# K-Loops: Loop Skewing for Reconfigurable Architectures

Ozana Silvia Dragomir and Koen Bertels

*Computer Engineering, EEMCS, TU Delft, The Netherlands*

{O.S.Dragomir, K.L.M.Bertels}@tudelft.nl

***Abstract*—In this paper, we propose new techniques for improving the performance of applications running on a reconfigurable platform supporting the Molen programming paradigm. We focus on parallelizing loops that contain hardware-mapped kernels in the loop body (called K-loops) with wavefront-like dependencies. For this purpose, we use traditional transformations, such as loop skewing for eliminating the dependencies and loop unrolling for parallelization. The first technique presented in this paper improves the application performance by running in parallel on the reconfigurable hardware multiple instances of the kernel. The second technique extends the first one and determines how many kernel instances should be scheduled for software execution in each iteration, concurrently with the hardware execution, such that the hardware and software times are balanced. In the experimental part, we present results when parallelizing the Deblocking Filter (DF), which is part of the H.264 encoder and decoder, after skewing the main DF loop to eliminate the data dependencies. For the unroll factor 8, we report a loop speedup of up to 4.78.**

## I. Introduction

Reconfigurable Computing is an increasingly popular approach to increase application performance, as it combines the flexibility of the General Purpose Processor (GPP) with the speed of the (reconfigurable) hardware. The common solution is to identify the application kernels and to accelerate them in hardware. Loops are an important source of performance improvement, as they represent or include kernels of modern real-life applications (audio, video, image processing, etc).

In our research, we focus on parallelizing loops that contain hardware-mapped kernels in the loop body (called K-loops). Assuming the Molen machine organization [1] as our framework, we aim at improving application performance by running multiple kernel instances in parallel on the reconfigurable hardware, while there is also the possibility of concurrently executing code on the General Purpose Processor (GPP).

In this paper, we address the case when data dependencies that prevent parallelization by simple loop unrolling can be eliminated by transforming the loop body using loop skewing. The contributions of this paper are: a) a technique for parallelizing K-loops with wavefront-like dependencies, running all kernel instances on the reconfigurable hardware; b) a technique for parallelizing K-loops with wavefront-like dependencies, running part of the kernel instances in hardware and part in software; c) experimental results for the Deblocking Filter (DF) K-loop, part of the H.264 encoder/decoder. The techniques are based on profiling information about area consumption, memory transfers and execution times. Since the DF kernel execution time is not constant, in the experimental part we take into account the average time and the maximum time of a number of kernel executions when applying the presented techniques. For the unroll factor 8, we report a loop speedup for different picture sizes between 3.47 and 4.4, when the average execution time is considered. When the maximum kernel execution time is considered, the speedup for the unroll factor 8 is between 3.77 and 4.78.

This work is supported by the FP6 EU project hArtes, with integrated project number 035143

The rest of this paper is organized as follows. Section II introduces the framework and background work, while a comparison with related work is performed in Section III. Section IV presents the problem statement and Section V presents the proposed methodology. Experimental results are shown in Section VI, while we draw the final conclusions in Section VII.

## II. Background (the framework)

The work presented in this paper is related to the Delft Workbench(DWB)[1] project. The DWB is a semi-automatic toolchain platform in the context of Custom Computing Machines (CCM) which targets the Molen polymorphic machine organization [1], supporting the entire design process. The Molen framework allows multiple kernels/applications to run simultaneously on the reconfigurable hardware. The architecture is based on the tightly coupled processor co-processor paradigm, where a general purpose core processor (GPP) controls the execution and reconfiguration of a reconfigurable co-processor. The Molen machine organization has been implemented on Virtex-II Pro device [2].

In the preliminary stage of profiling and code estimation the application kernels are identified. Note that in this context, a **kernel** is a part of the application where a large percentage of the execution time is spent. Also, the kernel itself has to be quite large in order to justify the hardware mapping, because the transfer of the parameters and any other memory reads are expensive in terms of execution time.

The source code is then transformed such that the application kernels reside in independent functions which are annotated with specific pragma directives needed by the compiler. The purpose is to speed up the application execution by mapping and executing the identified kernels on the reconfigurable hardware.

[1]http://ce.et.tudelft.nl/DWB/

Then, the Molen compiler identifies functions annotated with specific *pragma* directives and replaces the function calls with associated `set/execute` pseudo-functions. Based on the Architecture Description File, the compiler generates the executable file, replacing and scheduling function calls to the kernels implemented in hardware with specific instructions for hardware reconfiguration and execution, according to the Molen programming paradigm [3]. The Molen compiler is based on the GCC compiler for PowerPC.

The DWARV automatic hardware generator [4] is used to transform the selected kernels into VHDL code targeting the Molen platform. The automated code generation is envisioned for fast prototyping and fast performance estimation during the design space exploration.

```
for (j=0; j<N; j++){
  for (i=0; i<M; i++)
    A(i,j) = F(A(i-1,j),A(i,j-1));
}
```

a) loop before skewing

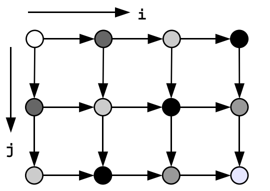
```
for (t=1; t<M+2*N; t++){
  pmin = max(t%2, t-2*N+2);
  pmax = min(t, M-1);
  for (p=pmin; p<=pmax; p+=2){
    i = p;
    j = (t-p)/2;
    A(i,j) = F(A(i-1,j),A(i,j-1));
  }
}
```

b) loop after skewing

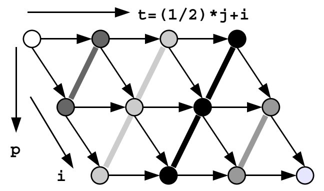
Fig. 1. Loop with wavefront-like dependencies.

## A. Loop skewing

Loop skewing is a widely used loop transformation for wavefront-like computations [5], [6]. Consider the example in Fig. 1a). In order to compute the element $A(i,\ j)$ in each iteration of the inner loop, the previous iteration's results $A(i-1,\ j)$ must be available already. Therefore, this code cannot be parallelized or pipelined as it is currently written. 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 $j$, we obtain the "skewed" loop in Fig. 1b). The iteration space and dependencies for the original and skewed loops are showed in Fig. 2 a) and b), respectively.



a) The dependencies for the loop before skewing



b) The dependencies for the loop after skewing

Fig. 2. Loop dependencies (different shades of gray show the elements that can be executed in parallel).

## III. Related work

Several research projects develop C to VHDL frameworks, trying to exploit as much as possible the advantages of reconfigurable systems by maximizing the parallelism in targeted applications and accelerating kernel loops in hardware. Directly connected to our work are [7], [8] and [9], where hardware is generated after optimizing the kernel loops.

The work of Guo et al. [7] is part of the ROCCC – C to hardware compilation project whose objective is the FPGA-based acceleration of frequently executed code segments (loop nests). The ROCCC compiler applies loop unrolling, fusion and strip mining and creates pipelines for the unrolled loops in order to efficiently use the available area and memory bandwidth of the reconfigurable device.

Weinhardt and Luk [8] introduce pipeline vectorization, a method for synthesizing hardware pipelines based on software vectorizing compilers. In their approach, full loop unrolling, as well as loop tiling and loop merging are used to increase basic block size and extend the scope of local optimizations.

The work of Gupta et al.[9] is part of the SPARK project and uses shifting to expose loop parallelism and then to compact the loop by scheduling multiple operations to execute in parallel. In this case, loop shifting is performed at low level, unlike us who perform it at high (functional) level. The shifted loops are scheduled and mapped on the hardware, as in the case of the projects presented previously.

COMPAAN [10], [11] is a compiler (method and a tool set) for transforming affine nested loops programs written in Matlab into a Kahn Process Network specification. The transformations supported by the Compaan compiler are: unfolding, plane cutting, skewing and merging. Compaan is used in conjunction with the Laura [12] tool, which operates as a back-end for the compiler. Laura maps a KPN specification onto hardware, for example, FPGAs.

Our approach is different than the ones above, as we do not aggressively optimize the kernel implementation to improve the application's performance, but work at high (functional) level. Our approach is to speedup the application by executing multiple kernel instances in parallel. The benefit of this approach is that it improves the performance irrespective of the kernel's hardware implementation.

In this paper, we extend the work in [13], where loop unrolling and loop shifting were considered for parallelizing loops by executing multiple instances of a kernel concurrently. Parallelizing with loop unrolling and/or shifting imposed several constraints regarding the inter- and intra-iterations data dependencies of the loops. In our current work, we relax some of these constraints and thus we can address a wider range of applications. We analyze loops with wavefront-like dependencies and break these dependencies by applying loop skewing. Although the parallelization is performed using the loop unrolling technique, the skewed loop will no longer be a rectangular loop as in the previous cases[2]. New formulas are therefore needed to compute the estimated speedup. In addition, a new executional strategy is proposed, exploring more the architectural capabilities. It assumes running part of the kernel instances in software, in parallel with the kernel instances that run concurrently in the hardware.

## IV. Problem statement

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. The loop may contain also code that is not mapped on the FPGA, but will always execute on the GPP (in software). The software code and the hardware kernels may appear in any order in the loop body.

```
for (i=0; i<N; i++) {
  /* Function that executes always on the GPP */
  do_SW (blocks, i, ...);
  /* Kernel function */
  K (blocks, i, ...);
}
```

Fig. 3. K-loop with one hardware mapped kernel.

A simple example of a K-loop is illustrated in Fig. 3. One challenge we address in our work is to improve the performance for such loops by applying standard loop transformations to maximize the parallelism inside the loop.

A number of loop transformations (such as loop unrolling, software pipelining, loop shifting, loop distribution, loop merging, or loop skewing) can be used successfully to maximize the parallelism inside the loop and improve the performance. In previous work [13], the effects of loop unrolling and loop shifting on K-loops containing well known multimedia kernels: DCT, SAD, Quantizer, convolution (Sobel) were studied. This work imposed several constraints regarding the inter- and intra-iterations data dependencies for the K-loop, which are the following. In order to be able to apply loop unrolling and run in parallel multiple instances of the kernel, data dependencies between K($i$) and K($j$), for any iterations $i$ and $j$, $i \neq j$, may not exist. In order to perform loop shifting and then concurrently execute the code on the GPP and on the FPGA, one more constraint needs to be satisfied: there should be no data dependencies between do_SW($i$) and K($j$), for any iterations $i$ and $j$, $i \neq j$.

In this paper, we relax the data dependencies constraints and analyze the impact of loop skewing on accelerating K-loops. The problem statement is the following: for a skewed K-loop containing a kernel K, find the optimal unroll factor $u$ which maximizes the performance, such that a maximum of $u$ identical instances of K run in parallel on the reconfigurable hardware. Note that the innermost loop of a skewed loop has a variable number of iterations, depending on the value of the outer index variable.

The method proposed in this paper addresses this problem, given a C implementation of the target application and a VHDL implementation of the kernel. Our algorithm computes at compile time the optimal unroll factor, taking into consideration the memory transfers, the execution times in software and hardware, the area requirements for the kernel, and the available area (we assume no constraints regarding the placement of the kernel). Note that in this paper we consider that the execution time in hardware may vary for different kernel instances. Our assumptions regarding the application and the framework are summarized in Table I.

[2]In Fig. 5 from the 'Methodology' section we illustrate how the number of iterations of the inner skewed loop varies.

TABLE I
General and Molen-specific assumptions

| |
|---|
| **Loop nest**<br>⋆ loop bounds are known at compile time; |
| **Memory accesses**<br>⋆ memory reads in the beginning, memory writes in the end;<br>⋆ on-chip memory shared by the GPP and the Custom Computing Units (CCUs) is used for program data;<br>⋆ all necessary data are available in the shared memory;<br>⋆ all transactions on shared memory are performed sequentially;<br>⋆ kernel's local data are stored in the FPGA's local memory, not in the shared memory; |
| **Area & placement**<br>⋆ shape of design is not considered;<br>⋆ placement is decided by a scheduling algorithm such that the configuration latency is hidden;<br>⋆ interconnection area needed for CCUs grows linearly with the number of kernels. |

```
for (j=0; j<N; j++) {
  for (i=0; i<M; i++) {
    /* Function that executes always  on the GPP */
    do_SW (blocks, i, j...);
    /* Kernel function */
    K (blocks, i, j, i-1, j-1...);
  }
}
```

Fig. 4. K-loop with one hardware mapped kernel and inter-iteration data dependencies.

*Motivational example:* Throughout the paper, we will use the motivational example in Fig. 4. It consists of a K-loop with two functions: do_SW — which is executed always on the GPP — and K, which is the application's kernel and will be executed on the reconfigurable hardware. Implicitly, the execution time for do_SW is smaller than the execution time of K on the GPP.

We assume that in each iteration $(i,\ j)$ data pointed by $(blocks, i, j)$ are updated based on data previously computed in iterations $(i-1,\ *)$ and $(*,\ j-1)$. Thus there are data dependencies between instances of K in different iterations, similar to the dependencies shown in Fig. 2a).

## V. Methodology

In this section, we present techniques that can be applied to K-loops that cannot be parallelized directly because of data dependencies. Using the loop skewing transformation

TABLE II
NOTATIONS

| | |
|---|---|
| $M$, $N$ | the dimensions of the initial loop (before skewing); |
| $T_{\text{sw}}$ | number of cycles for one instance of the software function (the function that is always executed by the GPP — in our example, the `do_SW` function); |
| $T_{K(\text{sw})}/T_{K(\text{hw})}$ | number of cycles for K() running in software/hardware (average of several instances or maximum, depending on the case); |
| $T_{K(\text{hw})}(u)$ | number of cycles for $u$ instances of K() running in hardware, defined in (3) (the case $u \le \text{u}_\text{m}$); |
| $T_{\text{loop(sw)}}$ | number of cycles for the loop nest executed completely in software; |
| $T_{\text{it}}(t)$ | the hardware execution time for an iteration $t$, assuming the unroll factor $u$; |
| $T_{\text{h/h}}(u)/T_{\text{h/s}}(u)$ | the hardware execution time for the skewed loop when all/part of the kernels instances are running in hardware, assuming the unroll factor $u$; |
| $S_{\text{h/h}}(u)/S_{\text{h/s}}(u)$ | the speedup for the skewed loop when all/part of the kernels instances are running in hardware, assuming the unroll factor $u$. |

presented in Section II, the data dependencies are eliminated. For the resulted loop, we show how the optimal unroll factor is computed and what is the estimated speedup.

The unroll factor depends on the area, memory transfers and execution times for the software/hardware functions. The area bound for the unroll factor is pretty straightforward, as you cannot run in parallel more kernels than they fit on the available FPGA area.

*Memory accesses:* Performance increases until the computation is fully overlapped by the memory transfers performed by the kernel instances running in parallel and we denote by $\text{u}_\text{m}$ the unroll factor where this case happens. A very detailed explanation of how the memory bound $\text{u}_\text{m}$ is computed is provided in [13]. We consider that $T_\text{r}$, $T_\text{w}$ and $T_\text{c}$ are, respectively, the times for memory read, write, and computation on hardware for kernel K, as indicated by the profiling information. Using the notations:

$$T_{\text{min(r,w)}} = \min\left(T_\text{r}, T_\text{w}\right);\ T_{\text{max(r,w)}} = \max\left(T_\text{r}, T_\text{w}\right), \quad (1)$$

the memory bound is derived as:

$$u \le \text{u}_\text{m} = \left\lfloor \frac{T_\text{c}}{T_{\text{min(r,w)}}} \right\rfloor + 1 \quad (2)$$

Then, the time for running $u$ instances of $K$ on the reconfigurable hardware is:

$$T_{K(\text{hw})}(u) = \begin{cases} T_\text{c} + T_{\text{min(r,w)}} + u \cdot T_{\text{max(r,w)}}, & \text{if } u \le \text{u}_\text{m} \\ u \cdot (T_\text{r} + T_\text{w}), & \text{if } u > \text{u}_\text{m} \end{cases} \quad (3)$$

*Speedup:* We use the notations presented in Table II. The loop in our motivational example has the same iteration domain as the loop in Fig. 1a), therefore after skewing the iteration domain will be the same as in Fig. 1b). The outer loop will have $N+M-1$ iterations. For simplicity, we assume that $M \le N$. The number of iterations of the inner loop ($N_{\text{it}}$) varies according to Fig. 5. The maximum number of iterations of the inner loop is $\min(M,N) = M$ and there are $(N - M + 1)$ such iterations. This means that $M$ is the maximum available parallelism. A mathematical formulation for $N_{\text{it}}$ is:

$$N_{\text{it}}(t) = \begin{cases} M, & M \le t \le N \\ t, & 1 \le t < M \text{ or } N < t < M + N \end{cases} \quad (4)$$

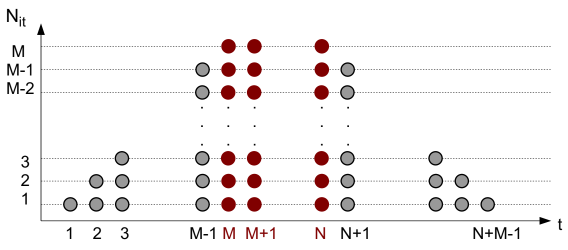
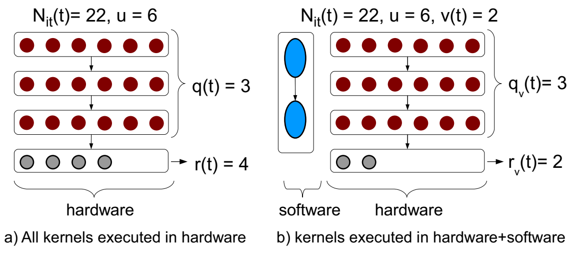


Fig. 5. The number of iterations of the inner skewed loop.



N_it(t)= 22, u = 6
q(t) = 3
r(t) = 4
hardware
a) All kernels executed in hardware
N_it(t) = 22, u = 6, v(t) = 2
q_v(t)= 3
r_v(t)= 2
software
hardware
b) kernels executed in hardware+software

Fig. 6. Kernels'execution pattern in an iteration.

Next, we compute the hardware execution time for the skewed loop assuming the unroll factor $u$ as the sum of the execution times in each outer iteration $t$. First, we consider the method with all kernel instances running in hardware and second, the method with part of the kernels running in software. Note that the time to execute the loop nest completely in software ($T_{\text{loop(sw)}}$) does not depend on the unroll factor:

$$T_{\text{loop(sw)}} = (T_{\text{sw}} + T_{K(\text{sw})}) \cdot M \cdot N \quad (5)$$

*a)* **With all kernels in hardware:** The hardware execution time for the skewed loop with all kernel instances running in hardware when unrolling with factor $u$ is:

$$T_{\text{h/h}}(u) = \sum_{t=1}^{M+N-1} T_{\text{it}}(t) \quad (6)$$

$T_{\text{it}}(t)$ represents the execution time of the $N_{\text{it}}(t)$ kernels in the iteration $t$. Considering $N_{\text{it}}(t)$ from (4), $T_{\text{h/h}}(u)$ becomes:

$$T_{\text{h/h}}(u) = 2 \cdot \sum_{t=1}^{M-1} T_{\text{it}}(t) + (N - M + 1) \cdot T_{\text{it}}(M) \quad (7)$$

After skewing, the inner loop can be parallelized to improve the performance. Depending on the relation between $N_{\text{it}}(t)$, $\text{u}_\text{m}$ and $u$, the $N_{\text{it}}(t)$ kernels in outer iteration $t$ should execute in groups of $u$ concurrent kernels and not all kernels in parallel. The groups execute sequentially, as depicted in Fig. 6a). The number of groups with $u$ kernels is the quotient of the division of $N_{\text{it}}(t)$ by $u$, denoted by $q(t)$. One more group consisting of $r(t)$ kernels will be executed, where $r(t)$ is the remainder of the division of $N_{\text{it}}(t)$ by $u$. Then, the time to execute the $N_{\text{it}}(t)$ kernels in hardware is:

$$T_{\text{it}}(t) = q(t) \cdot T_{K(\text{hw})}(u) + T_{K(\text{hw})}(r(t)), \quad (8)$$

Assuming the unroll factor $u$, let $q$ and $r$ be the quotient and the remainder of the division of $M$ by $u$, respectively. Then $M = q \cdot u + r$, $r < u$. The hardware execution time becomes:

$$\begin{aligned} T_{\text{h/h}}(u) \;=\;& 2 \cdot [\,T_{\text{it}}(1) + \ldots + T_{\text{it}}(u)\,] \\ +\;& 2 \cdot [\,T_{\text{it}}(u+1) + \ldots + T_{\text{it}}(u+u)\,] \\ + \ldots +\;& 2 \cdot [\,T_{\text{it}}(q \cdot u + 1) + \ldots + T_{\text{it}}(q \cdot u + r)\,] \\ &+ (N - M - 1) \cdot T_{\text{it}}(q \cdot u + r) \end{aligned} \quad (9)$$

By expanding each $T_{\text{it}}(t)$ from (9) according to (8), we obtain:

$$\begin{aligned} T_{\text{h/h}}(u) &= 2\cdot[\,T_{K(\text{hw})}(1)+\ldots+T_{K(\text{hw})}(u)\,] \\ &+\ 2\cdot[\,T_{K(\text{hw})}(u)+\ldots+2\cdot T_{K(\text{hw})}(u)\,] \\ +\ldots &+\ 2\cdot[\,q\cdot T_{K(\text{hw})}(u)+T_{K(\text{hw})}(1)+\ldots+ \\ &\quad +q\cdot T_{K(\text{hw})}(u)+T_{K(\text{hw})}(r)\,] \\ &\quad +(N-M-1)\cdot[\,q\cdot T_{K(\text{hw})}(u)+T_{K(\text{hw})}(r)\,] \end{aligned} \tag{10}$$

Then, by grouping all the similar terms, $T_{\text{h/h}}(u)$ becomes:

$$\begin{aligned} T_{\text{h/h}}(u) &= 2\cdot q\cdot\sum_{i=1}^{u-1}T_{K(\text{hw})}(i)+2\cdot\sum_{i=1}^{r-1}T_{K(\text{hw})}(i) \\ &+(N-M+1)\cdot T_{K(\text{hw})}(r) \\ &+q\cdot(N-u+1+r)\cdot T_{K(\text{hw})}(u) \end{aligned} \tag{11}$$

The speedup at loop level will be:

$$S_{\text{h/h}}(u)=\frac{T_{\text{loop(sw)}}}{T_{\text{h/h}}(u)+T_{\text{sw}}\cdot M\cdot N}. \tag{12}$$

*b)* **With part of the kernels in software:** It is possible that better performance is achieved if not all kernel instances from an iteration are executed in hardware. Since the degree of parallelism varies with the iteration number according to Fig. 5, balancing the number of kernels that run in software and in hardware for each iteration would lead to a significant overhead. Assuming a certain unroll factor $u$ ($u < M$), we look only at the iterations $t$ with $N_{\text{it}}(t) > u$ and compute the number of kernels (denoted by $v(t)$) that need to run in software such that the iteration execution time is minimized. There are $N + M - 2\cdot u - 1$ such iterations, namely the iterations $t$ with $u < t \leq N + M - u$.

Then, $v(t)$ is the number of kernels for which the kernels running in software execute in approximately the same amount of time as the kernels running in hardware. The time to execute $v(t)$ kernels in software is:

$$T_{K(\text{sw})}(v(t))=v(t)\cdot T_{K(\text{sw})} \tag{13}$$

The execution pattern for $N_{\text{it}}(t) = 22$, with $v(t) = 2$ kernel instances running in software is depicted in Fig. 6b). In this context, $q_v(t)$ and $r_v(t)$ are the quotient and the reminder of $N_{\text{it}}(t) - v(t)$ divided by $u$, respectively.

The time to execute the remaining $N_{\text{it}}(t) - v(t)$ kernels in hardware is:

$$T_{K(\text{hw})}(t,v(t))=q_v(t)\cdot T_{K(\text{hw})}(u)+T_{K(\text{hw})}(r_v(t)) \tag{14}$$

By using (3) in (14), $T_{K(\text{hw})}(t, v(t))$ becomes:

$$\begin{aligned} T_{K(\text{hw})}(t,v(t)) &= (N_{\text{it}}(t)-v(t))\cdot T_{\max(\text{r,w})}+ \\ &+\left\lceil\frac{N_{\text{it}}(t)-v(t)}{u}\right\rceil\cdot(T_{\text{c}}+T_{\min(\text{r,w})}) \end{aligned} \tag{15}$$

Then, the optimum number of kernels that should run in software in the iterations $t$ with $N_{\text{it}}(t) > u$ is the closest integer to the value of $v(t)$ that satisfies the relation:

$$T_{K(\text{sw})}(v(t))=T_{K(\text{hw})}(t,v(t)) \iff \tag{16}$$

$$\begin{aligned} v(t)\cdot(T_{K(\text{sw})}+T_{\max(\text{r,w})}) &= N_{\text{it}}(t)\cdot T_{\max(\text{r,w})}+ \\ &+\left\lceil\frac{N_{\text{it}}(t)-v(t)}{u}\right\rceil\cdot(T_{\text{c}}+T_{\min(\text{r,w})}) \end{aligned} \tag{17}$$

The value of $v(t)$ depends on the number of kernels in iteration $t$ ($N_{\text{it}}(t)$),which represents the available parallelism, and on the weight of the memory transfers ($T_{\max(\text{r,w})}$)). The larger the value of $N_{\text{it}}(t)$ and the weight of the memory transfers compared to the kernel's execution time (in software and hardware), the larger the value of $v(t)$. The unroll factor $u$ also influences the number of kernels to execute in software, as a larger value of $u$ implies that more kernels will execute in hardware and fewer in software. These considerations are illustrated in Section VI, where we present the experimental results.

By rounding the value of $v(t)$ to the nearest lower integer, the execution time for the kernels in software in iteration $t$ will not be larger than the execution time for the kernels in hardware. Therefore, the overall time for executing all the kernel instances in iteration $t$ is given by the hardware time:

$$T_{\text{it}}(t,v(t))=T_{K(\text{hw})}(t,v(t)) \quad \Rightarrow \tag{18}$$

$$T_{\text{it}}(t,v(t))=q_v(t)\cdot T_{K(\text{hw})}(u)+T_{K(\text{hw})}(r_v(t)) \tag{19}$$

The hardware execution time for unroll factor $u$ when part of the kernels execute in software is:

$$T_{\text{h/s}}(u)=2\cdot\sum_{t=1}^{u}T_{\text{it}}(t)+\sum_{t=u+1}^{M+N-u}T_{\text{it}}(t,v(t)) \tag{20}$$

$$T_{\text{h/s}}(u)=2\cdot\sum_{t=1}^{u}T_{\text{it}}(t)+2\cdot\sum_{t=u+1}^{M-1}T_{\text{it}}(t,v(t))+\sum_{t=M}^{N}T_{\text{it}}(M,v(M)) \tag{21}$$

Then, by expanding each $T_{\text{it}}$ and grouping all the similar terms, $T_{\text{h/s}}(u)$ becomes:

$$\begin{aligned} T_{\text{h/s}}(u) &= 2\cdot\sum_{i=1}^{u-1}T_{K(\text{hw})}(i)+ \\ &+\left(2+2\cdot\sum_{i=u+1}^{M-1}q_v(i)+(N-M+1)\cdot q_v(M)\right)\cdot T_{K(\text{hw})}(u)+ \\ &+\left(2\cdot\sum_{i=u+1}^{M-1}T_{K(\text{hw})}(r_v(i))\right)+(N-M+1)\cdot T_{K(\text{hw})}(r_v(M)) \end{aligned} \tag{22}$$

Similarly to (12), the speedup at loop level when balancing the kernel execution in software and hardware will be:

$$S_{\text{h/s}}(u)=\frac{T_{\text{loop(sw)}}}{T_{\text{h/s}}(u)+T_{\text{sw}}\cdot M\cdot N}. \tag{23}$$

Note that (12) and (23) are valid under the assumption that the software code (`do_SW`) and the hardware-mapped kernel execute sequentially. In [13] it is proved that it is always beneficial to apply loop shifting for breaking a specific kind of dependencies (task-chain dependencies) between the `do_SW` function and the kernel, in order to enable the software-hardware parallelism. The same technique can be applied also in conjunction with the loop skewing and thus the final execution time and the speedup could be improved by concurrently executing the software and the hardware code.

*Area considerations:* When multiple kernels are mapped on the reconfigurable hardware, the goal is to determine the optimal unroll factor for each kernel, which would lead to the maximum performance improvement for the application. For this purpose, the model needs one more parameter: the calibration factor – denoted here by $F$. The calibration factor has been previously introduced in [14] and a more detailed explanation and an example of how it influences the choice of the unroll factor is presented in [13]. It is a positive number decided by the application designer, which determines a limitation of the unroll factor according to the targeted trade-off. (One may not want to increase the unrolling if the gain in speedup would be only by a factor of 0.1%, but the area usage would increase by 15%.)

Denoting by $\Delta S(u+1,u)$ the relative speedup increase between unroll factors $u$ and $u+1$ and by $\Delta A(u+1,u)$ the relative area increase (which is constant since all kernel instances are identical), the simplest relation to be satisfied between the speedup and necessary area is:

$$\Delta S(u+1,u) > \Delta A(u+1,u) \cdot F, \qquad \text{where:} \tag{24}$$

$$\Delta A(u+1,u) = A(u+1) - A(u) = Area_{(K)} \in (0,\ 1) \tag{25}$$

$$\Delta S(u+1,u) = \frac{S_{\text{loop}}(u+1) - S_{\text{loop}}(u)}{S_{\text{loop}}(u)}. \tag{26}$$

Only in the ideal case $\frac{S_{\text{loop}}(u+1)}{S_{\text{loop}}(u)} = \frac{u+1}{u}$, meaning that

$$S_{\text{loop}}(u+1) < 2 \cdot S_{\text{loop}}(u),\ \forall u \in \mathbb{N},\ u > 1 \tag{27}$$

and the relative speedup satisfies the relation:

$$\Delta S(u+1,u) \in [0,\ 1),\ \forall u \in \mathbb{N},\ u > 1. \tag{28}$$

Thus, $F$ is a threshold value which sets the speedup bound for the unroll factor. How to choose a good value for $F$ is not within the scope of this research. However, it should be mentioned that a greater value of $F$ would lead to a lower bound, which translates to 'the price we are willing to pay in terms of area compared to the speedup gain is small'. Also, the value of $F$ should be limited by $\frac{\Delta S(2,1)}{Area_{(K)}}$, which is the value that would allow the unroll factor of 2 — a larger value would lead to the unroll factor 1 (*i.e.*, no unroll):

$$F \in \left[0,\ \frac{\Delta S(2,1)}{Area_{(K)}}\right]. \tag{29}$$

Based on these formulas, the optimal unroll factor and the estimated achieved speedup can be computed in linear time.

## VI. Results

In this section, we illustrate the methods presented in Section V. The performance of our method depends on the kernel implementation, and the order of magnitude of the achieved speedup is not relevant for the algorithm.

The analysis was performed on one of the most time consuming parts of the H.264 video codec, Deblocking Filter (DF)[15], which is a part of both encoder and decoder. The code presented in Fig. 7 represents the main loop of the DF. In the H.264 standard, the image is divided in blocks of $4 \times 4$ pixels. The color format is YCbCr 4:2:0, meaning that the chrominance (chroma) components are sub-sampled to half the sample rate of the luminance (luma) - in both directions. The blocks are grouped in macroblocks (MB), each MB being a $4 \times 4$ block matrix for the luma and $2 \times 2$ block matrix for the chroma components. Each block edge has to be filtered, the DF being applied to each decoded block of a given MB for luma and chroma samples. The pseudocode for the DF is shown in Fig. 8. For the processed MB, first vertical filtering is applied (luma and chroma), followed by horizontal filtering. The vertical and horizontal filtering cannot be interchanged or executed in parallel because of data dependencies, however the luma and chroma filtering can be performed in parallel.

The VHDL code for the filter was automatically generated with the DWARV [4] tool. The synthesis results using Xilinx XST tool of ISE 8.1 for different Xilinx boards are presented in Table III. The code was executed on the Virtex II Pro–XC2VP30 board. The execution times were measured using the PowerPC timer registers. For the considered implementa-

```
DF main loop:

foreach frame f
  for y = 0 .. nCols
    for x = 0 .. nRows
    {
      compute_MB_params (&MB);
      filter_MB (MB);
    }
```

Fig. 7. Deblocking Filter main loop.

```
filter_MB:

foreach Edge in MB.edges[VERT]
  if (is_picture_edge(Edge) == FALSE)
  {
    filter_MB_edge(MB, VERT);
    if (odd == TRUE) {
      odd == FALSE;
      filter_MB_edge_cv(MB, VERT, Cr);
      filter_MB_edge_cv(MB, VERT, Cb);
    }
    else odd = TRUE;
  }
foreach Edge in MB.edges[HORIZ]
  if (is_picture_edge(Edge) == FALSE)
  {
    filter_MB_edge(MB, HORIZ);
    if (odd == TRUE) {
      odd = FALSE;
      filter_MB_edge_ch(MB, HORIZ, Cr);
      filter_MB_edge_ch(MB, HORIZ, Cb);
    }
    else odd = TRUE;
  }
```

Fig. 8. Deblocking Filter applied at MB level.

TABLE III
SYNTHESIS RESULTS

| Platform | Slices | [%] | Freq[MHz] |
|---|---|---|---|
| XC2VP30 | 2561 | 18.70 | 178.296 |
| XC2VP70 | 2552 | 7.72 | 177.953 |
| XC2VP100 | 2606 | 5.91 | 151.400 |

TABLE IV
EXECUTION TIMES FOR THE DF (CYCLES)

| | Software | Hardware |
|---|---|---|
| average | 87119 | 106802 |
| maximum | 103905 | 119166 |

tion, the shared memory has an access time of 3 cycles for reading and storing the value into a register and 1 cycle for writing a value to memory;

In the profiling stage, the execution times for the DF running in both software and hardware for 80 different MBs were measured. The average and the maximum values that were measured are presented in Table IV. Also, the profiling indicates the (maximum) number of memory transfers per kernel – 2424 memory reads and 2400 memory writes. The execution time for the software function that computes the parameters is 2002 cycles.

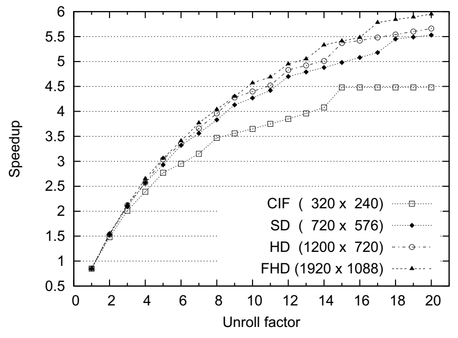


Fig. 9. DF speedup for different picture sizes (average MB exec time).

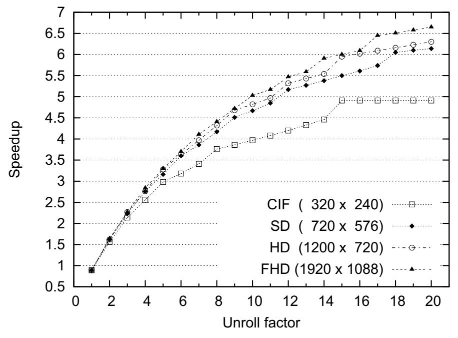


Fig. 10. DF speedup for different picture sizes (maximum MB exec time).

Figures 9 and 10 illustrate the results when the first of the presented methods is considered, with all kernel instances running on the FPGA. Equations (5), (11) and (12) were used for computing the different speedups for the skewed and unrolled DF loop for various image sizes and unroll factors. The speedup considering the average execution times for the kernel are illustrated in Fig. 9, while the speedups for the maximum execution times are presented in Fig. 10.

The results are heavily influenced by the performance of the kernel implementation in hardware. As shown in Table IV, the hardware implementation is slower than the software one, which results in a performance decrease (0.85 and 0.89 speedup) when no unrolling is performed. However, unrolling even with a factor of 2 already compensates the slow hardware implementation, giving a speedup of 1.5 – 1.6. The performance does not increase linearly with the unroll factor because of the variable number of iterations of the skewed loop, being influenced also by the relation between the unroll factor and $M$ ($M$ is the maximum degree of parallelism for most of the iterations of the skewed loop).

The results show that better performance is obtained when parallelizing larger loops. The reason is that the bigger the loop size, the larger the number of iterations that have a higher available degree of parallelism ($M$). Note that for the CIF picture format ($320 \times 240$ pixels), the speedup saturates at the unroll factor 15. This happens because the DF loop size is equal to the picture size divided by 16, meaning that in this case $N = 20$ and $M = 15$.

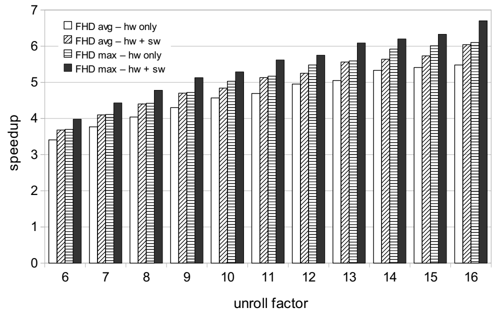


Fig. 11. DF speedup for FHD format for different unroll factors.

Figure 11 presents the speedup for the FHD picture format for unroll factors between 6 and 16, considering both types of execution presented in Section V. The first type of execution consists of running all kernel instances in hardware, while the second places part of the kernel instances in software. A performance increase between 4% and 15% is noted for the second type. The performance would increase more for a larger number of iterations or if the memory I/O ($T_{\max(\mathrm{r,w})}$) would represent more of the execution time.

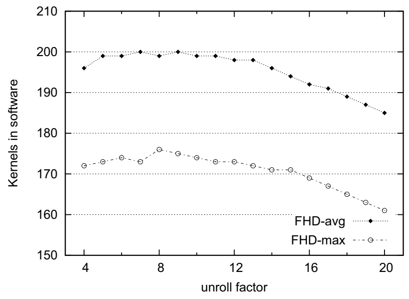


Fig. 12. How the number of kernels in software varies with the unroll factor.

Figure 12 shows how the number of kernel instances that execute in software varies with the unroll factor for the FHD picture format. The depicted numbers represent the sum of software executed kernels across all loop iterations, approx. 2–2.45% of the total number of kernels for the FHD format. The general trend is that a larger unroll factor means fewer kernels to execute in software, because more kernels will

TABLE V
SPEEDUPS FOR THE UNROLL FACTOR U=8

| | CIF | SD | HD | FHD |
|---|---|---|---|---|
| average (hw only) | 3.47 | 3.83 | 3.96 | 4.04 |
| average (hw + sw) | 3.65 | 4.11 | 4.20 | 4.40 |
| maximum (hw only) | 3.77 | 4.18 | 4.32 | 4.42 |
| maximum (hw + sw) | 3.93 | 4.47 | 4.55 | 4.78 |

execute in hardware in each group. See Fig. 6 for the kernels' execution pattern, where one line of kernels represents a group. On the same figure it can be seen that the total number of kernel instances that execute in software is larger for FHD-avg than for FHD-max. This happens because the memory I/O represents a larger percentage from the total execution time for FHD-avg, compared to FHD-max. The reason is that the weight of the I/O time from the total execution time directly influences the number of software scheduled kernels, as seen from equation (17).

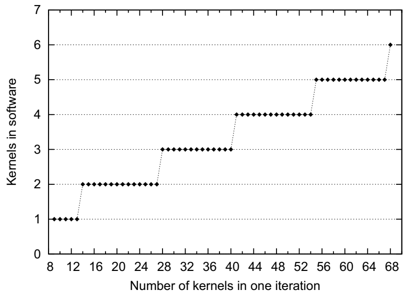


Fig. 13. The software scheduled kernels vs. the total number of kernels.

On the basis of one iteration which has a variable number of kernels over time (see Fig. 5), Fig. 13 presents how the number of kernel instances that execute in software varies with the total number of kernels in that iteration. When the total number of kernels in one iteration increases, the total execution time increases and therefore more kernel instances can be scheduled for software execution.

Taking into account also area requirements, the optimum unroll factor depends on the desired trade-off between area consumption and performance. If for instance the metric chosen for the optimum unroll factor is to have a speedup increase of at least 0.3 comparing to the previous unroll factor, then the optimum unroll factor for most picture sizes in our example is 8. The estimated speedups achieved for the unroll factor $u = 8$ for all pictures sizes and using the average/maximum kernel execution times are depicted in Table V.

## VII. CONCLUSION

In this paper, we address parallelizing K-loops with wavefront-like dependencies, thus relaxing the constraints imposed by using only loop unrolling and/or shifting for loop parallelization. Two methods were presented, the first one considering that all instances of a kernel K will run on the reconfigurable hardware. The second method considers that part of the kernel instances run in hardware and part in software. Both methods are based on loop skewing and unrolling for computing the optimal number of kernel instances that will run in parallel on the reconfigurable hardware.

The second method is more suitable for a large number of iterations and/or when the I/O represents a significant part of the kernel, limiting the degree of parallelism.

The input data for the algorithms consist of profiling information about memory transfers, execution times in software and hardware, and information about area usage for one kernel instance and area availability.

The presented methods can be used to improve performance for any VHDL implementation of the kernel, if there are enough resources available (for instance, when moved to a different platform). Moreover, their implementation in the compiler decreases the time for design-space exploration and makes use efficiently of the hardware resources.

In the experimental part, we analyzed the case of the Deblocking Filter loop of the H.264 encoder/decoder. Using an automatically generated VHDL code for the edge filtering part of the DF, we showed that a speedup between 3.4 and 4.8 can be achieved with the unroll factor 8, depending on the input picture size.

In our future work, we consider extending the model by supporting loops with an arbitrary number of hardware kernels and with pieces of software code also occurring in between the kernels.

## REFERENCES

[1] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The MOLEN polymorphic processor," *IEEE Transactions on Computers*, November 2004.
[2] Xilinx Inc., "Virtex-II Pro and Virtex-II Pro X platform FPGAs: Complete data sheet," http://www.xilinx.com/bvdocs/publications/ds083.pdf.
[3] S. Vassiliadis, G. Gaydadjiev, K. Bertels, and E. M. Panainte, "The Molen programming paradigm," in *SAMOS 2003*.
[4] Y. Yankova, G. Kuzmanov, K. Bertels, G. Gaydadjiev, Y. Lu, and S. Vassiliadis, "DWARV: DelftWorkbench automated reconfigurable VHDL generator," *FPL 2007*.
[5] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Surveys*, 1994.
[6] M. Wolfe, "Loop skewing: the wavefront method revisited," *Int. J. Parallel Program.*, vol. 15, no. 4, pp. 279–293, 1986.
[7] Z. Guo, B. Buyukkurt, W. Najjar, and K. Vissers, "Optimized generation of data-path from C codes for FPGAs," *DATE 2005*.
[8] M. Weinhardt and W. Luk, "Pipeline vectorization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2001.
[9] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Loop shifting and compaction for the high-level synthesis of designs with complex control flow," *DATE 2004*.
[10] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere, "System Design Using Kahn Process Networks: The Compaan/Laura Approach," in *DATE 2004*.
[11] T. Stefanov, B. Kienhuis, and E. Deprettere, "Algorithmic transformation techniques for efficient exploration of alternative application instances," in *CODES 2002*.
[12] C. Zissulescu, T. Stefanov, and B. Kienhuis, "Laura: Leiden Architecture Research and Exploration Tool," in *FPL 2003*.
[13] O. Dragomir, T. P. Stefanov, and K. Bertels, "Optimal Loop Unrolling and Shifting for Reconfigurable Architectures," *ACM Transactions on Reconfigurable Technology and Systems*, Vol. 2, Nr. 4, September 2009.
[14] O. S. Dragomir, E. M. Panainte, K. Bertels, and S. Wong, "Optimal unroll factor for reconfigurable architectures," in *ARC 2008*.
[15] P. List, A. Joch, J. Lainema, G. Bjøntegaard, and M. Karczewicz, "Adaptive Deblocking Filter," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 614 – 619, 2003.