Automatic Optimisation of MapReduce Designs by Geometric Programming

Qiang Liu [1], Tim Todman [2], Wayne Luk [3] and George A. Constantinides [4]

[1,2,3]*Department of Computing,* [4]*Department of Electrical Engineering, Imperial College, London SW7 2AZ, UK*
{qiang.liu2, timothy.todman, w.luk, g.constantinides}@imperial.ac.uk

***Abstract*—Many important applications can be expressed using the *MapReduce* pattern, where a computation is decomposed into a *Map* phase on which each element of source data is independently operated, followed by a *Reduce* phase in which the mapped elements are combined with an associative operator. We develop an approach for compiling applications with the MapReduce pattern into parallel hardware. Using optimisation techniques based on geometric programming, we map the computation onto a resource-constrained architecture. Furthermore, we explore important variations of MapReduce, such as making the Reduce a linear structure rather than a tree structure. Results for four benchmarks show that our approach can improve system performance by up to 170 times compared to the initial designs.**

I. Introduction

Recently, there has been a great deal of interest in how to program parallel applications, especially as parallel hardware is now common on 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. By using a sequential programming language like C, programmers can focus on the algorithm level without worrying about the underlying hardware; furthermore, parallelism can be automatically extracted. In this paper, we propose a framework that automatically identifies a common computation pattern in C-like descriptions and exploits the MapReduce pattern to transform it onto an efficient parallel computing architecture.

We have proposed a methodology [2] 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. Using such patterns to transform code into the form expected by model-based approaches makes the models simpler and easier to implement.

MapReduce is a technique widely used to improve parallelism of large-scale computations [1], [3], [4]. 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 accumulating the results of the Map phase with an associative operator. MapReduce can apply to computations where: (a) there is no dependence between computations working on different parts of the input data set; and (b) the accumulation operation is associative and commutative. These characteristics can be obtained from data flow analysis tools such as SUIF [5]. 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. All previous methods require designers to identify the MapReduce pattern and specify the *Map* and *Reduce* functions explicitly.

In this work, we build a geometric programming model for efficiently mapping applications which have the two MapReduce characteristics onto a parallel computing structure. Customized loop-level parallelization and pipelining are used to balance system performance and hardware resource utilization. We use pattern-based transforms to convert the source into a form the geometric progrmming model can use.

The contributions of this paper are thus:

- a framework combining model-based and pattern-based transforms for automatically identifying the MapReduce computation pattern and transforming it to a parallel computing architecture (Section III);
- a geometric programming (GP) model mapping the MapReduce computations onto a parallel computing system with constraints on memory bandwidth and hardware resources, by concurrently exploiting loop strip-mining and pipelining, and using a tree structure in the Reduce phase (Section IV-A);
- a variation of the geometric programming model using a linear reduce structure, which can save system area for some applications (Section IV-B); and
- an evaluation of the framework using matrix-matrix multiplication, Monte Carlo simulation, 1-D correlation, Sobel edge detection, and motion estimation, showing performance improvement up to 170 times compared to the initial designs (Section V).

II. Related Work

The MapReduce programming model, named after the *Map* and *Reduce* functions in Lisp and other functional languages, has been developed and used in Google [3]. The Haskell

functional language and Google's Sawzall are used to describe the *Map* and *Reduce* functions. Yeung *et al.* [4] 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. In [4], source codes with the *Map* and *Reduce* functions have been manually transformed into Handel-C descriptions, which are then synthesized and mapped onto hardware. Our proposed framework combining pattern-based and geometric programming (GP) based transforms can automatically identify MapReduce and extract parallelism.

In this work, we use loop strip-mining and pipelining to extract parallelism in MapReduce patterns. Loop-level parallelization has been widely used for improving performance [6]. Loop transformations, such as loop merging, permutation and strip-mining, are used to reveal parallelism. A GP model [7] is used to determine the tile size of multiple loops to improve data locality in a hierarchical memory system. Eckhardt *et al*. [8] recursively apply locally sequential, globally parallel and locally parallel, globally sequential schemes to map an algorithm onto a processor array with a two-level memory hierarchy. Loop pipelining is applied to pipeline the innermost loop [9] and outer loops [10]. An integer linear programming model is proposed [11] 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*. [12] 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 [5] 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 [13], but such systems do not incorporate hardware-specific knowledge into the transforms.

We have previously combined model-based data reuse and loop parallelization and pattern-based transforms into a single framework [2]. This paper extends our previous work to cover the MapReduce pattern.

III. Problem Statement

This paper considers how to map an obvious but inefficient description, with a possibly implicit MapReduce pattern, onto highly parallel hardware. Our proposed framework applies model-based and pattern-based transforms to automatically identify and extract parallelism, using loop strip-mining and pipelining, from sequential loops. When a loop is strip-mined for parallelism, the loop that originally executes sequentially is divided into two loops, a new *Do* loop running sequentially and a *Doall* 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. Fig. 1 shows how we use model-based and pattern-based transforms to achieve design goals.

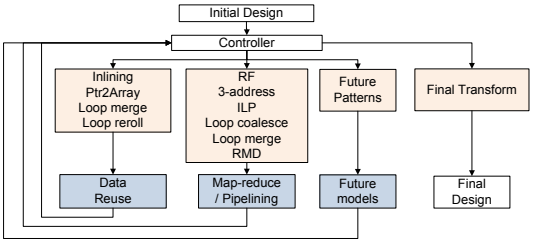

Fig. 1. Combining model-based (such as data reuse / parallelisation and MapReduce) and pattern-based approaches (such as loop coalescing and unrolling) into a single approach. Each model-based approach has an associated pattern-based approach. After multiple model-based transforms have been applied, a final pattern-based transform produces the output hardware. Note: Ptr2Array: convert pointer to array, RF: reduce fanout, ILP: Instruction-Level Parallelism, RMD: remove multiplications and divisions.

To illustrate this problem, Fig. 2 (a) 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. 2 (b), the original complex expression is decomposed into simple operations: two memory accesses, one multiplication and the result accumulation. This can reduce the number of logic levels of the generated circuits, improving system latency [2]. Next, we generate a data flow graph (DFG) for the code, as shown in Fig. 2 (c), 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 accumulation operation is addition which is associative and commutative.

We thus map the dot product onto a parallel computing structure. Given a hardware platform with sufficient amount of multipliers and each assignment takes one clock cycle, Table I shows some design options for 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 *Do* loop. Ideally, if the memory bandwidth is $2N$ bytes per execution cycle as shown in the second row of Table I, then N multiplications can execute in parallel and the final result can be merged using a tree structure, as shown in Fig. 2 (e) where pipeline registers are not shown. In this case, the loop i is fully strip-mined and the dot product needs $log_2N + 2$ execution cycles: log_2N 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

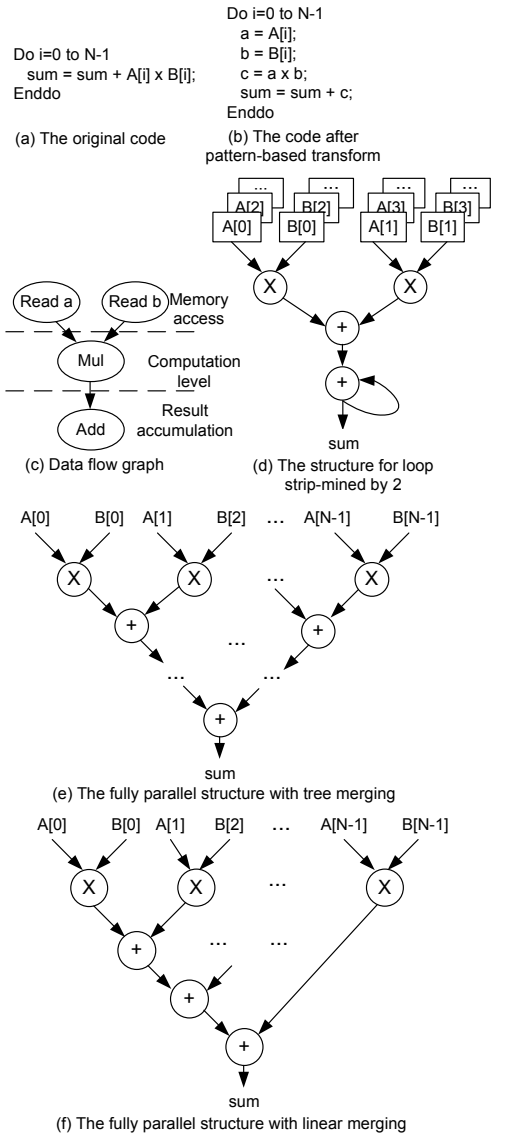

Fig. 2. Motivating example: dot product

byte per execution cycle, then the fully parallel implementation in Fig. 2 (e) 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 I. 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. 2 (d). This structure can be further pipelined with $ii = 2$. This design, combining local parallelization and global pipelining,

TABLE I
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
1	$(N, 1)$	$2N + log_2 N + 1$	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

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 I 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 a framework with model-based and pattern-based transforms to deal with the problem of mapping the MapReduce pattern. The exploration of the design space of mapping the MapReduce pattern onto a hardware platform is formulated in a geometric programming (GP) model. This model allows a quick and automatic determination of the performance-optimal designs in the design space, under different memory bandwidth and computation resources constraints. The design is transformed from the initial sequential descriptions, and the resulting design is performance-optimal in the MapReduce design space and may be further improved when other optimisations are applied. Our pattern-based approach will be used to identify the pattern and transform it to the form required by the GP model. Therefore, the proposed framework tackles several variations of the MapReduce pattern:

- implicit MapReduce pattern, such as the motion estimation algorithm [14] used in X264;
- MapReduce pattern at any loop level, such as the Monte Carlo simulation of Asian Options [15]; and
- 2-D MapReduce patterns, such as edge detection [16].

IV. GP MODEL FOR MAPPING MAPREDUCE

As the MapReduce pattern is often present in sequential loops, we target a loop containing off-chip memory accesses, computations and result accumulation, with the number of iterations fixed at compile time. The design variables for mapping the MapReduce pattern loop onto a parallel computing structure include the number of parallel partitions of the loop and the pipeline initiation interval; the loop is mapped to

TABLE II
NOTATION USED IN THE MODEL.

Notations	Description
k	# parallel partitions of a loop
ii	initiation interval of pipeline
v	# iterations in a loop partition after strip-mining
x_f	# resource f being used
d_i	# latency in execution cycles of computation level i of DFG
N	# loop iterations of a loop
$RecII$	data dependence constraint on ii in the computation
W_f	# computation resource f required in a loop iteration
R_{if}	# computation resource f required in computation level i of DFG
I	# computation levels of DFG
B	the required memory bandwidth in one loop iteration
M_b	the memory bandwidth
C_f	# computation resource f available
F	F types of computation resources involved
$notAlign$	0: data are aligned; 1: data are not aligned
$notFull$	0: loop is fully strip-mined; 1: loop is partially strip-mined

a locally parallel, globally pipelined design. In fact, there is no difference in execution time between locally parallel, globally pipelined and globally parallel, locally pipelined. The DFG of a loop nest divides into three sections: memory access, computation operations and result accumulation, as in Fig. 2 (c). The operations from different loop iterations execute in multiple parallel processing units after loop strip-mining; the three sections are pipelined. Without loss of generality, in line with Handel-C, we assume that each assignment in the loops takes one clock cycle.

We formulate the design exploration problem as a piecewise integer geometric program. Geometric programming (GP) [17] is the following optimization problem:

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

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.

A. Piecewise GP Model for MapReduce with Tree Reduce

Table II lists the notation used in the following model, where capital letters are compile-time constant model parameters and the others are integer variables. Using this notation, the mapping of a MapReduce pattern onto a parallel computing structure is formulated in problem (1)–(9).

Expression (1) is the objective function, the number of execution cycles of a loop after loop strip-mining and pipelining. It is the standard modulo scheduling length of pipelining [9], [10]. We assume that the number of execution cycles for loading data from memory dominates the initiation interval (ii). In this way, the negative one originally in the formulation of the execution time of standard pipelining is removed from the first term of expression (1), meeting our GP model requirement. Section V shows that this assumption is true for the tested benchmarks. Inequality (6) defines the number of loop iterations v executing in each parallel partition.

$$\text{minimize } v \times ii + \sum_{i=1}^{I} d_i + \lceil \log_2 k \rceil + notFull \qquad (1)$$

subject to

$$B \times k \times M_b^{-1} \times ii^{-1} + notAlign \times ii^{-1} \le 1 \qquad (2)$$

$$W_f \times x_f^{-1} \times ii^{-1} \le 1, f \in F \qquad (3)$$

$$RecII \times ii^{-1} \le 1 \qquad (4)$$

$$k \times x_f \le C_f, f \in F \qquad (5)$$

$$N \times k^{-1} \times v^{-1} \le 1 \qquad (6)$$

$$R_{if} \times x_f^{-1} \times d_i^{-1} \le 1, 1 \le i \le I, f \in F \qquad (7)$$

$$1 \le k \le N \qquad (8)$$

$$1 \le x_f \le C_f, f \in F \qquad (9)$$

The second term in the objective function (1) is the execution cycles taken by computation. There may be I ($I \ge 1$) 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 resource f in level i and the number of allocated resource x_f, as defined in inequalities (7). The example in Fig. 2 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 1 or 2 depending on the resource constraint, leading to $d_1 = 2$ or $d_1 = 1$ cycles.

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. 2 (d) and (e), the number of intermediate results is reduced by the power of two. The boolean parameter $notFull$ is added here, because for the case where the loop is partially strip-mined an accumulation of the results generated sequentially is needed; Fig. 2 (d) shows an example.

Inequality (2) is the memory bandwidth constraint. Given a memory bandwidth M_b, the more parallel partitions k require more data loading time, leading to 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.

Inequalities (3) and (4) are respectively the computation resource constraints and the data dependence constraint on ii. Inequalities (5) are the computation resource constraints on the parallelization. The F types of computation resources are allocated between loop parallelization (k) and pipelining (ii) to achieve an efficient solution. Inequalities (8) and (9) are the ranges of integer variables k and x_f, respectively.

Overall, we can see that the only item that makes the relaxed problem (1)–(9) (allowing all variables to be real number) not a GP problem is the logarithm in the objective function (1). However, it can be noted that $\lceil \log_2 k \rceil$ is constant in certain ranges of k. Therefore, the problem (1)–(9) can be seen as

a piecewise integer geometric problem in different ranges of k; the number of subproblems increases logarithmically with k. For large $k = 1000$, there are 11 integer GP problems and each problem can be quickly solved using a branch and bound algorithm used in [18] with a GP solver as the lower bounding procedure. Section V shows the performance of the piecewise GP model.

The formulation (1)–(9) targets mapping a loop onto a parallel computing structure. The loop needs 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)–(9). 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 V exhibits this case. Inequality (10) 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 (11) and (12) 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.

$$(N' - 1) \times ii' \times ii^{-1} + ii^{-1} \times \sum_{i=1}^{I'} d'_i \leq 1 \qquad (10)$$

$$W'_f \times x_f \times ii'^{-1} \leq 1, f \in F \qquad (11)$$

$$RecII' \times ii'^{-1} \leq 1 \qquad (12)$$

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 (2) to cover 2-D data accesses, allowing for the 2-D data to not necessarily be contiguous:

$$\#row \times (B \times k \times M_b^{-1} \times ii^{-1} + notAlign \times ii^{-1}) \leq 1 \qquad (13)$$

where $\#row$ is the number of rows of a 2-D data block.

B. 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 replace the tree structure; Fig. 2 (f) shows an example for dot product. For k parallel computations, the linear structure needs k execution cycles to accumulate all intermediate computation results. Although longer than the tree structure, we observe that for a set of applications containing the MapReduce pattern, using the linear structure to merge results can both keep the same system throughput as the tree structure and reduce system area.

Fig. 3 (a) shows an example of 1-D correlation. The innermost loop j of the code has the 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. 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. 3 (b). An important characteristic of this structure is the regular shape, which could benefit the place and routing of hardware resources. After mapping loop j, fully pipelining loop i will produce 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.

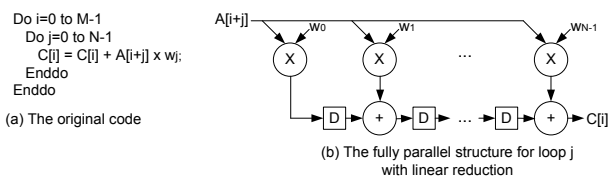

Fig. 3. An example: 1-D correlation.

TABLE III
THE DETAILS OF FIVE BENCHMARKS

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

Many signal and image processing applications use 1-D correlation or convolution or show similar behavior, so we extend the formulation in Section IV-A for linear reduction. In the objective function (14), we replace $\lceil \log_2 k \rceil$ with k. 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. Also, as every element of A is multiplied with the N coefficients w_j in order to linearly reduce, the data dependence distance of the computations in loop j decreases as the number of parallel partitions increases. Therefore, we add inequality (15).

$$\text{minimize } v \times ii + \sum_{i=1}^{I} d_i + k + notFull \qquad (14)$$

$$N \times k^{-1} \times ii^{-1} \leq 1 \qquad (15)$$

Now, the formulation with objective (14) and constraints (2)–(15) is a GP model, mapping MapReduce patterns to a parallel computing platform with a linear reduction.

V. Experimental Results

We apply the framework to five kernels: multiplication of two 64×64 matrices (MAT64), the motion estimation (ME) algorithm [14] used in X264, the Sobel edge detection algorithm (Sobel) [16], Monte Carlo simulation (MCS) of Asian Option Pricing [15] and 1-D correlation; Table III shows benchmark

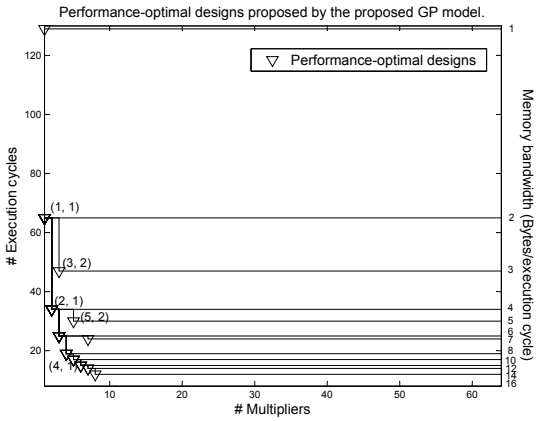

Fig. 4. Design Pareto frontiers proposed by the framework for MAT64. The bracket next to each design shows (k, ii).

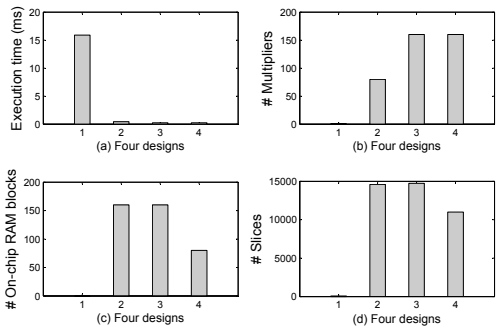

Fig. 5. Implementation results of some designs of MAT64.

parameters. Each benchmark contains the MapReduce pattern; two levels of MapReduce pattern are identified in ME, MAT64 and Sobel. We apply the transforms [2] to the outer level exploiting data reuse and loop-level parallelism. For the inner level MapReduce pattern in the inner loops, we apply inner loop parallelization and pipelining, constrained by memory bandwidth and computation resources. Different performance-optimal designs under different constraints are represented by the number of parallel loop partitions and the pipelining initiation interval, (k, ii).

In our experiments, the target platform is an FPGA-based system with off-chip SRAM. Without loss of generality, we assume the off-chip SRAM to be accessed by a single port with two cycle latency. All array references with input data are stored in off-chip SRAMs. The Monte Carlo simulation of Asian Option involving floating point arithmetic is implemented in Xilinx XC4VFX140 with 192 DSP48, and the other four kernels are implemented in XC2v8000, which has 168 embedded hard multipliers and 168 dual-port RAM blocks. For ME, MAT64 and Sobel, the on-chip RAM is configured as a scratch-pad to buffer reused data, and thus we can feed the parallel computations after mapping the inner loop MapReduce pattern. The on-chip RAM in the XC2v8000 can be configured in 9, 18, 36 bits. For ME and Sobel the frame size is the QCIF luminance component (144×176 pixels).

a) 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. Fig. 4 shows results from applying our proposed piecewise GP model (1)–(9) to the innermost loop of the MAT64, with the performance Pareto frontiers under different memory bandwidths and different numbers of hard multipliers. Given a memory bandwidth and the number of multipliers, the figure reveals the performance-optimal design: for example, when the memory bandwidth is 3 bytes/execution cycle and 3 multiplier is available, the performance-optimal design is ($k = 3$, $ii = 2$). 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 in the target platform. Fig. 5 shows the real execution time and the utilization of on-chip multipliers, RAMs and slices of four designs. The first is the original design without any optimisation. The second is optimised by the transforms [2], 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 in the target platform. The third design in Fig. 5 shows the result after mapping the two MapReduce pattern levels. The system performance speeds up about 66 times compared to the original design and speeds up about twice compared to the second design which only maps the outer MapReduce patter 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 [2] to the outer loops. For the inner level, the design corresponding to 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 [2]. The optimised design after mapping the MapReduce patterns in this order is the fourth design in Fig. 5. Fig. 5 (a) shows that this design has the performance almost equal to the third design that is 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. 5 (c) and (d).

b) The fast motion estimation algorithm: [14] used in X264 is highly computation-intensive. To determine the motion vectors for each macro block of the frame under encoding, the algorithm performs multiple searches on the reference frames. During a search the current macro block under encoding 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. Therefore, there are also two MapReduce pattern levels: matching the current block with several candidate blocks at the same time with a min function to merge results and the computation of SAD. 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. 6 (b). Our proposed pattern-based transforms perform function inlining to reveal this MapReduce pattern, and the model-based transforms are applied for data reuse [2] as the block matching process reuses much data. The proposed framework in this paper is applied to SAD computation. Results are shown in Fig. 6 (a). Memory bandwidth is the main constraint, given that logic for addition and comparison operations is adequate in the target platform. The performance-optimal designs proposed by the piecewise GP model (1)–(9) under a range of memory bandwidth are shown in downward-pointing triangles and are connected to form the Pareto frontier. Some other possible designs, which are automatically discarded by the GP model, with different values in (k, ii) are shown in stars all above the Pareto frontier.

We have implemented a search path of the motion estimation algorithm and the sequential searches can use the same circuit. The original design, three designs (1, 1), (16, 9) and (32, 5) proposed by the framework and four non-optimal designs as shown in Fig. 6 (a), are implemented in the target platform to obtain the real execution time and memory bandwidth. Comparing the results in Fig. 6 (b) to (a) shows that the designs proposed by the GP framework are performance optimal under the corresponding memory bandwidth constraints. The designs after mapping the two MapReduce patterns improve the performance of the search path of ME up to 170 times compared with the original design and up to 2 times when compared with the design (the down-point triangle with memory bandwidth 2 Bytes/execution cycles in Fig. 6 (b)) that only maps the outer level MapReduce pattern.

c) Monte Carlo simulation: has wide applications in finance. Asian Option prices depend on the average price observed over a number of discrete dates in the future; the mathematical model has no closed solution [15]. Therefore, MCS is used to estimate the 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 IV-A. The extended piecewise GP model applies to the outer loop. As this application uses

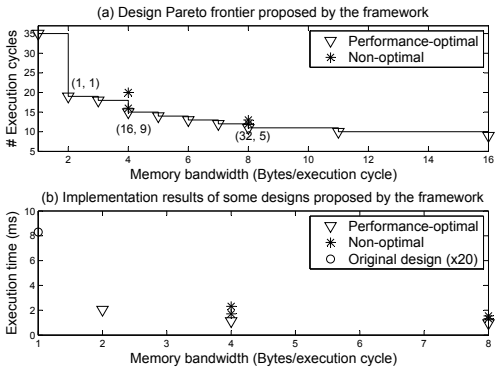

Fig. 6. Experimental results of ME. (a) Design Pareto frontier; (b) Implementation results of some designs on the FPGA. The bracket next to each design shows (k, ii).

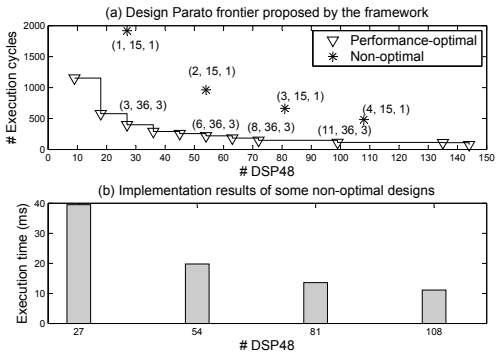

Fig. 7. Experimental results of MCS of asian option. (a) Design Pareto frontier. (b) Implementation results of some designs on the FPGA.

floating point, the number of DSP blocks that execute the floating point operations is the constraint in the target platform. Fig. 7 (a) shows the performance Pareto frontier. Again, as the number of DSPs increases, more iterations of the outer loop can execute in parallel and the execution cycles decrease.

We have implemented the MCS of Asian Option Pricing using HyperStream library [19] in 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 implement four non-optimal designs as shown in stars in Fig. 7 (a). The results in Fig. 7 (b) 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.

d) 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

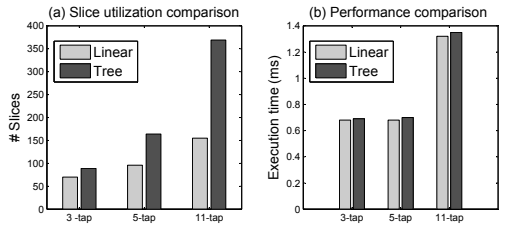

Fig. 8. Comparison between the linear and the tree structure. (a) Resource utilization comparison. (b) Performance comparison.

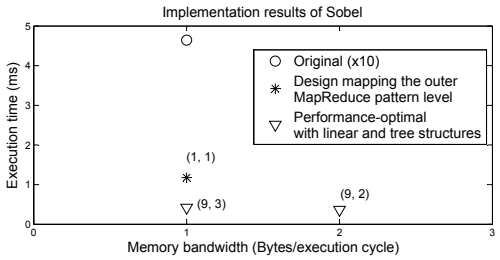

Fig. 9. Experimental results of Sobel designs proposed by the framework.

FPGA platform with both the linear and the tree accumulating structures; Fig. 8 shows results. It can be seen that in all three cases, tree reduction costs more slices, up to 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. 8 (a). This is because for correlation operations, the MapReduce pattern with linear reduction does not require input shift registers and has regular circuits that are easier to place and route. Also, the implementations with linear reduction show a small performance increase, as shown in Fig. 8 (b). This is because for this benchmark the designs with the linear and tree structures have the same latency, but the tree case could have more logic levels and thus more pipeline stages.

e) Sobel edge detection algorithm: [16] 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 [2] 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 IV-B to map the innermost loop with linear reduction, and then apply the piecewise GP model in Section IV-A 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 only mapping the outer MapReduce pattern level, and the designs that map the two MapReduce pattern levels are implemented in the target platform, together with the original design. Fig. 9 shows that when memory bandwidth is 2 Bytes/execution cycle, the design with the hybrid tree and linear reduction improves the performance about 3 times compared to the design only mapping the outer level MapReduce pattern, and shows 125 times speedup compared to the original design.

The above experiments show that the proposed GP framework can explore the design space with MapReduce and pipelining effectively, and can correctly determine the performance-optimal designs under different constraints in the design space. Moreover, the proposed framework identifies two result accumulating structures and adapts the GP model to different applications. The execution time of the piecewise GP model increases linearly as the number of pieces increases. On average, for the five benchmarks, an optimal design is generated by the piecewise GP framework within 15 seconds.

VI. Conclusion

We present optimisations for compiling designs in a sequential language with the MapReduce pattern onto parallel hardware. Our approach, based on geometric programming, achieves up to 170 times speedup compared to unoptimised designs. Future work includes dealing with other patterns for parallel applications in addition to MapReduce, and automating the selection of models and patterns at each step in the optimisation. These optimisations may further improve the system performance.

Acknowledgement. We thank FP6 hArtes project, the EPSRC, AlphaData and Xilinx for their support.

References

[1] K. Asanovic et al., "The Landscape of Parallel Computing Research: A View from Berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006.
[2] Q. Liu et al., "Optimising designs by combining model-based and pattern-based transformations," in *Proc. FPL*, 2009.
[3] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *6th Symp. on OSDI*, December 2004, pp. 137–150.
[4] J.H. Yeung et al., "Map-reduce as a programming model for custom computing machines," in *Proc. FCCM*, 2008, pp. 149–159.
[5] M. W. Hall et al., "Maximizing multiprocessor performance with the SUIF compiler," *IEEE Computer*, December 1996.
[6] U. K. Banerjee, *Loop Parallelization*. Kluwer Academic, 1994.
[7] L. Renganarayana and S. Rajopadhye, "A geometric programming framework for optimal multi-level tiling," in *Proc. Conf. Supercomputing*. IEEE Computer Society, 2004, p. 18.
[8] U. Eckhardt and R. Merker, "Hierarchical algorithm partitioning at system level for an improved utilization of memory structures," *IEEE Trans. CAD*, vol. 18, pp. 14–24, Jan. 1999.
[9] V. Allan, R. Jones, R. Lee, and S. Allan, "Software pipelining," in *ACM Computing Surveys*, vol. 27, no. 3, 1995, pp. 367–432.
[10] H. Rong et al., "Single-dimension software pipelining for multi-dimensional loops," in *IEEE Proc. on CGO*, 2004, pp. 163–174.
[11] K. Turkington et al, "Outer loop pipelining for application specific datapaths in FPGAs," *IEEE Trans. VLSI.*, vol. 16:10, pp. 1268–1280, 2008.
[12] B. di Martino et al., "A technique for FPGA synthesis driven by automatic source code synthesis and transformations," *Proc. FPL*, 2002.
[13] "The TXL programming language," http://www.txl.ca/.
[14] L. Merritt and R. Vanam, "Improved rate control and motion estimation for h.264 encoder," in *Proc. ICIP*, 2007, pp. 309–312.
[15] http://www.interactivesupercomputing.com/success/pdf/caseStudy_financialmodeling.pdf, "Financial modeling–monte carlo analysis," accessed 2009.
[16] http://www.pages.drexel.edu/∼weg22/edge.html, accessed 2006.
[17] S. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge University Press, 2004.
[18] J. Löfberg, "YALMIP : A toolbox for modeling and optimization in MATLAB," in *Proc. CACSD*, 2004.
[19] G. W. Morris and M. Aubury, "Design space exploration of the european option benchmark using hyperstreams," in *Proc. FPL*, 2007, pp. 5–10.