Kernel-Centric Acceleration of High Accuracy Stereo-Matching

Tobias Kenter, Henning Schmitz, Christian Plessl
Department of Computer Science, University of Paderborn
33098 Paderborn, Germany
Email: {kenter - mod - christian.plessl}@uni-paderborn.de

***Abstract*—Stereo-matching algorithms recently received a lot of attention from the FPGA acceleration community. Presented solutions range from simple, very resource efficient systems with modest matching quality for small embedded systems to sophisticated algorithms with several processing steps, implemented on big FPGAs. In order to achieve high throughput, most implementations strongly focus on pipelining and data reuse between different computation steps. This approach leads to high efficiency, but limits the supported computation patterns and due the high integration of the implementation, adaptions to the algorithm are difficult.**

In this work, we present a stereo-matching implementation, that starts by offloading individual kernels from the CPU to the FPGA. Between subsequent compute steps on the FPGA, data is stored off-chip in on-board memory of the FPGA accelerator card. This enables us to accelerate the AD-census algorithm with cross-based aggregation and scanline optimization for the first time without algorithmic changes and for up to full HD image dimensions. Analyzing throughput and bandwidth requirements, we outline some trade-offs that are involved with this approach, compared to tighter integration of more kernel loops into one design.

Keywords*—*Stereo-Matching; Kernel Acceleration; Dataflow Computing; Performance Modeling;

I. Introduction

In order to achieve the best possible performance for application acceleration on FPGAs, the entire design process, from selection of the algorithm, compute and data structures to customization of operations and their precision needs to be considered [1]. However, the usage of FPGAs is increasingly extending beyond established domains such as embedded systems and specialized high performance compute scenarios, where this design paradigm is acceptable, towards usage scenarios in general-purpose computing and big data centers [2]. In those scenarios, where large code bases exist, at least parts of the application are subject to ongoing changes and the impact of small changes in the algorithm or data representation of some part of the entire application can not be easily assessed, such a comprehensive design method is often infeasible. Instead, a more pragmatic approach is required, which we denote as kernel-centric, where individual parts of the application, identified as computational hotspots and suitable for acceleration, are translated and offloaded to functionally equivalent FPGA implementations. In this work we apply such a design approach to a stereo-matching application and evaluate its potential and cost.

Stereo-matching, the computation of a depth image from a pair of stereo images, is one of the most important challenges in computer vision, with applications ranging from robot navigation [3] to 3D movie production [4]. Common design goals for all types of applications are high matching quality and high processing speed, yet with varying priorities and additional constraints, e.g. on image resolution, on latency or throughput, or on power and resource limitations. For example in robot navigation, low latency real-time processing and often a constrained power budget are of paramount importance and in order to achieve them, low image resolutions and limited matching quality have to be accepted, whereas for 3D movie production support for high image resolutions is required and matching quality is much more important.

Over the last couple of years, a number of FPGA implementations for stereo-matching with different design goals has been published, however, almost all of them are the result of some form of HW/SW-codesign process, where the algorithmic approach as well as data representation, are specifically tailored to a throughput oriented design and the capabilities of FPGA fabric. While this is per-se a good principle, it is not suitable for an envisioned role of FPGAs as complement to general-purpose CPUs. In particular, the qualitatively best performing algorithms for stereo-matching have not yet been implemented on FPGAs.

In this paper we present an FPGA-accelerated stereo-matching application with—to the best of our knowledge—the highest accuracy of all published FPGA implementations and with support for up to full HD resolution, which only few FPGA implementations currently achieve. Along with this implementation, we present a memory management mechanism that facilitates kernel-centric acceleration of existing software implementations. We also demonstrate the use of simple performance models for choosing an appropriate amount of kernel parallelism to saturate available off-chip bandwidths. Finally we investigate some of the trade-offs involved in the kernel-centric acceleration approach.

In the remainder of this paper, we will present as background the software algorithm we started off to accelerate, the Maxeler platform we developed our FPGA accelerated stereo-matching on and selected related work. Then we present the general concept of kernel-centric acceleration and our solution to facilitating memory management in this context. Afterwards we present the design of the specific kernels we implemented on FPGA and our design decisions based on a simple performance model. These kernels are first measured individually, then integrated into the entire stereo-matching

978-1-4799-5944-0/14/$31.00 ©2014 IEEE

Published in: *Proc. Int. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*
 DOI: 10.1109/ReConFig.2014.7032535

implementation. Finally we evaluate the overall outcome of our approach.

II. Stereo Matching

Most stereo-matching algorithms compute a disparity value for each pixel, which represents how far (distance in pixels) the object that this pixel belongs to appears shifted between the left and right stereo image. As auxiliary metric to compute disparities, many algorithms use a cost value for each pixel at each possible disparity, where a low cost signifies that it is plausible that this pixel should have the corresponding disparity.

In our work, we follow the stereo-matching algorithm published by Mei et al. [5]. It consists of four phases. In the first phase, *cost initialization*, for each pixel and each possible disparity, a local cost value is computed, based on two similarity metrics. In the second phase, *cost aggregation*, the costs of neighboring pixels of the same disparity are aggregated in adaptive support regions, which are determined by color differences and absolute distances. In the third phase, *scanline optimization*, an energy minimization approach is mimicked by dynamic programming along 1-dimensional scanlines. The *refinement* phase performs a consistency check between the left and right disparity images resulting from the previous phase and applies several local optimizations for pixels which are not classified consistently. As the most time-consuming parts, we outline the basic mechanisms of cost aggregation and scanline optimization.

The idea of cost aggregation is to reduce the huge amount of noise contained in the local cost metrics. For that purpose, the costs are aggregated over a limited area around each pixel, which likely belongs to the same objects of the image and thus should have similar disparity values. Therefore aggregation areas should track object boundaries in shape and size as good as possible. However, computing individual aggregation areas for each pixel and summing up the costs inside them can be very compute intense. The cross-based aggregation method utilized here was first proposed by Zhang et al [6]. The areas are defined by the length of four arms for each pixel, two extending to the left and right, two up and down. Two possible aggregation areas are now formed by all vertical arms that belong to pixels on the horizontal arms of each pixel and respectively the other way round as illustrated in Fig. 1. Horizontal first aggregation areas can cover vertical object boundaries better, vertical first aggregation is more precise for horizontal object boundaries. For both aggregation areas the actual aggregation can be performed in linear time with the help of integral sums. To outline this technique for the horizontal step, first for each row r of costs a new sum row s is computed, containing at each position p the sum of all cost values from the left to exactly this position p: $s_p = \sum_{i=0}^{p} r_i$. Now the aggregated cost a_{pq} between two positions p and q, $p < q$, is easily computed with $a_{pq} = s_q - s_{p-1}$.

The scanline optimization follows Hirschmüller's [7] semiglobal matching strategy. Global matching would perform a 2-dimensional energy minimization for the entire image, minimizing the weighted sum of the energy in the final disparity image and of the involved matching costs for this

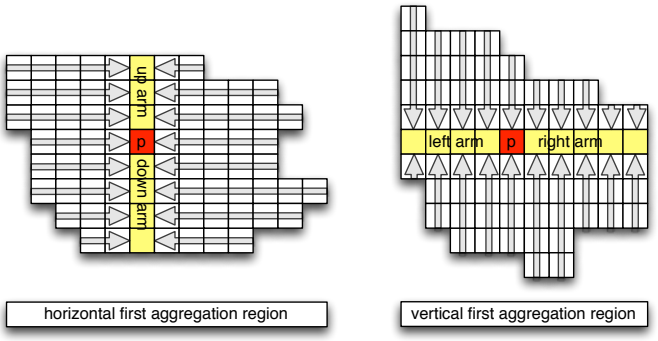

Figure 1. Illustration of cross-based cost aggregation regions, left projecting all horizontal arms on the vertical arms of a pixel, right projecting vertical arms on the horizontal arms of a pixel.

disparity image. The scanline optimization mimics this idea along 1-dimensional lines, but avoids costly minimization steps and instead uses a dynamic programming approach, where the previous disparity decisions along the scanline are fixed and only the energy trade-off for the current step is considered. Equation 1 outlines the basic recursion equation.

$$\begin{aligned} C_r(p,d) =& C_a(p,d) + \min[C_r(p-r,d), \\ & C_r(p-r,d\pm 1) + P_1, \\ & \min_k C_r(p-r,k) + P_2] - \min_k C_r(p-r,k) \end{aligned} \tag{1}$$

The scanline cost C_r for disparity d along a scanline path r at pixel p is computed by the aggregation cost C_a at this position and disparity level, plus the minimum of the following three: 1) the scanline cost from the previous position along the scanline path at the same disparity $C_r(p-r,d)$, 2) the scanline cost from the previous position along the scanline path at a neighboring disparity $C_r(p-r,d\pm 1)$ plus a small penalty P_1 and 3) the minimal scanline cost at the previous position along the scanline path at any disparity $\min_k C_r(p-r,k)$ plus a larger penalty P_2. Finally the minimal scanline cost at the previous position is subtracted. In this equation, C_a reflects the energy added to the path from matching costs, whereas the penalty values reflect the energy of the resulting disparity profiles. Both penalty values additionally depend at each specific position on the color differences of the original images.

As starting point for our acceleration, we use our own software implementation for stereo-matching, which follows these concepts, but offers additional features, such as different, parametrizable cost initialization metrics (for more metrics see e.g. [8]), an adjustable sequence of aggregation steps including scaling, and an optional OpenGL visualization of aggregation areas, cost volumes and cost metric profiles. The precision of intermediate cost values required for stable results depends highly on the actual images processed. In general, quality degradation with reduced precision is graceful, but in some cases with single precision floating point, costs after computing differences in the aggregation step can falsely become 0, leading to artifacts. Thus, we use in our software implementation double precision and also require this from the FPGA acceleration. Our implementation reaches an accuracy in the Middlebury benchmark [9] of average 5.73% bad pixels and we make sure during our acceleration process to still produce the same results.

III. Maxeler Platform

The Maxeler platform comprises two 6-core (12 threads) Intel Xeon X5650 CPUs as host platform and is equipped with four MAX3424A PCIe accelerator cards with a Xilinx Virtex-6 SX475T FPGA for user logic and 24GB of local SDRAM memory. This local memory is called LMem and has to be read or written in bursts of 384 adjacent bytes. However, in order to come close to the possible bandwidth of 38.4GB/s, several bursts, either adjacent or with a fixed stride, should be accessed with a single memory command, e.g. commands with only 1 burst each will lead to an efficiency of only 11%, whereas with 8 consecutive bursts already 80% efficiency are reached. The PCIe interface on the other hand can be used to stream data from or to host memory and reaches a bandwidth of 2 GB/s.

The distinctive feature of the Maxeler systems is their development environment, which allows programming the FPGAs with a Java extension called MaxJ and integrating them to the host (CPU) part of an application with automatically generated API calls for each functionality implemented on FPGA. The MaxJ language offers a much higher abstraction than HDL languages like VHDL and Verilog, but much finer control on the design than when generating hardware via HLS. Conceptually, MaxJ is built around streams of data, where typically one data element per cycle is processed in a so-called hardware kernel. A sequence of operations on one or several streams is automatically translated into a corresponding compute pipeline, where pipelining may also happen inside individual operations, in particular when they utilize DSP blocks. The streams can be connected to other kernels or to LMem or via PCIe to host memory and the Maxeler toolflow will automatically generate the required buffers and interfaces.

IV. Related Work

We demonstrated a similar approach of accelerating individual kernels of a stereo-matching algorithm in [10]. However, we did not build custom kernel designs, but targeted an instruction based vector coprocessor on FPGA, which shares a common address space with the host CPU and whose memory interface allows random accesses on a much smaller granularity than the Maxeler platform does.

The following stereo-matching implementations on FPGA are all custom designs implementing the entire processing pipeline without off-chip memory accesses. Shan et al. [11] implemented a slightly modified variant of the presented cost aggregation for adaptive support regions on FPGAs. By aggregating only once and in a fixed order, first vertically and then horizontally, they are able to stream the required data only through on-chip buffers. Wang et al. [12] try to follow the algorithm of Mei et al. [5] in their FPGA implementation more closely. In addition to the aggregation technique of Shan et al. [11], they propose a reduced scanline optimization which runs in three downward directions, following the order the data is generated in the previous aggregation stage. Both implementations try to exploit parallelism both in the spacial domain of the images, working on several rows at once, and in the disparity domain of the cost volume, working on several disparity images at once. Shan and Wang reach an average of 7.65% and 6.17% bad pixels respectively.

```
1 kernelAtoB(double* a, double* b){
2   mm.writeIfInvalid(a);
3   callKernel(mm.getLMem(a), mm.getLMem(b));
4   mm.setModified(b);
5   [mm.readIfModified(b);]
6 }
```

Listing 1. Illustration for using the Memory Manager.

V. Kernel Acceleration and Memory Management

We selected 8 kernels for acceleration, based on three criteria: first, by profiling we identified the most runtime intense kernels. The 8 selected kernels cover 93.2% of the total program runtime for the benchmarks we present in this paper, which would allow a speedup of at most 14.7x if the selected kernels could be accelerated infinitely. No other potential kernel of the stereo matching implementation accounted for more than 1.5% of the total runtime, which makes further candidates much less promising. Second, we analyzed the compute patterns of the candidate kernels and found them to contain data parallelism suitable for FPGA acceleration. Third, we made sure that the selected kernels cover a consecutive sequence of compute steps allowing that intermediate results between kernel calls can remain in LMem on the single accelerator card we utilize. Therefore, when all kernels are executed on the FPGA, all input data could be transferred to LMem before the first kernel call and results be transferred back after the final kernel call.

Moving all data before and after the complete sequence of accelerated kernels would however require modifications to the top level execution flow of the host algorithm. Also, when not all kernels are available for FPGA execution, when additional compute steps are performed in the host application, or when intermediate results are to be visualized, additional transfers in-between may be required. Therefore we decided to include an interface to transfer data to and from LMem to every kernel implementation on the FPGA and move data exactly when required.

On the host side, we implemented a memory manager for data in LMem. It can allocate memory areas in LMem that correspond to specific host data structures and supports tracking the state of those LMem memory areas as `invalid`, when LMem data is not valid, `shared`, when host memory and LMem are identical and `modified`, when data has been modified or computed for the first time in LMem, following the simple MSI coherency protocol.

The update steps of this protocol are manually triggered by API calls to the memory manager as illustrated in Listing 1. Before a hardware kernel is called, the input array `a` is written to LMem, but only if no valid copy is located there already. For the actual call to the hardware kernel, the locations of `a` and `b` are translated to the corresponding LMem locations. Since the kernel writes to `b`, this array is manually set as `modified` in LMem. Now in order to have a standalone replacement for a corresponding software function `AtoB`, `b` is read back to host memory. If on the other hand other hardware kernels work on the data of `b`, as it is typically the case for our kernels, the read-back can be omitted or delayed until the data of `b` is required on the host. Investigating where this is the case is the

only modification to our software implementation that is not restricted to the local kernels that are accelerated.

VI. Kernel design and performance model

In this section we present the identified time-consuming kernels and outline their parallelization opportunities, taking dependencies and data locality into account. Subsequently, we discuss the compute and memory access patterns we implemented in hardware. Most kernels are designed with a flexible amount of parallelism, which is specified by an unrolling factor f_u prior to synthesis. We present a simple performance model for each individual kernel and derive the optimal unrolling factor from it. The design process is outlined in some detail for the first kernel whereas for the other kernels we restrict ourselfs to noteworthy aspects due to space limitations.

A. Cost aggregation for adaptive support regions

The cost aggregation involves five different kernels: horizontal integral sums and differences, vertical integral sums and differences and scaling. All aggregation steps are independent for each different disparity value, however we chose to focus on parallelization in the dimensions of the images instead of the disparity dimensions. Parallelizing different disparities does on its own not guarantee sufficient parallelism, in particular when some of the double precision floating point operations need to be internally pipelined, as it is the case in our designs. The work of Shan et al. [11] suggests that utilizing disparity level parallelism in addition to image dimension parallelism might allow to save BRAM resources at the cost of additional logic utilization, which we did not investigate further considering the synthesis results that will be presented in Section VII.

As outlined in Section II, for the **horizontal integral sums** (denoted as `HorSum`), in each row the cost values from the beginning of the row to each current position are summed up. Listing 2 illustrates the corresponding software function for one image slice. There are dependencies along the rows, but we can parallelize computation by vertical unrolling, i.e. computing several rows in parallel. Figure 2 illustrates unrolled computation on the right. We additionally buffer more parallel lines in order to hide the feedback latency caused by internal pipelining. In memory, the data is arranged in row-major order, which means that entire rows are stored in continuous memory locations one after the other, with LMem using the same data layout as the host application to allow for the memory management outlined in Section V. Thus each burst of 384 bytes will read 48 subsequent double values from each row. The left side of Figure 2 illustrates the order data is read from LMem. While the sequence of big memory blocks needs to follow the same sequence in memory command generator and compute kernel, the order inside blocks is changed through stream offsets implemented with BRAM buffers to allow for the vertical unrolling.

We analyze the performance of this and all upcoming individual kernels with regards to the required LMem bandwidth for different unrolling factors f_u at a target clock frequency of 100 MHz. The plan is to utilize sufficient unrolling to saturate the available bandwidth of approximately 35 GB/s, whenever this is possible. In contrast to roofline-based performance models like [13], the computational intensity in terms of operations

```
1 void horSum(double **in, double **out){
2   for(int y=0; y<height; y++){
3     out[y][0] = in[y][0] ;
4     for(x=1; x<width; x++){
5       out[y][x] = out[y][x-1] + in[y][x];
6     }
7   }
8 }
```

Listing 2. Horizontal integral sums.

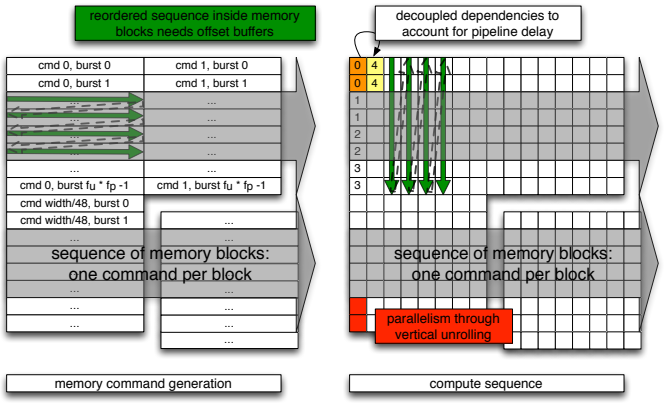

Figure 2. Illustration of memory accesses on the left and compute order on the right. Unrolling with a factor $f_u = 2$ is illustrated, two elements are computed in each cycle.

per memory bandwidth doesn't change in our kernels, because there is no data reuse beyond keeping the value from the last column in a buffer until it is used when the the next value along a row is computed. The required bandwidth for a selection on unrolling factors f_u is summarized in Table I. Following this analysis we synthesized the kernel with $f_u = 24$.

After the computation of the horizontal integral sums follows the step of computing **horizontal differences** (`HorDiff`). For each pixel, a left and right arm is required, which define the two positions in the integral cost rows to access and subtract from each other. There are no dependencies in this kernel and having the required integral cost data available in local buffers seems straightforward when unrolling horizontally, since neighboring pixels in one row require largely overlapping areas of possible input values defined by the arms. Figure 3 illustrates the selection of the position specified by right arms for an unrolling factor f_u of 4 and with possible values for the arm length of 0 to 4 (in practice we use the a length of up to 34 as reported in [5]). The orientation of the 4 element blocks is here given by the layout of data in memory. As Table I shows, for this kernel again an unrolling factor

Table I. Required bandwidth in GB/s for implemented kernels with 100 MHz at different unrolling factors f_u. Implemented versions are highlighted.

Kernel	$f_u = 8$	$f_u = 12$	$f_u = 16$	$f_u = 24$	$f_u = 48$
HorSum	11.9	17.9	23.8	**35.8**	71.5
VerSum	11.9	17.9	23.8	**35.8**	71.5
Scale	11.9	17.9	23.8	**35.8**	71.5
HorDiff	13.4	20.1	**26.8**	40.2	80.5
VerDiff	13.4	20.1	26.8	**40.2**	80.5
Turn	6.0	11.9	17.9	23.8	**35.8**
Scan	13.4	**20.1**	26.8	40.2	80.5
SAvg	29.8	**44.7**	59.6	89.4	178.8

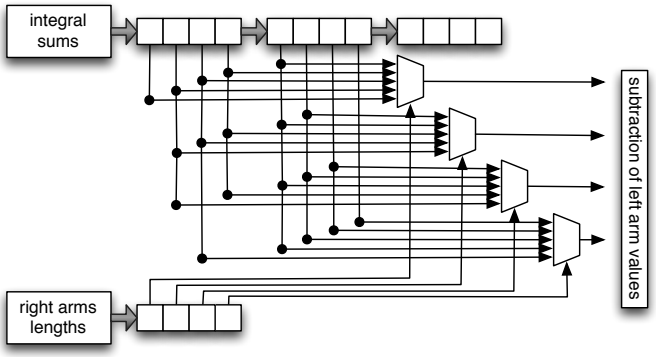

Figure 3. Illustration of right arm part of HorDiff for a maximum arm length of 5 (multiplexer size) and an unrolling factor f_u of 4.

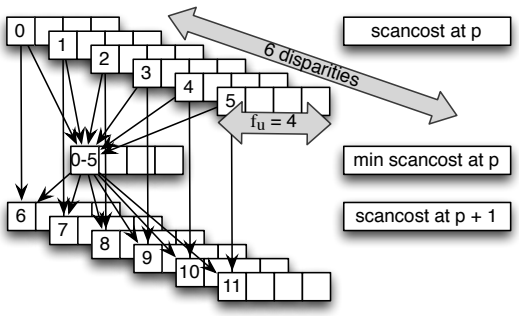

Figure 4. Illustration of downward scanline compute pattern for 6 disparities and unrolling factor f_u of 4. Arrows indicate dependencies for the leftmost column, numbers the compute step in which a value is updated.

f_u of 24 would guarantee full bandwidth utilization. However, even though resource utilization would permit it, the synthesis tools were not able to route a design with more than $f_u = 8$ anywhere near 100 MHz. The illustration in Figure 3 may give an intuitive idea that high number of overlapping connections to the different multiplexers causes this problem. On the other hand, trying to unroll computation vertically, like for `HorSum` lead to prohibitively large resource consumption. For our final design we implemented four separate processing units working on distinct tiles of each image and each unrolled 4 times, achieving a synthesizable design with overall parallelism of 16.

The **vertical integral sum** kernel (`VerSum`) is orthogonal to the `HorSum` kernel. Consequently, we now unroll computation horizontally. In contrast to `HorSum`, the blocks to allow unrolling and internal pipelining of operations don't need to be artificially created by specialized memory access patterns. Instead, we create line buffers for entire rows and use a linear memory access pattern.

Similarly to the `HorDiff` kernel, the computation of **vertical differences** (`VerDiff`) does not contain any dependencies. However, in contrast to the `HorDiff` kernel, it is more eligible to horizontal unrolling because now selection of arm positions is realized independently for each column. Thus for both vertical kernels we were able to use the desired unrolling factor of $f_u = 24$ (Table I).

Finally in the **scaling kernel** (`Scale`), each aggregated value is scaled i.e. divided by the size of its specific aggregation region. It is a straightforward streaming kernel without dependencies and can be readily unrolled horizontally. An IEEE compliant division of a double precision floating point values is not advisable on FPGA. However, since there is a fixed number of discrete sizes s_a, any aggregation region can have, we can precompute the inverse values $1/s_a$, look them up in BRAM and multiply by those inverses. Having the same bandwidth requirements as the summation kernels, an unrolling factor of $f_u = 24$ saturates the available bandwidth (Table I).

B. Scanline optimization

In contrast to the aggregation, the **scanline** (`Scan`) optimization is not independent for different disparity values. On the contrary, for the computation of the scanline costs of a new pixel, the minimal scanline costs of the previous pixel over all disparities need to be known. On the other hand, we also have a dependency along the scanlines, such that unrolling can only be performed orthogonal to the scanline direction. Figure 4 illustrates the dependency pattern for a downward scanline computation and how it can be unrolled horizontally. In our actual implementation, due to the burst size of our LMem interface, actual data blocks of 48 horizontal elements are loaded from memory and computed in $48/f_u$ cycles before proceeding to the next line. Since the previous minimum of step 0 is required to update the minimum for step 1, we reordered the datapath for the recursion of Equation 1 to have a deeper pipeline for the computation of the individual scancosts and a simple comparison for the selection of the current minimal scanline value. Nevertheless, similar to the `HorSum` kernel, we use additional unrolling in space to account for the feedback delay of this datapath. This delay of 3 cycles also limits the potential unrolling factor f_u to 12 without increasing the block size, even though for optimal throughput $f_u = 24$ would have been desirable (Table I). Doubling the block size to 96 on the other hand would have reduced the efficiency of the LMem interface, but also the resulting design failed to meet timing.

The same kernel also works on upward scanlines by just changing the order of reads and writes generated by our memory command generator. However, for horizontal scanlines this concept was not applicable due to prohibitive BRAM requirements, since bursts were still aligned horizontally, but unrolling would have to be done vertically and additionally the buffers would have to cover all disparity dimensions due to the compute sequence. Therefore we decided to implement a **turn kernel** (`Turn`) that reads data in row-major data layout and writes it back to LMem in column-major data layout, or vice-versa. Now we can execute horizontal scanlines by a sequence of turning data, applying vertical scanlines and turning data back. The overhead of this turning steps gets mitigated by using the upward and downward variant of the `Scan` kernel to mimic both rightward and leftward scanline computation on the same turned data. The `Turn` kernel uses 48 BRAM blocks in which data is written to and read from with a diagonally shifted addressing, providing the flexibility that either an entire row or column of 48 values can be accessed. The dimensions of blocks to be turned has to match at least the 48 elements per burst from the LMem interface, so in contrast to all other kernels we implemented this with a fixed unrolling factor f_u of

Table II. RESOURCE UTILIZATION OF THE IMPLEMENTED KERNELS, CRITICAL RESOURCE HIGHLIGHTED.

Design	Logic	LUTs	FFs	DSP	BRAM
Available	297600	297600	297600	2016	2128
HorSum_{24}	27%	19%	23%	1%	**39%**
VerSum_{24}	**30%**	21%	25%	0%	18%
Scale_{24}	**23%**	15%	19%	6%	22%
HorDiff_{16}	38%	27%	33%	1%	**48%**
VerDiff_{24}	**47%**	40%	42%	1%	23%
Turn_{48}	**32%**	23%	27%	3%	29%
Scan_{12}	**53%**	42%	46%	1%	31%
SAvg_{12}	**35%**	25%	30%	0%	23%

48. Without a double buffering technique, at any time either a block of data is written or is read from the BRAMs, but according to our model this suffices to saturate the available bandwidth (Table I).

After computing the costs along all scanline directions, the final scanline costs for each position and disparity is computed by averaging the values of all directions. The resulting `SAvg` kernel is a straightforward streaming kernel that we unrolled horizontally with $f_u = 12$. Since this step reduces the amount of result data to be transferred back, it is still particularly valuable for the overall implementation.

VII. SYNTHESIS AND RESULTS

In this section we present the synthesis results of the individual kernels and their individually measured performance compared to the presented models as well as to CPU execution times.

In Table II we summarize the resource consumption after targeting a clock frequency of 100MHz. Since not always a LUT and FF can be placed together into one logic slice, we report both individual requirements as well as resulting overall logic utilization. The results indicate that by adapting the unrolling factor to the available bandwidth, we leave some resources unused, between 47% and 77% of the critical resource. This will be considered in Section VIII. For the majority of kernels, six in number, logic slices are the most utilized resource, only the two kernels for horizontal aggregation use a higher fraction of BRAMs.

By working on blocks of data streamed from LMem, most of the presented kernels are virtually unlimited in all dimensions making up a cost volume, image width and height as well as number of possible disparities, with the only practical limit being the 24GB of LMem available on our MAX3424A FPGA cards. The `VerDiff` kernel in its current form uses line buffers which are for the presented results synthesized with a limit of 1920 elements. Thus our system is able to process full HD images, which is rarely reached by other stereo-matching systems on FPGA and if so is limited in the number of disparity levels [11].

In Tables III and IV we present the individual performance of the synthesized kernels applied to cost volumes of a pair of 1152x768 images with 128 disparity levels and for a pair of full HD (1920x1080) images with 256 disparity levels respectively. We compare the measured execution time t_{exe} to the expected execution time t_{model} when modeling the pure compute throughput of the kernels and compute a factor of the observed gap between those two as f_{gap}. Next we compare

Table III. PERFORMANCE OF INDIVIDUAL KERNELS FOR IMAGE DIMENSIONS 1152X768 WITH 128 DISPARITY VALUES. EXECUTION TIMES t IN SECONDS. *TURN KERNEL IS NOT ACTUALLY EXECUTED IN SOFTWARE.

Kernel	t_{model}	t_{exe}	f_{gap}	t_{sw}	$f_{speedup}$
HorSum_{24}	0.047	0.138	2.9	0.404	2.9
VerSum_{24}	0.047	0.068	1.4	0.220	3.2
Scale_{24}	0.047	0.101	2.1	0.833	8.2
HorDiff_{16}	0.071	0.077	1.1	0.321	4.2
VerDiff_{24}	0.047	0.158	3.4	1.107	7.0
Turn_{48}	0.047	0.113	2.4	*0.935	*8.3
Scan_{12}	0.094	0.140	1.5	4.852	34.7
SAvg_{12}	0.094	0.167	1.8	0.446	2.7

Table IV. PERFORMANCE OF INDIVIDUAL KERNELS FOR IMAGE DIMENSIONS 1920X1080 WITH 256 DISPARITY VALUES. EXECUTION TIMES t IN SECONDS. *TURN KERNEL IS NOT ACTUALLY EXECUTED IN SOFTWARE.

Kernel	t_{model}	t_{exe}	f_{gap}	t_{sw}	$f_{speedup}$
HorSum_{24}	0.221	0.788	3.6	1.892	2.4
VerSum_{24}	0.221	0.316	1.4	1.059	3.4
Scale_{24}	0.221	0.471	2.1	3.888	8.3
HorDiff_{16}	0.332	0.366	1.1	1.486	4.1
VerDiff_{24}	0.221	0.928	4.2	5.296	5.7
Turn_{48}	0.221	0.696	3.1	*6.086	*8.7
Scan_{12}	0.442	0.543	1.2	19.717	36.3
SAvg_{12}	0.442	0.759	1.7	2.517	3.3

the achieved kernel performance to the corresponding kernels runtimes on CPU t_{sw} and report the speedup factor $f_{speedup}$.

Analyzing the gaps f_{gap} between possible compute throughput and achieved real-world performance, we see considerable variations, from kernels closely tracking the possible performance to discrepancies up to 4x. Some parts of these discrepancies are expected when a kernels possible throughput exceeds the best possible memory bandwidth of approximately 35 GB/s in Table I, e.g. the 12 times unrolled `SAvg` kernel. Another part of these gaps is attributable to call overheads, since we measure on the host from kernel initialization to return of the kernel call. However, some of the larger gaps must be attributed to either significantly lower memory performance for the implemented memory access patterns or to insufficient data buffers to keep the compute pipelines fed. It is important to note in this context, that in contrast to the somewhat erratic gaps f_{gap}, the individual measured hardware execution times t_{exe} are highly reproducible, with standard deviations (not in the tables) typically below 0.1% of t_{exe}.

Comparing the achieved hardware performance to CPU execution times of the same kernels, we observe a wide range of speedup $f_{speedup}$ compared to the corresponding CPU execution times. The general pattern of modest speedups for kernels with few simple operations and linear data access patterns and highest speedups for the `Scan` kernel with the highest compute effort per data element is not surprising. Yet, the range speedup factors between a barely worthwhile 2.4x and quite nice 36.3x is notable.

VIII. INTEGRATION

When executing the entire stereo-matching process with the presented 8 kernels, the overall speedup depends on the image dimensions and possible disparity values, as summarized in Table V. We present the total execution time t_{total} and the individual execution times for the aggregation and scanline phases t_{agg} and t_{scan}. For the FPGA-accelerated designs, we also measure the total time spend for data transfers between

Table V. EXECUTION TIME [S] OF ENTIRE STEREO-MATCHING PROCESS AND INDIVIDUAL PARTS FOR 3 IMAGE DIMENSIONS AND 3 DIFFERENT DESIGNS.

Configuration	t_{total}	t_{agg}	t_{scan}	t_{trans}	t_{switch}
384x384x16					
CPU	2.18	0.57	0.69	0.00	0.00
8 Kernels	21.49	19.13	1.49	0.15	20.12
$\texttt{Agg}_{4*12+4}$	3.01	0.77	1.42	0.17	1.60
1152x768x128					
CPU	96.88	28.91	55.21	0.00	0.00
8 Kernels	49.17	31.44	5.12	5.84	20.12
$\texttt{Agg}_{4*12+4}$	33.92	16.02	5.19	5.84	1.75
1920x1080x256					
CPU	580.60	137.68	392.07	0.00	0.00
8 Kernels	155.38	83.67	20.84	28.79	22.58
$\texttt{Agg}_{4*12+4}$	154.08	83.19	20.48	28.58	1.66

host and LMem t_{trans} and the time for reconfiguring the FPGAs t_{switch}. In addition to the previously investigated image dimensions, we added a very small image pair with limited disparity to outline the limitations of our approach. For this small example, we see a huge slowdown of approximately factor 10x when running all 8 kernels, caused by a high number of bitstream changes, particularly in the aggregation phase, taking around 0.35s each on our platform.

However, in Section VII we noted that several individual kernels didn't use all resources. Now we explore the options of implementing several kernels in one design — with reduced unrolling factor where required — to reduce the overhead of switching kernels. Due to the repeated aggregation steps, each of which uses all 5 aggregation kernels, we focused on a common aggregation design `Agg`. Other than for the individual kernels, we here considered also different clock frequencies, since kernels with reduced unrolling factor may no longer be bandwidth limited. The best design we found reached 130MHz with an unrolling factor of 12 for both `Sum` kernels, `VerDiff` and `Scale` and only 4 times unrolling for `HorDiff`, which otherwise again caused timing issues. We denote this fused design as $\texttt{Agg}_{4*12+4}$ and added its performance to Table V. With this design, the slowdown for the small image pair is reduced to approximately 1.5x, with the middle image pair we reach an overall speedup of almost 3x and for the largest dimensions the speedup of the entire stereo-matching process approaches 4x. Coincidently, at these dimensions the performance of single kernels with highest processing speed and more reconfiguration time matches that of the fused aggregation kernel $\texttt{Agg}_{4*12+4}$.

IX. DISCUSSION AND OUTLOOK

We presented a kernel-centric acceleration approach that was for the first time able to support a software algorithm that, due to the nature of its dependencies, has not been successfully implemented on FPGAs so far. Our approach achieves a speedup of up to 3.8x compared to the pure software implementation.

Our implemenation exploits similar levels of parallelism as those of Shan et al. [11] and Wang et al. [12]. However, the overall performance is several orders of magnitude less than that of those implementations. To a large degree, this is due to a lack of pipelining between the different computation phases of our implementation, where each kernel is executed exclusively on the FPGA. To a smaller extent, it is caused by the higher number of computation steps (e.g. four aggregation steps instead of one) we execute, following Mei et al. [5]. Also our performance data contains practical bandwidth limitations, inefficiencies of some memory accesses patterns and data setup times on the host, whereas many pure FPGA stereo matchers are characterized in terms of raw throughput of the datapath.

What our kernel-centric methodology achieved on the other hand was acceleration as a 1:1 replacement of an existing software implementation, delivering exactly the same results. Other software implementations spend comparatively much more time (e.g. running several minutes on the smallest datasets) for small improvements in matching quality. Further more, with our approach most parameters of the overall stereo-matching application can still be adapted with software compile times (e.g. a few seconds), like the metrics used for cost initialization, the sequence of aggregation steps or the scaling step after aggregation. However, some important design parameters are fixed or require a time-consuming re-synthesis of the hardware kernels. These are the datatype of the cost values and for some kernels the maximal supported arm length and image width.

Such kernel-centric acceleration approaches will be required, when FPGAs are to be increasingly utilized as accelerators for high performance and data center workloads, which require exact reproduction of software results. In environments with frequent updates to the executed tasks, the reusability of hardware kernels when only part of the software is changed is a welcome bonus. Depending on the computation time per kernel, configuration changes can become a major problem. Multi-context FPGAs as repeatedly proposed in academia may be a solution, or when the application allows, configurable kernels as utilized by Putnam et al. [2].

However, we have seen that kernel-centric acceleration comes at a considerable performance cost compared to a full processing pipeline. A major performance limiting factor in this project was the off-chip memory bandwidth and the coarse granular memory access patterns required by the memory interface for all data that exceeds the capacity of available BRAMs. Upcoming FPGA platforms with stacked memory with capacities and bandwidths in-between the wide gap between current BRAM and off-chip DRAM may very much mitigate this problem.

ACKNOWLEDGEMENT

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901), the European Union Seventh Framework Programme under grant agreement no. 610996 (SAVE), and the Maxeler university program MAXUP.

REFERENCES

[1] M. Herbordt, T. Van Court, Y. Gu, B. Sukhwani, A. Conti, J. Model, and D. DiSabello, "Achieving high performance with FPGA-based computing," *Computer*, vol. 40, no. 3, pp. 50–57, March 2007.

[2] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proc. Int. Symp. on Computer Architecture (ISCA)*, 2014.

[3] B. Tippetts, D. J. Lee, K. Lillywhite, and J. K. Archibald, "Hardware-efficient design of real-time profile shape matching stereo vision algorithm on FPGA," *Int. Journal of Reconfigurable Computing*, 2014.

[4] K. Ruhl, M. Eisemann, and M. Magnor, "Cost volume-based interactive depth editing in stereo post-processing," in *Proc. European Conference on Visual Media Production (CVMP)*. ACM, 2013.

[5] X. Mei, X. Sun, M. Zhou, S. Jiao, H. Wang, and X. Zhang, "On building an accurate stereo matching system on graphics hardware," in *Proc. ICCV Workshop on GPU in Computer Vision Applications (GPUCV)*, 2011.

[6] K. Zhang, J. Lu, and G. Lafruit, "Cross-based local stereo matching using orthogonal integral images," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 19, no. 7, pp. 1073–1079, Jul. 2009.

[7] H. Hirschmüller, "Stereo processing by semiglobal matching and mutual information," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 30, no. 2, pp. 328–341, Feb. 2008.

[8] H. Hirschmüller and D. Scharstein, "Evaluation of stereo matching costs on images with radiometric differences," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 31, no. 9, pp. 1582–1599, Sep. 2009.

[9] D. Scharstein and R. Szeliski, "A taxonomy and evaluation of dense two-frame stereo correspondence algorithms," *Int. Journal on Computer Vision*, vol. 47, no. 1-3, pp. 7–42, Apr. 2002.

[10] T. Kenter, H. Schmitz, and C. Plessl, "Pragma based parallelization – trading hardware efficiency for ease of use?" in *Proc. Int. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE Computer Society, Dec. 2012, pp. 1–6.

[11] Y. Shan, Y. Hao, W. Wang, Y. Wang, X. Chen, H. Yang, and W. Luk, "Hardware acceleration for an accurate stereo vision system using mini-census adaptive support region," *IEEE Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 4s, pp. 132:1–132:24, Apr. 2014.

[12] W. Wang, J. Yan, N. Xu, Y. Wang, and F.-H. Hsu, "Real-time high-quality stereo vision system in FPGA," in *Int. Conf. on Field Programmable Technology (ICFPT)*, Dec 2013, pp. 358–361.

[13] B. da Silva, A. Braeken, E. D'Hollander, and A. Touhafi, "Performance modeling for FPGAs: extending the roofline model with high-level synthesis tools," *Int. Journal of Reconfigurable Computing*, vol. 2013, pp. 1–10, 2013.