J Sign Process Syst (2012) 67:65–78
DOI 10.1007/s11265-010-0563-9

Automated Mapping of the MapReduce Pattern onto Parallel Computing Platforms

Qiang Liu · Tim Todman · Wayne Luk · George A. Constantinides

Received: 19 February 2010 / Revised: 15 July 2010 / Accepted: 9 November 2010 / Published online: 15 December 2010
© Springer Science+Business Media, LLC 2010

Abstract The *MapReduce* pattern can be found in many important applications, and can be exploited to significantly improve system parallelism. Unlike previous work, in which designers explicitly specify how to exploit the pattern, we develop a compilation approach for mapping applications with the MapReduce pattern automatically onto Field-Programmable Gate Array (FPGA) based parallel computing platforms. We formulate the problem of mapping the MapReduce pattern to hardware as a geometric programming model; this model exploits loop-level parallelism and pipelining to give an optimal implementation on given hardware resources. The approach is capable of handling single and multiple nested MapReduce patterns. Furthermore, we explore important variations of MapReduce, such as using a linear structure rather than a tree structure for merging intermediate results generated in parallel. Results for six benchmarks show that our approach can find performance-optimal designs in the design space, improving system performance by up to 170 times compared to the initial designs on the target platform.

Q. Liu · T. Todman (✉) · W. Luk
Department of Computing, Imperial College London,
London SW7 2AZ, UK
e-mail: tjt97@doc.ic.ac.uk

Q. Liu
e-mail: qiang.liu2@imperial.ac.uk

W. Luk
e-mail: w.luk@imperial.ac.uk

G. A. Constantinides
Department of Electrical Engineering, Imperial College
London, London SW7 2AZ, UK
e-mail: g.constantinides@imperial.ac.uk

Keywords Parallel computing · MapReduce · Pipelining · Geometric programming

1 Introduction

Recently, there has been a great deal of interest in how to program parallel applications, especially since parallel hardware is now common in desktop and portable computers. However, implementing complex applications efficiently on parallel computing structures is still highly challenging. Researchers [1] have explored ways to improve programmer productivity and maximize system efficiency. In this paper, we target advanced compilation techniques, which can compile sequential descriptions for parallel execution. This allows programmers in the traditional sequential programming environment, to focus on the algorithm level without worrying about the underlying hardware; parallelism can be automatically extracted afterwards by compilers. We propose an approach that automatically identifies a common computation pattern, the MapReduce pattern, in C-like descriptions and maps it onto an efficient parallel computing platform.

MapReduce is a technique widely used to improve parallelism of large-scale computations [1–3]. It partitions the computation into two phases: first, the *Map* phase, in which the same computation is performed independently on multiple data elements; second, the *Reduce* phase, in which the final result is calculated by reducing the results of the Map phase with an associative operator. A simple example of MapReduce is vector dot product, where the Map phase multiplies corresponding vector elements and the Reduce phase sums the products. MapReduce applies to

computations where: (1) there is no dependence between computations working on different parts of the input data set; and (2) the reduction operation is associative. These characteristics can be obtained from data flow analysis tools such as SUIF [4]. Examples of the MapReduce pattern include: matrix multiplication, Monte Carlo simulation and pattern match and detection.

In practice, MapReduce can be limited by memory bandwidth and the number of processing units (hardware computation resources). Given a hardware platform, it is often not obvious how to map a MapReduce pattern onto the platform to maximise system performance. Most previous methods require designers to identify the MapReduce pattern and specify the *Map* and *Reduce* functions explicitly. To our best knowledge, the work [5] is the first one that targets automating the exploitation of the MapReduce pattern. However, it is limited to a single-level MapReduce pattern and does not consider resource constraints on the reduction operation in the Reduce phase that affects the final designs of some applications.

We have proposed a methodology [6] for automatically optimising hardware designs written in C-like descriptions by combining both model-based and pattern-based transforms. Given system parameters, model-based approaches map a design into underlying mathematical models to enable rapid design space exploration. Pattern-based approaches perform optimisations by matching and transforming syntax or data flow patterns.

In this paper, we show how the methodology [6] can be applied to automate mapping the MapReduce pattern. We extend a previous model [5] to map applications with multiple MapReduce patterns onto a parallel computing structure. Customized loop-level parallelization and pipelining are used to balance system performance and hardware resource utilization.

The contributions of this paper are thus:

- an approach for automatically identifying the MapReduce computation pattern and mapping it to a parallel computing architecture;
- an extended geometric programming (GP) model considering resource constraints in the Reduce phase when mapping MapReduce computations onto a parallel computing platform, with constraints on memory bandwidth and hardware resources;
- a geometric programming model for concurrently mapping multiple nested MapReduce patterns on to a parallel platform; and
- an evaluation of the proposed automation approach using six application kernels, showing performance improvement up to 170 times compared to the initial designs on the target platform.

The rest of the paper is organised as follows. Section 2 details related work. Section 3 illustrates the problem under consideration using a simple example; Section 4 describes the proposed approach to deal with the problem. The geometric programming model formulating the design space of mapping the MapReduce pattern is presented in Section 5. Section 6 presents experimental results from applying the proposed approach to six real applications, and is followed by the conclusion and future work in Section 7.

2 Related Work

In this paper, we apply the methodology proposed in [6], which does not support the MapReduce pattern optimisation, to automate the mapping of multiple MapReduce patterns onto parallel computing platforms by combining pattern-based and geometric programming (GP) based transformations. Loop-level parallelization and pipelining are exploited to extract parallelism in MapReduce patterns.

The MapReduce programming model, named after the *Map* and *Reduce* functions in Lisp and other functional languages, has been developed and used in Google [2]. The Haskell functional language and Google's Sawzall are used to describe the *Map* and *Reduce* functions. Yeung et al. [3] apply the MapReduce programming model to design high performance systems on FPGAs and GPUs. All these methods require designers to identify the MapReduce pattern and specify the *Map* and *Reduce* functions explicitly. Liu et al. [5] proposes an approach for optimising designs with the MapReduce pattern at compile time. However, it is limited to a single-level MapReduce pattern and does not consider resource constraints on the reduction operation in the Reduce phase that affects the final designs of some applications. This paper extends [5] to map nested loops with multiple MapReduce patterns, as well as considering the additional resource constraints.

Loop-level parallelization has been widely used for improving performance [7]. Loop transformations, such as loop merging, permutation and strip-mining, are used to reveal parallelism. A GP model [8] is used to determine the tile size of multiple loops to improve data locality in a hierarchical memory system. Eckhardt et al. [9] recursively apply locally sequential, globally parallel and locally parallel, globally sequential schemes to map an algorithm onto a processor array

with a 2-level memory hierarchy. Loop pipelining is applied to pipeline the innermost loop [10] and outer loops [11]. An integer linear programming model is proposed [12] for pipelining outer loops in FPGA hardware coprocessors. Our proposed GP framework determines loop parallelization and pipelining for a MapReduce pattern at the same time.

Pattern-based transforms for hardware compilation have been explored by researchers like di Martino et al. [13] on data-parallel loops, as part of a synthesis method from C to hardware. Unlike our method, they do not allow user-supplied transforms. Compiler toolkits such as SUIF [4] allow multiple patterns to be used together, but give no support for including model-based transforms. Pattern matching and transforming can be achieved in tree rewriting systems such as TXL [14], but such systems do not incorporate hardware-specific knowledge into the transforms.

3 Problem Statement

This paper considers how to map an obvious but inefficient description, with a possibly implicit MapReduce pattern, onto highly parallel hardware. We use loop strip-mining and pipelining to extract parallelism from the sequential description. When a loop is strip-mined for parallelism, the loop that originally executes sequentially is divided into two loops, a new **for** loop running sequentially and a **for all** loop that executes in parallel, with the latter inside the former. The main constraints on mapping MapReduce are the computational resources of the target platform, which affect the size of each strip in strip-mining, and the bandwidth between processing units and memories that affects how operations are scheduled after loop partitioning. We use a locally parallel, globally pipelined structure to balance use of memory bandwidth and hardware resources.

To illustrate this problem, Fig. 1a shows a simple example, the dot product of two 8-bit vectors. Assume that arrays A and B are stored in off-chip SRAM with single port and can be accessed every cycle after pipelining. Our proposed framework first applies pattern-based transforms to this code. As shown in Fig. 1b, the original complex expression is decomposed into simple operations: two memory accesses, one multiplication and the result accumulation (result merging). This can reduce the number of logic levels of the generated circuits, improving system latency [6]. We refer the statements merging results as *merging statements* in this paper. Next, we generate a data flow graph (DFG) for the code, as shown in Fig. 1c, and use this to determine pipelining parameters. From the DFG, we can see that this example has the characteristics of the MapReduce pattern: multiplication executes independently on element pairs of array A and B, and the result merging operation is addition which is associative.

We thus map the dot product onto a parallel computing structure. Given a hardware platform with sufficient multipliers and each assignment in the code takes one clock cycle, Table 1 shows some design options for

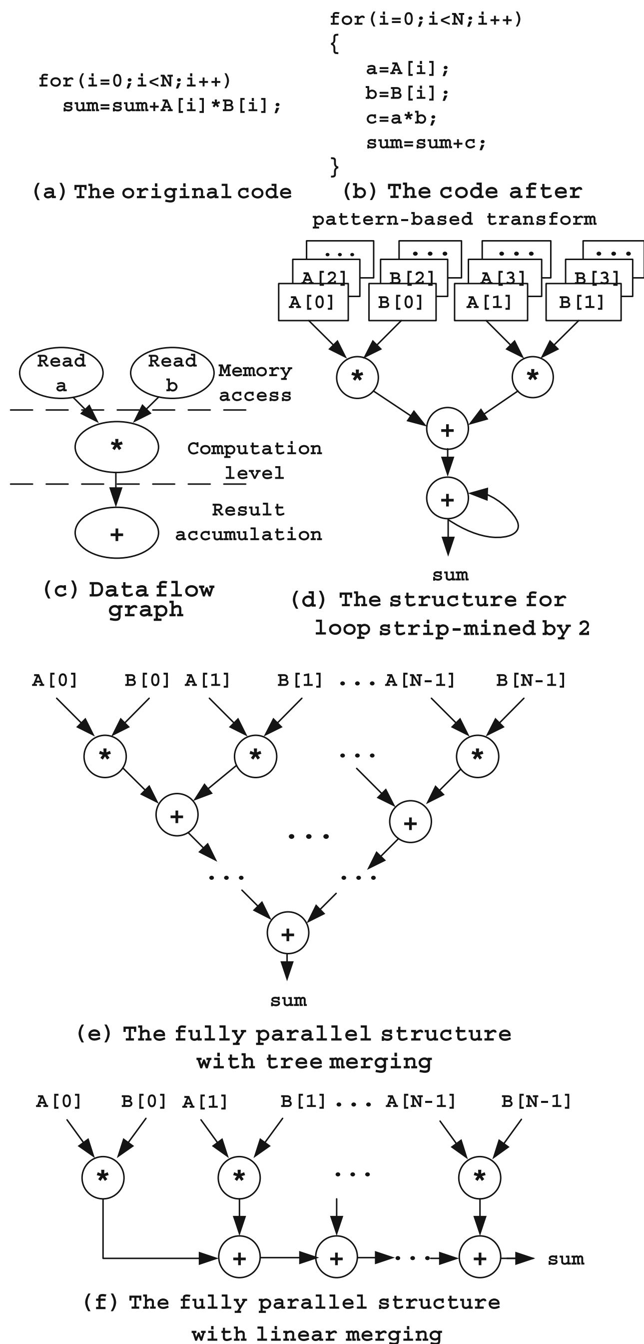


Figure 1 Motivating example: dot product.

dot product under different memory bandwidth constraints. Each design is represented by the number of parallel partitions (k) of the strip-mined loop, each with its own processing unit, and the initiation interval (ii) of pipelining the outer **for** loop. Ideally, if the memory bandwidth is $2N$ bytes per execution cycle as shown in the second row of Table 1, then N multiplications can execute in parallel and the final result can be merged using a tree structure, as shown in Fig. 1e where pipeline registers are not shown. In this case, the loop i is fully strip-mined and the dot product needs $\log_2 N + 2$ execution cycles: $\log_2 N$ execution cycles for result merging, one execution cycle for loading $2N$ data and one for multiplication.

In practice, $2N$ bytes per execution cycle memory bandwidth is unrealistic for large N. If memory bandwidth is one byte per execution cycle, then the fully parallel implementation in Fig. 1e needs $2N + \log_2 N + 1$ execution cycles to finish the dot product, where $2N$ execution cycles are used to load $2N$ data. In this scenario, pipelining the loop i with $ii = 2$ due to the two sequential memory accesses needs $2N + 2$ execution cycles to compute the dot product, as the design $(1, 2)$ in Table 1. If $N > 2$, the pipelined implementation is more promising than the fully parallel version, not just in speed but also in resource usage. Furthermore, if the memory bandwidth is 2 bytes per execution cycle, elements from each of A and B can be read at the same time, and the fully parallel version needs $N + \log_2 N + 1$ execution cycles.

Alternatively, if the loop i is strip-mined by 2, then every two iterations of the loop i execute in parallel on two processing units and the results are merged, shown in Fig. 1d. This structure can be further pipelined with $ii = 2$. This design, combining local parallelization and global pipelining, can perform the dot product in $2\lceil N/2 \rceil + 3$ execution cycles. The third design option is to pipeline loop i with $ii = 1$ without loop strip-mining and it is still promising as $N + 2$ execution cycles are spent on the product. Similarly, if the memory bandwidth is 3 bytes per execution cycle, Table 1 shows three possible design options. Here, the second option combining loop strip-mining and pipelining achieves a balance between speed and resource utilization.

We can see that there are multiple options for mapping the dot product onto a given hardware platform under different memory bandwidth constraints. Even more design options result if multipliers are also constrained. Finding the best design in terms of various criteria is not easy, and requires exploration of the design space to find the optimal design.

This paper proposes an approach with model-based and pattern-based transforms to deal with the problem of mapping the MapReduce pattern.

4 The Proposed Approach

Figure 2 shows how our proposed approach combines model-based and pattern-based transforms to map the MapReduce pattern onto a target hardware platform while achieving design goals.

The approach takes as input the sequential description of a design. We initially apply pattern-based transformations, such as function inlining and converting pointers to arrays, to the design to ease the dependence analysis. We apply further pattern-based transformations, including loop merging and loop coalescing, to the design to ensure it is in a certain form. This form enables us to simplify the mathematical model by removing variables, enabling it to be solved faster.

After dependence analysis, we check that the design contains the two characteristics of the MapReduce pattern given in Section 1. If they are found, we apply the model-based transformation for the MapReduce pattern, shown in the shaded block in Fig. 2.

We formulate the problem of mapping the MapReduce pattern onto a target parallel computing platform

Table 1 Possible design options of the dot product example.

Mem_bandwidth (Bytes/exe_cycle)	Designs (k, ii)	Exe_cycles	Multipliers
$2N$	$(N, 1)$	$\log_2 N + 2$	N
	$(N, 1)$	$2N + \log_2 N + 1$	N
	$(1, 2)$	$2N + 2$	1
2	$(N, 1)$	$N + \log_2 N + 1$	N
	$(2, 2)$	$2\lceil N/2 \rceil + 3$	2
	$(1, 1)$	$N + 2$	1
3	$(N, 1)$	$\lceil 2N/3 \rceil + \log_2 N + 1$	N
	$(3, 2)$	$2\lceil N/3 \rceil + 4$	3
	$(1, 1)$	$N + 2$	1

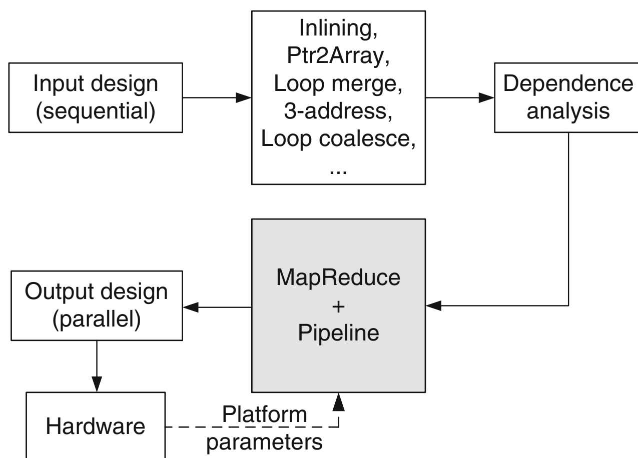


Figure 2 Combining model-based (such as MapReduce) and pattern-based approaches (such as loop coalescing) into a single approach.

as a geometric programming model. Solving this problem automatically generates an optimized design exploiting loop-level parallelization and pipelining for given hardware platform parameters such as memory size and computation resources. This paper focusses on presenting this geometric programming model.

As output our approach produces a design with explicit timing and parallelism. The design can then be implemented on the hardware platform using vendor tools to synthesize, place and route it. Currently, the model is used to optimize design parameters; we do not currently automate output hardware generation, but this is relatively straightforward.

Our approach can adapt to several variations on the MapReduce pattern:

- applications with implicit MapReduce pattern, such as the motion estimation algorithm [15] used in X264;
- applications with MapReduce pattern at any loop level, such as the Monte Carlo simulation of Asian Options [16];
- applications with 2-D MapReduce patterns, such as edge detection [17]; and
- application with multi-level MapReduce patterns, such as k-means clustering [18].

5 GP Model for Mapping MapReduce

As the MapReduce pattern is often present in sequential loops, we target a loop nest containing the MapReduce patterns with the number of iterations fixed at compile time.

We formulate the design exploration problem as a piecewise integer geometric program. Geometric programming (GP) [19] is the following optimization problem:

$$\begin{aligned} &\text{minimize} \quad f_0(x) \\ &\text{subject to} \quad f_i(x) \le 1,\ i = 1, \ldots, m \\ &\qquad\qquad\quad h_i(x) = 1,\ i = 1, \ldots, p \end{aligned}$$

where the objective function and inequality constraint functions are all in *posynomial* form, while the equality constraint functions are *monomial*. Unlike general nonlinear programming problems, GP is convex and thus has efficient solution algorithms with guaranteed convergence to a global solution [19].

In the next three subsections, we present the proposed piecewise integer geometric programming model for mapping a single loop with the MapReduce pattern, an extension of the model to a variant of the MapReduce pattern, and a further extension to map nested loops with multiple MapReduce patterns. Table 2 lists notation used in the models: capitals denote compile time constants, while lower case denotes integer variables. The models assume that assignments take one clock cycle, like Handel-C [20]. This limits clock speed to the slowest expression used; we ameliorate this by breaking expressions into three-address code.

5.1 Piecewise GP Model for MapReduce with Tree Reduce

In this section, a GP model for mapping a loop containing the MapReduce pattern with a tree reduce structure is presented. This model extends [5] to consider the constraints of the resources for the merging operations in the Reduce phase. The design variables include the number of parallel partitions of the loop and the pipeline initiation interval; the loop is mapped to a locally parallel, globally pipelined design.

The objective is to minimize the number of execution cycles of a loop after loop strip-mining and pipelining as below:

$$\text{minimize } v \times ii + \sum_{i=1}^{I} d_i + \sum_{j}^{\lceil \log_2 k \rceil} c_j + notFull \tag{1}$$

This expression is derived from the standard modulo scheduling length of pipelining [10, 11], assuming that the number of execution cycles for loading data from memory dominates the initiation interval (ii). Section 6 shows that this assumption is true for the tested benchmarks. The number of loop iterations v executing in each parallel partition is defined in Inequality 2.

$$N \times k^{-1} \le v \tag{2}$$

Table 2 Notation used in the models.

Notations	Description
k (k_l)	Parallel partitions of a loop (loop l)
ii	Initiation interval of pipeline
v (v_l)	Iterations in one partition of a loop (loop l)
x_f	No. resources f being used
d_i	Latency in execution cycles of computation level i of DFG
c_j (c_{sj})	Execution cycles of level j of merging tree (of statement s)
m_j (m_{sj})	Merging operations of level j of merging tree (of statement s)
r_s	Execution cycles of merging tree of statement s
L	No. nested loops
I_s	The loop level where statement s is located, $1 \leq I_s \leq L$
N (N_l)	No. iterations of a loop (loop l)
A	No. array references
$RecII$	Data dependence constraint on ii in the computation
W_f	Computation resources f required in a loop iteration
R_{if}	Computation resources f required in level i of DFG
I	No. computation levels of DFG
B (B_a)	Required memory bandwidth (of array a) in one loop iteration
Q_a	Set of loop variables in the indexing function of array a
M_b	Memory bandwidth
C_f	Computational resources f available
M_r	Resources r available for merging operations
F	F types of computation resources involved
$notAlign$ ($notAlign_a$)	0: data are aligned; 1: data are not aligned (of array a)
$notFull$ ($notFull_l$)	0: loop (l) is fully strip-mined; 1: loop (l) is partially strip-mined

The second term in the objective function 1 is the execution cycles taken by computation. There may be I ($I \geq 1$) computation levels in the DFG and the computation cycles of each loop iteration consist of the execution cycles of every level. The execution cycle d_i of computation level i is determined by R_{if} the requirement of each computation resource f in level i and the number of allocated resources x_f, as defined in inequalities 3.

$$R_{if} \times x_f^{-1} \times d_i^{-1} \leq 1, 1 \leq i \leq I, f \in F \tag{3}$$

The example in Fig. 1 needs one multiplier in the single computation level. For this simplest case, x_f taking one results in $d_1 = 1$. Assuming the computation level requires two multipliers, x_f could take on the value 1 or 2 depending on the resource constraint, leading to $d_1 = 2$ or $d_1 = 1$ cycles, respectively.

The last two terms of the objective 1 are the execution cycles taken by final result merging. When a tree structure is used as in Fig. 1d and e, the number of levels of the merging tree is $\lceil \log_2 k \rceil$, and thus the number of execution cycles taken by the merging tree is the sum of the execution cycles c_j spent on every level j. We arrange the number of merging operations in each level such that there are m_1 merging operations in the top level, and subsequently the following level has m_j merging operations, as defined in Eqs. 4 and 5.

$$m_1 \geq k/2 \tag{4}$$

$$m_j \geq m_{j-1}/2, 2 \leq j \leq \lceil \log_2 k \rceil \tag{5}$$

The number of execution cycles c_j of level j is determined by the number of merging operations in level j and M_r the number of resources available for the merging operations in Eq. 6, which is the resource constraint on the merging operations.

$$c_j \geq m_j/M_r, 1 \leq j \leq \lceil \log_2 k \rceil \tag{6}$$

The Boolean parameter $notFull$ is added to Eq. 1, for the case where the loop is partially strip-mined and sequential accumulation of the results is needed; Fig. 1d shows an example.

There are four constraints on the pipelining initiation interval ii, as shown in Eqs. 7–10. Inequality 7 is the memory bandwidth constraint. Given a fixed memory bandwidth M_b and the memory bandwidth required in one loop iteration B, more parallel loop iterations k require more input data and thus more data loading time, leading to a larger initiation interval (ii). Here, the Boolean parameter $notAlign$ is used to capture the situation where data are not aligned between storage and computation; this case may need one extra memory access to obtain requested data. Inequality 8 gives the computation resource constraints, where W_f is the number of each computation resource f required in one loop iteration. Inequality 9 is the data dependence constraint on ii, as used in [10, 11]. The available resource for merging operations also has impact on ii, as shown in inequality 10. For k intermediate computation results generated in parallel, up to k merging operations are needed to obtain the final result.

$$B \times k \times M_b^{-1} \times ii^{-1} + notAlign \times ii^{-1} \leq 1 \tag{7}$$

$$W_f \times x_f^{-1} \times ii^{-1} \leq 1, f \in F \tag{8}$$

$$RecII \times ii^{-1} \leq 1 \quad (9)$$

$$k \times M_r^{-1} \times ii^{-1} \leq 1 \quad (10)$$

Inequality 11 captures how computation resources constrain parallelization. We achieve an efficient solution by balancing the resources allocated to loop parallelization (k) and pipelining (ii), given that the number of resource f available is C_f.

$$k \times x_f \leq C_f,\ f \in F \quad (11)$$

Inequalities 12 and 13 are the ranges of integer variables k and x_f, respectively. Inequality 13 may appear to be redundant, but the GP solver requires explicit ranges for all variables.

$$1 \leq k \leq N \quad (12)$$

$$1 \leq x_f \leq C_f,\ f \in F \quad (13)$$

Overall, the only item making the relaxed problem 1–13 (allowing all variables to be real numbers) not a GP problem is the logarithm $\lceil \log_2 k \rceil$. The logarithms must be eliminated in order to use a GP solver. However, we note that $\lceil \log_2 k \rceil$ is constant in certain ranges of k. Therefore, the problem 1–13 can be seen as a piecewise integer geometric problem in different ranges of k; the number of subproblems increases logarithmically with k. For large $k = 1{,}000$, there are 11 integer GP problems and each problem can be quickly solved using a branch and bound algorithm used in [21] with a GP solver as the lower bounding procedure. Section 6 shows the performance of the piecewise GP model. The solution (k, ii) to the problem 1–13 tells us how to MapReduce and pipeline the loop under consideration.

The formulation 1–13 targets mapping a loop onto a parallel computing structure. The loop need not be innermost. For example, in order to map a 2-level loop nest, unrolling the innermost loop allows the formulation to work. Equivalently, we can add several constraints as shown below, to the formulation 1–13. Then we can automatically determine pipelining the innermost loop (ii'), strip-mining (k) the outer loop and global pipelining (ii); the Monte Carlo simulation of Asian Option benchmark in Section 6 exhibits this case. Inequality 14 shows the constraint, the innermost loop (with N' iterations) scheduling length, on the initiation interval ii of the outer loop, as the outer loop cannot start the next iteration before the innermost loop finishes. Inequalities 15 and 16 are the computation resource and data dependence constraints on the initiation interval of pipelining the innermost loop. The required resources W'_f in the innermost loop are included in W_f, in order to share resources among operations.

$$(N' - 1) \times ii' \times ii^{-1} + ii^{-1} \times \sum_{i=1}^{I'} d'_i \leq 1 \quad (14)$$

$$W'_f \times x_f^{-1} \times ii'^{-1} \leq 1,\ f \in F \quad (15)$$

$$RecII' \times ii'^{-1} \leq 1 \quad (16)$$

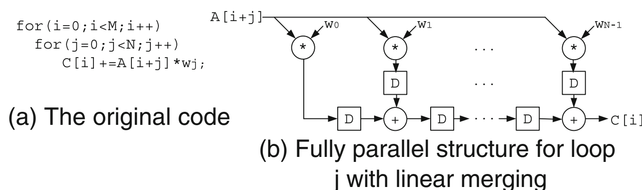
Finally, the MapReduce pattern is also present in some applications in 2-D form. Our pattern-based transforms could coalesce 2-D loops into 1-D, so the GP formulation described above can apply. We extend inequality 7 to cover 2-D data block ($Row \times Col$) access, allowing for the 2-D data block access to not necessarily be linearized:

$$Row \times \left(B \times k \times M_b^{-1} \times ii^{-1} + notAlign \times ii^{-1}\right) \leq 1. \quad (17)$$

5.2 Extension of the Model for Linear Reduce

In the previous subsection, the logarithm expression in the objective function results from using a tree structure for Reduce. Alternatively, a linear structure could be used; Fig. 1f shows an example for dot product. For k parallel computations, the linear structure needs k execution cycles to merge all intermediate results. Although longer than the tree structure, some applications with the MapReduce pattern using the linear structure to merge results can both keep the same throughput as the tree structure and reduce the memory bandwidth requirement. Other merging structures are possible; we have not yet found a compelling use case for these.

Figure 3a shows an example of 1-D correlation. The innermost loop j of the code has a similar computation pattern to the dot product, and can map onto a parallel computing structure using MapReduce. The differences from the dot product are that one multiplication operand w_j is constant and when the outer loop i iterates, the data of A are shifted invariantly into the computation. This is potentially suitable for pipelining.



(a) The original code

(b) Fully parallel structure for loop j with linear merging

Figure 3 An example: 1-D correlation.

If the innermost loop j can be fully strip-mined, then the N multiplications execute at the same time, and the intermediate computation results are linearly merged, as shown in Fig. 3b. An important characteristic of this structure is its regular shape, which could benefit the placement and routing of hardware resources. After mapping loop j, fully pipelining loop i produces a $C[i]$ every cycle after N execution cycles. If any multipliers and memory access bandwidth are still available, loop i could be further partitioned.

Many signal and image processing applications use 1-D correlation or convolution or show similar behavior, so we extend the formulation in Section 5.1 for linear reduction. In the objective function 18, we replace $\sum_j^{\lceil \log_2 k \rceil} c_j$ with k. Constraints 4–6 are no longer applied. When the loop is partially strip-mined, $notFull$ here represents a delay of the accumulated result which feeds back to the linear structure in the next iteration. Because the number of cycles taken to multiply each element of A with the N coefficients decreases as the number of parallel partitions increases, the data dependence distance of loop j similarly decreases. Therefore, we add inequality 19.

$$\text{minimize } v \times ii + \sum_{i=1}^{I} d_i + k + notFull \tag{18}$$

$$N \times k^{-1} \times ii^{-1} \le 1 \tag{19}$$

Now, the formulation with objective 18 and constraints 7–19 is a GP model, mapping MapReduce patterns to a parallel computing platform with a linear reduction.

5.3 Extension of the Model for Multi-level MapReduce

The GP models presented above are for mapping a single loop level with the MapReduce pattern onto a parallel computing platform. This section extends the models to map nested loops with the MapReduce patterns. We target a rectangular loop nest with L loops $(I_1, I_2, \ldots, I_L)$, where I_1 is the outermost loop, I_L is the innermost loop and the loop length of loop l is N_l. For simplicity, we assume that all computational operations reside in the innermost loop, and only the innermost loop is pipelined. The statements outside the innermost loop are divided into two disjoint sets: $S_1 = \{$merging statements outside the innermost loop$\}$ and $S_2 = \{$non-merging statements outside the innermost loop$\}$.

Expressions 1 and 18 correspond to the execution cycles of the innermost loop alone using tree or linear reduction structures. We redefine them as $e = v_L \times ii + \sum_{i=1}^{I} d_i + r_L + notFull$, where r_L is the number of execution cycles of the merging statement s_L in the innermost loop. Then, we can formulate the total execution cycles of the target loop nest as the new objective:

$$\text{minimize } \prod_{l=1}^{L-1} v_l \times e + \sum_{s \in S_2} \prod_{l=1}^{I_s} v_l + \sum_{s \in S_1} \left(v_{I_s} \times notFull_{I_s} + r_s \right) \tag{20}$$

where I_s is the loop level in which statement s is located, and

$$r_s = \sum_{j}^{\lceil \log_2 k_{I_s} \rceil} c_{sj} \;\; \textbf{or} \;\; k_{I_s} \tag{21}$$

is the number of execution cycles taken by merging statement s using tree reduction structure or linear reduction structure, where c_{sj} have the same meaning as c_j in Eq. 1, but for merging statement s outside the innermost loop. r_L also complies with Eq. 21 for the merging statement in the innermost loop.

The number of iterations v_l of loop l after partitioning into k_l parallel segments is

$$N_l \times k_l^{-1} \le v_l, 1 \le l \le L \tag{22}$$

$$1 \le k_l \le N_l, 1 \le l \le L. \tag{23}$$

Moreover, the constraints of the resources used for merging operations in the tree structure (4–6) and (10) need to be defined according to each merging statement in the loop nest.

$$m_{s1} \ge k_{I_s}/2, s \in S_1 \cup \{s_L\} \tag{24}$$

$$m_{sj} \ge m_{sj-1}/2, s \in S_1 \cup \{s_L\}, 2 \le j \le \lceil \log_2 k_{I_s} \rceil \tag{25}$$

$$c_{sj} \ge m_{sj}/M_r, s \in S_1 \cup \{s_L\}, 1 \le j \le \lceil \log_2 k_{I_s} \rceil \tag{26}$$

$$k_L \times M_r^{-1} \times ii^{-1} \le 1 \tag{27}$$

Correspondingly, the memory bandwidth constraint 7 is redefined as:

$$B_a \times \prod_{l \in Q_a} k_l \times M_b^{-1} \times ii^{-1} + notAlign_a \times ii^{-1} \le 1, a \in A \tag{28}$$

where each of A array references under operations is stored in a memory bank with bandwidth M_b, each reference a requires B_a memory bandwidth in one loop iteration and Q_a is the set of loop variables in the indexing function of reference a. Unlike Eq. 7, here, each array reference is treated individually, because different array references may have different bandwidth

Table 3 Benchmark details.

Benchmark	# loops	Refs	MapReduce pattern
MAT64	3	2	Two levels: in the outer loops and the innermost loop
ME	Implicit	2	Two levels: implicit and in SAD
Sobel	4	2	Two levels: in outer two loops and inner two loops
MCS	2	0	In the outer loop
1-D correlation	2	1	In the innermost loop
k-means	4	1	In the inner three loops

requirements after mapping multiple MapReduce patterns. Meanwhile, the computational resource constraint 11 is redefined as:

$$\prod_{l=1}^{L} k_l \times x_f \le C_f,\ f \in F \quad (29)$$

This model allows us to map the multi-level MapReduce pattern and thus extends the applicability of the proposed approach. This model is still a piecewise GP problem. With multiple nested patterns, however, this problem has more variables and more subproblems caused by the logarithm, and thus could take longer to solve.

6 Experimental Results

We apply the proposed approach to six kernels: multiplication of two 64 × 64 matrices (MAT64), the motion estimation (ME) algorithm [15] used in X264, the Sobel edge detection algorithm (Sobel) [17], Monte Carlo simulation (MCS) of Asian Option Pricing [16], 1-D correlation and *k*-means clustering [18]; Table 3 shows benchmark parameters. Each benchmark contains the MapReduce patterns. One level of the MapReduce pattern exists in MCS and 1-D correlation, to which the models presented in Sections 5.1 and 5.2 are applied, respectively. Two levels of the MapReduce pattern are identified in ME, MAT64 and Sobel. We apply the transforms [6] to the outer level, exploiting data reuse and loop-level parallelism. For the MapReduce pattern in the inner loops, we apply the model in Section 5.1 to exploit inner loop parallelization and pipelining. Lastly, we apply the extended model in Section 5.3 to *k*-means clustering, which has three levels of the MapReduce pattern. All GP models are solved by a MATLAB tool box, named YALMIP [21]. Different designs are represented by the number of parallel loop partitions and the pipelining initiation interval, (k_l, ii).

In our experiments, the target platform is an FPGA-based system with off-chip SRAM. The Monte Carlo simulation of Asian Option involving floating point arithmetic is implemented in Xilinx XC4VFX140 with 192 DSP48, and the other five kernels are implemented in XC2v8000, which has 168 embedded hard multipliers and 168 dual-port RAM blocks. On-chip RAMs configured as scratch-pad buffers and registers are used to increase memory bandwidth. All experimental results are obtained on the target platform after synthesis, placement and routing. For ME and Sobel the frame size is the QCIF luminance component (144 × 176 pixels).

Matrix–matrix Multiplication is a typical arithmetic kernel used in many applications. Each element of the output matrix is the dot product of one row of one input matrix and one column of the other input matrix. The dot product is the inner MapReduce pattern level and computing multiple elements of the output matrix in parallel is the outer MapReduce level. In our implementation, each matrix element is 8 bits. Memory bandwidth and the number of hard multipliers embedded in the FPGA are the constraints on mapping the inner level MapReduce pattern. Figure 4 shows results from applying our proposed piecewise GP model (1–13) to the innermost loop of MAT64. Given a memory bandwidth and the number of multipliers, the figure reveals

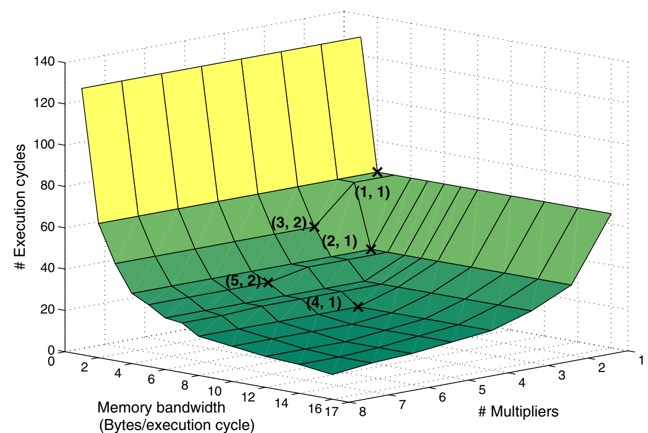


Figure 4 Designs proposed by the framework for MAT64. The bracket next to each design shows (k, ii).

the performance-optimal design: for example, when the memory bandwidth is 3 bytes/execution cycle and 3 multipliers are available, the performance-optimal design is ($k = 3, ii = 2$); when the memory bandwidth is 5 bytes/execution cycles and 5 multipliers available, the performance-optimal design is ($k = 5, ii = 2$), as shown in Fig. 4. As the memory bandwidth and the number of multipliers available increase, the execution cycles of the performance-optimal designs decrease.

To verify these designs, we implement several representative designs on the target platform. Figure 5 shows the real execution time and the utilization of on-chip multipliers, RAMs and slices of four designs, where *y* axes are in a logarithmic scale. The first is the original design without any optimisation. The second is optimised by the transforms [6], where the inner MapReduce pattern is not mapped. Based on the second design, the third design applies our proposed framework to parallelize the innermost loop by 2 and pipeline the sequential iterations with $ii = 1$ with memory bandwidth 4 bytes/execution cycle. This corresponds to the design (2, 1) in Fig. 4. Here, we only partition the innermost loop into two parallel segments, because the outer level MapReduce pattern has been mapped onto 80 parallel processing units and thus each unit has two multipliers available as there are in total 168 multipliers on the target platform. The third design in Fig. 5 shows the result after mapping the two MapReduce pattern levels. The system performance speeds up by about 66 times compared to the original design and speeds up about twofold compared to the second design which only maps the outer MapReduce pattern level.

To test the impact of mapping the outer level and the inner level of MapReduce patterns in MAT64 in different orders, we also first apply the proposed GP framework to the innermost loop and then apply the transforms [6] to the outer loops. For the inner level, the design (4, 1) in Fig. 4 is chosen, as the maximum bandwidth of the on-chip RAMs is 8 bytes/execution cycle, given that two input matrices are stored in two independent RAM blocks and each is accessed through one port. The outer level, then, is mapped onto 40 parallel processing units by [6]. The optimised design after mapping the MapReduce patterns in this order is the fourth design in Fig. 5. Figure 5a shows that this design has a performance almost equal to the third design generated by mapping the outer MapReduce pattern first, whereas the fourth design reduces the requirement of on-chip RAM blocks and slices, as shown in Fig. 5c and d.

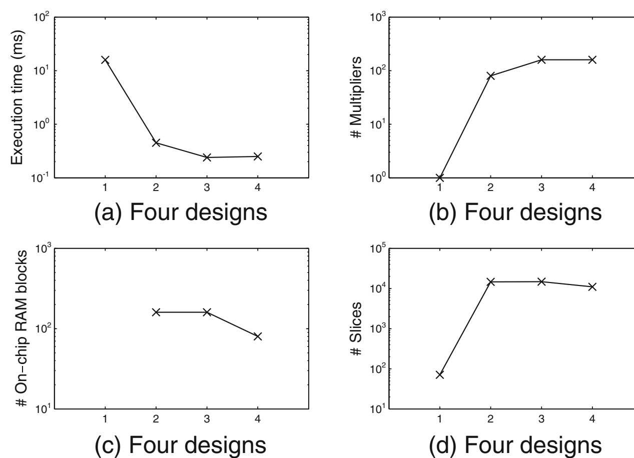


Figure 5 Implementation results of some designs of MAT64. *y* axes use a logarithmic scale and in **c** the first design does not involve on-chip RAM.

Fast Motion Estimation Algorithm by Merritt and Vanam [15], used in X264, is highly computation-intensive. To determine the motion vectors for each macro block of the frame being encoded, the algorithm performs multiple searches on the reference frames. During a search the current macro block being encoded is matched with several candidate blocks of the same size, and each matching involves the calculating the sum of absolute differences (SAD) of the pixels from the current block and a candidate block. Finally, the candidate block with the minimum SAD is chosen as the reference block to estimate the current macro block. This algorithm has the MapReduce pattern at two loop levels: 1) the *outer-level* loop matching the current block with several candidate blocks, merging results with a *min* function, and 2) within the SAD computation. Due to the use of pointers and function calls, the former is not obvious in the algorithm level description. Directly mapping the algorithm onto a parallel computing structure leads to an inefficient design: the original design shown in Fig. 6b. Our proposed pattern-based transforms perform function inlining to reveal this MapReduce pattern, and the model-based transforms are applied for data reuse [6] as the block matching process reuses much data. The proposed approach in this paper is applied to SAD computation. Results are shown in Fig. 6a. Memory bandwidth is the main constraint, given that logic for addition and comparison operations is adequate on the target platform. The design Pareto frontier proposed by our approach under a range of memory bandwidths are shown.

We implement one search path of the motion estimation algorithm; sequential searches can use the same circuit. The original design, three designs (1, 1), (16, 9) and (32, 5) proposed by our approach and four possible designs are implemented on the target platform to

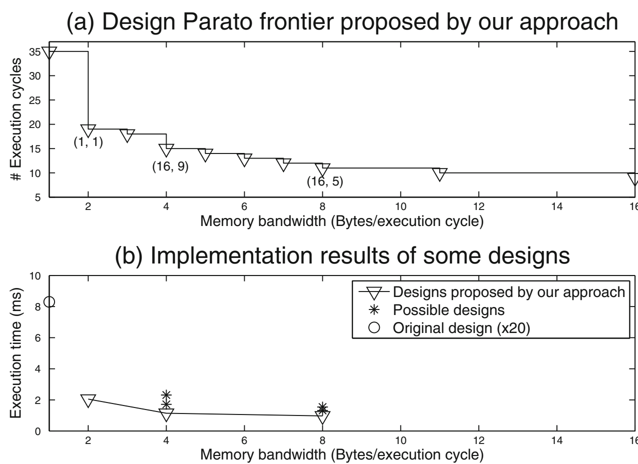


Figure 6 Experimental results of ME. **a** Design Pareto frontier; **b** Implementation results of some designs on the FPGA. The bracket next to each design shows (k, ii).

obtain the real execution time and memory bandwidth. Figure 6 shows how different memory bandwidth constraints affect the designs from our approach. The design applying our approach to both loop levels with the MapReduce pattern improves performance by up to 170 times compared to the original design, and by up to two times compared to the design which only applies our approach to the outer-level loop, shown by the downward-pointing triangle in Fig. 6b.

Monte Carlo Simulation (MCS) has wide applications in finance. Asian Option prices depend on the average price observed over a number of discrete dates in the future, and the mathematical pricing model has no closed solution [16]. Therefore, MCS is used to estimate the average price. This simulation contains two loops: the outer loop that controls the times of executing MCS and sums the results together, and the inner loop that sets several observation points during a contract period; we mention this case in Section 5.1. Therefore, the extended piecewise GP model applies to the outer loop. As this application uses floating point, the number of DSP blocks that execute the floating point operations is the constraint on the target platform limiting the computations in both Map and Reduce phases. Figure 7a shows the performance Pareto frontier generated by the proposed approach. Again, as the number of DSPs increases, more iterations of the outer loop can execute in parallel and the execution cycles decrease. Four promising but sub-optimal designs are also shown in the figure. The design parameters of these four designs and the corresponding optimal designs are shown in Fig. 7a as well. We can see that our proposed approach allocates DSP blocks between loop parallelization and pipelining to achieve better balanced and faster designs.

We have implemented the MCS of Asian Option Pricing using the HyperStream library [22] on the target platform. Because the library implementation tries to greedily pipeline all operations, we can only implement the application by first allocating the DSPs to pipeline the innermost loop, and thus the resources to parallelize the outer loop iterations are limited. We have been able to only implement the four sub-optimal designs as shown in stars in Fig. 7a. The implementation results of these four designs in Fig. 7b show that the extended GP model can correctly reflect the relative performance figures of different designs and thus is able to determine the performance-promising one. We believe that this can also be observed for the performance-optimal designs.

In the three benchmarks above, the MapReduce pattern has been mapped to the target platform and the computation results have been accumulated using a tree structure. In the next two benchmarks, the linear structure is examined.

1-D Correlation is an important tool used in image and signal processing; Fig. 3 shows an example. In this experiment, we implement three instantiations of 1-D correlation: 3-tap, 5-tap and 11-tap. We map each case onto the target FPGA platform with both the linear and the tree accumulating structures; Fig. 8 shows results when memory bandwidth is limited, such as 1 byte per execution cycle as shown here. It can be seen that in all three cases, tree reduction costs more slices, up to

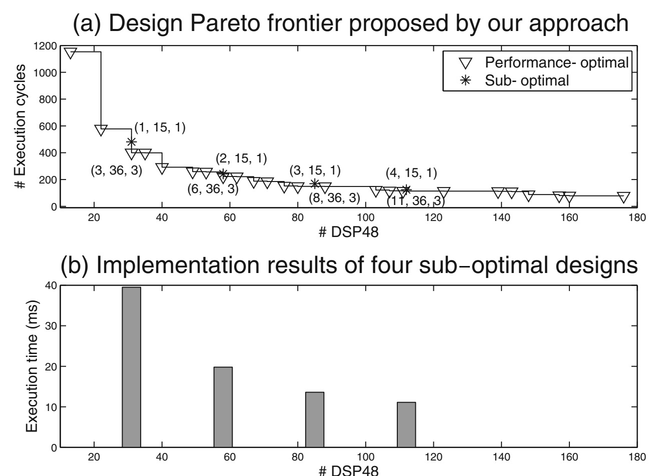


Figure 7 Experimental results of MCS of asian option. **a** Design Pareto frontier. **b** Implementation results of some designs on the FPGA. The bracket next to each design shows (k, ii, ii').

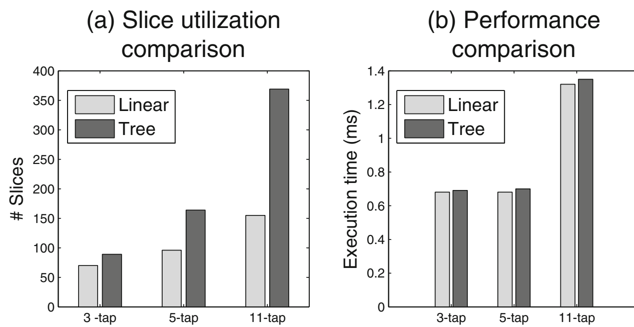


Figure 8 Comparison between the linear and the tree structure. **a** Resource utilization comparison. **b** Performance comparison.

two times for the 11-tap case, and as the number of taps increases the cost difference between the linear and the tree structures increases, as shown in Fig. 8a. Also, implementations with linear reduction show a small execution time decrease, as Fig. 8b shows. These are because for correlation operations, MapReduce with tree reduction requires input shift registers to feed data for parallel processing, resulting in larger area and longer latency.

Sobel Edge Detection algorithm [17] is a well known image processing operator, comprising four loop levels with two 2-level loop nests in the inner two levels. The transforms in [6] merge the inner two 2-level loop nests, buffer reused data in on-chip RAMs and parallelize the outer two loops. In the inner loops, there is a 2-D correlation with a 3×3 filter. We first apply the extended GP model in Section 5.2 to map the innermost loop with linear reduction, and then apply the piecewise GP model in Section 5.1 to map the loop next to the innermost loop with tree reduction. We do not map the two loops with a linear reduction because linearly reducing results from 2-D has a large latency and requires many registers to buffer intermediate results. The design mapping only the outer MapReduce pattern level and designs that map the two MapReduce pattern levels are implemented on the target platform, together with the original design. Figure 9 shows that when memory bandwidth is 2 Bytes/execution cycle, the design with the hybrid tree and linear reduction improves the performance about three times compared to the design mapping only the outer level MapReduce pattern, and shows 125 times speedup compared to the original design.

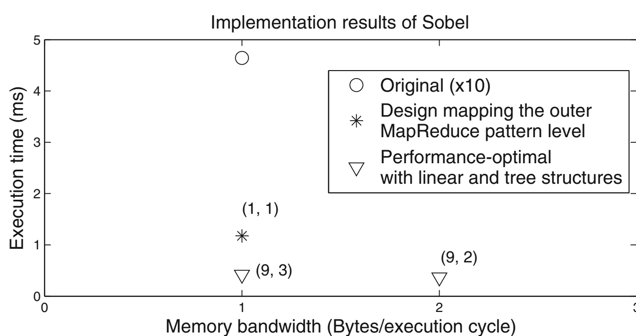


Figure 9 Experimental results of Sobel designs proposed by the framework. The bracket next to each design shows (k, ii).

K-means Clustering [18] is often used in computer vision. Data are segmented into k clusters, depending on similarity. The inner three loops of this algorithm [18] exhibit three nested MapReduce patterns. Therefore, we apply the extended model presented in Section 5.3 to map the inner three loops onto the target parallel platform. The experimental results are shown in Fig. 10 for clustering 64 randomly generated 8-dimensional vectors into four clusters. The design Pareto frontier proposed by our approach is plotted in Fig. 10a when the target platform has 168 embedded multipliers available. We implement the original design, which does not exploit embedded MapReduce patterns and some designs automatically mapped by our approach on the target platform, as shown in Fig. 10b. The parameters, which guide the mapping of the three loops under the different multiplier constraints, of some designs are presented. We can observe that the Pareto frontier proposed by our approach clearly matches the real

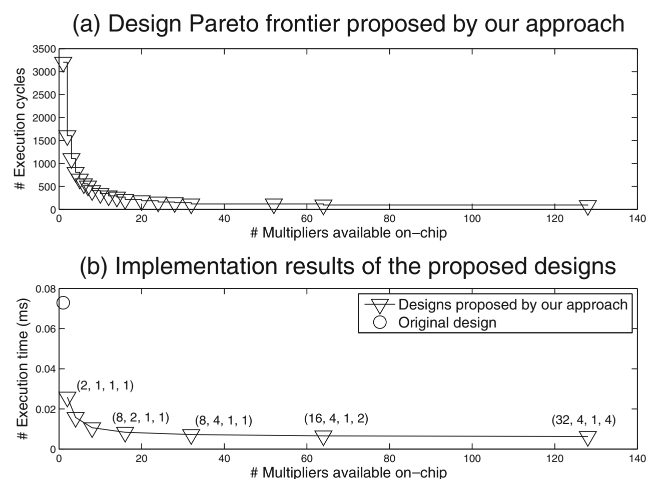


Figure 10 Experimental results of k-means clustering. **a** Design Pareto frontier; **b** Implementation results of the proposed designs on the FPGA. The bracket next to each design shows (k_1, k_2, k_3, ii).

design trend over the multiplier constraints. On the target platform, the designs proposed by our approach improve the speed of the tested clustering algorithm by up to 12 times compared to the initial design.

The execution time for solving the piecewise GP models proposed in this paper increases linearly as the number of pieces increases and exponentially with the number of nested MapReduce pattern levels. On average, for the first five benchmarks, an optimal design is generated by the piecewise GP framework within 15 s, while for the last one with three nested MapReduce patterns about 190 s are needed to obtain a design; the execution times are obtained on a machine with a single 3 GHz CPU. Since each piece of the piecewise GP models is independent, it is possible to speed up the solving process on a multi-core machine.

7 Conclusion

In this paper, we present an automated approach for mapping designs in a sequential description exhibiting the MapReduce pattern onto parallel computing platforms. Our approach combining model-based and pattern-based transformations automates the identification and mapping of MapReduce patterns using geometric programming. Using our approach on six applications achieves up to 170 times speedup on the target platform compared to the unoptimised initial designs. The experimental results also demonstrate that our approach can find performance-optimal designs within the design space of each application under different platform constraints.

Future work includes identifying other possible variants of the MapReduce pattern and dealing with other common patterns for parallel applications. These may further improve system performance.

Acknowledgements We thank FP6 hArtes project, the EPSRC, AlphaData and Xilinx for their support. We also thank the anonymous reviewers for their comments.

References

1. Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., et al. (2006). *The landscape of parallel computing research: A view from Berkeley*. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183.
2. Dean, J., & Ghemawat, S. (2004). MapReduce: Simplified data processing on large clusters. In *6th symp. on OSDI* (pp. 137–150).
3. Yeung, J. H., Tsang, C., Tsoi, K., Kwan, B. S., Cheung, C. C., Chan, A. P., et al. (2008). Map-reduce as a programming model for custom computing machines. In *Proc. FCCM* (pp. 149–159).
4. Hall, M. W., Anderson, J. M., Amarasinghe, S. P., Murphy, B. R., Liao, S.-W., Bugnion, E., et al. (1996). Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer, 29*(12), 84–89.
5. Liu, Q., Todman, T., Luk, W., & Constantinides, G. A. (2009). Automatic optimisation of mapreduce designs by geometric programming. In *Proc. int. conf. on FPT, Sydney, Australia* (pp. 215–222).
6. Liu, Q., Todman, T., Coutinho, J. G. de F., Luk, W., & Constantinides, G. A. (2009). Optimising designs by combining model-based and pattern-based transformations. In *Proc. FPL* (pp. 308–313).
7. Banerjee, U. K.: *Loop parallelization*. Kluwer Academic (1994).
8. Renganarayana, L., & Rajopadhye, S. (2004). A geometric programming framework for optimal multi-level tiling. In *Proc. conf. supercomputing* (p. 18). IEEE Computer Society.
9. Eckhardt, U., & Merker, R. (1999). Hierarchical algorithm partitioning at system level for an improved utilization of memory structures. *IEEE Trans. CAD* (Vol. 18, pp. 14–24).
10. Allan, V., Jones, R., Lee, R., & Allan, S. (1995). Software pipelining. *ACM Computing Surveys, 27*(3), 367–432.
11. Rong, H., Tang, Z., Govindarajan, R., Douillet, A., & Gao, G. R. (2004). Single-dimension software pipelining for multidimensional loops. In *IEEE Proc. on CGO* (pp. 163–174).
12. Turkington, K., Constantinides, G. A., Masselos, K., & Cheung, P. Y. K. (2008). Outer loop pipelining for application specific datapaths in FPGAs. *IEEE Transactions on VLSI, 16*(10), 1268–1280.
13. di Martino, B., Mazzoca, N., Saggese, G. P., & Strollo, A. G. M. (2002). A technique for FPGA synthesis driven by automatic source code synthesis and transformations. In *Proc. FPL*.
14. The TXL programming language (2010). http://www.txl.ca/. Accessed 2010.
15. Merritt, L., & Vanam, R. (2007). Improved rate control and motion estimation for h.264 encoder. In *Proc. ICIP* (pp. 309–312).
16. Financial modeling–Monte Carlo analysis. http://www.interactivesupercomputing.com/success/pdf/caseStudy_financial%modeling.pdf. Accessed 2009.
17. Green, B. (2002). Edge detection tutorial. http://www.pages.drexel.edu/~weg22/edge.html. Accessed 2010.
18. Russell, M. (2004). C programme to demonstrate k-means clustering on 2D data. University of Birmingham. Accessed 2010.
19. Boyd, S., & Vandenberghe, L. (2004). *Convex optimization*. Cambridge University Press.
20. Mentor Graphics. DK design suite: Handel-C to FPGA for algorithm design. http://www.mentor.com/products/fpga/handel-c/dk-design-suite/. Accessed 2010.
21. Löfberg, J. (2004). YALMIP: A toolbox for modeling and optimization in MATLAB. In *Proc. CACSD*.
22. Morris, G. W., & Aubury, M. (2007). Design space exploration of the European option benchmark using hyperstreams. In *Proc. FPL* (pp. 5–10).

Qiang Liu received the B.S. and M.Sc. degrees in electrical and electronic engineering from Tianjin University, Tianjin, China, and the PhD degree from Imperial College London, in 2001, 2004, and 2009, respectively. He is currently a research associate in the Department of Computing, Imperial College London, London, U.K. From 2004 to 2005, he was with the STMicroelectronics Company Ltd., Beijing, China. His research interests include high level synthesis, design methods and optimization techniques for heterogeneous computing systems.

Tim Todman received the BSc degree from the University of North London, and the MSc and PhD degrees from Imperial College London, in 1997, 1998 and 2004, respectively. He is currently a research associate in the Department of Computing, Imperial College London, London, UK. His research interests include hardware compilation and implementation of graphics algorithms on reconfigurable architectures.

Wayne Luk is Professor of Computer Engineering with Imperial College London. He was a Visiting Professor with Stanford University, California, and with Queen's University Belfast, UK. His research includes theory and practice of customizing hardware and software for specific application domains, such as multimedia, financial modeling, and medical computing. His current work involves high-level compilation techniques and tools for high-performance computers and embedded systems, particularly those containing accelerators such as FPGAs and GPUs. He received a Research Excellence Award from Imperial College, and 11 awards for his publications from various international conferences. He is a Fellow of the IEEE and the BCS.

George A. Constantinides (S.96.M.01.SM.08) received the M.Eng. degree (with honors) in information systems engineering and the Ph.D. degree from Imperial College London, London, U.K., in 1998 and 2001, respectively.

Since 2002, he has been with the faculty at Imperial College London, where he is currently Reader in Digital Systems and Head of the Circuits and Systems research group. Dr. Constantinides is a Fellow of the BCS. He is an Associate Editor of the IEEE Transactions on Computers and the Journal of VLSI Signal Processing. He was a Program Cochair of the IEEE International Conference on Field-Programmable Technology in 2006 and Field Programmable Logic and Applications in 2003, and is a member of the steering committee of the International Symposium on Applied Reconfigurable Computing. He serves on the technical program committees of several conferences, including DAC, FPT and FPL.