This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

Optimizing Hardware Design by Composing Utility-Directed Transformations

Qiang Liu, Tim Todman, Wayne Luk and George A. Constantinides

Abstract—*Utility-directed* transformations involve changing a design to optimize for given constraints while preserving behavior. These changes are often achieved by techniques such as linear programming or geometric programming. We present a systematic approach composing multiple utility-directed transformations for optimizing and mapping a sequential design onto a customizable parallel computing platform such as a Field-Programmable Gate Array (FPGA). Our aim is to enable automatic design optimization at compile time. Design goals specified by users drive the design transformations. Each utility-directed transformation achieves part of the overall goal, and multiple utility-directed transformations, connected by pattern-directed transformations, are composed to fulfil the overall design requirements. The utility-directed transformations in this work produce performance-optimized designs by exploiting data reuse, MapReduce and pipelining for the target parallel computing platform. Moreover, it is shown that performing transformations in different orders allows users to trade speed for resources, and design performance for compile time. Several applications are used to evaluate this approach on FPGAs. The system performance of a 64-bit matrix multiplication is shown to improve up to 98 times compared to the original design, in the target hardware platform.

Index Terms—Design optimization, data reuse, MapReduce, pipelining, geometric programming.

1 Introduction

Hardware designers increasingly use high-level descriptions, such as C, to ease design description, development and simulation, and to enable fast design space exploration. To meet design goals, designers must apply multiple optimizations to their design, making it more efficient but without affecting its intended functionality. This makes it difficult for designers to quickly optimize designs.

Compilation techniques have been developed to automate optimization by using design transformations. Like the classification of *adaptation policies* in automatic computing [1], most of the transformations can be divided into two types: *pattern-directed* transformations (PDTs) [2], [3], [4], [5] and *utility-directed* transformations (UDTs) [6], [7], [8], [9]. Similar to *action policies* [1], a PDT defines explicitly the design to be transformed, the transformed design, and the conditions that trigger the transformation. What is also needed is a mechanism for designers to specify the desirable properties of the transformed design and algorithms that can find such a design, rather than the transformed design itself which may not yet be identified. UDTs *model design goals that capture desirable properties of the transformed design using utility functions, and characterize design spaces using constraint functions*. Similar to adaption in *utility function policies* [1], the transformation in a UDT is selected to achieve the minimum/maximum value of the utility function in certain design contexts including algorithm and hardware characteristics. As a result, the UDTs provide designers with a convenient means of describing complex optimizations.

PDTs have been widely used in existing compilation tools as described in Section 2, while UDTs have been less frequently used in design optimizations [6], [7], [8], [9]. This is because there are challenges in using UDTs. First, UDTs require modeling techniques to capture design contexts and the impacts of optimization techniques on designs, *i.e.* the capability of evaluating different transformation options. The complexity of the models should also be taken into account. Second, UDTs usually impose particular requirements on input designs, and thus need support from front-end and back-end tools or manual direction to complete the optimization process.

In this paper, we propose a new approach for design optimization. The proposed approach provides multiple UDTs with different optimization objectives, or with the same objectives but using different optimization techniques. PDTs are used to preprocess designs, transforming them to the required input forms for UDTs. In this way, multiple UDTs can be composed to enable automatic and powerful design optimizations. This approach allows users to work at more abstract design levels, where design goals are described and, where necessary, desired transformations and transformation orders can be specified. The overall design goal is then achieved by a sequence of transformations, each fulfilling a particular subgoal. For instance, if one wants to minimize the execution time of a design, then one could first choose loop pipelining, then reusing data and finally parallelizing loops.

The contributions of this paper are:

- a systematic approach composing utility-directed transformations for automatic design optimization (Section 3);
- utility-directed optimizations based on geometric programming (GP) models, concurrently exploiting optimization techniques: data reuse, MapReduce, and pipelining (Section 4); and
- an evaluation of our approach based on three benchmarks

Q. Liu, T. Todman and W. Luk are with the Department of Computing, Imperial College, London SW7 2BT, U.K. e-mail: {qiang.liu2, timothy.todman, w.luk}@imperial.ac.uk.
G. A. Constantinides is with the Department of Electrical and Electronic Engineering, Imperial College, London SW7 2BT, U.K. e-mail: g.constantinides@imperial.ac.uk.

Digital Object Indentifier 10.1109/TC.2011.205 0018-9340/11/$26.00 © 2011 IEEE

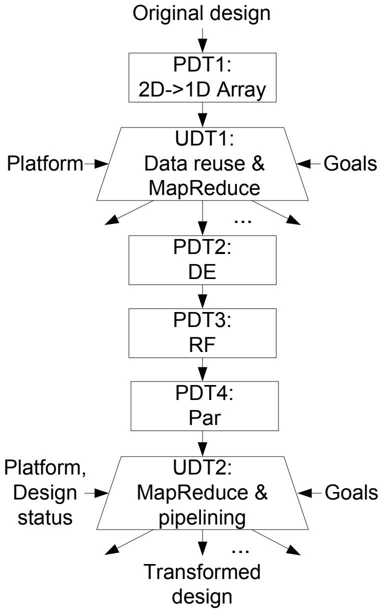


Fig. 1. A sequence of transformations applied to 2-D matrix multiplication: four PDTs and two UDTs.

from digital signal processing, image processing, and scientific computing (Section 5).

In this paper, we target designs captured in sequential languages like C, and we transform and map them automatically onto a parallel computing structure. The dependence between transformations is investigated to determine the order in which to apply them. The output is a transformed C-like hardware description, preserving input design functionality, with explicit timing and parallelism descriptions. A meta language CML [10] is used to specify PDTs. Three UDTs are presented in this paper for optimizing design speed. Based on geometric programming, these UDTs explore the design space of using data reuse, MapReduce and pipelining for design optimization.

Fig. 1 shows an example sequence of PDTs and UDTs applied to matrix-matrix multiplication. The original design is transformed by four PDTs and two UDTs, and the output is an optimized design. The optimization techniques are explained in the rest of the paper. Each UDT has multiple transformation paths, and only one path is chosen according to design goals and target platform specified by users. Note that the second UDT also considers the design status after previous transformations. This example is used throughout the paper to explain how the proposed approach works.

The rest of the paper is organized as follows: Section 2 describes related work and tools. Section 3 presents our optimization methodology composing UDTs. Section 4 details these transformations. Section 5 presents experimental results from applying our approach to several real applications. Section 6 concludes with plans for future work.

2 Background

Pattern-directed transformations (PDTs) match and transform syntax or dataflow patterns of input programs. Each user-defined pattern occurring in a program is transformed to a user-specified form if Boolean conditions hold. Fig. 2 shows a PDT `decompose` (DE), described in CML, with an empty condition. This transformation can reduce logic levels of the generated circuit, improving system clock frequency.

```
Transform decompose
{
        pattern { Var = Var + Exp1 * Exp2 }
        conditions {  }
        result { NewVar = Exp1 * Exp2;
                 Var = Var + NewVar; }
}
```

Fig. 2. Example of pattern-directed transformation: decomposing (DE) an expression with two arithmetic operators into two expressions each with one operator.

PDTs for hardware compilation have been explored by researchers such as di Martino *et al.* [3] on data-parallel loops written in C source code, as part of a synthesis method from C to hardware. Compiler toolkits such as SUIF [4] and CoSy [5] allow multiple syntax patterns to be used together. However, these approaches give no support for including utility-directed transformations. Syntax pattern matching and transforming can also be done in tree rewriting systems such as TXL [11], but such general systems make it hard to incorporate hardware-specific knowledge into the transformations.

In our approach, PDTs are written in a domain-specific language called CML [10], based on CTT [12] and compiled into a C++ description; the resulting program then runs a source-to-source transformation. PDTs could be written once by domain or hardware experts, then used many times by non-experts. We identify several kinds of transformation: input (transforming code into a suitable form for a UDT), tool-specific and hardware-specific (optimizations for particular synthesis tools or hardware platforms). We provide a library of useful transformations: general-purpose ones such as loop restructurings, and special-purpose ones such as transforming Handel-C arrays to RAMs.

In contrast, *utility-directed* transformations (UDTs) do not explicitly specify how to transform programs. Instead, they involve optimization problems; user-specified design goals and specifications form the objectives and constraints of the optimization problems. The solutions to the optimization problems determine how transformations are carried out. For example, one may want a power-efficient design that meets specific $Speed$ and $Area$ requirements. This design could be generated by a utility-directed transformation according to design parameters $\vec{x}$ determined by the following optimization problem:

$$\begin{aligned} \text{minimize } & \mathbb{P}(\vec{x}) \\ \text{subject to } & \mathbb{S}(\vec{x}) \geq Speed \\ & \mathbb{A}(\vec{x}) \leq Area \end{aligned}$$

where $\mathbb{P}(\vec{x}), \mathbb{S}(\vec{x})$ and $\mathbb{A}(\vec{x})$ are the system power, speed and area models, respectively.

Programs with composite objects such as arrays and iteration statements such as loops usually need UDTs for more efficient designs. For example, a geometric programming (GP) model [6] determines loop tile size for multiple loops to improve data locality in a hierarchical memory system. Liu *et al.* [7] propose a GP framework to automate exploration of the data reuse and loop-level parallelization design space in the context of FPGA targeted hardware compilation. An integer linear programming model is proposed in [8] for pipelining outer loops in FPGA hardware coprocessors. Lam *et al.* [9] use a tabu search approach to determine loop unrolling factors. System speedup over a single CPU implementation is considered as the utility function. In [13], a model of affine recurrent equations represents a biological application, for acceleration on FPGAs. All these approaches need support from front- and back-end tools or manual transformations to complete the optimizations.

Table 1 lists several commercial hardware compilers, each targeting different C-like languages. Optimizations used in these tools are mostly pattern-directed and specific programming rules are added upon the traditional C. Rewriting a design from C/C++ to their target inputs, *e.g.* Impulse C or Handel-C, is not trivial. Our approach complements these tools by automating code transformation and, more importantly, automating complex design optimizations.

High-level synthesis frameworks, SPARK [14], ROCCC [15], DEFACTO [16], Hyper-LP [17], and LegUp [18], perform optimization transformations, such as code motion, loop transformation, dynamic renaming, pipelining, retiming, scalar replacement, data reuse, operation chaining, and area-saving binding, to optimize hardware circuit performance. In these frameworks heuristic and probabilistic optimization algorithms are used to guide the transformations. However, these approaches do not allow users to specify transformations, do not exploit large scale parallelism, and the optimization algorithms only give local suboptimal solutions.

In this paper, we combine UDTs to allow the automation of sophisticated design optimizations. The proposed approach is instantiated in this paper as follows. We generalize the transformations used in [28], [29] into PDTs and UDTs, so that more transformations can be integrated. We extend the UDT [7] which optimizes data locality and data-level parallelism to considering practical design constraints, and we combine it with UDT [29], which optimizes data-level and instruction-level parallelism, using PDTs. This instantiation can automatically exploit multiple techniques to optimize designs. Among them, data reuse [30], [31], [32], pipelining [8], [33] and MapReduce [34], [35] are particularly powerful.

Each of these three design optimization techniques requires design space exploration to determine the optimal designs for a given target hardware platform. Moreover, these techniques are related [7], [29]. We show in the result section that combining these techniques in utility-directed transformations can find more efficient designs and allows designers to trade off design speed and area, automating design space exploration.

3 Systematic Approach

Our systematic approach starts with a sequential but possibly inefficient design, and applies multiple transformations, driven by user design goals and specifications, to achieve a more efficient design. User design goals could involve maximizing or minimizing a system metric (*e.g.* minimizing execution time) or a specific target for a system metric (*e.g.* execution time less than 1 second). User specifications could include the target hardware platform (*e.g.* Xilinx XC5VSX240 FPGA), chosen transformations, the execution order of transformations, and so on.

Our approach provides multiple UDTs that users can choose; different UDTs perform different optimization objectives. A UDT involves an optimization problem (OP) in the following form and can be used as a library or an executable.

```
UDT:Name{Objective, ExecutionOrder,
         Requirements, Related PDT,
         InputProgram,
         OP(Input: Program parameters,
                   Platform parameters,
                   Design status;
            Output: Design parameters)}
```

ExecutionOrder is a whole number, indicating the execution order of a UDT. A UDT with ExecutionOrder=0 does not execute, and a UDT with ExecutionOrder=a executes prior to a UDT with ExecutionOrder=b iff $a < b$. Requirements describes the required input form for a UDT, *e.g.* loop affinity. Related PDT specifies those PDTs which help transforming input programs for the associated UDT.

In our current approach, there is a default order of executing a set of transformations, depending on the design goals. An example is shown in Fig. 1, where the design goal is to optimize speed and the order of applying the transformations follows hardware optimization principles. The effects of these transformations on matrix multiplication are shown in Section 5. Users are allowed to experiment with different transformation orders to choose the best. We are developing methods (such as [36]) which enable automatic exploration of the transformation ordering, so that users have the options to manually tune the order in applying transformations or rely on automatic methods.

Our approach meets user design goals by combining multiple optimization stages. Different stages may achieve different goals. For example, one stage optimizes speed and another optimizes power consumption. At each stage, an appropriate UDT is chosen by default (if users do not specify it), in terms of the characteristics of the input code and the design goal. If the input code is not in a suitable form, PDTs are used to transform the code into a form such that an appropriate UDT can be applied. The input program parameters (such as the number of loop levels), specified platform parameters (such as the number of available DSP blocks), and the design status after prior UDTs (such as DSP block utilization) instantiate the optimization problem involved in the UDT. The solution to the optimization problem tells us

TABLE 1
Some commercial hardware compilers.

Tools	Source	Target	Optimization techniques
AccelDSP Synthesis [19]	MATLAB	RTL	automatic conversion of floating-point to fixed point, user specified loop unrolling, user specified pipestage operation, user specified RAM/ROM mapping
Catapult C Synthesis [20]	C++	RTL	loop unrolling, pipelining and merging, memory mapping and allocation
PICO Express [21]	C	SystemC, RTL,	exploitation of parallelism at multiple levels, pipeline architecture of processing array
CoDeveloper [22]	Impulse C Streams-C	HDL	software/hardware co-design support, loop unrolling and pipelining
SC Compiler [23]	SystemC	EDIF, RTL	automatic tree balancing, logic sharing, re-timing and rewriting, variable width reduction
Cynthesizer [24]	SystemC	RTL	datapath optimization: pipeline, loop unrolling, *etc.*
DK Design Suite [25]	Handel-C	EDIF, RTL	re-timing, tree balancing, ALU mapping, memory pipelining transformations
C2H [26]	C	HDL	loop pipelining, memory access pipelining, parallel scheduling, arithmetic resource sharing
AutoESL [27]	C/C++/SystemC	RTL	loop unrolling and pipelining, resource binding, function inlining, user specified array mapping

whether a more efficient design exists. If it does, then the input code is transformed according to the output `design parameters` determined by the optimization problem, and the design status after the transformation (such as system speed and resource utilization) is logged.

The transformed design status provides information, such as the number of execution cycles and on-chip memory resource utilization, based on mathematical models used in utility functions which capture system behavior in a cycle accurate way and represent on-chip embedded resource utilization accurately. The effects of clock frequency are not explored in the transformations presented in this paper, since the overall execution time depends also on number of clock cycles. Some transformations such as pipelining would tend to improve clock frequency while preserving number of clock cycles, while parallelization can improve number of cycles and degrade clock frequency. However, we observe that the overall execution time is dominated by the number of execution cycles when multiple UDTs and PDTs are composed, as shown in Section 5. During design exploration we assume a fixed clock frequency (which can be known for a target hardware platform), to avoid repeatedly running the placement and routing toolchain.

After each stage, the transformed design is evaluated: if it meets design requirements specified by users, then the transforming procedure stops, otherwise further transformations follow. Finally, the chosen sequence of UDTs and PDTs is recorded. This sequence of transformation choices documents the design process, a) forming an audit trail to check design correctness after many transformations, and b) allowing reuse for other applications. If the design goals and specifications set by users are met after the sequence of transformations, then an optimized design is generated. Otherwise, users need to look at the log information, identify the reasons for failure and change the design goals and specifications accordingly.

The advantages of composing UDTs are:

- individual UDTs can be simpler, thus allowing fast and accurate solution to optimization problems;
- complex optimizations are supported;
- customization of transformation sequences allows optimizations to be adapted to applications, and trade-offs between system speed, power and area.

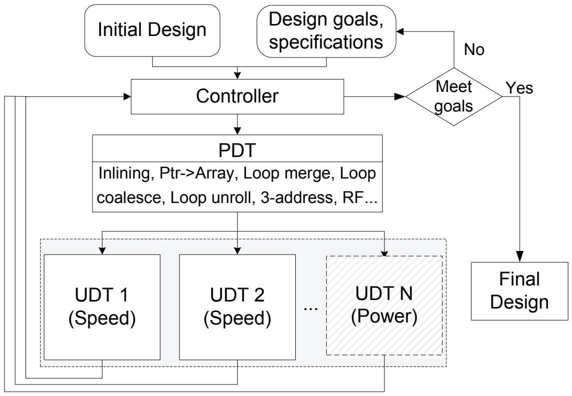

Initial Design
Design goals, specifications
No
Controller
Meet goals
Yes
PDT
Inlining, Ptr->Array, Loop merge, Loop coalesce, Loop unroll, 3-address, RF...
UDT 1 (Speed)
UDT 2 (Speed)
...
UDT N (Power)
Final Design

Fig. 3. Combining utility-directed transformations into a single approach. UDTs have a set of associated PDTs.

Fig. 3 shows the implementation of our approach. The input design is a sequential C description, and the final design output could be any behavioral hardware description; this paper uses Handel-C [25]. The transformation choice and design metric evaluation at each stage, as described above, are done in the controller [37]. PDTs perform common transformations, such as function inlining, multi-D array to 1-D array, pointer to array dereference, and subexpression elimination. These transformations reveal the data flow of the input code and ease extraction of input code parameters for UDTs. Moreover, some PDTs, as shown in Section 5, can optimize the logic circuit and exploit fine-grain parallelism of hardware.

In Fig. 3, there could be multiple UDTs, each addressing a particular design goal. In this paper, two UDTs are implemented for exploring combined data reuse and MapReduce transformations (`UDT1`) and combined MapReduce and pipelining transformations (`UDT2`), since these optimization techniques are inter-linked. The design goal is to optimize speed, and hardware resources in the user target platform constrain the transformations. `UDT1` optimizes data locality and loop-level parallelism for data-dominated applications, which have the MapReduce pattern and process large amounts of data stored in off-chip memory, with predictable memory access pattern. The controller directs input codes with these characteristics to this transformation. `UDT2` exploits loop-level parallelization and pipelining. These two transformations are based on geometric programming (GP) [38] models and are described in Section 4, where the two GP models are also combined to concurrently make decisions on both transformations.

The specification of transformation ordering can be described using script. Python code for the sequence of transformations in Fig. 1 applied to matrix multiplication is presented below. Specific meaning of some of parameters listed below is out of the scope of this paper; details can be found in [37].

```
1: InDesign=parse("matmult.c")
2: des1=2DArrayTo1DArray(InDesign)
3: Program_params=[("numLoop1", 3),
   ("loopBound1", [64, 64, 64]),
   ("loopParallelizability1", [1, 1, 1]),
   ("numArray1", 3), ...]
4: Platform_params=[("numRAMBlock", 168),
   ("blockSize", 16384), ("memBandwidth", 32),
   ("numMultiplier", 168)]
5: des2=UDT1(Speed, 1, "affineLoop", [], des1,
   OP(Program_params, Platform_params, [],
   Out_design_params))
6: des3=decomposeExpressions(des2)
7: des4=reduceFanout(des3)
8: des5=parallelize(des4)
9: Program_params=[("loopLevel", innermost),
   ("loopBound2", [64]),
   ("loopParallelizability2", [1]), ...]
10: Design_stat=[("numMultiplier", utilization),
    ("numRAMBlock", utilization), ...]
11: OutDesign=UDT2(Speed, 2, "affineLoop", [],
    des5, OP(Program_params, Platform_params,
    Design_stat, Out_design_params))
12: unparse(OutDesign, "out.c")
```

The following explains each line in the above code:

1) The input file is parsed into a variable `InDesign`.
2) PDT 2-D array to 1-D array is applied, producing `des1`.
3) Program parameters are placed into a variable `Program_params` as a list of key-value pairs. Each transformation is responsible for checking its parameters are correct. Note that values can be integers, booleans, strings or arrays of other values; we omit some of the parameters for space reasons.
4) Platform parameters are similarly placed in another variable.
5) UDT1 is applied to `des1`, yielding `des2`.
6) Three PDTs are applied to `des2`, yielding `des5`.
9) Parameters for UDT2 are placed in a key-value list.
10) Design status for UDT2 is stored. Note that the utilization of embedded multipliers and RAM blocks is updated after UDT1.
11) UDT2 transformation takes place.
12) Finally, the design is unparsed to an output file.

This script is just a simple sequence, and in practice we use more features of the scripting language and provide more parameters for each transformation [37].

Other UDTs can be integrated into the proposed approach, *e.g.* a low power UDT [39]. Although users can define their own UDTs, UDTs are expected to be defined by domain experts and are used by application builders who may not be experienced in hardware optimization. UDTs should be defined based on application characteristics, in order to apply to a set of applications. Asanovic *et al.* [40] identify seven classes of computation and communication patterns, which cover a large range of numerical applications. The UDTs presented in this paper can work on three of the seven classes and we intend to work on the others.

The design space exploration in each UDT with M variables we present in this paper can be finished in polynomial time $O(M^k)$ [38] for some constant k, and exploring the order of N UDTs needs $O(N!)$. Therefore, in the worst case the complexity is $O(M^k N!)$. However, N is not large (3 UDTs in our experiments), and not all transformation orders are sensible in practice. Domain experts can choose a limited set of useful transformation orders in advance. Furthermore, different orders of applying transformations can be evaluated concurrently to reduce exploration time.

This section gives an overview of our approach to enable the automation of system design optimization; the next section presents the above mentioned UDTs in detail.

4 UDTs WITH GEOMETRIC PROGRAMMING

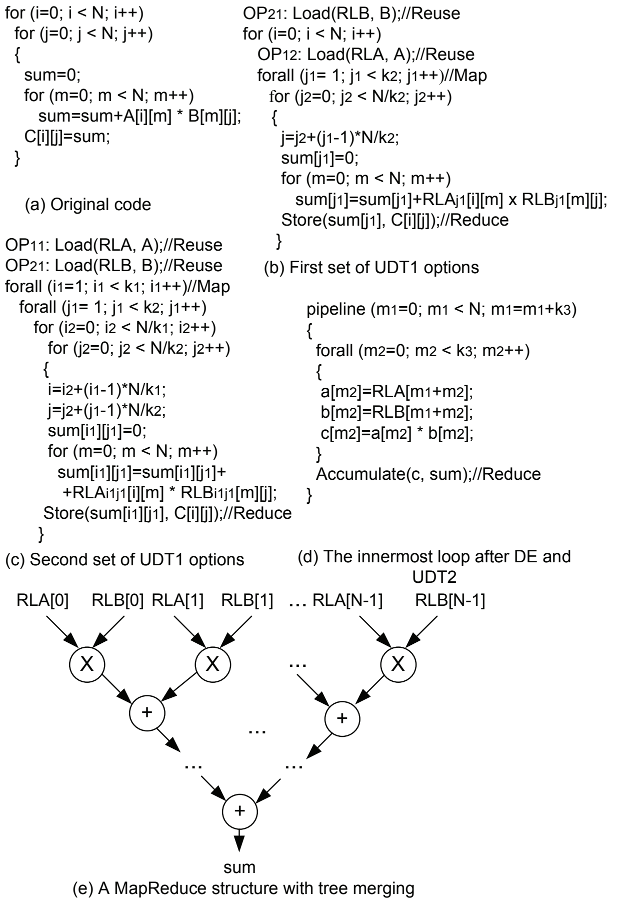
We observe that three optimization techniques, data reuse, pipelining, and MapReduce, are inter-related. Memory resources constrain all three techniques. Data reuse, after distributing buffered data across multiple memory banks, can improve memory bandwidth, benefiting pipelining and MapReduce. Computational resources constrain pipelining and MapReduce. This inter-relationship means that application of these techniques separately may not lead to efficient designs. Thus we present one UDT for optimizing both data locality and loop-level parallelism within a single step, and another UDT [29] exploiting MapReduce and pipelining. Some formulae in the optimization problem models shown below are nonlinear, due to the combination of the techniques. We convert them into geometric programming problems and use existing techniques to solve them.

Geometric programming (GP) [38] is the following optimization problem:

$$\begin{array}{lll} \text{min :} & f_0(x) & \\ \text{subject to} & f_i(x) \leq 1, & i = 1, \ldots, m \\ & h_i(x) = 1, & i = 1, \ldots, p \end{array}$$

where $x \succ 0$, and 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 can be transformed into a convex form with efficient solution algorithms with guaranteed convergence to a global minimum [38]. Also, as the addition and multiplication of posynomials are still posynomial, multiple GP models can be combined or extended to cover more complex transforming tasks.

Since our target output design in this paper is a Handel-C description, as mentioned in Section 3, the design execution model described in the rest of the paper is in line with Handel-C semantics that each assignment in the C description takes one clock cycle, although the formulation described below can be extended to other cases. Our utility-directed transformations



```
for (i=0; i < N; i++)
  for (j=0; j < N; j++)
  {
    sum=0;
    for (m=0; m < N; m++)
      sum=sum+A[i][m] * B[m][j];
    C[i][j]=sum;
  }
```

(a) Original code

```
OP21: Load(RLB, B);//Reuse
for (i=0; i < N; i++)
  OP12: Load(RLA, A);//Reuse
  forall (j1= 1; j1 < k2; j1++)//Map
    for (j2=0; j2 < N/k2; j2++)
    {
      j=j2+(j1-1)*N/k2;
      sum[j1]=0;
      for (m=0; m < N; m++)
        sum[j1]=sum[j1]+RLAj1[i][m] x RLBj1[m][j];
      Store(sum[j1], C[i][j]);//Reduce
    }
```

(b) First set of UDT1 options

```
OP11: Load(RLA, A);//Reuse
OP21: Load(RLB, B);//Reuse
forall (i1=1; i1 < k1; i1++)//Map
  forall (j1= 1; j1 < k2; j1++)
    for (i2=0; i2 < N/k1; i2++)
      for (j2=0; j2 < N/k2; j2++)
      {
        i=i2+(i1-1)*N/k1;
        j=j2+(j1-1)*N/k2;
        sum[i1][j1]=0;
        for (m=0; m < N; m++)
          sum[i1][j1]=sum[i1][j1]+
            +RLAi1j1[i][m] * RLBi1j1[m][j];
        Store(sum[i1][j1], C[i][j]);//Reduce
      }
```

(c) Second set of UDT1 options

```
pipeline (m1=0; m1 < N; m1=m1+k3)
{
  forall (m2=0; m2 < k3; m2++)
  {
    a[m2]=RLA[m1+m2];
    b[m2]=RLB[m1+m2];
    c[m2]=a[m2] * b[m2];
  }
  Accumulate(c, sum);//Reduce
}
```

(d) The innermost loop after DE and UDT2

(e) A MapReduce structure with tree merging

Fig. 4. Motivating example: matrix multiplication. For brevity, N is a multiple of $k_{l_{1\leq l\leq 3}}$. forall: all for loop iterations executing in parallel. pipeline: loop pipelining.

take care of the operation scheduling and resource allocation, and the resource binding is left to the Handel-C synthesizer.

4.1 UDT1 with Data Reuse and MapReduce

This section presents UDT1, which uses a GP model to optimize both data locality and loop-level parallelism in one step. The target applications are data-dominated. The transformation introduces for an array A an on-chip buffer RLA, loads frequently used data of A from off-chip memory into RLA at a loop level and replaces the array reference A with RLA for accesses (*data reuse*) [30], [31]. Then, the buffered data are distributed into different scratch-pad memory banks [7], [32] to increase memory bandwidth. Finally, operations from different loop iterations are mapped onto parallel processing units (*MapReduce*), using loop strip-mining and interchange [41]. This UDT extends [7] to apply to MapReduce patterns, and to consider computational resource constraints and ensure that only data accessed by a processing unit are transferred to that unit's buffers.

We use $N \times N$ integer matrix-matrix multiplication, as shown in Fig. 4, to illustrate the transformation. We assume arrays A, B and C are stored in off-chip memory with a single access port. Simple analysis shows that all elements of the matrices A and B in the critical data path are accessed more than once, loops i and j are parallelizable and the only dependence between iterations of loop m is the accumulation of sum. We introduce on-chip buffers for arrays A and B to reduce off-chip memory access, and apply MapReduce to loops i and j, so elements of C can be generated in parallel in the map phase; the reduce phase outputs results sequentially as there is only one off-chip memory port.

The possible transformation options are shown in (b) and (c) of Fig. 4. There are two data reuse options (OP_{11} and OP_{12}) for array A and one (OP_{21}) for array B. If the matrix size is 64×64 elements (8 bits), the on-chip memory required and the number of off-chip memory accesses for options OP_{11} and OP_{12} of array A are (32768 bits, 4096) and (512 bits, 4096), respectively. Both options have the same off-chip memory accesses, but different on-chip memory requirements. The data reuse options govern which loop to partition. For example, for options OP_{12} and OP_{21}, only loops j and m can be parallelized, because the single-port off-chip memory accesses for loading RLA exist in loop i, as in Fig. 4 (b). If loop i must be partitioned for higher parallelism, then OP_{11} must be chosen as in Fig. 4 (c), requiring more on-chip memory. This shows the inter-relationship between data reuse and MapReduce.

The design space combining data reuse and MapReduce increases exponentially with the number of array references and the number of loop levels [7]. We thus formulate the design space exploration as a GP optimization problem as shown in the left-hand side of Fig. 5, with the objective of minimizing execution time (1), subject to on-chip resource constraints (2)–(8). This GP model allows the optimal solution and is scalable with the problem size [7].

Table 2 lists notation used in this paper. Lower case letters represent integer variables, corresponding to design parameters used for transformation; capitals represent compile-time constants: input program parameters and platform parameters specified by users. These constant parameters are used to instantiate the models.

The objective (1) of this transformation is to minimize the number of execution cycles. There are three parts to (1): the number of execution cycles taken by S statements, except for the result reducing statement S_r (*e.g.* the statement Store in Fig. 4 (b)) after mapping the loop nest to a parallel structure, the number of execution cycles taken by the reduce phase, and the cycles for loading reused data from off-chip to on-chip memory. Together with the computational resource constraints (6), the number of execution cycles spent on the reduce phase and the upper bound on the number of data elements accessed in one parallel segment of the partitioned loops in (3), not considered in [7], complete the formulation.

Users can specify the constraints on the on-chip memory B and the number of computational resources C_f in inequalities (2) and (6) to trade area for speed. The requirements of UDT1 for input programs are rectangular loop structure, no pointers and statically determined memory access patterns. Related PDTs are function inlining, pointer-to-array converting, loop restructuring, and so on. The design parameters, data reuse variables ρ_{ij} and loop parallelization variables k_l, determine how transformations are carried out in UDT1.

$$\text{min:} \quad \sum_{s=1,s\neq S_r}^{S} \prod_{l=1}^{W_s} v_l + \prod_{l=1}^{W_r} (v_l \times k_l) + \sum_{i=1}^{R} \sum_{j=1}^{E_i} (\rho_{ij} \times C_{ij}) \tag{1}$$

subject to: for $1 \leq l \leq N, 1 \leq j \leq E_i, 1 \leq i \leq R, f \in F$

$$\prod_{l=1}^{W_r} k_l \times \sum_{i=1}^{R} \sum_{j=1}^{E_i} (\rho_{ij} \times b_{ij}) \leq B \tag{2}$$

$$\prod_{l\in Q_i} v_l \times \prod_{l \in P_i} L_l \times Size_{RAM}^{-1} \times b_{ij}^{-1} \leq 1 \tag{3}$$

$$\sum_{j=1}^{E_i} \rho_{ij} = 1 \tag{4}$$

$$\rho_{ij} \in \{0,1\}, 1 \leq j \leq E_i \tag{5}$$

$$\prod_{l=1}^{N} k_l \times W_f \leq C_f \tag{6}$$

$$k_l - (L_l - 1) \times \sum_{j=1}^{l} \rho_{ij} \leq 1 \tag{7}$$

$$L_l \times k_l^{-1} \times v_l^{-1} \leq 1 \tag{8}$$

$$k_l^{-1} \leq 1 \tag{9}$$

$$\text{min:} \ v_l \times ii + C_{data} + \sum_{i=1}^{I} d_i + \lceil \log_2 k_l \rceil + notFull \tag{10}$$

subject to: for $f \in F, 1 \leq i \leq I$

$$BW \times k_l \times M_b^{-1} \times ii^{-1} + notAlign \times ii^{-1} \leq 1 \tag{11}$$

$$W_f \times x_f^{-1} \times ii^{-1} \leq 1 \tag{12}$$

$$RecII \times ii^{-1} \leq 1 \tag{13}$$

$$d_i \times ii^{-1} \leq 1 \tag{14}$$

$$k_l \times x_f \leq C_f \tag{15}$$

$$L_l \times k_l^{-1} \times v_l^{-1} \leq 1 \tag{16}$$

$$R_{if} \times x_f^{-1} \times d_i^{-1} \leq 1 \tag{17}$$

$$1 \leq k_l \leq L_l \tag{18}$$

$$1 \leq x_f \leq C_f \tag{19}$$

Fig. 5. Optimization formulations for UDT1 (1)–(9) and UDT2 (10)–(19).

4.2 UDT2 with MapReduce and Pipelining

This section describes UDT2 exploiting MapReduce and pipelining optimizations, and briefly describes the GP model [29] involved in UDT2. This transformation can always be applied to applications for performing loop pipeline. Target platform computational resources restrict the number of parallel loop iterations, and bandwidth between processing units and memories affects how operations are scheduled after loop partitioning. This transformation generates a locally parallel, globally pipelined structure to balance use of memory bandwidth and hardware resources.

Fig. 4 (d) illustrates the problem, using the innermost loop of matrix multiplication. Our approach first applies PDTs. The original complex expression is decomposed into simple operations: two memory accesses, one multiplication and the result accumulation. This example exhibits the MapReduce pattern: multiplication executes independently on element pairs of array RLA and RLB, while accumulation is achieved by integer addition which is associative.

We thus map the innermost loop of matrix multiplication onto a parallel computing structure, by loop strip-mining. We need to determine the number of iterations (k_3) of each loop strip, running in parallel, and the initiation interval (ii) of pipelining the outer loop controlling the strip counter, as shown in Fig. 4 (d). Given a hardware platform with sufficient multipliers, there are multiple design options for mapping the innermost loop under different memory bandwidth constraints. For example, if the memory bandwidth is $2N$ bytes per execution cycle, Fig. 4 (e) (pipeline registers are not shown) shows a design with loop m fully parallelized, which needs $\log_2 N + 2$ execution cycles; assuming each assignment takes one clock cycle. When memory bandwidth is smaller, several designs exist, combining local parallelization and global pipelining. Even more design options result if multipliers are also constrained. Finding the best design in terms of various criteria is not easy, and requires design space exploration.

We therefore formulate mapping of loop l of a loop nest onto a parallel computing platform in problem (10)–(19) shown in the right-hand side of Fig. 5; detailed discussion can be found in [29]. The design parameter variables include the number of parallel partitions k_l of loop l and pipeline initiation interval ii. Once these parameters are determined, the corresponding transformation is carried out. This transformation can be applied to a single loop level with compile-time known loop bounds. Related PDTs are loop restructuring, loop coalescing, and loop peeling.

If this utility-directed transformation (UDT2) is executed after the transformation (UDT1) described in Section 4.1, then the design status needs to be considered. For example, UDT1 transformation generates a design that utilizes C'_f DSP blocks, increases memory bandwidth to M'_b when introducing on-chip buffers, and achieves a system speed T'. When the problem (10)–(19) is instantiated, M_b in (11) is replaced with M'_b and C_f in the constraint (15) for DSP resource becomes $C_f - C'_f$. After the instantiation, the problem is solved. If the execution time achieved by UDT2 is slower than T', then UDT2 transformation is ignored and the design transformed by UDT1 is retained.

4.3 Combined GP Model

The previous two sections respectively show two GP models used in two UDTs. As the models are both GP, we could combine them to simultaneously make design decisions on data reuse, multi-level MapReduce and pipelining for applications with the target characteristics; for example matrix multiplication has data reuse and two MapReduce patterns in different loop levels. The combined model formulates the whole design space and can find more efficient designs.

TABLE 2
A list of notation (# means the number of).

Variable	Description
ρ_{ij}	binary data reuse variables
k_l	# partitions of loop l
v_l	# iterations in one partition of loop l
d	# duplications of reused data
ii	initiation interval of pipeline
x_f	# resource f being used
d_i	# execution cycles taken by computation level i of DFG
b_{ij}	# on-chip RAM blocks required by the data reuse option j of reference i in one segment of the partitioned loops
Program parameters	Description
S	# statements in a program
S_o	# statements outside the innermost loop
S_r	result reducing statement associated with MapReduce
W_r	the innermost level of the loops under MapReduce
W_s	loop level of statement s in a loop nest
N	# loop levels
R	# array references
E_i	# data reuse options of array reference i
L_l	# iterations of loop l
$RecII$	data dependence constraint on ii in the computation
Q_i	set of loop indices in the indexing function of reference i among loops $(1, \ldots, W_r)$
P_i	set of loop indices in the indexing function of reference i among loops $(W_r + 1, \ldots, N)$
W_f	# computational resource f required in a loop iteration
R_{if}	# resource f required in computation level i of DFG
I	# computation levels of DFG
BW	required memory bandwidth in one loop iteration
F	F types of computational resources involved
$notFull$	0: loop is fully partitioned; 1: loop is partially partitioned
Platform parameters	Description
B	# on-chip RAM blocks available
$Size_{RAM}$	# data elements accommodated in on-chip RAM block
C_{ij}	# loading cycles of the data reuse array of option j of reference i
M_b	memory bandwidth available
C_f	# computational resource f available
$notAlign$	0: data are aligned; 1: data are not aligned
C_{data}	# cycles to read one datum from on-chip RAM to registers

As hardware resources constrain MapReduce, the available resources need to be allocated among multiple MapReduce levels to obtain an efficient design. This is the link used to combine the two GP models described in the previous two sections. For simplicity, we present a new GP model exploring data reuse, two-level MapReduce and pipelining with the inner-level MapReduce pattern in the innermost loop of a loop nest; it can be extended to many-level MapReduce cases.

The total number of execution cycles of a design comprises four parts:

$$\text{min: } cyc = cyc_s + cyc_{in} + cyc_r + \sum_{i=1}^{R} \sum_{j=1}^{E_i} (\rho_{ij} \times C_{ij}) \quad (20)$$

The number of cycles taken by statements outside the innermost loop cyc_s is given by (21). The number of cycles taken by statements inside the innermost loop after two-level MapReduce and pipelining is (22). The number of cycles cyc_r taken by the reduce phase of the outer loop MapReduce is (23), where two expressions for cyc_r correspond respectively to using a linear structure or a tree structure in the reduce phase [29]. The resource link between the MapReduce levels is (24), where we assume all computations are in the innermost loop. It can easily extend to cover cases where some computations execute in outer loops.

Together with the constraint models described in Section 4.1 and 4.2, this new model is used in providing a utility-directed transformation, which does the same job as the previous two UDTs but could generate more efficient designs.

$$cyc_s = \sum_{s=1}^{S_o} \prod_{l=1}^{Ws} v_l \quad (21)$$

$$cyc_{in} = \prod_{l=1}^{N-1} v_l \times (v_N \times ii + C_{data} + \sum_{i=1}^{I} d_i + \lceil \log_2 k_N \rceil + notFull) \quad (22)$$

$$cyc_r = \prod_{l=1}^{W_r} (v_l \times k_l) \textbf{ or } cyc_r = \prod_{l=1}^{W_r} v_l \times \log_2 \prod_{l=1}^{W_r} k_l \quad (23)$$

$$\prod_{l=1}^{N} k_l \times x_f \leq C_f, f \in F \quad (24)$$

A branch and bound algorithm used in [42] solves the integer GP in all three models, using the geometric programming relaxation as a lower bounding procedure. Section 5 shows the performance of these GP models; as expected, the combined GP model produces more promising designs, but large problem sizes are slow to solve.

5 Experimental results

In this section, we show results from applying our approach to three kernels: multiplication of two 64 × 64 matrices (MAT64), the motion estimation (ME) algorithm [43] used in X264 and Sobel edge detection (Sobel) [44]. In addition to these, applications such as correlation in signal processing and the widely used Monte Carlo simulation are all cases where our approach succeeds [29]. Table 3 shows benchmark properties. Data reuse opportunities exist in all kernels, and each benchmark contains the MapReduce pattern; we identify two levels of MapReduce pattern in ME, MAT64 and Sobel. We apply UDT1 to the outer level and UDT2 to the inner level. Performance-optimized designs under different constraints are represented by the chosen data reuse option for each array reference, the number of parallel loop partitions of each loop and the pipelining initiation interval, (ρ_{ij}, k_l, ii).

Our experiments use an FPGA-based system with off-chip SRAM. Without loss of generality, we assume the off-chip SRAM is accessed by a single port with two cycle latency; these off-chip SRAMs store input data. The FPGA is an XC2v8000, which has 168 embedded hard multipliers, 168 dual-port RAM blocks and runs at 100 MHz when all hard multipliers are used. Users can specify other hardware platforms, resulting in different platform parameters. The generated parallel computing structure consists of multiple processing units, each with its own two-level on-chip buffers: registers and on-chip RAM configured as scratch-pad memory. For ME and Sobel the frame size is that of the QCIF luminance

TABLE 3
The properties of the benchmarks.

Benchmark	# loops	Refs	MapReduce pattern	Reuse
MAT64	3	2	2 levels: outer loops and innermost loop	yes
ME	implicit	2	2 levels: implicit and in SAD	yes
Sobel	4	2	2 levels: outer two loops and inner two loops	yes

TABLE 4
Transformations used in experiments.

Utility-directed	Description	Purpose
UDT1 UDT2	Data reuse and MapReduce MapReduce and Pipeline	Improve data locality and parallelism
Pattern-directed	**Description**	**Purpose**
LM	Loop merging	Reduce data reuse dist-
DE	Decompose expressions with the 3-address rule	Reduce logic level
RF	Reduce fanout of variables	
Par	Parallelize independent statements	Improve parallelism

component (144 × 176 pixels). All results are obtained after synthesis, placement and routing.

The experimental results include:

- examining a series of pattern-directed and utility-directed transformations on MAT64 and Sobel to show the performance of the proposed approach;
- verifying the proposed UDTs on MAT64, sobel and ME, and comparing the transformations executing in different orders that result in different designs and allow a trade-off between optimization results and running time for solving optimization problems; and
- comparing the designs proposed by our approach with other two approaches on MAT64.

5.1 Results after Each Transformation

Table 4 shows the transformations used in our experiments. Table 5 and Figs. 6 and 7 show the effects of transformations on MAT64 and Sobel, with the results normalized over original designs.

UDT1 improves data locality and parallelism simultaneously, as shown in Table 5. We see that the number of off-chip memory accesses is significantly reduced, up to 64 times for the two benchmarks, resulting in significant power reduction in off-chip accesses. Also, the outer loops are partitioned into multiple parallel segments, considerably decreasing system execution time, as shown in Figs. 6 and 7. The costs of these improvements are the use of on-chip RAMs and slices. For example, in Table 5, the second design of Sobel after UDT1 has 18 times less off-chip memory accesses and the outer two loop levels are mapped into 144 executing parallel segments, but requires 144 blocks of on-chip RAMs and nearly 150 times more slices, compared to the original design. The large increase in on-chip resources is due to the transformed design executing in a single-program-multiple-data (SPMD) model. Users can trade speed for area, by specifying less number of available RAM blocks, for example, leading to slower speed, as shown in Fig. 9.

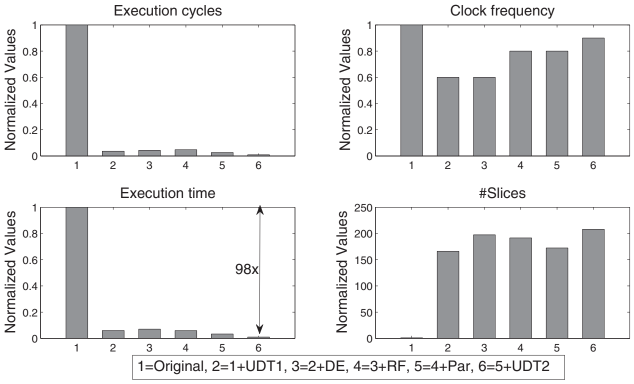


Fig. 6. Effects of a series of transformations on MAT64.

As a result of the large and complex circuits, the transformed designs also degrade clock frequency, as shown in Figs. 6 and 7 (compare the second bar to the first). The PDTs listed in Table 4 are used to optimize these circuits.

The last row of Table 5 shows the benefit of merging loops in the Sobel code before UDT1, as an example of PDT helping UDT. Without merging, UDT1 partitions the second loop into 59 parallel segments, while after merging loops UDT1 partitions the outer two loops of the code into 144 parallel segments. With the same on-chip RAM constraint and off-chip memory access reduction, the latter halves the execution time of the former, as Fig. 7 shows.

After UDT1, we apply multiple PDTs, listed in Table 4, one by one to the two benchmarks. The order of applying these transformations follows hardware optimization principles. DE and RF improve clock frequency of the resultant designs by reducing logic level and latency (Figs. 6 and 7). These transformations often increase execution cycles and area as they introduce intermediate registers to implement operations in multiple cycles. Afterwards, Par parallelizes statements to reduce execution cycles.

After each transformation, some design metrics may worsen, but further transformations can still be applied to try to meet the overall goal. For instance, execution times for MAT and Sobel in Fig.6 and 7 increase after DE and RF, but decrease after Par. In fact, without DE and RF to improve clock frequency, the execution time after UDT2 would not improve significantly.

Moreover, PDTs reveal more instruction-level parallelism that can be exploited by UDT2. As Table 5 shows, UDT2 further partitions the innermost loop into parallel segments and pipelines the operations. For example, the innermost loop of MAT64 is partitioned by 2 and pipelined with $ii = 1$. We partition the innermost loop into two parallel segments; given that there are 168 multipliers in this device, UDT1 maps the outer level MapReduce pattern onto 80 parallel processing units, with two multipliers in each unit.

On the target platform, composing UDT1 and UDT2 with several PDTs speeds up MAT64 and Sobel, by about 98 times and 88 times, respectively, compared to the initial designs.

TABLE 5
Results after UDT1 and UDT2 utility-directed transformations (a Before merging loop. b After merging loop).

Benchmark	UDT1			UDT2
	(ρ_{ij}, k_l)	Num of on-chip block rams	Off-chip access reduction	(ρ_{ij}, k_l, ii)
MAT64	$(1, 0, 1, 5, 16, 1)$	160	64×	$(1, 0, 1, 5, 16, 2, 1)$
Sobel	$(0, 1, 0, 1, 59, 1, 1)^a$ $(1, 0, 0, 144, 1, 1, 1)^b$	59 144	18× 18×	$(0, 1, 0, 1, 59, 1, 1, 3)^a$ $(1, 0, 0, 144, 1, 1, 3, 1)^b$

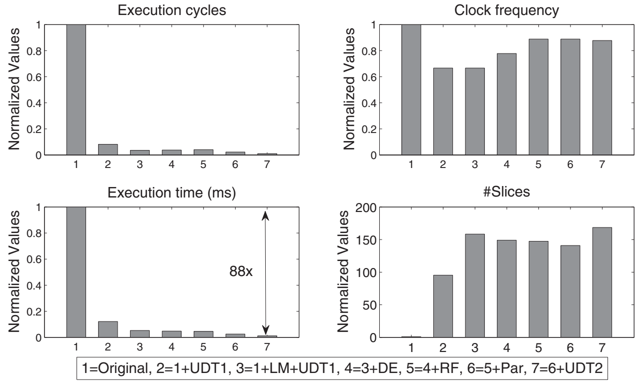
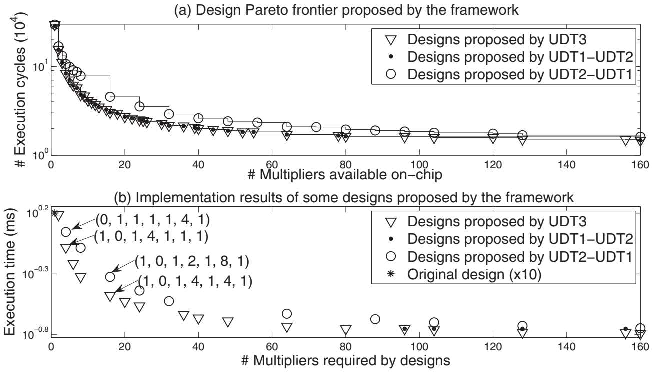
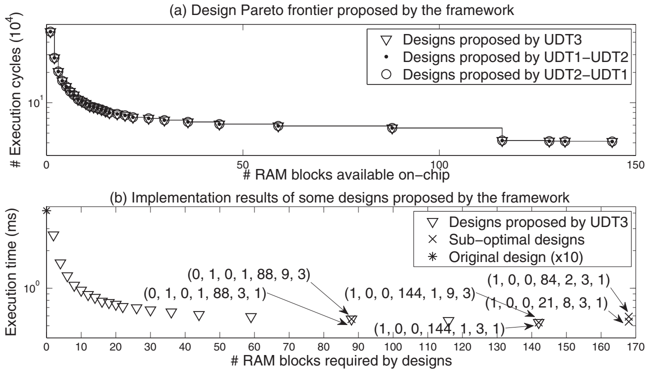


Fig. 7. Effects of a series of transformations on Sobel.


(a) Design Pareto frontier proposed by the framework

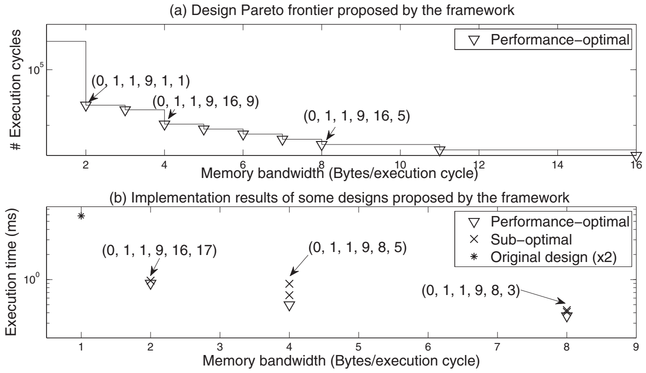
(b) Implementation results of some designs proposed by the framework

Fig. 8. Design space exploration results for MAT64.


(a) Design Pareto frontier proposed by the framework

(b) Implementation results of some designs proposed by the framework

Fig. 9. Design space exploration results for Sobel.


(a) Design Pareto frontier proposed by the framework

(b) Implementation results of some designs proposed by the framework

Fig. 10. Design space exploration results for ME.

5.2 Design Space Exploration with UDTs

To verify the UDTs in Section 4, we apply them to MAT64, Sobel and ME, and implement the resulting designs on the target platform. The results in Section 5.1 use a series of transformations, where UDT1 executes prior to UDT2. In this section, We perform them in three orders: UDT1 before UDT2 (UDT1-UDT2), UDT2 before UDT1 (UDT2-UDT1) and concurrent UDT1 and UDT2 (UDT3) as described in Section 4.3.

Given the number of computational resources, memory size and bandwidth, a performance-optimized design in the design space is generated by the UDTs in the three orders; all designs from the same order form a performance Pareto frontier. All results are shown in Figs. 8, 9 and 10, where the y axis follows a logarithmic scale. Fig. 8 (a) shows results for MAT64. First, designs from the three orders exhibit the same trend, *i.e.* execution cycles decrease as the available on-chip multipliers increase, due to greater parallelism; Figs. 9 (a) and 10 (a) show similar design trends. Users can constrain the design exploration to obtain expected performance. Second, given the same number of multipliers and comparing with the designs given by UDT1-UDT2 and UDT2-UDT1, the designs from UDT3 speed up to 6% and 72% respectively, and use about 0.5 and 8 times the respective number of on-chip memories. In Sobel, multiplication operations are replaced automatically by shift operations; Fig. 9 (a) shows designs from the three orders under different on-chip memory constraints. Here, due to no computational resource (multiplier) constraints between the two MapReduce levels, the three orders generate the same designs. Similarly, ME results are the same for each order, so Fig. 10 (a) only shows the Pareto frontier from UDT1-UDT2 under different memory bandwidths. We observe that the combined GP model UDT3 guarantees the performance-optimal design in the design space, as it combines design spaces of data reuse, MapReduce and pipelining, while UDT2-UDT1 achieves best resource utilization. Therefore, by specifying the order of composing different transformations, users can balance the system metrics. Moreover, the combined model needs over twice the run time, compared to separate approaches as Table 6 shows. Run times are for calling YALMIP [42] from

TABLE 6
Solution time comparison of three execution orders of our UDTs.

Benchmark	UDT3 (sec)	UDT1-UDT2 (sec)	UDT2-UDT1 (sec)
MAT64	136.0	49.3	57.3
Sobel	69.6	34.0	36.1
ME	136.8	15.3	16.7

TABLE 7
Performance comparison of our approach and [45] and [46] in matrix-matrix multiplication.

Dimensions	Virtex II Pro30		Virtex-5 VSX240	
of Matrices	[45] (ms)	Ours (ms)	Ours (ms)	[46] (ms)
[64, 64]	0.799	0.213	0.026	0.022
[128, 128]	5.122	1.264	0.157	0.071
[256, 256]	45.318	15.366	1.763	0.454

MATLAB on a 3 GHz PC.

To verify these results, we implement several designs of MAT64, Sobel and ME. Figs. 8 (b), 9 (b) and 10 (b) show real execution times and the use of on-chip multipliers, RAM blocks and memory bandwidth, respectively. These figures, firstly, show similar trends in performance Pareto frontiers to those in subfigures (a) under different design constraints. This proves that our models can distinguish the performance-optimal design from different design options.

The figures also show some sub-optimal designs. In Fig. 8 (b), some designs proposed by `UDT1-UDT2` and `UDT2-UDT1` lie above the Pareto line formed by results from `UDT3`. Note that most designs from `UDT2-UDT1` are worse than those of `UDT3`, while `UDT1-UDT2` has only four designs worse than `UDT3`. This agrees with Fig. 8 (a). Also, Fig. 8 (b) shows four designs with different parameters in angle brackets from `UDT3` and `UDT2-UDT1`. From these design parameters we can see that `UDT3` optimizes the design as a whole and thus achieves better results, while `UDT2-UDT1` gives the priority of parallelism optimization to the inner loops. Fig. 9 (b) explicitly shows parameters of performance-optimal and sub-optimal designs of Sobel. For example, when the on-chip RAM block constraint is 88, design (0, 1, 0, 1, 88, 9, 3) runs as fast as design (0, 1, 0, 1, 88, 3, 1) from our GP model, but needs more logic resources due to greater innermost loop parallelization. There are two designs using all available on-chip RAM blocks, as in the bottom-right corner of Fig. 9 (b), but they are still slower than our design (1, 0, 0, 144, 1, 3, 1). Likewise, for the same 8 bytes/cycle memory bandwidth in Fig. 10, the design from our approach is (0, 1, 1, 9, 16, 5) rather than design (0, 1, 1, 9, 8, 3) which matches the memory bandwidth; our model identifies the 0.04ms speed difference. Finally, the original unoptimized designs of MAT64, Sobel and ME are also implemented, shown in Figs. 8 (b), 9 (b) and 10 (b).

The results above demonstrate that our approach can find the performance-optimized design in the design space of data reuse, MapReduce and pipelining, and that different transformation orders enable trading off a) speed and resources, and b) design performance and compile time. These give users flexibility to customize designs.

5.3 Comparison with Existing Approaches

To evaluate our approach, we compare matrix-matrix multiplication against two existing approaches [45] and [46]. These approaches implement a blocked matrix multiplication algorithm with fixed-point arithmetic on FPGAs. Each approach uses different hardware platforms and FPGA devices.

Table 7 compares estimated execution times for various matrix sizes. Compared to [45], our approach is up to 4 times faster, since our approach extracts higher parallelism by exploiting MapReduce and pipelining. Compared to [46], our approach is slower by a factor of 1.2 to 3.8, because [46] uses double buffering to pipeline data input/output with computation. As matrix sizes increase, time spent on data loading/unloading increases and thus the performance difference between our approach and [46] increases. In future, we will integrate outer loop pipelining into our model to bridge this performance gap.

6 Conclusions

We present a systematic approach composing utility-directed transformations for optimizing and mapping a sequential design onto an FPGA-based parallel computing platform. Our approach provides multiple UDTs, each performing optimization transformations, and uses PDTs to connect UDTs. This enables the automation of complex hardware design optimizations. The combination of modular and parameterized UDTs allows users to work at a high level to describe design goals and specification. The approach is illustrated in this paper with two UDTs connected by several PDTs. Two geometric programming (GP) models guide speed optimizing UDTs, exploiting data reuse, multi-level MapReduce, and pipelining techniques.

Results from applying our approach to three applications show that design speed can improve up to 98 times compared to sequential designs in the same platform. Our UDTs can produce performance-optimized designs in the design space of exploiting data reuse, MapReduce and pipelining under different design constraints. Moreover, performing transformations in different order allows users to trade speed for resources, and design performance for compile time. Finally, compared to two existing approaches for matrix multiplication, our automated approach achieves performance between them.

Current and future work includes extending the current UDTs to include techniques such as outer loop pipelining, supporting more applications, adding more transformations to capture the whole hardware system design flow such as hardware/software partitioning and data representation optimization, and moving appropriate compile-time optimizations to run time. A direction of particular interest is to investigate extensions of the proposed approach to support self-optimization of designs to adapt to internal and external changes at run time. To avoid the time-consuming process of placement and routing at run time, we can generate multiple configurations and switch between them by either a) partial run-time reconfiguration [47], or b) compiling them into a single configuration and activating them at run time by clock gating [48]. In both cases, placement and routing takes place at design time while system performance optimization takes place at run time. In the future, we also intend to propose transformations for GPGPUs to improve their performance with reduced design effort.

Acknowledgment

This work was supported in part by UK EPSRC, by the European Union Seventh Framework Programme under Grant agreement number 248976 and 257906, by the HiPEAC NoE, by Alpha Data, by Celoxica, by nVidia, and by Xilinx.

References

[1] J. O. Kephart and R. Das, "Achieving self-management via utility functions," *IEEE Internet Computing*, vol. 11, no. 1, pp. 40–48, 2007.

[2] A. Armonas and L. Nemuraite, "Pattern based generation of full-fledged relational schemas from UML/OCL models," *Information Technology and Control*, vol. 35, no. 1, pp. 27–33, 2006.

[3] B. di Martino, N. Mazzoca, G. P. Saggese, and A. G. M. Strollo, "A technique for FPGA synthesis driven by automatic source code synthesis and transformations," *Proc. Int. Conf. on FPL*, 2002.

[4] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam, "Maximizing multiprocessor performance with the SUIF compiler," *IEEE Computer*, December 1996.

[5] ACE, "CoSy Compilers: Overview of Construction and Operation." [Online]. Available: http://www.ace.nl/compiler/paper-construct.pdf

[6] L. Renganarayana and S. Rajopadhye, "A geometric programming framework for optimal multi-level tiling," in *Proc. Conf. Supercomputing*. IEEE Computer Society, 2004, p. 18.

[7] Q. Liu, G. A. Constantinides, K. Masselos, and P. Y. K. Cheung, "Combining data reuse with data-level parallelization for FPGA-targeted hardware compilation: a geometric programming framework," *IEEE Trans. CAD.*, vol. 28, no. 3, pp. 305–215, 2009.

[8] K. Turkington, G. A. Constantinides, K. Masselos, and P. Y. K. Cheung, "Outer loop pipelining for application specific datapaths in FPGAs," *IEEE Trans. VLSI.*, vol. 16:10, pp. 1268–1280, 2008.

[9] Y. Lam, J. Coutinho, W. Luk, and P. Leong, "Optimising multi-loop programs for heterogeneous computing systems," in *Southern Programmable Logic Conference*, 2009, pp. 129 –134.

[10] T. Todman, J. G. d. F. Coutinho, and W. Luk, "Customisable hardware compilation," *The Journal of Supercomputing*, vol. 32, no. 2, pp. 119–137, 2005.

[11] "The TXL programming language," (2009 Oct.). [Online]. Available: http://www.txl.ca/

[12] M. Boekhold, I. Karkowski, H. Corporaal, and A. Cilio, "A programmable ANSI C transformation engine," *Proc. 8th Int. Conf. on Compiler Construction*, pp. 292–295, 1999.

[13] S. Derrien and P. Quinton, "Parallelizing HMMER for hardware acceleration on FPGAs," in *IEEE Int. Conf. on ASAP*, July 2007, pp. 10–17.

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

[15] Z. Guo, B. Buyukkurt, and W. Najjar, "Input data reuse in compiling window operations onto reconfigurable hardware," in *ACM SIGPLAN/SIGBED Conf. on LCTES*. New York, NY, USA: ACM Press, 2004, pp. 249–256.

[16] B. So, M. W. Hall, and P. C. Diniz, "A compiler approach to fast hardware design space exploration in fpga-based systems," in *Proc. ACM SIGPLAN Conf. on Programming language design and implementation*. New York, NY, USA: ACM, 2002, pp. 165–176.

[17] A. P. Chandrakasan, M. Potkonjak, J. Rabaey, and R. W. Brodersen, "Hyper-lp: a system for power minimization using architectural transformations," in *Proc. IEEE/ACM Int. Conf. on Computer-aided design*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1992, pp. 300–303.

[18] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-level synthesis for FPGA-based processor/accelerator systems," in *ACM/SIGDA Int. Symp. on FPGA*, 2011, pp. 33–36.

[19] "Introducing AccelDSP Synthesis," (2008 May). [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/acceldsp_user.pdf

[20] (2008 Oct.). [Online]. Available: http://www.mentor.com/products/c-based_design/catapult_c_synthesis/index.cfm

[21] (2005 Oct.). [Online]. Available: http://www.synfora.com/products/picoexpress.html

[22] (2005 Oct.). [Online]. Available: http://www.impulsec.com/C_to_fpga_overview.htm

[23] (2008 May). [Online]. Available: http://www.agilityds.com

[24] (2008 May). [Online]. Available: http://www.forteds.com/products/cynthesizer_datasheet.pdf

[25] (2010 Jan.). [Online]. Available: http://www.mentor.com

[26] "Nios II C2H Compiler user guide," (2008 May). [Online]. Available: http://www.altera.com/literature/ug/ug_nios2_c2h_compiler.pdf

[27] (2011 Jan.). [Online]. Available: http://www.autoesl.com

[28] Q. Liu, T. Todman, J. G. de F. Coutinho, W. Luk, and G. A. Constantinides, "Optimising designs by combining model-based and pattern-based transformations," in *Proc. Int. Conf. on FPL*, 2009, pp. 308–313.

[29] Q. Liu, T. Todman, W. Luk, and G. A. Constantinides, "Automatic optimisation of MapReduce designs by geometric programming," in *Proc. Int. Conf. on FPT*, Sydney, Australia, 2009, pp. 215–222.

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

[31] Q. Liu, G. A. Constantinides, K. Masselos, and P. Y. K. Cheung, "Automatic on-chip memory minimization for data reuse," in *Proc. IEEE Symp. on FCCM*, 2007, pp. 251–260.

[32] N. Baradaran and P. C. Diniz, "A compiler approach to managing storage and memory bandwidth in configurable architectures," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 13, no. 4, pp. 1–26, 2008.

[33] H. Rong, Z. Tang, R. Govindarajan, A. Douillet, and G. R. Gao, "Single-dimension software pipelining for multi-dimensional loops," in *IEEE Proc. on CGO*, 2004, pp. 163–174.

[34] J. H. Yeung, C. Tsang, K. Tsoi, B. S. Kwan, C. C. Cheung, A. P. Chan, and P. H. Leong, "Map-reduce as a programming model for custom computing machines," in *Proc. Int. Symp. on FCCM*, 2008, pp. 149–159.

[35] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *6th Symp. on OSDI*, December 2004, pp. 137–150.

[36] T. Todman, Q. Liu, W. Luk, and G. Constantinides, "A scripting engine for combining design transformations," in *Proc. IEEE Int. Symp. on FCCM*, 2010, pp. 255–258.

[37] ——, "Customizable composition and parameterization of hardware design transformations," in *13th Euromicro Conf. on DSD*, 2010, pp. 595–602.

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

[39] Q. Liu, T. Todman, and W. Luk, "Combining optimizations in automated low power design," in *DATE Conf.*, 2010, pp. 1791–1796.

[40] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, 2006.

[41] U. K. Banerjee, *Loop Parallelization*. Kluwer Academic, 1994.

[42] J. Löfberg, "Yalmip : A toolbox for modeling and optimization in MATLAB," in *Proc. Conf. on CACSD*, Taipei, Taiwan, 2004.

[43] L. Merritt and R. Vanam, "Improved rate control and motion estimation for H.264 encoder," in *Proc. ICIP*, 2007, pp. 309–312.

[44] (2006). [Online]. Available: http://www.pages.drexel.edu/∼weg22/edge.html

[45] N. Dave, K. Fleming, M. King, M. Pellauer, and M. Vijayaraghavan, "Hardware acceleration of matrix multiplication on a Xilinx FPGA," in *Proc. Int. Conf. on Formal Methods and Models for Codesign*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 97–100.

[46] I. Sotiropoulos and I. Papaefstathiou, "A fast parallel matrix multiplication reconfigurable unit utilized in face recognitions systems," in *Proc. Int. Conf. on FPL*, Prague, Czech, 2009, pp. 276–281.

[47] M. Koester, W. Luk, J. Hagemeyer, M. Porrmann, and U. Ruckert, "Design optimizations for tiled partially reconfigurable systems," *IEEE Trans. VLSI.*, vol. 19, no. 6, pp. 1048 –1061, June 2011.

[48] Q. Liu, T. Mak, J. Luo, W. Luk, and A. Yakovlev, "Power adaptive computing system design in energy harvesting environment," in *Proc. Int. Conf. on SAMOS*, 2011, pp. 33–40.

Qiang Liu received the B.S. degree and the MSc degree in School of Electronic Information Engineering from Tianjin University, Tianjin, China, in 2001 and 2004, respectively, and the Ph.D. degree from Department of Electrical and Electronic Engineering at Imperial College London, London, U.K., in 2008. From 2004 to 2005, he worked for STMicroelectronics Co. Ltd, Beijing, China. From 2009 to 2011, he was a Research Associate in Department of Computing at Imperial College London. He is currently an Associate Professor in School of Electronic Information Engineering at Tianjin University, with research interests in hardware compilation and synthesis, VLSI design optimization and automation, and reconfigurable computing.

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 (Associate Professor) in Digital Systems and Head of the Circuits and Systems research group. Dr. Constantinides is a Fellow of the BCS and a Senior Member of the IEEE. 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, FPGA, FPT and FPL.