Optimal pipeline design for Recursive Variable Expansion

Zubair Nawaz*, Thomas Marconi*,
Koen Bertels*,[1], Todor Stefanov†,[2]

* *Computer Engineering Lab, TU Delft, Mekelweg 4, 2628 CD, Delft, The Netherlands*
† *LIACS, Leiden University, Niels Bohrweg 1, 2333 CA, Leiden, The Netherlands*

ABSTRACT

This paper presents an automated flexible pipeline design algorithm for our unique acceleration technique called Recursive Variable Expansion. The preliminary experimental results on a kernel of real life application shows comparable performance to hand optimized implementation in reduced design time. This make it a good choice for generating high performance code for kernels, for which hand optimized codes are not available.

KEYWORDS: Loop Optimization; high performance computing

1 Introduction

Loops are an important source of performance improvement, for which there exist a large number of compiler optimizations. A major performance can be achieved through *loop parallelization*. By doing extensive loop parallelization, reconfigurable systems beat general purpose processor (GPP) albeit the higher frequency of GPP. Recursive Variable Expansion (RVE) is a technique which removes the loop carried data dependencies among the various statements of the program to execute every statement in parallel. By doing this, we can see the maximum parallelism that can be achieved assuming we have unlimited hardware resources on a FPGA. To be more practical, we would like to achieve maximum parallelism given some limited hardware resources. Pipelining is one of the widely used technique, in which resources can be reduced with out much degradation of parallelism. In this paper, we introduce a flexible pipelining design algorithm for RVE which not only fulfills the area constraints on a given FPGA but also hides the memory access latency behind computation.

1.1 Recursive Variable Expansion

The basic idea is the following. If any statement G_i is waiting for some statement H_j to complete for some iteration i and j respectively due to some data dependency, both of the

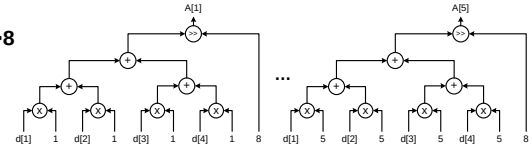
[1]E-mail: {z.nawaz,t.m.thomas,k.l.m.bertels}@tudelft.nl
[2]E-mail: stefanov@tudelft.nl

```
for i=1 to 5
  A[i]=0
  for j=1 to 4
    A[i]=A[i]+d[j]*i
  end for
  A[i]=A[i]>>8
end for
```

(a) sample code

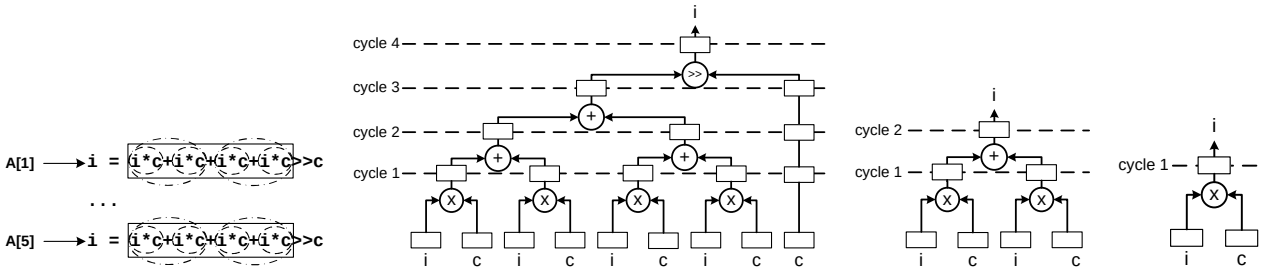
```
A[1]=A[1]>>8
    =A[1]+d[4]*1>>8
    =A[1]+d[3]*1+d[4]*1>>8
    =A[1]+d[2]*1+d[3]*1+d[4]*1>>8
    =A[1]+d[1]*1+d[2]*1+d[3]*1+d[4]*1>>8
    =0+d[1]*1+d[2]*1+d[3]*1+d[4]*1>>8
    =d[1]*1+d[2]*1+d[3]*1+d[4]*1>>8
...
A[5]=d[1]*5+d[2]*5+d[3]*5+d[4]*5>>8
```

(b) RVE expanded expressions



(c) Circuit for Figure 1b

Figure 1: Motivational example



(a) Generic Expressions (b) Pipelined circuit for *i*c+i*c+i*c+i*c>>c* (c) Pipelined circuit for *i*c+i*c* (d) Pipelined circuit for *i*c*

Figure 2: Generic expression and its pipeline circuit **??**

statements can be executed in parallel, if the computation done in H_j is replaced with all the occurrences of the variable in G_i which creates the dependency with H_j. This makes G_i independent of H_j. Similarly, computations can be substituted for all the variables which creates dependencies in other statements. This process can be repeated recursively till all the statements are function of known values and all data dependencies are removed. Hence, all the statements can be executed in parallel provided the required resources are available. RVE can be applied to a class of problems, which satisfy the following conditions.

1. The bounds of the loops must be known at the compile time.
2. The loops does not have any conditional statement.
3. Data is read at the beginning of a kernel from the memory and written back at the end of the kernel.
4. The indexing of the variables should be a function of surrounding loop iterators and/or constants.

2 Pipelining for Recursive Variable Expansion

We will use the simple example shown in Figure 1a in the rest of the paper to illustrate the RVE technique and how pipelining can be efficiently used. d[1], d[2], d[3] and d[4] are the four inputs and A[1], A[2], ..., A[5] are the five outputs to it. After applying the RVE, we get the *expanded expressions* shown in Figure 1b. As all loop carried dependencies are removed, all the expanded statements in Figure 1b can be computed efficiently by computing all the

Table 1: Comparison of automatically optimized DCT with Xilinx hand optimized DCT

	Frequency (MHz)	Initial latency (cycles)	Computation time for a block of 8×8 (cycles)	Time (ns)	Slices
Xilinx DCT core	171.223	92	64	373.8	1213
DCT full element	121.479	13	64	526.8	9215
DCT one-third element	265.354	8	192	723.6	2031

outputs in parallel by using a binary tree structure for each output as shown in Figure 1c. Computing like this gives a lot of parallelism, at the same time it requires a lot of area. This area can be reduced at the cost of little degradation in parallelism if all the circuits can be pipelined.

When a circuit is to be made from an expression, then the type and sequence of operators along with the type of operands is important. Therefore the expanded expressions in Figure 1b can be transformed to the *generic expressions* in Figure 2a by replacing variables with their types. In Figure 2a, *i* stands for integer and *c* for constant. The information in a generic expression is sufficient enough to infer the type and sequence of the operator along with the type of operands. Figure 2a shows that the generic expression (i.e. *i*c+i*c+i*c+i*c>>c*) for all outputs (A[1], A[2], ..., A[5]) is the same, which means that the sequence and type of operators in circuits of all outputs is the same. Therefore, we can map a circuit for an output along with intermediate registers on to an FPGA as shown in Figure 2b, provided it meets the area and memory constraints. The rest of the elements can be pipelined one after the other just by feeding the corresponding variables after each cycle in the circuit. However, if the memory or area constraints are not met, then the expression for an element has to be divided further and further into some smaller repeated equivalent sub-expressions such that when a circuit is to be made for any of those sub-expressions, it satisfies the area and memory constraints. This smaller sub-expression can be pipelined easily as it is small enough to satisfy the area and memory constraints and there are more than one such expression, for which corresponding data can be provided accordingly. For example in Figure 2a, some repeats are: *i*c+i*c* repeated 10 times and *i*c* repeated 20 times. The corresponding pipelined circuits are shown in Figure 2c and Figure 2d. This means that the problem of enumerating pipelining candidates for expanded expression is equivalent to finding repeated equivalent subexpressions or *repeats* in the corresponding generic expression. The chances of finding various repeats is very high in a RVE generic expression because it is generated from loop body without conditional statement which is doing some repetitive task, as shown in Figure 2a.

We would like to find the largest circuit which can be pipelinied and also satisfy the area and memory requirement. It is equivalent to finding the largest repeat in the generic expression which satisfy the area and memory requirment.

A better algorithm to find repeat is by using suffix tree [NMSB09]. It can find the optimal repeat in $O(L^2)$, which satisfies the area and memory requirement.

3 Experiments

The pipelining algorithm for RVE is implemented, which automatically finds the optimal repeat and the variables that need to be transferred after every cycle. We use a Molen pro-

totype implemented on the Xilinx Virtex II pro platform XC2VP30 FPGA, which contains 13696 slices. The automatically generated code is simulated and synthesized on ModelSIM and Xilinx XST of ISE 8.2.022 respectively. The integer implementation of DCT is used as benchmark to demonstrate the effectiveness of our pipelining algorithm. The results of the automatically optimized and different pipeline sizes for the DCT are compared with the hand optimized and pipelined DCT core[3] provided by Xilinx on the same platform. All the implementations take 8-bit input block elements and output DCT of 9-bit. A kernel of DCT outputs 64 elements whose generic expressions are the same. The repeat finding algorithm gives the optimal repeat in generic equation to be equal to expanded expression of one of these elements, we refer to it as *full element* in the experiment. This is the largest repeat which satisfy the area and memory requirements, therefore it is the optimal repeat. However, if there is less area available on FPGA, The next largest repeat is equal to one third of the expanded expression of the element, we refer to it as *one-third element*.

Table 1 shows the results for different implementation of DCT after synthesis. The Xilinx DCT core is hand optimized by knowing the properties of 2D DCT. 1D DCT is only implemented with buffering and taking the transpose of the 8×8 block. Our automatic optimization does not take advantage of the knowledge of the properties of 2D DCT. It takes the unoptimized code of 2D DCT, follows some generic steps to apply the RVE and then design a flexible deep pipeline trying to satisfy the area and memory constraints. The code generated for *DCT full element* is very large as compared to hand optimized, therefore it has low frequency. However, it extracts lots of parallelism and utilizes the resources to its capacity and produces a output of DCT block every 64 cycles with low initial latency of 13 cycles, which is basically the depth of the pipeline. The code for DCT one-third is relatively small but still larger than Xilinx DCT core. It produces better frequency than Xilinx core at the cost of 3 times more cycles and low initial latency of 8 cycles to compute one DCT block. The time to compute DCT using one third element is increased by 37% with a 78% decrease in area as compared to computing with full element.

4 Conclusion

The results show that our pipelining design algorithm for RVE which applies on some limited type of problems gives a comparable performance at the cost of extra hardware than the hand optimized code. Although it is not better than hand optimized in performance, the main benefits of our approach is automated design, optimization, and HW generation of kernels starting from a program code. Our algorithm is a good choice for kernels, for which hand optimized codes are not available, area is not major concern and high performance is the requirement in short design time. As a future work, we will apply our algorithm on more kernels for real life application.

References

[NMSB09] Z. Nawaz, T. Marconi, T. P. Stefanov, and K.L.M. Bertels. Flexible pipelining design for recursive variable expansion. In *Reconfigurable Architectures Workshop*, May 2009.

[3]https://secure.xilinx.com/webreg/clickthrough.do?cid=55758