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

Qiang Liu*, George A. Constantinides*, Konstantinos Masselos† and Peter Y.K. Cheung*
*Imperial College, London SW7 2BT, U.K.
†University of Peloponnese, Tripolis, Greece
{qiang.liu2, g.constantinides, k.masselos, p.cheung}@imperial.ac.uk

Abstract

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.

Keywords: FPGA, C-like language, data reuse, loop parallelization, hardware 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 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 different data reuse options with different numbers of off-chip memory accesses and different requirements of on-chip RAM size for each array reference. One of our concerns is how to minimize the on-chip memory overhead caused by exploiting data reuse. 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. 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. In the context of this paper, the parallelization decision is to decide an appropriate strip-mining for each loop level in the loop nest. 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 explain:

- 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,
- 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 efficient effect the transfer of off-chip data onto on-chip buffers [13],
- recognition of the dependence between data reuse and loop-level parallelization and an integer geometric programming framework for concurrent decision of the best data reuse and loop parallelization options under an on-chip memory constraint [14], 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

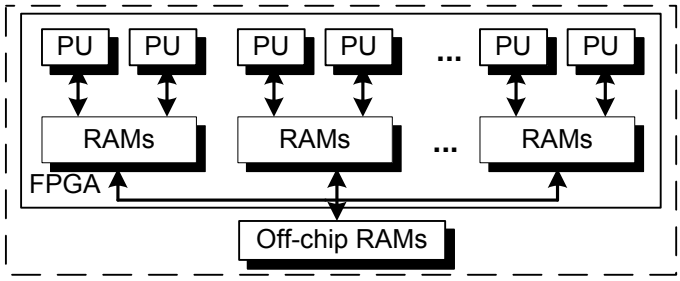


FIGURE 1: Target FGPA-based platform. PU stands for processor unit.

up to 33 times compared with the general-purpose processor and up to 4.3 times compared with a method that optimises data reuse and loop parallelization separately for an FPGA.

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. 1, 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, 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 a 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 ($E_i \leq N$) 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.

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.

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 [13] 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, $f_{A_i}(u, x) = Qu + Fx + c$,

Algorithm 1

Input: The iteration space $\mathcal{I}S_j$ 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), $\mathsf{Mem}_{\mathsf{on}}$, C_{ij} and the loading code for the data reuse array.

1: $K = \mathcal{I}S_j \oplus (-\mathcal{I}S_j)$, K is a 0-symmetric polytope enclosing all integer points in the iteration difference set $D = \{d \mid d = x - y,\ x, y \in \mathcal{I}S_j\}$
{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
{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 reuse basis $(a_m, a_{m+1}, ..., a_{n-1})$ by transforming the basis $(a'_m, a'_{m+1}, ..., a'_{n-1})$ into the polytope K
8: Combine the data reuse basis and the memory reuse 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 $\mathsf{Mem}_{\mathsf{on}} = \prod_{i=1}^{n} s_i$
{Compute an inverse mapping of the modular mapping for loading data into RLA_{i_j}}
10: Reveal the linear nature of the modular mapping, $g_{A_{i_j}}(x) = Gx + diag(s)k$
11: Call Feautrier's PIP software [15] to determine an integral vector k such that for $0 \leq g_{A_{i_j}}(x) \leq s - 1$, $x = G^{-1}g_{A_{i_j}}(x) - G^{-1}diag(s)k \in \mathcal{I}S_j$, 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
12: The loading time C_{ij} can be determined in terms of the loop structure

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 [13].

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.* $g_{A_{i_j}}(x) = Gx$ mod 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 $\mathcal{I}S_j = \{(I_j, \ldots, I_N)^T \mid LB_i \leq I_i \leq UB_i, I_i \in \mathbb{Z}, j \leq i \leq N\}$ 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 $Gx\ mod\ s$, on-chip memory requirement $\mathsf{Mem}_{\mathsf{on}}$, loading time C_{ij}, and a loading code for RLA_j. The idea is to incorporate memory reuse into data reuse exploration, in order to minimize the memory space required by exploiting data reuse. We combine a set of basis vectors (called *data reuse basis*), which preserves the data reuse present in the mapping $f_{A_{i_j}}(x) = Fx+c$, 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, to form a dense lattice for the kernel of the required modular mapping. 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 and can be solved by Feautrier's PIP software [15], and always leads a guaranteed solution by construction. Further details of Algorithm 1 given in [13].

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
C_{ij}	# loading cycles of the data reuse array of option j of reference i		of option j of reference i

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. 1, 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*.

4.2. Geometric programming formulation

This section provides an overview of the method we use to simultaneously optimize parallelism and memory utilization. For a detailed analysis, the reader is referred to [14].

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.

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

subject to

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

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

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

$$k_l^{-1} \le 1, 1 \le l \le N \quad (5)$$

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

$$L_l k_l^{-1} v_l^{-1} \le 1, 1 \le l \le N \quad (7)$$

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

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. 1. 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 at most 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.

The relaxation of problem (1)–(8), obtained by allowing k_l to be real valued solutions and replacing (4) by $1 \le \rho_{ij} \le 2$, 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 [16]. A branch and bound algorithm used in [17] 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.

5. EXPERIMENTS

The approaches described above have been applied to the full search motion estimation (FSME) algorithm [18] as partly shown in Fig. 2 (a), which is a typical kernel used in many video processing applications. In our experiments, the structure of the target platform is shown in Fig. 1 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.

```
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
            … = Current[(B*x+k)*M+B*y+l];
            … = Previous[(B*x+i+k-P)*M+B*y+j+l-P];
```

(a) Original code

```
Do x = 0, 35
  Load(RLcy1, Current);
  Load(RLpy1, Previous);
  Doall y1 = 1, 44
    Do y2=(y1-1), min(43, (y1-1))
      Do i = 0, 8
        Do j = 0, 8
          Do k = 0, 3
            Do l = 0, 3
              … = RLcy1[k mod 4][l mod 4][y2 mod 44];
              … = RLpy1[(i+k) mod 12][(4y2+j+l) mod 176];
```

(b) The code with data reuse arrays
RLcy1 and *RLpy1* and loop *y* strip-mined by 44

FIGURE 2: A code segment of FSME algorithm.

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

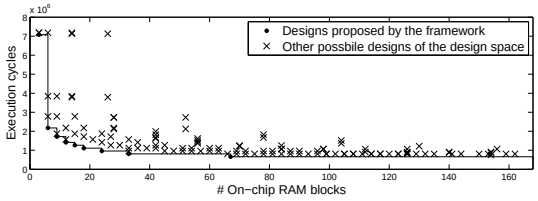
k_l	Reference	Reuse options	#Buffered data	Mem_{on} (DM)	Mem_{on} (MM)	B_{ij}	C_{ij}
$1 \leq k_1 \leq 36$	current	OP_{11}	25344	25344×8b	25344×8b	13	25344 (81×)
$1 \leq k_2 \leq 44$		OP_{12}	704	704×8b	704×8b	1	25344 (81×)
$k_3 = 1$		OP_{13}	16	532×8b	16×8b	1	25344 (81×)
$k_4 = 1$	previous	OP_{21}	25344	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×)

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

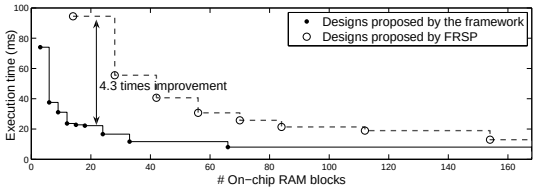
The first set of results concerns the on-chip memory requirement of different data reuse options of the two array references. Column Mem_{on}(DM) and Mem_{on}(MM) in Table 2 show the memory sizes required by data reuse arrays with a directly generated mapping (DM) and the modular mapping (MM) generated 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 [17]. In Fig. 3(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. 3(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. 3(a) and Fig. 3(b). This demonstrates that our approach has the ability to determine the performance-optimal designs for the FSME kernel under different on-chip memory constraints. In addition, the designs obtained by following the method (FRSP) that first explores data reuse and then loop parallelization [11] have been implemented and plotted in Fig. 3(b) in circles. Compared with the FRSP method, our approach achieves the performance improvement up to 4.3 times. This shows the advantage of the proposed approach that explores data reuse and loop parallelization at the same time.



(a) Designs in the design space combining data reuse with loop parallelization.



(b) Implementation of the proposed designs on an FPGA.

FIGURE 3: Experimental results of FSME

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.2 (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. 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 4.3 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. 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.celoxica.com*, accessed Nov. 2005.
[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., 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.
[14] 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 (accepted)*.
[15] Feautrier, P. (1988) Parametric integer programming. *RAIRO Recherche Opérationnelle*, **22**, 243–268.
[16] Boyd, S. and Vandenberghe, L. (2004) *Convex optimization*. Cambridge University Press.
[17] Lofberg, J. (2004) Yalmip : A toolbox for modeling and optimization in MATLAB. *Proc. CACSD '04*, Taipei, Taiwan.
[18] Bhaskaran, V. and Konstantinides, K. (1997) *Image and Video Compression Standards: Algorithms and Architectures*. Kluwer Academic Publishers, Norwell, MA, USA.