Loop Distribution for K-Loops on Reconfigurable Architectures

Ozana Silvia Dragomir and Koen Bertels
Computer Engineering, EEMCS, TU Delft, The Netherlands
Email: {o.s.dragomir, k.l.m.bertels}@tudelft.nl

***Abstract*—Within the context of Reconfigurable Architectures, we define a kernel loop (K-loop) as a loop containing in the loop body one or more kernels mapped on the reconfigurable hardware. In this paper, we analyze how loop distribution can be used in the context of K-loops. We propose an algorithm for splitting K-loops that contain more than one kernel and intra-iteration dependencies. The purpose is to create smaller loops (K-sub-loops) that have more speedup potential when parallelized. Making use of partial reconfigurability, the K-sub-loops can take advantage of having more area available for multiple kernel instances to execute in parallel on the FPGA. In order to study the potential for performance improvement of using the loop distribution on K-loops, we make use of a suite of randomly generated test cases. The results show an improvement of more than 40% over previously proposed methods in more than 60% of the cases. The algorithm is also validated with a K-loop extracted from the MJPEG application. A speedup of maximum 8.22 is achieved when mapping MJPEG on VirtexIIPro with partial reconfiguration and 13.41 when statically mapping it on the Virtex-4.**

I. Introduction

Loop distribution, also known as loop fission or splitting, is a fundamental transformation used for optimizing the execution of loops. Loop distribution consists of breaking up a single loop (loop nest) into multiple loops (or loop nests), each of which iterates over distinct subsets of statements from the body of the original loop. The placement of statements into loop must preserve its data and control dependencies.

This transformation has been used traditionally for the following purposes: i) to break up a large loop that does not fit into cache ([1], [2]); ii) to improve memory locality in a loop that refers too many different arrays ([2], [3]); iii) to enable other optimization techniques such as loop fusion, loop interchange, loop permutation ([2]–[4]); iv) to eliminate dependencies in a large loop whose iterations cannot be run in parallel, by creating multiple sub-loops, some of which can be parallelized (automated vectorization).

In this paper, we address the use of loop distribution, together with previously addressed loop optimizations, on kernel loops (K-loops) in order to increase processing performance. In our research, we define a K-loop as a loop containing in the loop body one or more kernels mapped on the reconfigurable hardware. The K-loop may contain code that is not mapped on the FPGA, but will always execute on the General Purpose Processor (GPP). The performance increase comes from running multiple kernel instances in parallel on the reconfigurable hardware, while there is also the possibility of concurrently executing code on the GPP. Our framework is the Molen machine organization [5] implemented on the VirtexIIPro and Virtex-4, allowing concurrent kernels/applications execution.

978-3-9810801-7-9/DATE11/©2011 EDAA

The contributions of this paper are: a) An algorithm to split K-loops with more than one kernel such that the maximum degree of parallelism can be achieved for the K-sub-loops; b) A study of the potential for performance improvement of applying loop distribution on K-loops by randomly generated test sets that simulate large K-loops; c) Validation of our algorithm using a real world application, MJPEG.

In the field of reconfigurable computing, loop distribution has been used to break a loop into multiple tasks [6]. Loop dissevering, a method for breaking the loop into multiple tasks for temporal partitioning has been presented in [7]. Each individual task will then be mapped onto the FPGA, making it possible to implement applications that exceed the size constraint of the FPGA. Differently from our work, there was no exploitation of parallelism to increase the performance.

In [8], loop fission is applied in conjunction with loop unrolling, and across multiple loops. The advantage of considering unrolling and fission of all loops globally is that unrolled sub-loops from various loops can be potentially executed in parallel. Our work is different in several ways: a) Loop granularity – in [8], the addressed loops are the application kernels. The K-loops in our work contain the application kernels inside them. b) The mapping and scheduling strategy – in [8], different instances of the same task can be scheduled on different Processing Elements (PEs) after unrolling. In our work the opposite is true, tasks are predefined (after profiling) as software or hardware tasks. In [8], parallel execution of different PEs is considered, but in our work we also consider parallel execution on the same PE (e.g. the FPGA). c) Area reuse – in [8], (partial) reconfigurability is not considered, while in our work is.

In this paper, we generalize the work in [9] where kernel loops with only one kernel were analyzed. For these types of loops, algorithms based on loop unrolling and loop shifting were proposed. We address the general case of K-loops with more than one kernel inside that can be split into smaller K-sub-loops. The unrolling and shifting techniques are then applied to the K-sub-loops. Intra-iteration dependencies are allowed in between the kernels. Randomly generated tests prove that splitting followed by unrolling and shifting performs better than just unrolling and shifting in most of the cases. For

```
for (i = 0; i < N; i++) do
    SW1(blocks[i]) ; // sw function
    K1(blocks[i]) ; // hw mapped kernel
    SW2(blocks[i]) ; // sw function
    K2(blocks[i]) ; // hw mapped kernel
    SW3(blocks[i]) ; // sw function
```

Figure 1: K-loop with 2 kernels

the MJPEG K-loop, the proposed technique gives more than 100% improvement over unrolling and shifting.

The rest of this paper is organized as follows. In Section II, we present the motivation behind our research. Section III introduces the algorithm for loop distribution, while experimental results are presented in Section IV . Final conclusions are presented in Section V.

II. Motivation

The work in [9] focuses on simple kernel loops containing only one hardware kernel that would be accelerated on the FPGA and some software code that will always execute on the GPP. For this kind of loops, the authors proposed algorithms based on loop unrolling and loop shifting with the purpose of maximizing parallelization and performance. The purpose of this paper is to extend the optimization framework to K-loops with an arbitrary number of kernels and pieces of software code occurring in between the kernels. The number of kernels in the loop determines the K-loop size. An example of a size 2 K-loop containing two hardware mapped kernels and three software functions illustrated in Fig. 1.

The K-loop in the example contains several functions – the SW_j functions will always execute on the GPP, while the K_j functions are the application kernels that are meant to be accelerated in hardware. This K-loop can be viewed as a task chain, where we assume that there are dependencies between consecutive tasks in the chain, but not between any two tasks from different iterations. Inter-iteration dependencies can be handled with other loop transformations such as loop skewing and are not within the scope of this paper. The software functions situated before and after a kernel are called the kernel's pre- and post- functions. The loops that result from the splitting of the original K-loop are called K-sub-loops.

It is obvious that for large K-loops, unrolling and/or shifting have limitations in regard to exposing the available parallelism. In the case of loop unrolling, different kernels in the K-loop may benefit from different unroll factors, due to various factors: the kernel's I/O, the kernel's area requirements, the kernel's acceleration factor, the relation between the kernel and its pre-/post- software functions. In the case of loop shifting, only the first and the last kernels in the K-loop will execute concurrently with their pre- or post- software functions. It is proven in [9] that it is always beneficial to apply loop shifting wherever the data constraints allow it. This means that performance can be further improved if loop shifting is applied on smaller K-loops.

In this paper, we propose a method for splitting a K-loop with more than one kernel, such that maximum performance is obtained when applying unrolling and shifting to the resulted K-sub-loops. Because of intra-iteration dependencies, the order of the functions is preserved when splitting the K-loop.

```
S = Loop.ComputeSpeedup() Klist.OrderKernels()
Llist.Init()
K = Klist.First()
while (K! = NULL) do
    if (!K.added) then
        L = new Loop();
        L.AddKernel(K); L.AddSwFunc(K);
        S_temp = L.ComputeSpeedup();
        if (S_temp < S) then
            next = K.GetNext();
            prev = K.GetPrev();
            while (next && !next.added && S_temp < S) do
                L.AddKernel(next); L.AddSwFunc(next);
                next = next.GetNext();
                S_temp = L.ComputeSpeedup();
            while (prev && !prev.added && S_temp < S) do
                L.AddKernel(prev); L.AddSwFunc(prev);
                prev = prev.GetPrev();
                S_temp = L.ComputeSpeedup();
        Llist.AddLoop(L);
    K = Klist.Next();
```

Figure 2: The distribution algorithm

III. Methodology

In this section, we propose a method for splitting a K-loop containing interleaved kernels and software functions. A K-sub-loop will contain one or more kernels. The K-loop breaking points are considered to be the points between two subsequent kernels that will be placed in different K-sub-loops. Assuming that the software functions and the kernels are placed evenly within the K-loop such that between any two kernels there is a software task, a decision has to be taken whether to distribute the software task with the first kernel or with the second one. This kind of decision is needed at each breaking point of the loop.

The algorithm uses the profiling information about memory transfers, execution times for all the software and hardware functions (in GPP cycles), area requirements for kernels, memory bandwidth and available area.

The algorithm proposed in this paper for K-loops distribution is a Greedy type of algorithm, applied to the sorted list of kernels. The sorting is performed according to a heuristic based on the hardware execution time, the memory constraints and the relation with the pre-/post- software function for each kernel. The first kernel in the list is the most promising kernel in terms of speedup. The kernel speedup is defined as the ratio between its software execution time and its hardware execution time. The K-loop speedup is defined as the ratio of the K-loop execution time with all kernels running in software and the K-loop's execution time with the kernels running in hardware. The algorithm is illustrated in Fig. 2.

The sorted list of kernels is processed in a Deep First Search manner. Since each function (either kernel or software) can

belong to only one K-sub-loop, the function will be processed (added to the current K-sub-loop) only if it has not been previously selected in a K-sub-loop. The order of the functions is preserved in the distributed K-loop because of intra-iteration dependencies. We will call a function that does not belong to a K-sub-loop a 'free' function. For each free kernel in the list, the following steps are performed.

1) A K-sub-loop is created (denoted by L), containing the current kernel. The pre- and post- functions are added to L if they are free.
2) While the speedup of L is less than the speedup of the original K-loop, and while the next kernel and its pre-function are free, they are added to L.
3) While the speedup of L is less than the speedup of the original K-loop, and while the previous kernel and its post- function are free, they are added to L.
4) L is saved in the list of K-sub-loops.

The following considerations apply regarding the software functions.

1) If the first function in the K-loop is a software function, it will always be grouped with the first kernel.
2) If the last function in the K-loop is a software function, it will always be grouped with the last kernel.
3) The software functions that are called in between kernel functions will be grouped with either the previous or the following kernel function. This decision is taken at the stage of kernel ordering.

A. Reconfigurability issues

The loop distribution can be performed either considering that all kernel instances should fit on the FPGA or considering that the FPGA will be (partially) reconfigured in between K-sub-loops to accommodate the new kernels.

Of course trying to fit all kernel instances on the FPGA imposes severe area constraints, especially in the case of small FPGAs such as the VirtexIIPro-XC2VP30. This may result in poor or no improvement when parallelizing the split loop because of the limitations imposed on the unroll factors.

However, larger FPGAs such as the Virtex4-XC4VLX80 may offer enough space for configuring all the needed kernel instances at once. Then the true potential for performance improvement of loop distribution can be seen.

Reconfiguration (either partial or total) can be expensive in terms of CPU cycles. Although part of the reconfiguration time can be hidden by the K-sub-loops prologue/epilogue (that appear because of loop unrolling and loop shifting), the reconfiguration overhead might be so big that the performance decreases instead of increasing. Because of the different technologies involved, different platforms have different reconfiguration times, as follows.

a) Reconfiguration on Virtex II Pro: The Virtex-II Pro configuration memory is arranged in vertical frames, that are the smallest addressable unit; therefore, all operations must act on whole configuration frames [10].

Configuring the entire device takes approx. 20ms. However, Virtex-II Pro devices allow partial reconfiguration – rewriting a subset of configuration frames. We consider that the time needed to reconfigure only a part of the device is proportional to the size of the module to be configured.

b) Reconfiguration on Virtex-4: The configuration architecture for the Virtex-4 FPGA family has a different layout than the earlier families, but it is still frame-based [11]. The number of frames that need to be configured determines the size of the reconfiguration bitstream. There is also a configuration overhead of 1312 words. In order to compute the configuration time, the throughput must also be known.

The idealized throughput on the Virtex-4 chip is 400MB/s, at 100MHz. The work of Liu et al [12] proposed a DMA engine design that yields a throughput of 399MB/s. Considering that the size of a frame is 41 words, the reconfiguration time is $0.411\mu s$/frame. When adding the overhead, the total reconfiguration time is

$$T_{reconfig} = \text{Number\_of\_frames} * 0.411 + 13.15(\mu s) \quad (1)$$

IV. Experimental results

In order to study the potential for performance improvement of distributing K-loops, a random test generator has been developed. The results of applying the distribution algorithm to the randomly generated tests are shown in Subsection IV-A. The algorithm is also validated with a K-loop extracted from the MJPEG application in Subsection IV-B.

A. Distribution on random K-loops

A suite of randomly generated tests has been used in order to study the impact of loop distribution on K-loops with sizes between 2 and 8. For each loop size, 1500 tests have been generated. For each test we compare the performance improvement when distributing the K-loop using the above algorithm and applying unrolling and shifting to the resulted K-sub-loops to the performance achieved by previously presented methods, based only on unrolling and shifting. For these experiments, the reconfiguration latency is not taken into account.

Each test case is determined by the following parameters:

- N - the number of iterations;
- $T_{\text{sw}}[j]$ - the execution time for each software function $\texttt{SW}_j$ (cycles);
- `MAX(sw)` - the maximum of the execution times of the software functions (cycles);
- $T_{K(\text{sw})}[i]$ - the execution time in software for each kernel $\texttt{K}_i$ (cycles);
- $S[i]$ - the speedup for $\texttt{K}_i$ kernel ($S[i] \in \mathbb{R}$);
- $T_{K(\text{hw})}[i]$ - the execution time in hardware for $\texttt{K}_i$ (cycles):

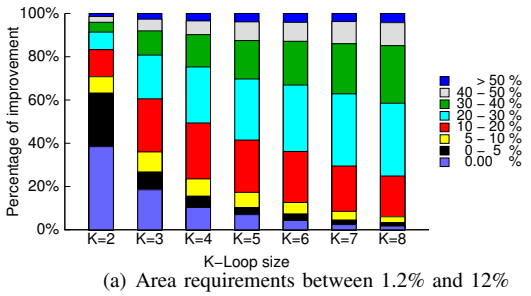
$$T_{K(\text{hw})}[i] * S[i] = T_{K(\text{sw})}[i]$$

- $A[i]$ - the area occupied by K_i in hardware, in percents:

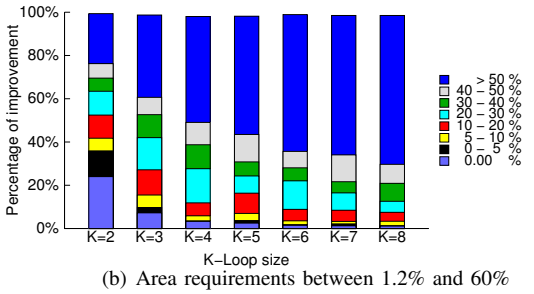
$$\sum(A[i]) \leq 90\%, (A[i] \in \mathbb{R});$$

- $T_{\text{r}}[i]$, $T_{\text{w}}[i]$, $T_{\text{c}}[i]$ - the memory read time, memory write time and computation time for $\texttt{K}_i$ running in hardware, respectively (cycles).

$$T_{\text{r}}[i] + T_{\text{w}}[i] + T_{\text{c}}[i] = T_{K(\text{hw})}[i];$$



(a) Area requirements between 1.2% and 12%



(b) Area requirements between 1.2% and 60%

Figure 3: Performance distribution for the random tests

Table I
PARAMETER VALUES

Parameter	Min. value	Max. value
N	64	256*256
$T_{\text{SW}}[j]$	0	800
$T_{K(\text{SW})}[i]$	1.5 * MAX(SW)	41.5 * MAX(SW)
$S[i]$	2.0	10.0
$A[i]$	1.2%	12.0% / 60%
$T_{\text{R}}[i]$	1	$\frac{1}{3} * T_{K(\text{HW})}[i]$
$T_{\text{W}}[i]$	1	$\frac{1}{6} * T_{K(\text{HW})}[i]$

The values for the presented parameters have been generated according to the Table I. Note that in some cases, the maximum value of a parameter depends on the generated value of another parameter (for instance, the kernel time for read is at most one third of the total execution time of the kernel in hardware, otherwise the I/O intensive kernel would not be a candidate for hardware acceleration).

Parallel hardware execution (enabled by loop unrolling) is a great source of performance improvement. The kernels' area requirements influence the maximum degree of parrallelism, thus the performance. To study the impact of the area requirements, two sets of tests with K-loop sizes between 2 and 8 were used. For the first set, all kernels have small area requirements (1.2 – 12%), that allows for a parallelism degree of 7 or more. The results are illustrated in Fig. 3(a). For the second set, the maximum area requirement for a kernel is 60%, that can lead to no hardware parallelism. All kernels in a K-loop should fit altogether on the available area, therefore the sum of all areas should be less than the maximum allowed (in our tests, this maximum is 90%, in order to avoid routing issues). The results are illustrated in Fig. 3(b).

In Fig. 3 a) and b) we illustrate the performance improvement of the loop distribution algorithm for different K-loop sizes, compared to the previously presented methods based on unrolling and shifting. For Fig. 3(a) the set of tests was generated with small area requirements(SAR) for each kernel, between 1.2% and 12%. In Fig. 3(b), the displayed results are for the case of larger area requirements (LAR), maximum 60% per kernel. The distribution algorithm brings no improvement for K-loops of size 2 for 38% of the SAR cases and 24% of the LAR cases, but these numbers decrease with the K-loop size, down to less than 2% for K-loop size 8. Little performance improvement (less than 10%) is shown for 11.2% of the SAR cases and 3.6% of the LAR cases. An improvement of 10–20% is shown for 23% of the SAR cases and 7% of the LAR cases. A performance gain between 20–40% is shown for most of the SAR cases (47.6%) and 19% of the LAR cases. A significant performance improvement of more than 40% is shown for 12% of the SAR cases and most of the LAR cases (67.3%).

The results show that the area requirements have a great impact on performance when loop distribution is taken into account. When the K-loop contains kernels with high area requirements, it is beneficial to split it into smaller K-sub-loops. This way, the smaller kernels can benefit from higher unroll factors.

Also, a large K-loop size is a good reason for splitting the loop. Having small K-sub-loops brings more benefit from loop shifting, when the software and hardware execute concurrently.

B. Distribution on the MJPEG K-loop

The MJPEG main loop is illustrated in Fig. 4. The data transmitted from task to task is a 8×8 pixel block. All functions need to perform the reading and writing of each pixel in the data block. The profiling information for all the functions in the MJPEG K-loop is presented in Table II. The software and hardware execution times for VLE that are presented in the profiling information are the average of the execution times of several instances of the VLE kernel. All other functions have constant execution time.

The application kernels are the DCT, Quantizer and VLE functions. Although the ZigZag function has a quite large execution time, it is a software only function as it is I/O intensive and implementing it in hardware would not be beneficial.

The compiler used to compile the MJPEG K-loop is based on GCC 4.3.1 and extended to support the MOLEN programming paradigm [5]. The VHDL code for the all kernels has been automatically generated with the DWARV [13] tool and synthesized using Xilinx XST tool of ISE 11.2 for the Xilinx VirtexIIPro-XC2VP30 and Virtex4-XC4VLX80 boards. The two chips' characteristics are complementary, making them suitable for illustrating the proposed method – The VirtexIIPro has small area and high reconfiguration time, while the Virtex-4 has large area and small reconfiguration time. The synthesis results for the MJPEG kernels are presented in Table III. The code was executed only on the Virtex II Pro-XC2VP30 board and the execution times were measured using the PowerPC timer registers.

```
for (i = 0; i < 16; i++) do
    for (j = 0; j < 8; j++) do
        PreShift(blocks[i][j]);
        DCT(blocks[i][j]);
        Bound(blocks[i][j]);
        Quantizer(blocks[i][j]);
        ZigZag(blocks[i][j]);
        VLE(blocks[i][j]);
```

Figure 4: MJPEG main K-loop

Table II
PROFILING INFORMATION FOR THE MJPEG FUNCTIONS

Kernel	$T_{K(\text{sw})}$(cycles)	$T_{K(\text{hw})}$(cycles)
PreShift	3096	–
DCT	106626	37278
Bound	1953	–
Quantizer	12510	2862
ZigZag	8112	–
VLE	51378	6252

In the remaining text, we will use the notation u for the unroll factor used for the whole K-loop. The u_i notations are used when the K-loop is split into several K-sub-loops and each u_i corresponds to the i-th kernel in the original loop. Then, u_1 is the unroll factor for DCT, u_2 is the unroll factor for Quantizer and u_3 is the unroll factor for VLE.

When the MJPEG K-loop is parallelized with loop unrolling or loop unrolling + shifting for parallelization, the same parallelization factor applies to all kernels (namely, the unroll factor). All kernel instances must be configured at the same time, no reconfiguration is considered. The following performance results can be obtained with unrolling and shifting:

- A maximum speedup of 12.93 for unroll factor $u = 28$ when there are no area constraints (unlimited area).
- A speedup of 3.56 for the VirtexIIPro chip. No unrolling can be performed because of the limited area ($u = 1$).
- A speedup of 5.88 for Virtex-4. The area constraints require that the maximum unroll factor is $u = 2$.

The loop distribution algorithm decides to partition the K-loop into two K-sub-loops. The K-sub-loops ($KL1$ and $KL2$) will be:

- $KL1$ = PreShift, DCT, Bound
- $KL2$ = Quantizer, ZigZag and VLE

The two K-sub-loops have now different optimal parallelization factors. For $KL1$, the optimal factor (assuming no area constraints) is $u_1 = 20$, leading to a speedup of 22.11 for $KL1$. For $KL2$, the optimal factor is $u_2 = u_3 = 1$, with speedup of 8.23. The reason for this optimal unroll factor follows. Performing loop shifting allows the software functions to execute in parallel with the hardware kernel. Since the execution time of the software will then be approximately the same as the execution time of the hardware VLE, there is no gain in unrolling in the case of $KL2$.

Table III
SYNTHESIS RESULTS FOR THE MJPEG KERNELS

Kernel	Area (slices)	% VirtexIIPro	%Virtex4
DCT	1692	12.35	4.72
Quantizer	409	2.98	1.14
VLE	10631	77.62	29.66

The K-loop speedup depends now on whether we choose to map all the needed kernel instances on the FPGA (no reconfigurability) or map the K-sub-loops kernels separately (partial reconfigurability before $KL2$).

Loop distribution with no reconfigurability.

No area constraints. The K-loop speedup for the optimal factors $u_1 = 20$, $u_2 = 1$ is 13.53.

The VirtexIIPro area constraints allow no unrolling for either $KL1$ or $KL2$, therefore applying loop distribution would bring no improvement. The speedup is 3.56.

The Virtex-4 area constraints allow a maximum unroll factor $u_1 = 13$ for $KL1$, while $KL2$ does not benefit from unrolling, hence the optimal factor is 1 ($u_2 = u_3 = 1$). The speedup for $KL1$ will be 21.65, while the K-loop speedup after distribution is 13.41. The final unroll factors are (u_1, u_2, u_3) = (13, 1, 1). Compared to the speedup of 5.88 when loop distribution is not used on Virtex-4, this is a 228% performance improvement.

Loop distribution with partial reconfigurability.

It makes no sense to consider partial reconfigurability with no area constraints, therefore we will analyze only the VirtexIIPro and Virtex-4 cases.

The VirtexIIPro area constraints allow for a maximum of 7 DCT kernels to be configured at a time (as mentioned before, we aim at occupying no more than 90% of the area). If reconfiguration overhead is not taken into account, the speedup for the K-loop with unroll factors $(7, 1, 1)$ is 12.71.

The reconfiguration latency is not negligible in this case. The VLE kernel from $KL2$ will need to be configured after the DCTs have finished. Since the Quantizer takes only 2.98% of the area and we are willing to minimize the reconfiguration time, the Quantizer can be configured before the end of $KL1$, since the DCTs occupy only 86.45% of the total area.

Considering the reconfiguration time of 19.39ms for the whole board, it would take 14.93ms to map VLE onto the FPGA. This means 985400 cycles at the recommended frequency of 66MHz. The maximum VLE reconfiguration latency that can be hidden by the epilogue of $KL1$ and prologue of $KL2$ and that is equal to only 14880 cycles. The MJPEG speedup increases when using loop distribution with partial reconfiguration from 3.56 to 8.22.

The Virtex-4 area constraints impose an unroll factor $u_1 = 18$ for $KL1$. Ignoring the reconfiguration overhead, the speedup for the K-loop with unroll factors $(18, 1, 1)$ is 13.51. Similar to the case of VirtexIIPro, the Quantizer can be configured before the end of $KL1$ and the VLE kernel will need to be configured after the DCTs have finished. The maximum VLE reconfiguration latency that can be hidden by the epilogue of $KL1$ and prologue of $KL2$ and that is equal to 28551 cycles. It differs from the one computed for VirtexIIPro because the $KL1$ epilogue is larger for $u_1 = 13$ (Virtex-4) than for $u_1 = 7$ (VirtexIIPro).

Table IV
PERFORMANCE RESULTS FOR DIFFERENT AREA CONSTRAINTS AND DIFFERENT METHODS ([1]-LOOP UNROLLING; [2]-LOOP SHIFTING; [3]-LOOP DISTRIBUTION).

Area constraints	(u_1,u_2,u_3)	methods	reconfig.	Speedup
∞	(28,28,28)	[1]+[2]	no	12.93
∞	(20, 1, 1)	[3]+[1]+[2]	no	13.53
VirtexIIPro	(1, 1, 1)	[1]+[2]	no	3.56
VirtexIIPro	(1, 1, 1)	[3]+[1]+[2]	no	3.56
VirtexIIPro	(7, 1, 1)	[3]+[1]+[2]	yes	8.22
Virtex-4	(2, 2, 2)	[1]+[2]	no	5.88
Virtex-4	(13, 1, 1)	[3]+[1]+[2]	no	13.41
Virtex-4	(18, 1, 1)	[3]+[1]+[2]	yes	11.95

To load the VLE on Virtex-4, a total of 883.9KB need to be transfered on the FPGA. The reconfiguration time of the VLE kernel computed with (1) is 2.21ms at 100MHz, that is the equivalent of 221000 CPU cycles. The K-loop speedup with partial reconfiguration is 11.95, less than the speedup achieved with static mapping of the kernels on the Virtex-4.

Table IV summarizes the results of mapping the MJPEG K-loop onto reconfigurable hardware, using the presented techniques and various area constraints. Note that ∞ means no area constraints.

The results on VirtexIIPro show that loop distribution with partial reconfiguration gives a significant performance improvement when the area is very restrictive (speedup increase from 3.56 to 8.22). Partial reconfiguration allows for a higher unroll factor in the first K-sub-loop and thus more parallelism. If the reconfiguration latency could be decreased, the performance would improve with maximum 54% (up to 12.71 speedup on VirtexIIPro).

The results of mapping MJPEG on the Virtex-4 show that loop distribution with no reconfigurability gives the best performance when the area is not a big constraint. The performance increased by 228%(from 5.88 to 13.41), that is 99% of the maximum speedup (13.53) that is possible with infinite resources.

V. CONCLUSION

In this paper, we discussed the use of loop distribution in the context of K-loops. When the kernels from a large K-loop are distributed into smaller K-sub-loops, they can benefit from different unroll factors and therefore more from the parallelism enabled by unrolling and shifting.

The results of experimental data that was randomly generated show that the K-loop size and the area constraints have a high impact on the performance. Splitting is very beneficial for large K-loops and for K-loops that contain kernels with high area requirements.

In more than 60% of the random test cases, distributing the K-loop lead to a significant performance improvement (more than 40%) compared to when loop unrolling and shifting are applyed to the K-loop without splitting it. Note that the partial reconfiguration overhead was not taken into account for the randomly generated tests because they were not designed for any platform in particular. Since Virtex-4 family chips have a small reconfiguration delay, we consider that for our tests this delay can be hidden by the K-sub-loops prologue and epilogue (resulted from loop unrolling and loop shifting).

The distribution algorithm proposed in this paper has been validated with the MJPEG K-loop. Our method proves to be efficient in both configurations: together with partial reconfiguration on small chips (the VirtexIIPro), as well as without reconfiguration on large chips (the Virtex-4).

The performance of the MJPEG K-loop on VirtexIIPro improved from 3.56 speedup without loop distribution to 8.22 speedup with loop distribution and partial reconfiguration in between the K-sub-loops.

On the Virtex-4, the best performance is achieved with loop distribution and static mapping of the kernels on the reconfigurable hardware (speedup of 13.41). This is a performance increase of more than 100% compared to the speedup of 5.88 when the MJPEG K-loop is parallelized only by unrolling and shifting.

ACKNOWLEDGMENT

This research is partially supported by Artemisia iFEST project (grant 100203), Artemisia SMECY (grant 100230), FP7 Reflect (grant 248976).

REFERENCES

[1] K. Kennedy and K. S. McKinley, "Loop Distribution with Arbitrary Control Flow," in *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*. Los Alamitos, CA, USA, 1990, pp. 407–416.

[2] ——, "Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution," in *Workshop on Languages and Compilers for Parallel Computing*, vol. 768, 1993, pp. 301–320.

[3] K. S. McKinley, S. Carr, and C.-W. Tseng, "Improving Data Locality with Loop Transformations," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 4, pp. 424–453, 1996.

[4] M. Liu, Q. Zhuge, Z. Shao, C. Xue, M. Qiu, and E. H.-M. Sha, "Maximum Loop Distribution and Fusion for Two-level Loops Considering Code Size," *Parallel Architectures, Algorithms, and Networks, International Symposium on*, vol. 0, pp. 126–131, 2005.

[5] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The MOLEN Polymorphic Processor," *IEEE Transactions on Computers*, November 2004.

[6] M. Kaul, R. Vemuri, S. Govindarajan, and I. Ouaiss, "An Automated Temporal Partitioning and Loop Fission Approach for FPGA Based Reconfigurable Synthesis of DSP Applications," in *DAC 1999*. New York, NY, USA: ACM, 1999, pp. 616–622.

[7] J. M. P. Cardoso, "Loop Dissevering: A Technique for Temporally Partitioning Loops in Dynamically Reconfigurable Computing Platforms," in *IPDPS 2003*. Washington, DC, USA, 2003, pp. 22–26..

[8] Y. M. Lam, J. G. F. Coutinho, C. H. Ho, P. H. W. Leong, and W. Luk, "Multiloop Parallelisation Using Unrolling and Fission," *International Journal of Reconfigurable Computing*, 2010.

[9] O. S.Dragomir, T. P. Stefanov, and K. Bertels, "Optimal Loop Unrolling and Shifting for Reconfigurable Architectures," *ACM Transactions on Reconfigurable Technology and Systems*, 2009.

[10] Xilinx Inc., "Virtex-II Pro and Virtex-II Pro X FPGA User Guide." [Online].

[11] ——, "Virtex-4 FPGA User Guide." [Online].

[12] S. Liu, R. N. Pittman, and A. Forin, "Energy Reduction with Run-Time Partial Reconfiguration (abstract only)," in *FPGA 2010*, New York, NY, USA: ACM, 2010, pp. 292–292.

[13] Y. Yankova, G. Kuzmanov, K. Bertels, G. Gaydadjiev, Y. Lu, and S. Vassiliadis, "DWARV: DelftWorkbench Automated Reconfigurable VHDL Generator," *FPL2007*, Amsterdam, Netherlands,2007.