lattice for $\ker(g_{A_{i_j}}(x)) \setminus \ker(F)$. Therefore, Step 6, following the method in [16], finds a set of basis vectors which generate a lattice Λ intersecting with K', which is a projection of K onto the orthogonal subspace, only at zero and thus when Step 7 transforms these basis vectors to the original coordinates, the obtained lattice still intersects with K, as a result the lattice also intersects with D at zero. In this way, the second condition can be met. Once both data reuse and memory reducing bases are found, step 8 constructs a modular mapping following the method in [16] and the modular mapping satisfies the two conditions. This modular mapping replaces the directly generated address function for the data reuse array RLA_{i_j} in the code.

However, the code must also load the reuse array by transferring data from the original array in off-chip memory. Hence for each element of the data reuse array, we must be able to correctly identify, with minimal control overhead, the corresponding element in the original array. This problem can be formulated as a *parametric integer linear program* with variable k as shown in step 10 of Algorithm 1 after we transform the modular mapping in a linear manner. This problem can

[1]Operating at the difference set and Steps 6 and 8 of Algorithm 1 are taken from the work of [16]. The rest steps of the algorithm are novel.

be solved by Feautrier's PIP software [17], and always leads a guaranteed solution by construction. According to this solution, a code for loading required data is generated and correspondingly the loading time can be determined.

Algorithm 1

Input: The iteration space $\mathcal{IS}_j$ and the coefficient matrix $F_{q\times n}$ of a data reuse array RLA_{i_j} of reference A_i.

Output: A modular mapping (G, s), Mem_{on}, C_{ij} and the loading code for the data reuse array.

1: $K = \mathcal{IS}'_j \oplus (-\mathcal{IS}'_j)$, K is a 0-symmetric polytope enclosing all integer points in the iteration difference set $D = \{d \mid d = x - y,\ x, y \in \mathcal{IS}_j\}$ and $\mathcal{IS}'_j$ is the convex hull of $\mathcal{IS}_j$
{/* Steps 2-3 obtain a data reuse basis characterizing data reuse in accesses to RLA_{i_j} */}

2: Compute $ker(F)$ by the Hermite normal form, $FV = [H\ \mathbf{0}]$, of F

3: Obtain a data reuse basis $(a_0, a_1, ..., a_{m-1})$ for the reuse characterization set P by Fourier-Motzkin elimination such that for every $x \in P = ker(F) \cap D$, $x = t_0a_0 + , \ldots, + t_{m-1}a_{m-1}$, t_i are non-zero integers
{/* Steps 4-7 obtain a memory reducing basis for mapping RLA_{i_j} onto on-chip memory */}

4: Find a basis of the subspace $P^{\perp}$ orthogonal to P by computing $ker(Q_{m\times n})$, where $Q_{m\times n}$ with a_i as its ith row vector

5: Project the polytope K onto the orthogonal space $P^{\perp}$ to construct a polytope K'

6: Find a basis $(a'_m, a'_{m+1}, ..., a'_{n-1})$ in the subspace $P^{\perp}$ to form a lattice $\Lambda = \{\sum_{i=m}^{n-1} t_i a'_i | t_i \in \mathbb{Z}\}$ and $\Lambda \cap K' = \{0\}$

7: Obtain memory reducing basis $(a_m, a_{m+1}, ..., a_{n-1})$ by transforming the basis $(a'_m, a'_{m+1}, ..., a'_{n-1})$ into the coordinates where polytope K is

8: Combine the data reuse basis and the memory reducing basis to construct a modular mapping by computing the Smith form, $S = GYU$, of a matrix $Y = (a_0, a_1, ..., a_{n-1})$, $S = diag(s)$

9: Reach the modular mapping $g_{A_{i_j}}(x) = (G, s)$ and $\text{Mem}_{\text{on}} = \prod_{i=1}^{n} s_i$
{/* Step 10 computes an inverse mapping of the modular mapping after revealing its linear nature, $g_{A_{i_j}}(x) = Gx + diag(s)k$ */}

10: Call Feautrier's PIP software [17] to determine an integral vector k such that for $0 \le g_{A_{i_j}}(x) \le s - 1$, $x = G^{-1}g_{A_{i_j}}(x) - G^{-1}diag(s)k \in \mathcal{IS}_j$, in order to obtain a loop structure and an inverse mapping $h_{A_{i_j}}(x) = FG^{-1}g_{A_{i_j}}(x) - FG^{-1}diag(s)k + c$ for loading data

11: Determine the loading time C_{ij} in terms of the loop structure.

As a result of this algorithm, we can measure the on-chip memory requirement and the loading time for each data reuse option at compile time. This makes the exploration of data reuse with loop parallelization carried out in the next section possible.

4. COMBINING DATA REUSE WITH LOOP PARALLELIZATION

In this section, it is shown that parallelism issues should be considered when making data-reuse decisions, and based on this conclusion we formulate the problem of exploring data reuse and loop parallelization to maximize the performance while meeting an on-chip memory constraint as an integer geometric programming problem, which can be solved with global optimization techniques.

4.1. Dependence between data reuse and loop parallelization

Data reuse mainly targets access locality improvement but can also increase the bandwidth of memory access, if a distributed memory architecture is available as in Fig. 2, allowing multiple loop iterations to execute in parallel.

If data reuse optimization and loop parallelization 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 into on-chip memories. 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 [11]. However, the choice of data-reuse option made first may not be able to completely exploit the possible parallelism that can be extracted from the code, resulting in suboptimal designs, as will be shown in Section 5.

The key 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.*

With respect to this constraint, the design space combining data reuse and loop parallelization for an N-level nested loops with R array references could include up to $(N+1)^R \prod_{l=1}^{N} L_l$ design options, where L_l is the total number of iterations of loop l. We can see that the design space increases exponentially with the number of array references and the number of loop levels. In the next section, it will be shown that how this problem can nevertheless be formulated in a manner admitting an efficient solution.

4.2. Geometric programming formulation

This section provides an overview of the method [15] we use to simultaneously optimize parallelism and memory utilization.

Data reuse option OP_{ij} is associated with an integral data reuse variable ρ_{ij} that can only take value one

or two. If data reuse option OP_{ij} is selected for reference A_i, then ρ_{ij} takes value two, otherwise one. 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. Loop l $(1 \leq l \leq N)$ is partitioned into k_l $(1 \leq k_l \leq L_l)$ parallel segments, together with ρ_{ij}, to be determined by the optimization procedure.

Based on the notations listed in Table 1, the problem of exploration of data reuse and data-level parallelization is defined in equation (1)–(8), which will be briefly described below.

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

subject to

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

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

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

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

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

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

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

In this formulation, all capitals are known parameters at compile time, and all variables (lowercase) are integers. The objective function minimizes the number of execution cycles of a program in the expression (1). Inequality (2) defines the constraint on the on-chip memory resources. 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. 2. 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). Inequality (3) ensures that exactly one data reuse option is chosen for each reference. Inequalities (6) show the link between data reuse variables ρ_{ij} and loop partition variables k_l. This link is exploited within this framework to remove redundant design options combing data reuse and data-level parallelization.

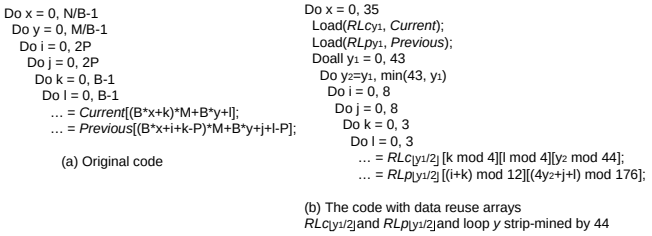

FIGURE 3. A code segment of FSME algorithm.

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 [18]. A branch and bound algorithm used in [19] is applied to the framework to solve the problem, using the geometric programming relaxation as a bounding procedure.

By solving problem (1)–(8), a performance-optimal design with both optimum data reuse options and loop parallelization options under the on-chip memory constraint can be determined for an application.

5. EXPERIMENTS

The approaches described above have been applied to the full search motion estimation (FSME) algorithm [20] as partly shown in Fig. 3 (a), which is a typical kernel used in many video processing applications. As described above, in the proposed framework, the number of data reuse variables and parallelism variables increases linearly in the depth of the loop nest. In practice, it is exceedingly rare to see a loop nest of depth greater than 6. Thus, the choice of FSME as a benchmark here is to test this practical limit. Similar results can also be observed on matrix-matrix multiplication and Sobel edge detection (see [15]). In our experiments, the structure of the target platform is shown in Fig. 2 with a Xilinx XC2v8000 FPGA, which has 168 blocks of 18 kbits on-chip RAM, while the approaches can be equally applicable to other modern FPGAs. Potentially frequently accessed elements of each array reference in the kernel are buffered in the on-chip RAMs and are duplicated in different banks to increase the bandwidth of memory accesses.

The details of the FSME algorithm is shown in Table 2. FSME has six regularly nested loops and two array references, corresponding to the *current* and *previous* video frames. The luminance component of QCIF image (144×176) is the typical frame size used in the experiments. Following the data reuse approach described in Section 2, we consider three beneficial data reuse options for each of the two array references. The

TABLE 1. A list of notations used in this paper. Capitals are parameters and lowercases are variables

Notation	Description	Notation	Description
ρ_{ij}	binary data reuse 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	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		

outermost two loops of the loop nest in the kernel are parallelizable and all parallelization options $\{k_l\}$ are also presented in Table 2.

The first set of results concerns the on-chip memory requirement of different data reuse options of the two array references. Column Mem_{on}(DM) and Mem_{on}(MM) in Table 2 show the memory sizes required by data reuse arrays with a directly generated mapping (DM) and the modular mapping (MM) generated using our algorithm in Section 3, respectively. We can see that the MM provides the minimum memory cost for data buffered on-chip, equal to the number of data buffered, while DM does not and in some case causes a significantly redundant requirement, up to 30 times. As a result, the number of on-chip RAM blocks (B_{ij}) and loading time C_{ij} can be evaluated, and is also shown in the table. The load time also corresponds to the number of off-chip memory accesses. The reduction in the off-chip memory accesses compared with the design without on-chip buffers is shown in the last column of the table in brackets.

Given B_{ij}, C_{ij} and the other input parameters listed in Table 1, the problem of determining the optimum data reuse options and loop parallelization options for the FSME algorithm, as formulated in Section 4.2, can be solved by YALMIP [19]. In Fig. 4 (a), some randomly generated feasible designs (shown in crosses) and the performance-optimal designs (shown in dots), are illustrated. The sub-optimal designs, above the performance-optimal Pareto frontier, have been automatically rejected by our approach. Moreover, the number of execution cycles decreases as the number of on-chip RAM blocks increases, because the allowable degree of parallelism increases.

All designs given by YALMIP have been implemented in Handel-C [9], a C-like language, and then have been synthesized and mapped onto the FPGA. The actual execution times and on-chip memory requirements are shown in Fig. 4 (b). In this figure, the designs proposed by our approach are shown in dots and the corresponding performance-optimal design Pareto frontier is drawn using a bold line. Clearly, there are the same trends of the frontiers in Fig. 4 (a) and Fig. 4 (b). This demonstrates that our approach has the ability to determine the performance-optimal designs for the FSME kernel in the context of different on-chip memory constraints. In addition, the designs obtained

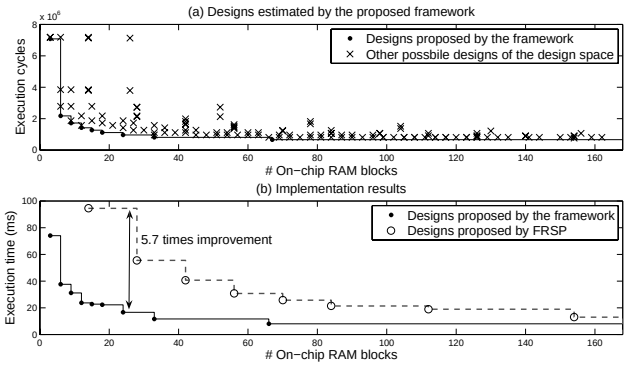

FIGURE 4. Experimental results of FSME. (a) Designs in the design space combining data reuse with loop parallelization. (b) Implementation of the proposed designs on an FPGA.

by following the method (FRSP) that first explores data reuse and then loop parallelization [11] have been implemented and plotted in Fig. 4 (b) in circles. Compared with the FRSP method, our approach achieves the performance improvement up to 5.7 times. This shows the advantage of the proposed approach that explores data reuse and loop parallelization at the same time.

For the FSME kernel, the modular mapping and the on-chip memory requirement for each data reuse array are generated within a second, and a performance-optimal design under an on-chip memory constraint is generated within 10 seconds by our approaches. A performance-optimal design with the data reuse options $\{OP_{12}, OP_{22}\}$ and the loop parallelization options $\{k_1 = 1, k_2 = 44, k_3 = 1, k_4 = 1, k_5 = 1, k_6 = 1\}$, as shown in Fig.3 (b), is implemented in the target platform with a Xilinx XC2v8000 FPGA running at 50 MHz. This design needs only 8.03 ms to perform the motion estimation for a frame, while the FSME algorithm running at a general-purpose computer with a 2.8 GHz CPU requires 270 ms.

6. RELATED WORK

Memory optimization and parallelization have been widely studied. However, to our best knowledge there has not been a method combining data reuse, memory reuse and loop parallelization to optimize memory

TABLE 2. The details of FSME and on-chip memory requirement and loading time of its data reuse options.

k_l	Reference	Reuse options	#Buffered data	Mem_{on} (DM)	Mem_{on} (MM)	B_{ij}	C_{ij}
$1 \leq k_1 \leq 36$	current	OP_{11}	25344	25344×8b	25344×8b	13	25344 (81×)
$1 \leq k_2 \leq 44$		OP_{12}	704	704×8b	704×8b	1	25344 (81×)
$k_3 = 1$		OP_{13}	16	532×8b	16×8b	1	25344 (81×)
$k_4 = 1$	previous	OP_{21}	25344	25344×8b	25344×8b	13	25344 (81×)
$k_5 = 1$		OP_{22}	2112	2112×8b	2112×8b	2	76032 (27×)
$k_6 = 1$		OP_{23}	144	1416×8b	144×8b	1	228096 (9×)

organization and parallelism at the same time as in this work.

Many approaches have been proposed for data locality optimization. To keep reused data in cache and reduce misses, loop permutation, reversal, skew, tiling, fusion, distribution and data layout transformations have been used in [21, 22, 23, 24]. Hardware prefetcher [25] and partitioned cache [26, 27] have been incorporated into the cache system to reduce cache misses.

Complementing the cache system, scratch-pad memory (SPM) has been recently given attention. Approaches for dynamic management and allocation of cache and SPM have been seen in [28, 29, 30]. There exists some research works on buffering reused data in FPGA on-chip embedded RAMs and registers [31, 32, 33]. A shift register is implemented in [31]. In [32] a simple algorithm, which stores whole arrays with repeatedly accessed data in registers if there are enough available registers, and otherwise in RAMs, is used to assign data. Baradaran *et al.* extend this in [33] and use a greedy algorithm to assign registers to arrays in the same cut of a graph which includes all critical paths of a design. Arrays can be partially buffered in registers and RAMs.

Several approaches have been proposed for optimizing memory usage. In [34] an approach is proposed for allocating local memory to arrays in designing application-specific embedded systems. The idea is to only store accessed data of each array to the local memory and obtain an indexing function for addressing these data in the local memory. However, the obtained indexing functions in some cases require redundant local memory space. Darte *et al.* [16] provide a mathematical framework to study generalized storage space reuse problem. The aim is to build a modular mapping from the original array indexing to reuse memory. This approach has been incorporated into, and forms the core of, our work.

Improvement of the parallelism of programs has been a hot topic in the computing community. Loop transformations have been used in [35] to enable loop parallelization of perfectly nested loops. Exploration of both data reuse and parallelization has been discussed in only a few previous works and has generally been done in two distinct steps. Catthoor *et al.* [11] first explore data reuse and determine the promising data reuse options, and then computations that execute in parallel and involve data accesses are scheduled and memory resources are allocated to provide the data availability on time. Schreiber *et al.* [36] propose an approach to synthesize a perfectly loop nest onto a systolic array. The approach first performs loop tiling on the loop nest and assigns the inner loop iterations within a tile to execute on the systolic array. Data are loaded into local registers to provide parallel accesses and reused data are shifted in the registers to reduce global memory bandwidth requirement. In [37] loop tile size selection models used in literature for optimizing data locality or parallelism have been generalized as a geometric programming framework.

None of approaches described above considers explicitly the link between memory reuse, data reuse and parallelization, and incorporates the link as a constraint which may be optimized over. This issue is addressed by our work and the proposed approaches complement the existing hardware compilers.

7. CONCLUSION

In this paper, we have described an approach for identifying data reuse options, an algorithm for evaluating data reuse options, and a framework for concurrent determining optimum data reuse and loop parallelization options. These approaches enable the automatic exploration of efficient memory subsystems around algorithms for high performance designs in FPGA-based platforms, and can be applied to the hardware compilation/synthesis stage in current C-to-gates flow. We have applied the proposed approaches to the full search motion estimation algorithm, resulting in the reduction in on-chip memory requirement up to 30 times, performance improvement up to 33 times compared with the general-purpose processor and up to 5.7 times compared with a method that optimises data reuse and loop parallelization separately for the FPGA. The results demonstrate that with minimized on-chip memory utilization, our approaches can produce the performance-optimal designs in the target platform with limited on-chip memories.

The proposed approaches in this work target affine nested loop nests, because they are often seen in many real applications. However, the approaches could be still applied to several situations where non-affine parameterized loop nests or non-affine array indexing expressions are involved, if those loops causing non-affinity could be moved to outer loops and data reuse arrays are introduced within these loops. For general non-affinely nested loop nests, apart from the cases

above, the linear algebraic approach central to our method can no longer be used. In the future, we will extend the work to considering non-affine loop structure and optimized data partition methods.

REFERENCES

[1] Todman, T., Constantinides, G., Wilton, S., Cheung, P., Luk, W., and Mencer, O. (2005) Reconfigurable computing: Architectures and design methods. *IEE Proceedings of Computers and Digital Techniques*, **152**, 193–207.

[2] Edwards, S. A. (2005) The challenges of hardware synthesis from C-like languages. *Proc. DATE '05*, Washington, DC, USA, pp. 66–67. IEEE Computer Society.

[3] Gupta, S., Dutt, N., Gupta, R., and Nicolau, A. (2003) SPARK: a high-level synthesis framework for applying parallelizing compiler transformations. *Proc. Int. Conf. on VLSI Design*, January, pp. 461–466.

[4] Weinhardt, M. and Luk, W. (Feb. 2001) Pipeline vectorization. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, **20**, 234–248.

[5] Buyukkurt, B., Zhi, G., and Najjar, W. A. (2006) Impact of loop unrolling on throughput, area and clock frequency in ROCCC: C to VHDL compiler for FPGAs. *Proc. ARC '06*, The Netherlands.

[6] Constantinides, G. A., Cheung, P. Y. K., and Luk, W. (Jan. 2005) Optimum and heuristic synthesis of multiple word-length architectures. *IEEE Trans. VLSI Syst.*, **13**, 39–57.

[7] Budiu, M. and Goldstein, S. C. (2002) Compiling application-specific hardware. *Proc. FPL '02*, London, UK, pp. 853–863. Springer-Verlag.

[8] McCloud, S. (2004) Catapult C synthesis-based design flow: speeding implementation and increasing flexibility. *White Paper*. Mentor Graphics.

[9] Handel-C language reference manual, http://www.agilityds.com/literature/handelc_language_reference_manual.pdf. accessed may 2008.

[10] Gokhale, M. B., Stone, J. M., Arnold, J., and Kalinowski, M. (2000) Stream-oriented FPGA computing in the Streams-C high level language. *Proc. FCCM '00*, pp. 49–56.

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

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

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

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

[15] Liu, Q., Constantinides, G. A., Masselos, K., and Cheung, P. Y. K. (2008) Combining data reuse exploitation with data-level parallelization for FPGA targeted hardware compilation: A geometric programming framework. *Int. Conf. FPL '08*, Heidelberg, Germany, pp. 179–184.

[16] Darte, A., Schreiber, R., and Villard, G. (2005) Lattice-based memory allocation. *IEEE Transactions on Computers*, **54**, 1242–1257.

[17] Feautrier, P. (1988) Parametric integer programming. *RAIRO Recherche Opérationnelle*, **22**, 243–268.

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

[19] Lofberg, J. (2004) Yalmip : A toolbox for modeling and optimization in MATLAB. *Proc. CACSD '04*, Taipei, Taiwan.

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

[21] Wolf, M. E. and Lam, M. S. (1991) A data locality optimizing algorithm. *PLDI '91: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, New York, NY, USA, pp. 30–44. ACM.

[22] McKinley, K. S., Carr, S., and Tseng, C.-W. (1996) Improving data locality with loop transformations. *ACM Trans. Program. Lang. Syst.*, **18**, 424–453.

[23] Kandemir, M., Choudhary, A., Ramanujam, J., and Banerjee, P. (1998) Improving locality using loop and data transformations in an integrated framework. *MICRO 31: Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, Los Alamitos, CA, USA, pp. 285–297. IEEE Computer Society Press.

[24] Kadayif, I. and Kandemir, M. (2004) Quasidynamic layout optimizations for improving data locality. *IEEE Trans. Parallel Distrib. Syst.*, **15**, 996–1011.

[25] Tang, W., Veidenbaum, A., Nicolau, A., and Gupta, R. (2003) Conflict miss elimination by time-stride prefetch. Technical report. University of California, Irvine.

[26] Aliagas, C., Valero, M., and Nord, C. (1995) A data cache with multiple caching strategies tuned to different types of locality. *Proceedings of the 1995 International Conference on Supercomputing*, pp. 338–347.

[27] Petrov, P. and Orailoglu, A. (2001) Performance and power effectiveness in embedded processors-customizable partitioned caches. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **20**, 1309–1318.

[28] Kandemir, M., Ramanujam, J., Irwin, M. J., Vijaykrishnan, N., Kadayif, I., and Parikh, A. (2004) A compiler-based approach for dynamically managing scratch-pad memories in embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **23**, 243–260.

[29] Udayakumaran, S. and Barua, R. (2003) Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. *CASES '03: Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, New York, NY, USA, pp. 276–286. ACM.

[30] Kandemir, M., Chen, G., and Li, F. (2006) Maximizing data reuse for minimizing memory space requirements and execution cycles. *ASP-DAC '06: Proceedings of the 2006 Conference on Asia South Pacific Design Automation*, Piscataway, NJ, USA, pp. 808–813. IEEE Press.

[31] Diniz, P. and Park, J. (2000) Automatic synthesis of data storage and control structures for FPGA-based computing engines. *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, Washington, DC, USA, pp. 91–100. IEEE Computer Society.

[32] Baradaran, N., Park, J., and Diniz, P. C. (2004) Compiler reuse analysis for the mapping of data in FPGAs with RAM blocks. *Proceedings of the 2004 IEEE International Conference on Field-Programmable Technology*, pp. 145–152.

[33] Baradaran, N. and Diniz, P. C. (2005) A register allocation algorithm in the presence of scalar replacement for fine-grain configurable architectures. *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pp. 6–11.

[34] Schreiber, R. and Cronquist, D. C. (2004) Near-optimal allocation of local memory arrays. Technical Report HPL-2004-24. Hewlett-Packard Laboratory, California.

[35] Banerjee, U. K. (1994) *Loop Parallelization*, . Norwell, MA, USA.

[36] Schreiber, R., Aditya, S., Mahlke, S., Kathail, V., Rau, B. R., Cronquist, D., and Sivaraman, M. (2001) Pico-npa: High-level synthesis of nonprogrammable hardware accelerators. Technical Report HPL-2001-249. HP Laboratory Palo Alto, California.

[37] Renganarayana, L. and Rajopadhye, S. (2008) Positivity, posynomials and tile size selection. *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, Piscataway, NJ, USA, pp. 1–12. IEEE Press.