International Journal of Computational Intelligence Techniques
ISSN: 0976-0466 & E-ISSN: 0976-0474, Volume 3, Issue 2, 2012, pp.-72-75.
Available online at http://www.bioinfopublication.org/jouarchive.php?opt=&jouid=BPJ0000221

K-LOOPS TRANSFORMATIONS FOR RECONFIGURABLE ARCHITECTURES

DUA R.[1]* AND KUSHWAHA A.K.[2]

[1]Department of Electronics & Communication Engineering, Ansal Institute of Technology, Gurgaon- 122003, Haryana, India.
[2]School of Engineering & Technology, Ansal University, Gurgaon- 122003, Haryana, India.
*Corresponding Author: Email- rakhi.dua@aitgurgaon.org

Received: October 25, 2012; Accepted: November 06, 2012

Abstract- The focus of this paper is on kernel loops (K-loops), which are loop nests that contain hardware mapped kernels in the loop body. In this paper, we propose methods for improving the performance of such K-loops, by utilizing standard loop transformations for exposing and exploiting the coarse grain loop level parallelism. The goal is to achieve a reconfigurable architecture that is a heterogeneous system consisting of a general purpose processor and a field programmable gate array (FPGA). In this work, different architectures are developed to give solutions to different problems: how to partition the application - decide which parts to be accelerated on the FPGA, how to optimize these parts (the kernels), what is the performance gain. So far, few researchers have exploited the coarse grain loop level parallelism.

Keywords- FPGA, GPP, CCU

Citation: Dua R. and Kushwaha A.K. (2012) K-Loops Transformations for Reconfigurable Architectures. International Journal of Computational Intelligence Techniques, ISSN: 0976-0466 & E-ISSN: 0976-0474, Volume 3, Issue 2, pp.-72-75.

Copyright: Copyright©2012 Dua R. and Kushwaha A.K. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution and reproduction in any medium, provided the original author and source are credited.

Introduction

In this paper, we propose a general framework that helps to determine the optimal degree of parallelism for each hardware mapped kernel within a K-loop, taking into account area, memory size, bandwidth and performance considerations. Furthermore, we design algorithms for several loop transformations in the context of K-loops which are used to determine the best degree of parallelism for a given K-loop, here the mathematical models are used to determine the corresponding performance improvement. The loop transformations that we analyze in this work are unrolling the loop, shifting the loop, K-pipelining, distribution of loop, and loop skewing. Also developed an algorithm that decides which transformations is to be used for a given K-loop.

Definition of K-Loop

Within the context of Reconfigurable Architectures, we define a kernel loop (K-loop) as a loop containing in the loop body. The loop may also contain code that is not mapped on the FPGA and that will always execute on the GPP (in software). The software code and the hardware kernels may appear in any order in the loop body. The number of hardware mapped kernels in the K-loop determines its size. A simple example of a size one K-loop is illustrated in [Fig-1], where SW is the software function & K is hardware mapped kernel. Here we focus on improving the performance for such loops by applying standard loop transformations to maximize the parallelism inside the K-loop.

```
for (i= 0; i< N; i ++) do
/* a general software function */
SW (blocks[i]);
/* a hardware mapped kernel */
K (blocks [i];
```

Fig. 1- A K-loop with one hardware mapped kernel

Assumptions for K-loop Framework

K-Loop Structure

- No inter-iteration dependencies, expect for wave front like dependencies;
- K-Loop bounds known at compile time.

Memory Accesses

- Reading or writing of memory at the beginning or end;
- Sharing of memory(on chip) by the GPP and CPPs
- All necessary data available in shared memory;
- Reading or writing of memory from or to the shared memory performed sequentially;
- Storing of kernel local data in the FPGA's local memory but not in the shared part of the memory.

Area and Placement:

- Shape of design not considered;
- Configuration of initial hardware is decided by a scheduling algorithm such the configuration latency is hidden.

Problem Overview

We need to improve the performance of such type of K-loop as K-loop is the part where applications spend most of their execution times. The proposed approach is to use standard loop transformations and maximize the parallelism inside the K-loop.

Loop unrolling is a transformation that replicates the loop body and reduces the iteration number. In our work, we use unrolling to expose the loop parallelism, thereby allowing us to execute concurrently multiple kernels on the reconfigurable hardware. [Fig-2] illustrates the unrolling transformation, where every task inside the initial loop is replicated for the same number of times inside the unrolled loop.

Loop shifting is a particular case of software pipelining. The shifting transformation moves operations from one iteration of the loop body to the previous iteration, as can be seen in [Fig-3]. The operations are shifted from the beginning of the loop body to the end of the loop body and a copy of these operations is also placed in the loop prologue. To be more specific, the prologue consists of A(1), the epilogue consists of a combination of B(N) and C(N), and each iteration from the transformed loop body consists of B(i), C(i), and A(i+1). In our research, loop shifting describes moving a function from the beginning of the K-loop body to the end thereby preserving the correctness of the program. We use loop shifting and loop pipelining to eliminate the data dependencies between software and hardware functions, allow them to execute concurrently. We use the loop shifting transformation for single kernel KL.

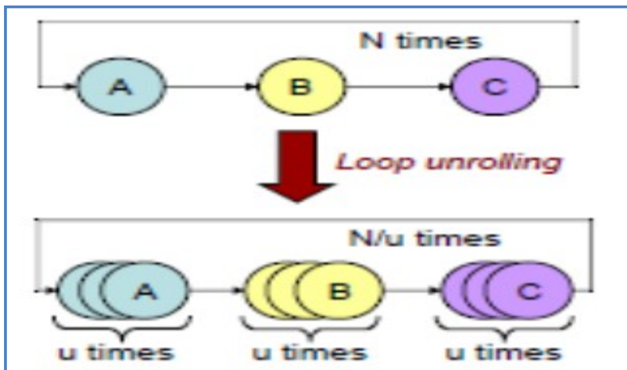

Fig. 2- Loop Unrolling

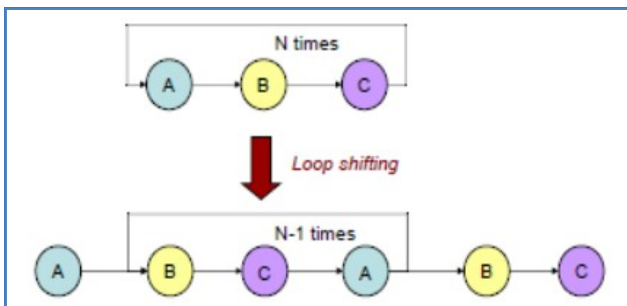

Fig. 3- Loop Shifting

Loop pipelining is used in our work for transforming K-loops with more than one kernel in the loop body. [Fig-4] illustrates the loop pipelining transformation. The prologue consists of A(1) followed by A(2) and B(1). The epilogue consists of B(N) and C(N-1), followed by C(N). Each iteration i from the transformed loop body consists of C(i), A(i + 2), and B(i + 1). When applied to K-loops, we define this transformation as K-pipelining. Assuming that the software and hardware functions alternate within the K-loop body, then half of the total number of function will need to be relocated.

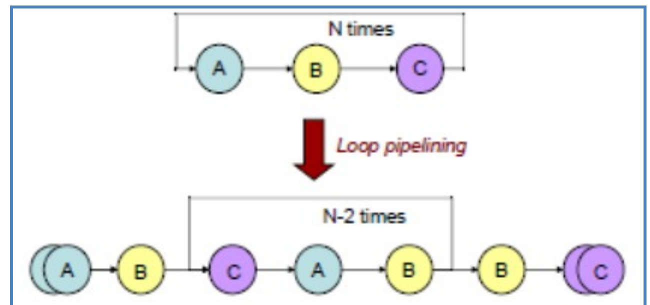

Fig. 4- Loop Pipelining

Loop distribution is a technique that breaks a loop into multiple loops over the same index range but each taking only a part of the loop's body. In our work, loop distribution is used to break down large K-loops (more than one kernel in the loop body) into smaller ones, so as to benefit more from parallelization with loop unrolling and loop shifting or K-pipelining. A generic representation of the loop distribution transformation is presented in [Fig-5]. In the top part, a loop with three tasks is illustrated. In the bottom part, each task is within its own loop and the execution order of the tasks has not changed.

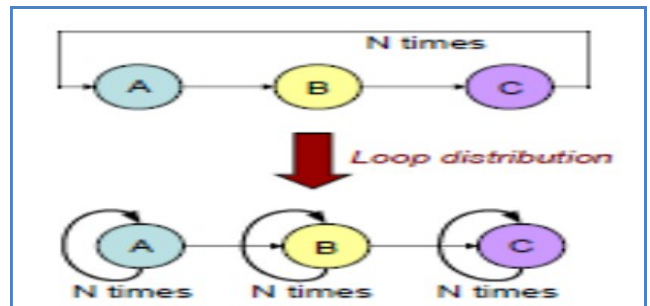

Fig. 5- Loop Distribution

Loop skewing is a widely used loop transformation for nested loops iterating over a multidimensional array, where an individual iteration of the inner loop depends on previous iterations. This type of code cannot be parallelized or pipelined in its original form. Loop skewing rearranges the array accesses so that the only dependencies are between iterations of the outer loop, and enables the inner loop parallelization.

Consider a nested loop whose dependencies are illustrated in [Fig-6a]. In order to compute the element (i, j) in each iteration of the inner loop, the previous iteration's results (i-1, j) must be available already. Performing the affine transformation (p, t) = (i, 2 * j + i) on the loop bounds and rewriting the code to loop on p and t instead of i and j, the iteration space as well as dependencies change are shown in [Fig-6b].

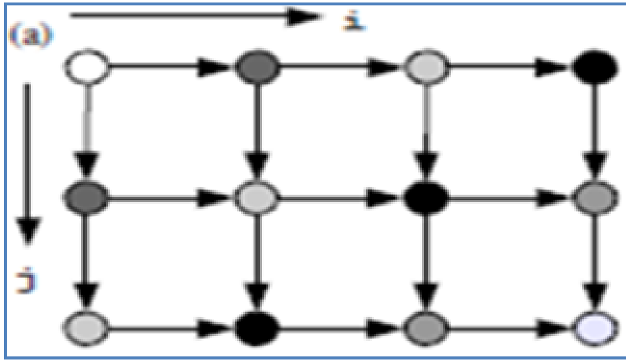

Fig. 6a- Loop dependencies Before skewing (different shades of gray show the elements that can be executed in parallel)

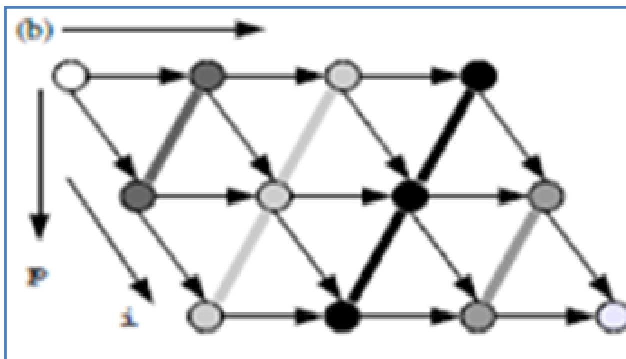

Fig. 6b- Loop dependencies After skewing(different shades of gray show the elements that can be executed in parallel)

Eps = 2; shift = 0;

Loop_trans KL

```
if (KL.Size == 1) then
  if (KL. Dependencies & WAVEWFRONT) then
    Skew (KL);
  End
  if (KL. Dependencies & TASKCHAIN) then
    Shift (KL);
    shift = 1
  End
  if shift && (KL.SW.ExecTime() / KL.HW_ExecTime( Eps)then
    Unroll (KL);
  End
Else
  go = KL.Distribute();
  if (go) then
    while (KLi[i]=KL GetNextSubLoop()do
      Loop-Trans(KLi[i]);
      i++
    End
  Else
    KPipeline(KL);
    Unroll(KL);
  End
End
End
```

Fig. 7- The loop transformations algorithm.

Table 1- Summary of loop transformations for K-Loops

Transformation	K-loop size	Impact	Reconfiguration	Can be used with	Exclusive with	Can schedule kernels in sw
Unrolling	Any	Enable hw parallelism	No	(2)-(5)	-	No
Shifting	1	Enable hw-sw Parallelism	No	(1),(4),(5)	-3	No
K-pipelining	≥2	Enable hw-sw Parallelism	No	(1),(4)	-2	No
Distribution	≥2	Split the K-loop into K-sub loops that are individually optimized	Yes/no	(1)-(3),(5)	-	No
Skewing	1	Eliminate dependencies enable unrolling	No	(1),(2)	-	Yes

Overview of Loop Transformations

Here, we develop an algorithm used to decide which loop transformations to apply to a given K-loop. [Table-1] shows an overview of the different loop transformations analyzed in this paper. For each transformation, the summary of loop transformations for K-loops is shown in [Table-1].

The K-loop size that is suitable for:

- The effect it has on the code (such as enabling parallelism or eliminating dependencies);
- The transformations that can be used in conjunction with it;
- The transformations that are used exclusive with it;
- How can transformation schedule kernels to run in software or not.

Simulation Technique

The 'Loop Trans' algorithm in [Fig-7] can be used to decide which loop transformations can be used on a given K-loop (KL). If the K-loop size is 1, depending on the existing loop dependencies, the K-loop is transformed with either loop skewing and/or loop shifting. Loop shifting is used to enable the software- hardware parallelism. If loop shifting is performed, it is important to look at the ratio between the execution times of the software part and of the hardware part of the K-loop, in order to understand if loop unrolling, which exposes the hardware parallelism, should be performed. If the software execution time is twice (or more) more than the hardware execution time, unrolling will bring no benefit because the hardware execution would be completely hidden by the software execution. If the ratio between the software and the hardware execution times is less than the threshold value (Eps = 2), unrolling will be performed.

If the K-loop size is larger than 1, the loop distribution algorithm will determine the best partitioning of the K-loop. In this way the algorithm run in a Greedy manner, selecting the most promising kernel in terms of speedup and adding functions around it to create a K-sub-loop, thereby constantly checking the performance of the K-

sub-loop against that of the original loop. It is possible that the partitioning that gives the best performance is into smaller K-sub-loops, each of the K-sub-loops will be transformed according to the original K-loop. If the loop distribution algorithm has decided that the original K-loop would perform best, then it will be transformed with K-pipelining and unrolling.

[Table 2] summarizes the formulas for computing the K-loop execution time (in cycles) for each of the proposed loop transformations.

The formulas presented for the loop unrolling, loop shifting, K-pipelining and loop skewing are dependent on the unroll factor, thereby assuming that we are always interested in exploiting the features of hardware parallelism. In loop distribution, the total execution time is the sum of the execution times of the K-sub-loops and their reconfiguration times, and each of the K-sub-loops may have a different degree of parallelism and different formula to compute its execution time. For this reason, we cannot explicitly write the formula in the case of loop distribution.

Table 2- Summary of the formulas for computing the k-loop execution time for each of the proposed loop transformations

Transformation		Execution Time
Unrolling	T- -unroll (u)=	N. T--sw + [N/u]. T-k(hw) (u) + T-k(hw) (R)
Shifting	T shift (u)=	u. T--sw + [N/u]. T-k(hw) (u) + T-k(hw) (R) u<U1; N/u].u.T sw + max (R.T--sw,T-k(hw) (u)) + T-k(hw) (R) u≥U1
K-pipelining	T- -pipe (u)=	(N/u-1).($\sum_{i=1}^{NK} \max(T_{ki}(u), u.T_{sw1}) + u.T_{swk+1}) (\sum_{i=1}^{NK} T_{ki}(R)^{-}$ + $\sum_{i=0}^{kr/2} u.T_{sw1}$ + $\sum_{i=1}^{NK} \max(T_{ki}(u), R.T_{sw1})$ +(R+u). .T swK+1
Distribution	T- -distr (u)=	$TkL_0 + \sum_{i=1}^{L-1}(T_r + T_{kL1})$, When the K-loop is split into L K sub-loops (KL..t-1), Tr is reconfiguration time
Skewing	T- -h/h (u)=	2.q. $\sum_{i=1}^{u-1}(T_{k(hw)}(i) + 2.\sum_{i=1}^{r-1}(T_{k(hw)}(i) + (N-M+1).T_{k(hw)}(r) + q.(N-u+1+r).T_{k(hw)}(u), \text{with } M = q.u + r, r < u))$

Conclusions

In this paper, we laid the foundations for the methods which are suitable for optimal implementation of formulas for computing K-loop Execution time for each loop transmission. In this paper, an algorithm has been developed that decides which of the five transformations is to use for a given K loop. The purpose is to eliminate the data dependencies over software and hardware functions and allow them to execute in parallel. The architecture is working well as per requirements.

References

[1] Alexander Aiken and Alexandru Nicolau (1988) 2*nd European Symposium on Programming*, 221-235.

[2] Alexander Aiken and Alexandru Nicolau (1990) 2*nd Workshop on Languages and Compilers for Parallel Computing*.

[3] Vicki H. Allan, Reese B. Jones, Randall M. Lee and Stephen J. Allan (1995) *ACM Computing Surveys*, 367-432.

[4] John R. Allen and Ken Kennedy (1984) *SIGPLAN Symposium on Compiler*.

[5] Yanko va Y., Kuzmanov G., Bertels K., Gaydadjiev G., Lu Y., Vassiliadis S. (2007) *Field Programmable Logic and Applications*.

[6] Dragomir O., Stefanov T.P. and Bertels K. (2009) *ACM Transactionson Reconfigurable Technology & Systems*, 2(4).

[7] Guo Z., Buykkurt B., Najjar W. and Vissers K. (2005) *Optimized Generation of Data Path from Codes for FPGA*.