Exploring Trade-Offs between Specialized Dataflow Kernels and a Reusable Overlay in a Stereo-Matching Case Study

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*—FPGAs are known to permit huge gains in performance and efficiency for suitable applications through customization and exploitation of parallelism on different levels. Recently, they are starting to show up in data center environments, accelerating e.g. search engines, finance applications or geophysical simulations through carefully designed and co-designed implementations. Once those systems with FPGA accelerators are in place, we see an additional demand to harvest at least some of their acceleration potential with reduced design efforts or shorter development cycles for further applications. In this work, we compare the resulting performance of two design concepts that in different ways promise such increased productivity.**

As common starting point, we employ a kernel-centric design approach, were computational hot-spots in an application are identified and individually accelerated on FPGA. By means of a complex stereo-matching application, we evaluate and compare two very different design philosophies for the implementation of the required kernels on FPGAs. On the one hand, we implement individually specialized dataflow kernels in a spatial programming language for a Maxeler FPGA platform, on the other hand, we target a vector coprocessor with large vector lengths that is implemented as a form of programmable overlay on the application FPGAs of a Convey HC-1. We assess both approaches in terms of overall system performance, raw kernel performance and performance relative to invested resources. Additionally, we report our personal experience in perceived implementation effort and tool runtimes.

The differences between the two fundamentally different approaches to FPGA kernel design turn out to be smaller than expected. After compensating for the effects of the underlying hardware platforms, the specialized dataflow kernels on the Maxeler platform are around 3x faster than kernels executing on the vector overlay on the application FPGAs of the Convey HC-1. In our concrete scenario, due to trade-offs between reconfiguration overheads and exposed parallelism, the advantage of specialized dataflow kernels is reduced to around 2.5x.

I. Introduction

In order to achieve the best possible performance for application acceleration on FPGAs, the entire design process, from selection of the algorithm, of compute patterns 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 such an holistic design paradigm is established, 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 where 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. This approach is a known concept in HW/SW co-design [3], [4] and also widely employed in the field of GPU acceleration [5], [6]. When we want to propagate more widespread utilization of FPGA accelerators with this approach, beneath good performance of accelerated designs, we necessarily need efficient design methods for the FPGA kernels.

In this work, we evaluate and compare two very different design philosophies for the implementation of such kernels on FPGAs. The first method applied is the design of specific dataflow implementations for each individual kernel in a spatial programming language, which promises high productivity while still retaining most of the result quality that can be achieved by low level design techniques. We evaluate this method on a Maxeler platform [7], using the MaxJ [8] language to specify the kernel designs and targeting a MAX3424A Vectis [7] accelerator card. The second method in contrast utilizes for all kernels the same instruction programmable overlay architecture, a vector coprocessor with very wide vectors, which is one usage mode of the Convey HC-1's [9] application FPGAs. In this work, the vectorized kernels executed on this architecture are hand-written assembly code. With the comparison between the two methods, we primarily want to assess the resulting performance in order to find out, whether such a quite generic overlay architecture as the utilized one can performance-wise be a viable solution when development time is limited. As secondary objective, we also quantify the actually required development effort and know-how.

For this comparison, we accelerated an existing stereo-matching algorithm with a kernel-centric acceleration ap-

Published in: *Int. Journal of Reconfigurable Computing (IJRC)*

proach for both design paths. The algorithm offers an interesting mix of parallelization opportunities in some problem dimensions and dependencies in other dimensions. Because of the dependency pattern, it can't be implemented as single design, that is fully pipelined through all of its compute stages, like other stereo-matchers on FPGAs. In this algorithm, we identified 10 runtime intense kernel functions and offloaded them to the target FPGAs. The group of kernels contains straightforward streaming kernels, kernels with mildly irregular data access patterns and kernels with the dependency pattern of a dynamic programming approach. Through slight generalization of one group of kernels, we implemented 8 specific dataflow designs to execute those kernel functions, one of the designs executing an auxiliary step that is required to efficiently support the data access pattern for one pair of functions. On the vector coprocessor, the 10 kernel functions are executed by 10 directly corresponding assembly functions.

In this work, we build closely upon a previous publication, in which most aspects of the kernel-centric acceleration of stereo-matching with specialized kernels in a spatial programming language are presented [10]. Additionally, earlier work using the vector overlay [11] was heavily modified to match the functionality and kernel selection on the dataflow platform.

Beyond the consolidated presentation of both design paths side-by-side, the specific contribution of this work is the in-depth comparison of the two approaches, compensating for the effects of the two different hardware platforms and thus allowing to assess the overheads and opportunities of using overlay architectures on FPGAs in a non-trivial and practically relevant use case.

The remainder of this paper is structured as follows: In Section II we first outline the general stereo-matching task, before presenting in Section III the concrete algorithm we accelerate. We then introduce in Section IV the two accelerator platforms and how they are programmed in this work. Before the concrete kernel implementations are described side-by-side for both platforms in Section VI, we outline the common acceleration principles and memory management concepts in Section V. Comparing both systems in Section VIII requires some normalization to account for the different hardware platforms, but gives us insights about the trade-offs in runtime, design effort and tool runtimes. Section IX discusses related work for three aspects of our paper. In Section X, we conclude with a careful generalization of the results and outline future directions.

II. Introduction to the Stereo-Matching Problem

Stereo-Matching is the computation of a disparity map from a pair of stereo images. The disparity specifies at every position in the image, how far the displayed object or feature appears displaced between the two images due to the different positions of the two camera lenses. In earlier work, the term *Stereo Correspondence* has been used for the same problem [12], [13]. By inverting the disparity information and scaling it according to the geometry of the camera system, actual depth information about the scene is obtained, which is denoted as *Stereo Vision* and is probably the most important method for computer vision.

Applications for computer vision in general and stereo-matching in particular range from automotive and industrial use cases over robot navigation [14] to 3D movie production and general 3D data acquisition. 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.

Most stereo-matching algorithms perform so called dense stereo-matching, that is, they compute a disparity map, containing a disparity value for each pixel. This value represents how far the object that this pixel belongs to appears shifted between the left and right stereo image. More formally, a disparity value $d = d_{left}(x, y)$ for a pixel in the left stereo image at position (x, y) signifies that the physical feature displayed by this pixel is believed to be found in the right stereo image at position $(x - d, y)$. If a corresponding right disparity image is computed, to be consistent, the corresponding disparity in the right image, $d_{right}(x-d, y)$ should also contain the same disparity value d, pointing back to position $([x - d] + d, y) = (x, y)$. For this definition of disparity and consistency to be precise, the two images need to be perfectly horizontally aligned.

As auxiliary metric to compute disparities, many algorithms use a cost value for each pixel at each possible disparity $C(x, y, d)$, thus forming a three dimensional cost volume, where a low cost signifies that it is plausible that this pixel should have the corresponding disparity.

The general sequence of modern stereo-matching approaches comprises three steps [15]: first, computation of a matching cost volume, second, an optimization method which computes a disparity map from the cost volume and third, post-processing of the disparity map. For computing the initial cost volume, metrics for local color similarity and for local structural similarity are commonly employed [16], [17]. To smoothen the cost volume, aggregation techniques can be employed [18], [12]. Optimization in the simplest form, often called WTA (winner-takes-all) [17], just selects the disparity with the lowest cost for each pixel: $d(x, y) = \underset{d}{\operatorname{argmin}} C(x, y, d)$. Other approaches like belief-propagation (BP) and graph-cuts (GC) seek to combine low matching costs with properties like low energy of the resulting disparity map. Beyond generic image augmentation approaches, post-processing often involves consistency checks between the disparity maps generated for left and right image and

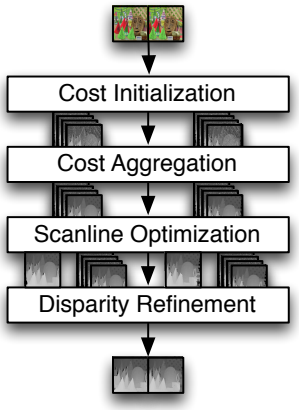

Figure 1. High-level overview of the stereo-matching algorithm following [16]. From a pair of stereo images, intermediate cost volumes are computed, which are used to generate disparity maps as final result.

handling of the identified inconsistencies [19], [16].

For many stereo methods, there exist variants which incorporate more than two input images, typically but not necessarily captured from a set of cameras placed along one horizontal line. The additional viewpoints open up additional opportunities for consistency checks among derived disparity maps and can particularly help to fill occluded areas, when they are visible in one of the additional input images. However, these variants still rely on a good underlying stereo-matching algorithm, like the one utilized as case study here.

III. Stereo-Matching algorithm with inherent parallelism

In our work, we algorithmically follow the stereo-matching implementation published by Mei et al. [16]. It follows the three basic steps outlined in Section II, but splitting the first step of cost computation into two separate phases, we subdivide it here into a total of four phases. Figure 1 gives a high-level overview of the stereo-matching sequence. In the first phase, cost initialization, two similarity metrics are applied on the input images to compute for each pixel and each possible disparity a local cost value, thus forming the first cost volumes. 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. This smooths the original cost volumes. In the third phase, scanline optimization, an energy minimization approach is mimicked by dynamic programming along 1-dimensional scanlines. This produces a first pair of disparity maps, but also another pair of cost volumes that are used in the fourth phase, disparity refinement. This fourth phase performs a consistency check between the left and right disparity maps and applies several local optimizations for pixels which are not classified consistently.

As the most time consuming parts and parts where the accelerated kernel functions are located, we present some details about the mechanisms of cost aggregation and scanline optimization, and briefly outline the two less time consuming steps cost initialization and disparity refinement, which are executed on CPU in our work.

A. Cost Initialization

The cost initialization following Mei et al. [16] provides the first cost metric $C_i(x, y, d)$ for each position and disparity based on two individual components. The first component is called the absolute difference cost C_{AD} for a pair of left- and right-image pixels in RGB format. This cost is defined as the difference of pixel intensities I, averaged over the three color channels: $C_{AD}(x, y, d) = \frac{1}{3} \sum_{i=R,G,B} |I_{left,i}(x, y) - I_{right,i}(x - d, y)|$

The second component is the census cost C_{census}, computed as the Hamming distance of the census transforms of a left and corresponding right pixel. The census transform captures the local structure in a 9×7 window around each pixel. As structural information, it is less sensitive to variations in lighting between the left and right image.

These two cost components are individually scaled by an exponential function that also enables the weighting of outliers and then added up to form the initial cost.

B. Cost Aggregation

The idea of cost aggregation is to reduce the huge amount of noise contained in the local cost metrics. Instead of simple smoothing, the costs for each possible disparity 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 [20]. 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 Figure 2. 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. Pseudocode for the horizontal aggregation step is given in Algorithm 1. As the outer loop indicates, the step is performed independently on each disparity d. The first loop nest computes for each row the running sum of costs from the row's first element to the current element. In the second nested loop, for each position (x, y), the difference between

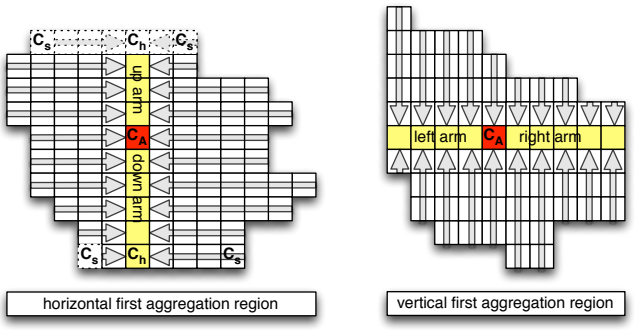

Figure 2. Illustration of cross-based cost aggregation regions, left projecting all horizontal arms on the vertical arms of a pixel, right projecting vertical arms on the horizontal arms of a pixel.

Algorithm 1 Horizontal aggregation step. Note: **for-all**-loops are independent, **for**-loops are ordered.

Require: $\forall(x, y, d)$: $C_i(x, y, d)$ = input cost
Require: $\forall(x, y, d)$: $A_{left/right}(x, y, d)$ = arm lengths
Ensure: $\forall(x, y, d)$: $C_h(x, y, d)$ = aggregated costs of row segment around position (x, y) in disparity d
  for all $d \in$ disparities **do**
    for all $y \in$ rows **do** {compute running sums}
      $C_s(0, y, d) \leftarrow C_i(0, y, d)$
      for $x = 1$ **to** #columns **do**
        $C_s(x, y, d) \leftarrow C_s(x-1, y, d)$ + $C_i(x, y, d)$
      end for
    end for
    for all y $\in$ rows **do** {compute row segment costs}
      for all $x \in$ columns **do**
        $a_l \leftarrow A_{left}(x, y, d)$
        $a_r \leftarrow A_{right}(x, y, d)$
        $C_h(x, y, d) \leftarrow C_s(x+a_r, y, d)$ - $C_s(x-a_l-1, y, d)$
      end for
    end for
  end for

two elements of the running sum is taken, with element positions defined by the arm lengths at (x, y). This difference is exactly the sum of costs in the horizontal segment around (x, y) that is specified by the two arms. In Figure 2, aggregation for one disparity is illustrated for the topmost and bottommost rows of the horizontal first aggregation region, where the horizontally aggregated costs C_h depend on two elements of running sums C_s. Afterwards, in the vertical aggregation step, vertically running sums (not illustrated in the Figure) are computed and the aggregated costs C_A is computed again as difference between the running costs at two positions, here at the two positions that are marked with C_h in the example. Note that the left and upper positions, from which the respective running sums are taken, are not part of the aggregation area itself.

A pair of horizontal and vertical aggregation steps forms one aggregation iteration with the illustrated horizontal first aggregation region. Following Mei et al. [16], we execute a total of four such aggregation iterations, the first and third one using horizontal first regions and the second and fourth one using vertical first regions. Not mentioned by Mei et al. [16] is a normalization step after each aggregation iteration, where the aggregated cost is scaled by the respective aggregation area. This was already proposed by Zhang et al [20], also utilized by Shan et al. [21] and we found it to be important for the result quality of our implementation.

C. Scanline Optimization

The scanline optimization follows Hirschmüller's [19] 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 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 and Algorithm 2 illustrates pseudocode for one scanline direction.

$$\begin{aligned} C_v(x, y, d) = C_A(x, y, d) + & \\ & \min\,[\, C_v(x - v_x, y - v_y, d), \\ & C_v(x - v_x, y - v_y, d \pm 1) + P_1, \\ & \min_k C_v(x - v_x, y - v_y, k) + P_2\,] \\ & - \min_k C_v(x - v_x, y - v_y, k) \\ (v_x, v_y) \in \{(\pm 1, 0), (0, \pm 1)\} & \end{aligned} \tag{1}$$

The scanline cost C_v in the equation is computed along a scanline path that depends on the direction (v_x, v_y), which in the pseudocode example is $(1, 0)$ to define a scanline to the right, with accordingly denoted scanline cost C_r. The scanline cost depends on the aggregation cost C_A and a term requiring all scanline costs at the previous pixel position along the scanline path. This previous pixel position is given by $(x - v_x, y - v_y)$ in the equation and by $(x - 1, y)$ in the pseudocode example. This term depending on the previous position reflects the energy minimization concept, selecting either the scanline cost from the previous position at the same disparity, or the scanline cost from the previous position at a neighboring disparity plus a small penalty P_1, or the minimal scanline cost of all disparities at the previous position plus a larger penalty P_2. These paths trade-off energy components added by the matching costs with energy components from the disparity profiles represented by the penalties P_1 and P_2. Not shown in the equation and pseudocode, both penalty values depend at each specific

Algorithm 2 Scanline optimization step in left to right orientation (denoted as `ScanRight`). Note: **for-all**-loops are independent, **for**-loops are ordered.

Require: $\forall(x, y, d)$: $C_a(x, y, d)$ = aggregated cost
Ensure: $\forall(x, y, d)$: $C_r(x, y, d)$ = right scanline cost
  for all $y \in$ rows **do**
    for all $d \in$ disparities **do**
      $C_r(0, y, d) \leftarrow C_a(0, y, d)$
    end for
    $C_{min}(0, y) \leftarrow \min_k C_r(0, y, k)$
    for $x = 1$ to #columns **do**
      for all $d \in$ disparities **do**
        $c_0 \leftarrow C_r(x-1, y, d)$
        $c_{-1} \leftarrow C_r(x-1, y, d-1) + P_1$
        $c_1 \leftarrow C_r(x-1, y, d+1) + P_1$
        $c_k \leftarrow C_{min}(x-1, y) + P_2$
        $C_{path} \leftarrow \min(c_0, c_{-1}, c_1, c_k)$
        $C_r(x, y, d) \leftarrow C_a(x, y, d) + C_{path} - C_{min}(x-1, y)$
      end for
      $C_{min}(x, y) \leftarrow \min_k C_r(x, y, k)$
    end for
  end for

position on the color differences of the original images. Finally, for normalization, the minimal scanline cost at the previous position is subtracted.

Figure 11, shown later in Section VI-B, serves us mainly to illustrate the compute and parallelization pattern of our implementations, but also contains a numeric example of a scanline computation, here of a downward scanline. For simplicity, costs are represented as integer values and with an aggregation cost of 0 for the second line. Green arrows indicate the minimization paths taken to compute the scanline costs in the second row depending on the previous row and the input aggregation costs. These green arrows reflect the best trade-off between minimization of the input costs and the scanline energy for any given position.

In the abstract description of stereo-matching approaches in Section II the optimization step was described to yield a disparity map, yet the scanline equation as described here computes a new cost volume, now incorporating a trade-off between raw matching costs and energy of the disparity map. This is convenient, as now the results of scanline optimization steps along different directions can simply be combined by computing the average of different scanline costs. On the combined scanline costs, now a WTA optimization selects the actual disparity for each pixel.

We use four directions, up, down, left and right like proposed by Mei et al. [16]. Each scanline by itself produces some streaking artifacts in the direction of the scanline, because the penalty values only favors persistence of previously optimal disparities along the scanline, but not in the reverse direction. Therefore it is important to utilize not only several different scanlines like in [22], but also to have pairs of reverse scanlines to symmetrically offset the streaking.

D. Disparity Refinement

The previous three phases are executed for both the left and right image, producing one disparity image for each side. As indicated earlier, their computed disparity values should match: $d_{left}(x, y) = d_{right}(x - d_{left}(x, y), y)$. Pixels for which this is not the case are classified as outliers and are treated with the refinement steps *Iterative Region Voting* and *Proper Interpolation* from Mei et al. [16]. Due to insufficient details given, we skip their *Depth Discontinuity Adjustment* step, but again perform the subsequent *Sub-pixel Enhancement* step, which aims to reduce errors caused by the discrete disparity levels.

E. Software Implementation

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. [23]), an adjustable sequence of aggregation steps, 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 get values of 0, leading to artifacts. Thus, we use in our software implementation double precision and also require this from the FPGA acceleration. With the settings of Mei et al. [16], our implementation reaches an accuracy in the Middlebury benchmark [13] of average 5.73% bad pixels and we make sure during our acceleration process to still produce the same results.

IV. Utilized FPGA Platforms and Programming Models

In this section, we introduce the two hardware platforms we target and outline how they are programmed in this work. We conclude the section with a brief comparison of the accelerator resources as used in our experiments.

A. Maxeler Platform and Programming Paradigm

The Maxeler platform we use [7] is illustrated in Figure 3. It comprises two 6-core (12 threads) Intel Xeon X5650 (Westmere microarchitecture) CPUs, running at 2.66 GHz, as host platform and is equipped with four *MAX3424A Vectis* PCIe accelerator cards, of which in this work only one is used. Each card contains a large Xilinx Virtex-6 SX475T [24] FPGA for user logic, a smaller, non-user-programmable FPGA for the PCIe interface, and 24GB of

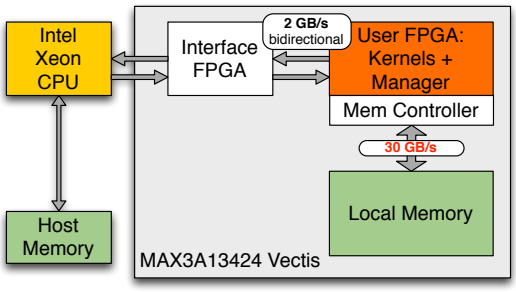

Figure 3. Illustration of the Maxeler platform with one MAX3 Vectis accelerator card.

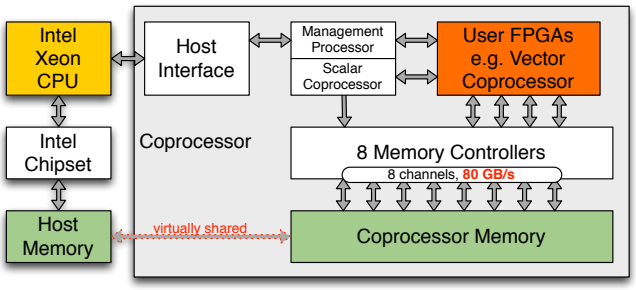

Figure 4. Illustration of the Convey HC-1 platform.

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 around 30 GB/s (with memory controllers synthesized at 300 MHz; up to 400 MHz are supported by the DDR3 DIMMs), several bursts, either adjacent or with a fixed stride, should be accessed with a single memory command. For example, commands with only 1 burst each lead to an efficiency of only 11%, whereas with 8 consecutive bursts, an efficiency of 80% is 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. Note that the memory controller is synthesized by the Maxeler tools onto the user FPGA alongside the custom logic.

The distinctive feature of the Maxeler systems is their development environment [8], which allows programming the FPGAs with a spatial programming language, denoted as *MaxJ* and realized as a Java extension. The kernel functionality implemented on FPGA is integrated with the host (CPU) part of an application through calls to an API automatically generated for the specified functionality. 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 automatically generates the required buffers and interfaces.

B. Convey HC-1 Platform with Vector Processor Overlay

The Convey HC-1 [9], illustrated in Figure 4, is a dual socket server system, where one socket is populated with a dual core Intel Xeon 5138 (Core microarchitecture) CPU, running at 2.13 GHz, while the other socket is connected to a stacked coprocessor board. The two boards communicate using the Intel Front-Side Bus (FSB) protocol. Both processing units have their own dedicated physical memory, which can be transparently accessed by the other unit through a common cache-coherent virtual address space, which distinguishes this platform from the Maxeler system. The coprocessor consists of multiple, individually programmable FPGAs. One FPGA implements the infrastructure that is shared by all coprocessor configurations. These functions include the physical FSB interface and cache coherency protocol as well as configuration and execution management for user programmable FPGAs. For implementing the application-specific functionality, four high-density Xilinx Virtex-5 LX330 [25] FPGAs are available. Eight memory controllers are implemented on one distinct Virtex-5 LX150 [25] FPGA per memory controller. Each of them accesses two DIMMs, which leads to an aggregated bandwidth of close to 80 GB/s with 16 memory modules. In our system configuration, custom-made scatter-gather DIMMs are installed, which allow accessing memory efficiently in 8-byte data blocks, while standard modules are designed for 64-byte block access.

The user FPGAs can be programmed with fully custom, problem specific designs, integrated into the rest of the system by interface libraries written in Verilog. Additionally, Convey offers a number of designs, so-called *Personalities*, which are developed as programmable accelerators for specific classes of tasks, such as graph traversal or local string alignment and, probably with the broadest scope, the so-called *Vector Personality*, which we use in this work. Since it is a programmable architecture on top of the programmable FPGAs, we consider this an overlay, which comes with abstraction benefits and overheads which we want to quantify in this work.

The Vector Personality provides the functionality of a vector coprocessor that executes programs targeting its vector instruction set. It comes in two variants, optimized for single- or double-precision floating point operations; both also support integer operations, e.g. for vectorized address calculations. According to our application, we use the double-precision Vector Personality. The vector instructions

Table I
HARDWARE RESOURCES OF THE TWO FPGA PLATFORMS AS USED IN OUR EXPERIMENTS. *MAXELER MAX3424A MEMORY CLOCK AND BANDWIDTH DEPEND ON USER DESIGN. **CONVEY HC-1 ACCESS GRANULARITY DEPENDS ON INSTALLED DIMMS.

Platform	Maxeler MAX3424A	Convey HC-1
Application FPGAs	1× Virtex-6 SX475T	4× Virtex-5 LX330
#6-input LUTs	**297600**	4× 207360 = **829440**
#36Kb BRAMs	**1064**	4× 288 = **1152**
#DIMMs	6	16
Memory controllers	on User FPGA	8 dedicated FPGAs
Memory Clock	300 MHz*, variable	300 MHz, fixed
Peak Bandwidth	**28.8** GB/s*	**74.4** GB/s
Min. Access Size	384 bytes	8 bytes**

are implemented for up to 1024 elements. A total of 64 vector registers are available and each can store such a set of 1024 elements. Besides the usual element-wise arithmetic vector operations, the vector instruction set contains memory instructions that distinguish it from typical SIMD vector instruction set extensions for general-purpose CPUs. It can load and store vectors where the elements are individually indexed and do not need to be aligned in a continuous memory location.

Convey includes a compiler to target this vector personality by annotating source code with pragmas, however we found it to be limited to simple array data structures and simple loop nesting patterns, which often requires significant code adaptations besides adding the vectorization pragmas. We fixed many of these shortcomings with the toolflow proposed in [26], however for the comparison of architectural overheads of the overlay, we wanted to achieve the best possible performance. Therefore for this work we designed all kernels by hand in assembly code, particularly exploiting on top of the capabilities of the automated toolflow additional opportunities as vector partitioning, vector register rotation and enhanced reuse of partially computed addresses.

C. Comparison of FPGA Platforms

Comparing the two hardware platforms, the Convey HC-1 is a few years older, with the utilized FPGAs being one generation behind, and the CPUs being two process shrinks (Intel *Tick*) and one microarchitectural change (Intel *Tock*) behind. On the other hand, when we compare a single Maxeler MAX3424A Vectis accelerator card to the coprocessor of the Convey HC-1, the latter incorporates a lot more hardware resources. Table I gives an overview of the accelerator hardware as used in our experiments. Together, the four FPGAs for the HC-1's application logic contain almost 3x more LUTs and some more BRAM resources than the single application FPGA of the MAX3424A. Similarly, the peak memory bandwidth of the Convey HC-1's coprocessor is around 2.5x higher than that of the Maxeler MAX3424A accelerator. This is essentially achieved by using more memory modules. Additionally, the Convey HC-1's memory controllers are implemented on dedicated FPGAs, in contrast to the Maxeler MAX3424A platform, where the memory controller is synthesized along with the application logic onto the same FPGA. For the Convey platform, this saves space on the application FPGAs and avoids timing issues when synthesizing new user designs. Finally, even though both platforms come closest to their peak bandwidth with linear access patterns, physically a much smaller access granularity is supported in the Convey HC-1 configuration we utilize.

In Section VIII, where we assess the effects of the two different approaches to kernel design, we need to compensate for the outlined differences of the hardware platforms.

V. KERNEL-CENTRIC ACCELERATION

The general idea of kernel-centric acceleration as followed here, is to identify runtime intense kernels with acceleration potential and execute them on FPGA, and to keep other, possibly complex parts of the application with small contributions to the overall runtimes on CPU. In order to identify the candidate kernels, we first performed profiling on CPU. The runtimes of all kernel functions with non-negligible runtimes, aggregated over all their invocations when they are executed more then once, are illustrated in Figures 5 and 6 for a FullHD input image pair on both CPU platforms. The kernels are sorted by the time of their first invocation, which reflects the overall sequence of cost initialization, aggregation, scanline optimization and disparity refinement, however there are repetition patterns spanning several of those kernels. Based on this result, we selected the 5 aggregation kernels from `horSum` to `scale` and the 5 scanline kernels from `scanUp` to `sumScanlines`. They cover 87% of the total program runtime, which permits by Amdahl's law a speedup of at most 7.8x.

Since both platforms investigated in this work have physically distinct accelerator memory, whenever possible, we want to leave data in this accelerator memory when it is read or modified by several different kernels or several invocations of the same kernel. Therefore beyond the raw execution times, possible data reuse between the kernels was considered. In case of our stereo-matching implementation, the selected kernels cover all cost-volume related compute steps of aggregation and scanline optimization, thus maximizing the reuse potential of data in accelerator local memory. Based on pure profiling runtimes, the final step of scanline optimization, `sumScanlines`, would be a less worthwhile acceleration target then e.g. the computation of census costs, but it reduces the amount of data to be transferred from accelerator memory to host memory significantly from four cost volume instances to a single one.

Both utilized target platforms require data to be moved between CPU and accelerator memory, but in different ways. The Maxeler platform [7] requires explicit data movement functionality added to each design by the designer and

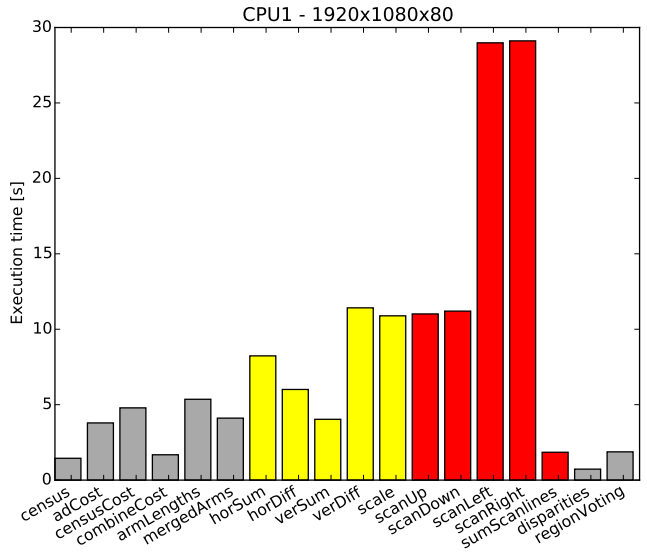

Figure 5. Runtimes that different kernels contribute to overall runtime on Maxeler CPU (denoted CPU1) for a FullHD image pair. Yellow and red bars indicate kernels in aggregation and scanline phase respectively.

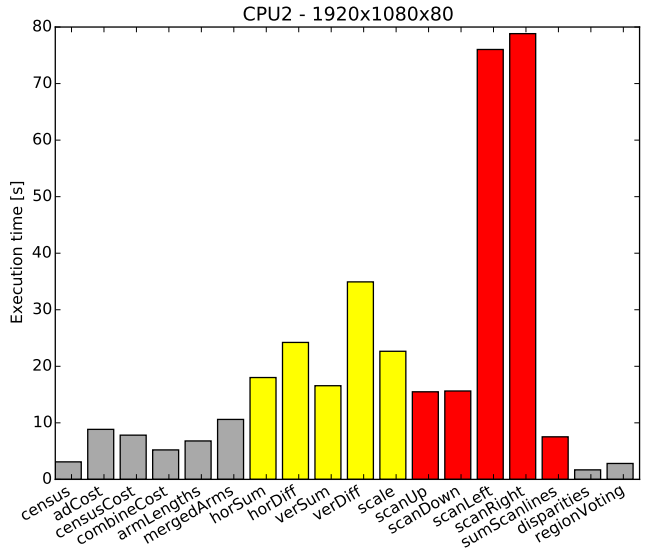

Figure 6. Runtimes that different kernels contribute to overall runtime on Convey CPU (denoted CPU2) for a FullHD image pair. Yellow and red bars indicate kernels in aggregation and scanline phase respectively.

the accelerator memory space is entirely managed by the developer [8]. The Convey platform [9] provides a shared memory space and different API functions for allocation on and movement between physical memory locations. In order to abstract these differences away from the application side, we modified and extended the memory manager presented in [10] for the Maxeler platform. An important feature of

```
1  cpuABtoB(double* a, double* b){
2    mm.reads(CPU, a);
3    mm.reads(CPU, b);
4    mm.writes(CPU, b);
5    // CPU kernel code here
6  }
7  cnyAtoB(double* a, double* b){
8    mm.reads(ACC, a);
9    mm.writes(ACC, b);
10   callCnyKernel(a, b);
11 }
12 maxAtoB(double* a, double* b){
13   mm.reads(ACC, a);
14   mm.writes(ACC, b);
15   callMaxKernel(mm.getLMem(a), mm.getLMem(b));
16 }
```

Listing 1. Different kernel functions using Memory Manager.

the memory manager, particularly useful during accelerator kernel development, is to support easy switching between CPU and accelerator execution of individual kernels with all required, but no unnecessary data movements.

Our means to achieve this was to express at the beginning of every kernel, which data structure it uses, whether it uses it at the host CPU or the accelerator and whether it reads or writes to this data structure. With this information, the memory manager keeps track of all data locations and initiates all required transfers prior to actual data accesses. In our new, extended memory manager concept, we applied these kernel annotations to both the kernels remaining on CPU and the wrappers for kernels executing on FPGAs. This goes beyond the modifications required for the methods presented in [10], where only data usage on FPGAs was indicated. The extension is however advantageous to the kernel centric acceleration concept, because it removes the only high-level application knowledge required for the previous version, where transfers from accelerator memory back to CPU had to be initiated manually, requiring changes for each accelerator kernel that is enabled or disabled during development.

Listing 1 illustrates some kernel functions using the memory manager interface. Before they actually use data, they indicate by calls to the memory manager API how (`mm.reads, mm.writes`) and where (locations `CPU, ACC`) they are going to use it. When a kernel both reads and writes data, or when it doesn't completely overwrite a structure, so previous data may still exist after writing, this has to be stated explicitly like in this example for the first function, using `b` both as input and output. The accelerator kernels (starting with `cny` for Convey, `max` for Maxeler) are mere wrappers and subsequently invoke execution on the respective accelerator. Due to the shared address space, the Convey kernel uses the original addresses, whereas the locations in Maxeler local memory are provided by the memory manager (`mm.getLMem`). Just like in [10], a memory region in Maxeler local memory is allocated lazily before the first usage of some data structure in this memory.

```
1 double* c_ad = (double*) mm.alloc(size);
2 double* c_init = (double*) mm.alloc(size);
3 double* c_agg = (double*) mm.alloc(size);
4 cpuABtoB(c_ad, c_init);
5 maxAtoB(c_init, c_agg);
6 cpuABtoB(c_init, c_agg);
```

Listing 2. Code sequence using Memory Manager with Maxeler kernel.

```
1 double* c_ad = (double*) mm.alloc(size);
2 double* c_init = (double*) mm.alloc(size);
3 double* c_agg = (double*) mm.alloc(size);
4 cpuABtoB(c_ad, c_init);
5 cnyAtoB(c_init, c_agg);
6 cpuABtoB(c_init, c_agg);
```

Listing 3. Code sequence using Memory Manager with Convey kernel.

Listing 2 now illustrates usage of two of those kernels. First, dynamic arrays are allocated through the memory manager, per default in host CPU memory. Then, for the first kernel call on CPU in Line 4, the memory manager determines at runtime that both arrays are already in the right location and no movement is required. In this example, the second kernel, Line 5, is executed on the Maxeler accelerator. For the data it reads, `c_init`, accelerator memory is lazily allocated and data is moved there from host. `c_agg` on the other hand is only written to, so it gets allocated in accelerator memory, but no data is actually moved. Line 6 now performs another kernel call on the host CPU. `c_init` was not modified in accelerator memory, so the memory manager internally still has it in a `shared` state and no data needs to be moved. `c_agg` however was `modified` in accelerator memory and on CPU it will now be read before it is possibly overwritten, so its data is transferred back by the memory manager.

Listing 3 repeats the same kernel pattern, just with the accelerated kernel being executed on the Convey platform instead of Maxeler. This time at the coprocessor kernel call in Line 5 no more memory is allocated since host CPU and accelerator share the same memory space. For the input data `c_init`, a similar data transfer is initiated as on the Maxeler platform, just using a different API with different arguments internally. For the output data `c_agg`, again no physical data transfer is required. For this purpose, the Convey API contains a `migrate_virtual` function, which doesn't actually move any data, but just lets the affected shared memory area point now to the physical accelerator memory. This function comes in two flavors, one that touches all affected memory pages to update internal state such as the TLB (Translation Lookaside Buffer) and the other one without this touching. The version with page touching guarantees the fastest raw execution time of subsequently executed accelerator kernels and thus is important for the later evaluation of kernel acceleration. On the other hand, we found the no-touch version in combination with allocation on host to yield fastest overall matching performance, because it partially overlaps the page touching effort with actual computation. It is even slightly faster than the alternative direct allocation as accelerator memory, even though the latter would require additional a-priori knowledge about the first usage location of a data structure. Therefore we measure and evaluate both versions in our experimental section.

These examples conclude this section on the selection of kernels for acceleration and the concepts and infrastructure to support memory management for both platforms through a common interface.

VI. Kernel-Designs for two FPGA platforms

In this section we present the compute and data-access patterns of 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 and data reuse patterns we implemented on the two accelerator platforms. The kernels for the Maxeler platform [8] are designed with a flexible amount of parallelism, which is specified by an unrolling factor f_u prior to synthesis. The actually utilized amount of parallelism, typically low two digit numbers, is either limited by resource or timing limitations during synthesis (horDiff and Scanline kernels) or by the known limits of the memory interface to feed the compute pipeline (all other kernels). For details of the synthesis results and bandwidth modeling, please refer to [10]. In order to hide feedback latencies in some kernels, in addition to this explicitly utilized parallelism, we also loop through different groups of work items in different clock cycles. For the Convey vector coprocessor [9], the desired amount of parallelism to be expressed by our kernel implementations is given by the size of the vector registers with up to 1024 elements. It internally contains 32 parallel function pipes and additionally makes use of further elements for latency hiding. We present for each kernel the designs for both platforms side-by-side to emphasize similarities and differences. We outline the designs of the first kernel in some detail whereas for the other kernels we restrict ourselves to noteworthy aspects.

A. Aggregation Kernels

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 and also for at least one of the image dimensions.

For the Maxeler platform, the independent image dimension suffices to support the required parallelism and latency hiding, so we restrict ourselves to unrolling in this dimension. The work of Shan et al. [21] suggests that utilizing disparity level parallelism in addition to image dimension parallelism might allow to save BRAM resources

```
1  void horSum(double *in, double *out){
2    long slice = height * width;
3    for(int d=0; d<=maxD; d++){
4      for(int y=0; y<height; y++){
5        out[d*slice + y*width] = in[d*slice + y*width] ;
6        for(x=1; x<width; x++){
7          out[d*slice + y*width + x] =
8            out[d*slice + y*width + x-1] + in[d*slice + y
                 *width + x];
9        }
10     }
11   }
12 }
```

Listing 4. Horizontal integral sums.

at the cost of additional logic utilization, which we did not investigate further for our kernels.

On the Convey platform, small image sizes do not suffice to fill the available vectors size. With vector partitioning, vectors can work on several groups of data, separated by so-called partition offsets. For the aggregation kernels, we use this feature to exploit both parallelism in image dimensions inside each partition and parallelism in disparity dimensions by multiple vector partitions.

Horizontal Integral Sums: After Section III already presented simplified pseudocode for the horizontal aggregation step, Listing 4 presents the corresponding function with the actual indexing used in our software implementation. There are dependencies along the rows, but we can parallelize computation by vertical unrolling, that is computing several rows in parallel, and additionally work on independent disparity dimensions for Convey vector partitions.

Figure 7 illustrates the computation pattern implemented on the Maxeler platform. The product of unrolling factor f_u and feedback latency l_f determines the number of rows that are in-flight at the same time as one common block. The latency is given by estimates from the Maxeler tools, whereas f_u is limited either by bandwidth estimations or synthesis results. More than $f_u * l_f$ rows in the same block are possible, but require larger buffers and provide no further advantages. After an entire block of rows is computed, the next block of rows, not shown in the illustration, is started. Finally, also not illustrated, after one entire image (a slice of the cost volume) is finished, computation continues with the next disparity. In this description, the presented compute pattern now governs the required memory access pattern, however in practice both are closely codesigned.

In memory, elements are arranged in row-major order, which means that entire rows are stored in continuous memory locations one after the other, because LMem uses the same data layout as the host application to allow for the memory management outlined in Section V. Thus each burst of 384 bytes reads 48 subsequent double values from each row. Figure 8 illustrates the way data is read from LMem with an appropriate memory command generator. We see that inside each block, between memory accesses and

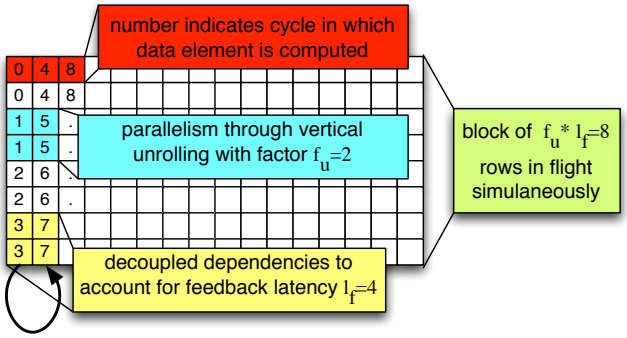

Figure 7. Illustration of compute order for horizontal sums on Maxeler. Here, an unrolling factor $f_u = 2$ and a feedback latency l_f is illustrated.

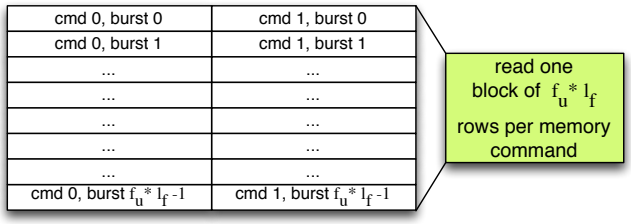

Figure 8. Illustration of memory accesses pattern for horizontal sums on Maxeler. Each burst spans 48 elements, every command generates accesses for an entire block of data. Inside each block, data needs to be reordered for the compute pattern.

compute step, the data needs to be reordered from horizontal to vertical order. This is relatively easy to do with the *MaxJ* concept of stream offsets, however the actual design may need considerable amounts of BRAM. Additional simpler, non-reordering buffers are required to keep the memory and compute pipelines fed.

The implementation for the Convey vector coprocessor, as indicated in the introduction to this section, not only uses the same unrolling into the independent image dimension, here vertically, but additionally can work on more than one disparity dimension in different vector partitions. Figure 9 illustrates this pattern with 4 partitions and 256 rows covered by each partition. The innermost loop runs horizontally inside the rows to reuse the vector register containing the previous integral sum as one of the two inputs for the next step. Before entering this innermost loop, for each group of rows, the number of partitions and size of the partitions is computed based on remaining dimensions and two offsets are written into configuration registers. One is the row offset between two consecutive vector elements inside each partition, the other one the image slice offset between two consecutive disparity levels in the cost volume. Vector load and store instructions use these offsets to determine the memory addresses of each vector element and, in this loop

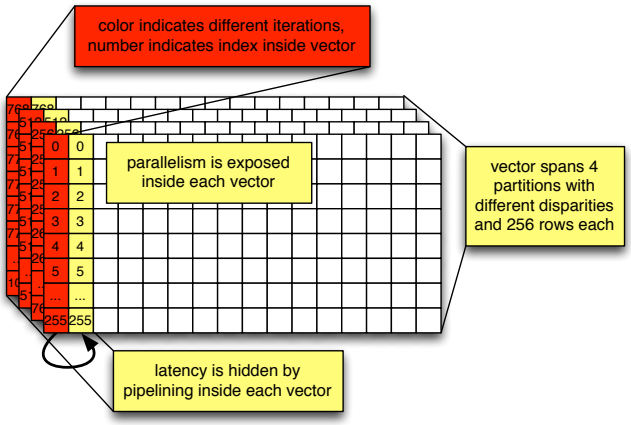

Figure 9. Illustration of compute order for horizontal sums on Convey vector coprocessor. Here, 4 partitions with 256 elements each are illustrated.

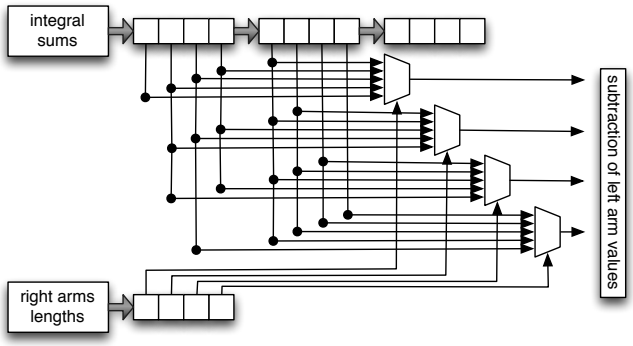

Figure 10. Illustration of right arm part of HorDiff for a maximum arm length of 5 (multiplexer size) and an unrolling factor f_u of 4.

profiting from the small access granularity of the scatter-gather RAM, only access the specified vector elements in memory.

Vertical Integral Sums: The vertical integral sum kernel (`VerSum`) is orthogonal to the `HorSum` kernel and contains vertical dependencies. Consequently, we now unroll computation horizontally for the implementations on both platforms.

Our Maxeler compute kernel combines the same combination of unrolling and feedback latency hiding as the `HorSum` kernel illustrated in Figure 7, just horizontally. When we buffer entire rows instead of blocks inside each row, the compute pattern exactly fits the data layout in memory, so we can use a linear memory access pattern instead of a customized memory command generator.

Similarly, the Convey `VerSum` vector kernel contains the same features, vector partitioning and data reuse in the innermost loop, as the `HorSum` kernel, but with vectors in horizontal image dimensions. Now the memory accesses inside each vector partition are continuous, which is beneficial for effective memory performance. In the vector processor instruction set, the only difference is, that the element stride is now set to the element size of 8 bytes.

Horizontal Differences: After the computation of the horizontal integral sums (`HorSum`) follows the step of computing horizontal differences (`HorDiff`). For each pixel, a left and a right arm length are required, which define the two positions in the integral cost rows to access and subtract from each other. So in this kernel we have data dependent memory accesses, however only with bounded offsets from a given position, which are limited by the maximal arm lengths. There are no dependencies in this kernel, so both horizontal and vertical unrolling are possible.

Since this kernel does not contain feedback, latency hiding as used on the Maxeler platform for the integral sum kernels is not needed here. With the burst oriented Maxeler memory interface, we need to have the window of possibly required integral cost data available in local buffers. This seems straightforward when unrolling horizontally, since neighboring pixels in one row require largely overlapping areas of possible input values defined by the arms. Figure 10 illustrates the use of multiplexers for 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 proposed in [16]).

Figure 10 illustrates, that the overlapping of the possible access windows makes the buffer very space efficient, in the example actually using only 8 registers to buffer the possible inputs for 4 parallel accesses with 5 different input options each. However, even though resource utilization would permit it, the synthesis tools were not able to route such a design with more than $f_u = 8$ anywhere near 100 MHz. The illustration in Figure 10 may give an intuitive idea that high number of overlapping signal routes to the different multiplexers causes this problem.

As an alternative, we tried vertical unrolling like in the previous kernel. Here, in addition to the resource consumption of reordering between row oriented memory access and column oriented compute step, for each parallel row a buffer for the possibly accessed input elements needs to be instantiated. With this approach, unrolling was limited by resource consumption after synthesis. Therefore, specifically for this kernel, in our final Maxeler design we combined horizontal and vertical unrolling, achieving the largest synthesizable design with overall parallelism of 16 through horizontal and vertical unrolling factors $f_{uh} = 4$ and $f_{uv} = 4$.

On the Convey vector processor platform, conceptually the vector registers might provide a similar line buffer for input cost values selected by arms. However, the instruction set does not support any form of parallel access to specific indexed elements of the vector, so this is not possible.

Instead, we resort to computing the address of each element of the horizontal integral sums that needs to be accessed by adding the arm length value to each respective base address. Then these addresses are used for indexed vector load operations, which are however less efficient then the accesses with regular strides as outlined for the previous kernel.

We again use multiple vector partitions covering several disparity values in each computation step for efficient utilization of the vector size with small images. Since there is no dependency of the two inner loops, the parallelism in each partition can be provided either by horizontal or by vertical vectors. After implementing and measuring both alternatives, we use vertical unrolling to form the vectors. When forming horizontal vectors, several loads of input cost values inside a vector load may point to the same location. This works functionally correct, but apparently causes additional latencies in the memory interface.

Vertical Differences: Similarly to the `HorDiff` kernel, the computation of vertical differences (`VerDiff`) does not contain any dependencies. On the Maxeler platform, horizontal unrolling does not suffer from the routing and timing difficulties of the `HorDiff` kernel, because now selection of arm positions is realized independently for each column. Thus, we can restrict unrolling to the horizontal dimension here and still reach unrolling values up to $f_u = 24$. On Convey we again use vector partitioning and this time unroll the vectors horizontally, following the data access pattern of the vertical summation and again avoiding indexed vector loads to contain several instances of the same address.

Scaling: Finally in the scaling kernel (`Scale`), each aggregated value is scaled, that is divided by the size of its specific aggregation region. It is a straightforward streaming kernel without dependencies and on both platforms can be readily unrolled horizontally, following a linear memory access pattern. On the other hand, the division of a double precision floating point values is neither easy nor efficient to implement on the Maxeler platform and not supported in the vector instruction set of the Convey coprocessor. Fortunately, there is a only fixed number of discrete sizes A_a that any aggregation region can have, so instead of division, we can precompute the inverse values $1/A_a$ and multiply by those inverses. On the Maxeler platform, those precomputed factors are stored in BRAM and looked up locally. For each parallel function pipe, a separate block of BRAM is instantiated. Due to the indexed access pattern, the Convey vector coprocessor again cannot use the vector registers to hold those values, but instead reads them with indexed vector loads from memory. Again, lookups to the same address impair performance, so on this platform we replicate the block of lookup values in memory and use the vector indices to distribute lookups to different blocks.

B. Scanline Kernels

In contrast to the aggregation, the scanline 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.

Vertical Scanlines: On the Maxeler platform, we implemented a common vertical scanline compute kernel (`ScanVer`), suitable both for the `ScanUp` and `ScanDown` kernels of the host application, switching between both modes by configuring the accompanying memory command generator for different access directions. Figure 11 illustrates the dependency pattern for a downward scanline computation and how it can be unrolled horizontally, here with boxes of size 4. All yellow boxes are required as inputs to compute the red boxes. The aggregation costs are read in in the required pattern as inputs, as well as the color difference information (not illustrated in the Figure) needed to determine the penalty values for each row (boxes P_1 and P_2). The resulting scanline costs are written out to LMem, but for an entire disparity range also buffered locally in BRAM for reuse in the next row. Therefore, computation is performed in blocks, but not in entire rows, as this would require excessive buffer space or additional read-backs.

In our actual Maxeler implementation, due to the burst size of the 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 from 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 scanline costs and a simple comparison for the selection of the current minimal scanline value. Nevertheless, similar to the integral sum kernels, we incur a feedback latency l_f of four cycles, which for the block size of 48 limits the possible unrolling in space with unrolling factor f_u to 12 ($l_f * f_u = 48$). We also tried larger block sizes to obtain more possible compute throughput, but the resulting large designs failed to meet timing.

On the Convey vector processor platform, we implemented two individual assembly kernels for `ScanUp` and `ScanDown` to save unnecessary selection instructions for the direction, but both implementations have identical structures. According to the dependence pattern, vectors cover entire rows or parts of rows, depending on image sizes. In contrast to the aggregation kernels, vector partitioning into different disparity dimensions is not possible.

The compute order looks very similar to the one illustrated for Maxeler in Figure 11, just with much larger blocks formed by the vectors. On this platform, the scanline costs of the previous line can't all be buffered for the next line,

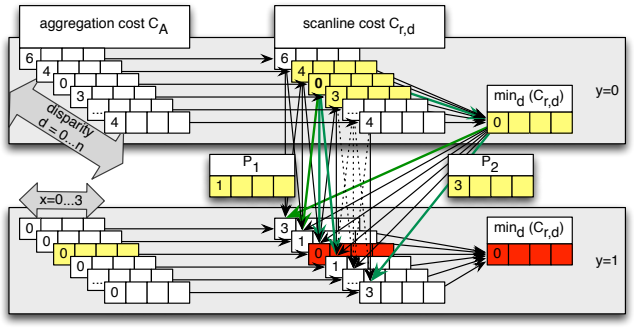

Figure 11. Illustration of downward scanline compute pattern with blocks of width 4 indicate the unrolling potential in x-dimension. Arrows indicate dependencies, for the computation of the red boxes (scanline cost is computed, minimum is updated), data from all yellow boxes is required. Paths are chose either as previous scanline cost at the same disparity without any penalty (red box), or as previous scanline cost at neighboring disparity plus penalty P_1 (boxes next to red box), or as global minimum of previous line plus penalty P_2 (outer boxes).

so they are read back at every iteration of the vertical loop. However, only one disparity block needs to be read for every newly computed block, the other two are reused from the previous iteration of the innermost loop, the disparity loop. E.g. in the Figure 11, the yellow block with scanline cost 3 was newly read for the current compute step, the other two yellow scanline costs are reused, which is done efficiently by using the vector register rotation feature of the vector instruction set. Also, our code makes heavy use of vector mask generation and vector element selection to find the different possible scanline paths inside one single vector, which can be used by the vector internal streaming of the coprocessor to skip masked out elements.

Horizontal Scanlines: On Maxeler, for horizontal scanlines (`ScanLeft` and `ScanRight`) the orthogonal unrolling concept from vertical scanlines with buffering the entire previous scanline costs was not applicable due to prohibitive BRAM requirements. This is because bursts were still aligned horizontally, but unrolling would have to be done vertically and additionally the buffers would have to cover all disparity dimensions. Therefore we decided to implement an auxiliary turn kernel (`Turn`) that reads cost arrays in row-major data layout and writes them back to LMem in column-major data layout, or vice-versa. Now we can execute horizontal scanlines by a sequence of turning input aggregation data, applying vertical scanlines and turning scanline result data back. The overhead of this turning steps gets mitigated, because both horizontal scan kernels use the same turned input data by utilizing the `ScanUp` and `ScanDown` variants of the vertical scan kernels.

The `Turn` kernel uses 48 BRAM blocks in which data is written to and read from with a diagonally shifted addressing scheme, which provides the flexibility that either an entire row or column of 48 values can be accessed. The size 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 48.

On the Convey coprocessor platform, the finer grained access capabilities of the memory interface allow direct implementation of the horizontal scanline kernels without prior turning. The kernel structure is very similar to the vertical one, just using row strides between the vertically unrolled vector elements as for the `HorSum` and `HorDiff` kernels.

Average over Scanline Directions: 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. On both platforms, the resulting `ScanAvg` kernel is a straightforward streaming kernel with four linear input and one linear output streams. As outlined in Section V, its particular value for the overall implementation lies in the reduction of output data size that has to be transferred back from accelerator memory to host memory.

This concludes the part of this Section covering kernel designs for both platforms. On the Convey platform, the kernels were directly integrated into an heterogeneous executable by filling empty proxy kernels with the proper signature compiled by the Convey compiler with the actual assembly code providing the described functionality. On the Maxeler platform, the kernels defined in the *MaxJ* language are synthesized to kernel-specific FPGA dataflow designs which is summarized in the following Subsection.

C. Synthesis and Integration

As indicated, most Maxeler dataflow kernel designs are parametrizable at synthesis time with an unrolling factor f_u, which is often constrained by several rules. It must be a whole number divisor of burst sizes, the product of f_u and feedback latency l_f must not exceed burst or block sizes. The `Turn` kernel has a fixed size for the diagonal buffer addressing scheme. The practically possible unrolling factors are further constrained by resource utilization and our decision to aim for a clock frequency of at least 100 MHz for the datapath. We furthermore analyzed the bandwidth requirements and did not investigate unrolling factors which would exceed those by much. The first data column of Table II summarizes the final unrolling factors utilized for individual kernels. Out of eight individual kernels, six were able to reach or exceed the bandwidth limit. Details of this analysis can be found in [10].

Anticipating some performance results from Section VIII, in the aggregation phase, there is a high reconfiguration overhead for switching between kernels in the sequence they are needed. Thus, for the five aggregation kernels that are repeated in different cycles during the application, we created a common design implementing all of their

Table II
UNROLLING FACTORS f_u OF SYNTHESIZED KERNELS. *ASTERISKS MARK UNROLLING THAT SUFFICES TO REACH BANDWIDTH LIMITS. MAXKERN DENOTES INDIVIDUAL KERNELS, MAXFUSED DENOTES THE INTEGRATED AGGREGATION DESIGN, FOR THE SCANLINE PHASE, THE SAME INDIVIDUAL KERNELS ARE USED.

Design	MaxKern	MaxFused
HorSum_{24}	24*	12
VerSum_{24}	24*	12
Scale_{24}	24*	12
HorDiff_{16}	16	4
VerDiff_{24}	24*	12
Turn_{48}	48*	—
Scan_{12}	12	—
SAvg_{12}	12*	—

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

Design	Logic	LUTs	Primary 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%
MaxFused	63%	50%	57%	8%	**77%**
Turn_{48}	**32%**	23%	27%	3%	29%
Scan_{12}	**53%**	42%	46%	1%	31%
SAvg_{12}	**35%**	25%	30%	0%	23%

functionality in the same FPGA configuration and thus saving the reconfiguration overhead. We had some headroom for this integration, because not all of the individual kernels hit resource limitations, but still we had to decrease the unrolling factors. The final integrated aggregation design was chosen as the optimal trade-off between unrolling and achievable timing and runs at 130 MHz. The second data column of Table II summarizes the decreased unrolling factors. For the scanline phase, no integrated design was found that increased overall performance, not even for small images. In this phase, less reconfigurations are required, so the overhead that can be saved is much smaller. On the other hand, severe reductions of the unrolling factors were required to get routable designs.

In Table III, we finally summarize the resource usage of all used dataflow kernel designs. The table highlights that the individual kernels don't hit hard limits in resource consumption, however for *HorDiff* and *Scan*, no larger design with valid unrolling factor could successfully be synthesized. The critical resources of all kernels are either logic slices or BRAMs.

VII. EXPERIMENTAL SETUP

In this section, we first present the setup and notation for the evaluated systems and their configurations. We then discuss the generation and selection of our input data.

A. Evaluated Systems

After implementing and testing all described kernels on both accelerators, we integrated them into our stereo-matching application and tested it in a total of six different configurations.

1) CPU1: The entire execution is performed on the Intel Xeon X5650 CPU with Westmere microarchitecture, running at 2.66 GHz, as host CPU of the Maxeler platform [7].

2) CPU2: The entire execution is performed on the Intel Xeon 5138 CPU with the older Core microarchitecture, and running at only 2.13 GHz, as host CPU of the Convey platform [9].

3) MaxKern: The first accelerated configuration executes the individual, maximally unrolled dataflow kernels on the Maxeler accelerator card. This design point guarantees the highest raw kernel performance, but induces considerable configuration overheads, in particular during the aggregation phase. The parts of the application that are not accelerated are executed on CPU1 and the memory manager presented in Section V handles transfers between host and accelerator memory.

4) MaxFused: The second configuration using Maxeler accelerator card uses the integrated aggregation design, containing five kernels with reduced parallelism. The remainder of the execution is identical to *MaxKern*, including utilization of individual kernels for the scanline phase. This configuration saves a lot of reconfiguration overhead during the aggregation phase in exchange for reduced raw kernel performance.

5) CnyVecTouch: On the Convey platform, the host parts of the application are executed on the slower CPU2 and the accelerated kernels are executed on the vector processor overlay on the FPGA accelerator. Thus, no bitstream reconfigurations are required during application runtime, but only the much smaller kernel code executed on the coprocessor is changed in the different matching phases. As coprocessor memory interleaving mode, we use a 31-31 interleaving mode, which maps memory addresses to the individual memory banks in a way that allows near peak throughput for most possible stride patterns. For our tests, we setup the 24 GB of physical host memory and 16 GB of physical accelerator memory with a windowed memory mode with an 12 GB window of mapped coprocessor memory, 12 GB of pure host memory and 4 GB of pure coprocessor memory. As suggested in Section V, when no actual data has to be transferred, we use two different strategies to migrate allocated memory areas between the distinct physical memory locations. Here, with the first one, all involved pages are touched on the new location to guarantee best raw kernel performance.

6) CnyVecNt: With the second strategy, no-touch, the migration step is much faster and overall matching performance is a bit higher, at the cost of some increased kernel runtimes. All other settings are identical to *CnyVecTouch*.

B. Input Data

Conceptually, all accelerated configurations profit from larger image sizes and higher maximal disparity values, as parallelism and pipelining can be exploited better and overheads are amortized better by longer computation times, whereas smaller sizes may help the pure host execution by better caching opportunities. Beyond this general rule of thumb, there are different characteristics specific to either the Maxeler or the Convey accelerator platform. On Maxeler, all LMem accesses need to be 384 byte aligned, so in practice we pad all data structures and memory accesses to fit these requirements. This padding is an overhead that doesn't occur for multiple-of-384 dimensions. On the Convey platform, for best performance it is important to fill the 1024 vector elements. This is trivially the case for multiple-of-1024 dimensions, but in the aggregation phase can also be achieved by nicely fitting vector partitions, e.g. for horizontal sums with height 256 and multiple-of-4 disparities as in our earlier illustration in Figure 9. Additionally more subtle effects occur when the image sizes interfere with the memory interleaving mode which defines distribution of memory space to different memory bank. However, the mentioned 31-31 interleaving mode makes our experiments relatively robust in this regard.

To summarize, absolute and relative performance significantly depend on the input dimensions for our stereo-matching systems. We therefore decided to perform our measurements with a series of different input dimensions and to use standardized real-world image sizes or screen resolutions, regardless of their suitability for either architecture. In order to generate the input images, we scaled image pairs from the Middlebury benchmark set [13] to the desired resolution with cubic scaling in Gimp. The number of disparity steps to investigate is scaled according to the scaling factor of image width. This is important, because with a too low limit to the possible disparities, matching artifacts occur, which lead to dis-proportionally longer runtimes of the disparity refinement step.

We created two test series, one starting from the *Tsukuba* image pair, which has a low ratio of maximal disparity to image width and one starting from the *Cones* image pair, which has a high ratio of maximal disparity to image width. Tables IV and V show the two series of input dimensions we investigated. We scaled the two image pairs to different commonly used sizes with pixel ratios between 5:4 (SXGA) and 64:35 (EGA), most of them at 4:3 like the original *Tsukuba* pair. We selected the set of sizes in a way that the number of pixels between two consecutive sizes increases by factors between 1.08 (from UXGA to FullHD) and 1.46 (from SXGA to UXGA) and the number of elements in a cost volume increases by factors between 1.29 (from UXGA and FullHD) and 1.82 (from SXGA to UXGA). This series are currently limited by two aspects. Firstly, a maximal line width of 1920 is synthesized in one of our Maxeler kernels. Secondly, for larger input dimensions, total memory usage starts to become an issue. On the Maxeler platform, with our current implementation of the MemoryManager, the 24GB of accelerator memory put a hard limit to the maximal input dimensions. On the Convey platform, some tests with larger input dimensions executed functionally correct, but performance was impaired by the Linux kernel starting to swap data between main memory and hard disk.

Table IV
DIMENSIONS OF LOW-DISPARITY IMAGE SERIES.

Name	Width	Height	Disp.	Pixels [$\times 10^6$]	Vol. Elems [$\times 10^6$]
Tsukuba	384	288	16	0.11	1.77
HVGA	480	320	20	0.15	3.07
Macintosh LC	512	384	22	0.20	4.33
EGA	640	350	27	0.22	6.05
VGA	640	480	33	0.31	8.29
WVGA	768	480	32	0.37	11.80
SVGA	800	600	34	0.48	16.32
DVGA	960	640	40	0.61	24.58
XGA	1024	768	43	0.79	33.82
XGA+	1152	864	48	1.00	47.78
SXGA	1280	1024	54	1.31	70.78
UXGA	1600	1200	67	1.92	128.64
FullHD	1920	1080	80	2.07	165.89
TXGA	1920	1400	80	2.69	215.04

Table V
DIMENSIONS OF HIGH-DISPARITY IMAGE SERIES.

Name	Width	Height	Disp.	Pixels [$\times 10^6$]	Vol. Elems [$\times 10^6$]
Cones	450	375	60	0.17	10.13
Macintosh LC	512	384	68	0.20	13.37
EGA	640	350	85	0.22	19.04
VGA	640	480	85	0.31	26.11
WVGA	768	480	102	0.37	37.60
SVGA	800	600	107	0.48	51.36
DVGA	960	640	128	0.61	78.64
XGA	1024	768	137	0.79	107.74
XGA+	1152	864	154	1.00	153.28
SXGA	1280	1024	171	1.31	224.13

VIII. EVALUATION AND COMPARISONS

We first present overall system performance for both platforms. For the main comparison between the approaches of specialized kernels and the reusable vector processor overlay, we focus on the raw kernel performance with both methods and abstract the underlying hardware away as far as possible. Finally, we give some estimates of the design efforts for both approaches.

A. Stereo-Matching System Performance

Our first charts present speedups for the execution of the entire stereo-matching process for different image sizes compared to pure host execution. For the two respective image series, Figures 12 and 13 show the speedups of the four configurations with accelerators, *MaxKern*, *MaxFused*, *CnyVecTouch* and *CnyVecNt*, compared to the host execution

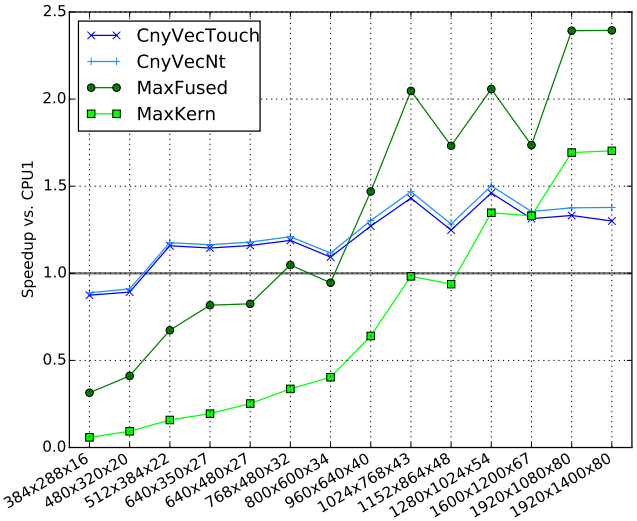

Figure 12. Low-disparity image series: Speedups of accelerated configurations compared to faster *CPU1*. *CnyVec* versions are at disadvantage because their host parts run on slower CPU2.

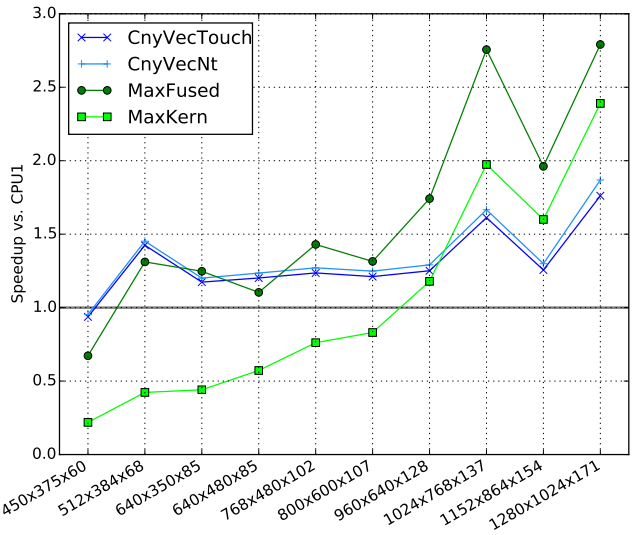

Figure 13. High-disparity image series: Speedups of accelerated configurations compared to faster *CPU1*. *CnyVec* versions are at disadvantage because their host parts run on slower CPU2.

on the faster *CPU1*. Since the host components of the two *CnyVec* versions are executed on *CPU2*, we also exemplarily include Figure 14, where *CPU2* is used as baseline for speedups of the low-disparity test series. Host CPU agnostic speedups of the *CnyVec* versions should be somewhere in between the values from Figures 12 and 14.

For both test series, we see that both *CnyVec* configurations on the Convey HC-1 [9] can achieve speedups, already at small image sizes which don't fully fill the

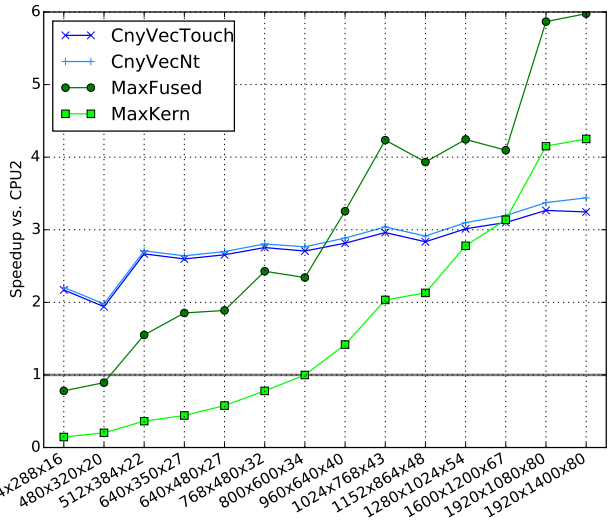

Figure 14. Low-disparity image series: Speedups of accelerated configurations compared to slower *CPU2*. *Max* versions have an additional advantage because their host parts run on the faster CPU1.

vector registers. 512x384 is the first image size, where *CnyVec* achieves little speedups over *CPU1*. The speedups increase slightly with increasing image sizes, but show some variations for specific sizes fitting vector register sizes or memory interface a bit better or worse. At 1280x1024x171 *CnyVecNt* reaches a peak speedup of 1.9x over *CPU1*. With the slower *CPU2* as baseline, the speedups are around 3x.

On the Maxeler platform [7], the *MaxFused* configuration with a common design for all aggregation kernels persistently outperforms the *MaxKern* configuration, with its individual, maximally parallel aggregation kernels. However, for small image sizes, *MaxFused* is still slower than *CPU1* and both *CnyVec* configurations. In the low-disparity test series, it takes the lead over all other designs for the first time at 960x640x40. In this test series, it's speedup peaks at 1920x1400x80 with 2.4x compared to *CPU1*. *MaxFused* profits from the higher disparities of the second test series, achieving a first speedup over *CPU1* already at 512x384x68 and a peak speedup of 2.8x at 1280x1024x171.

Figure 15 displays some additional details of the underlying data for *CPU1* and the respective faster versions for both accelerators, now showing absolute execution times and subdividing them into aggregation phase, scanline phase and all remaining parts of the application. We see that in the aggregation phase, only *MaxFused* outperforms *CPU1* by a small margin. However, execution of this phase on the accelerator has the additional benefit that afterwards, intermediate results are already in accelerator memory for the following scanline phase. During this scanline phase, now both accelerators achieve significant speedups compared to *CPU1*. During the execution phases remaining on

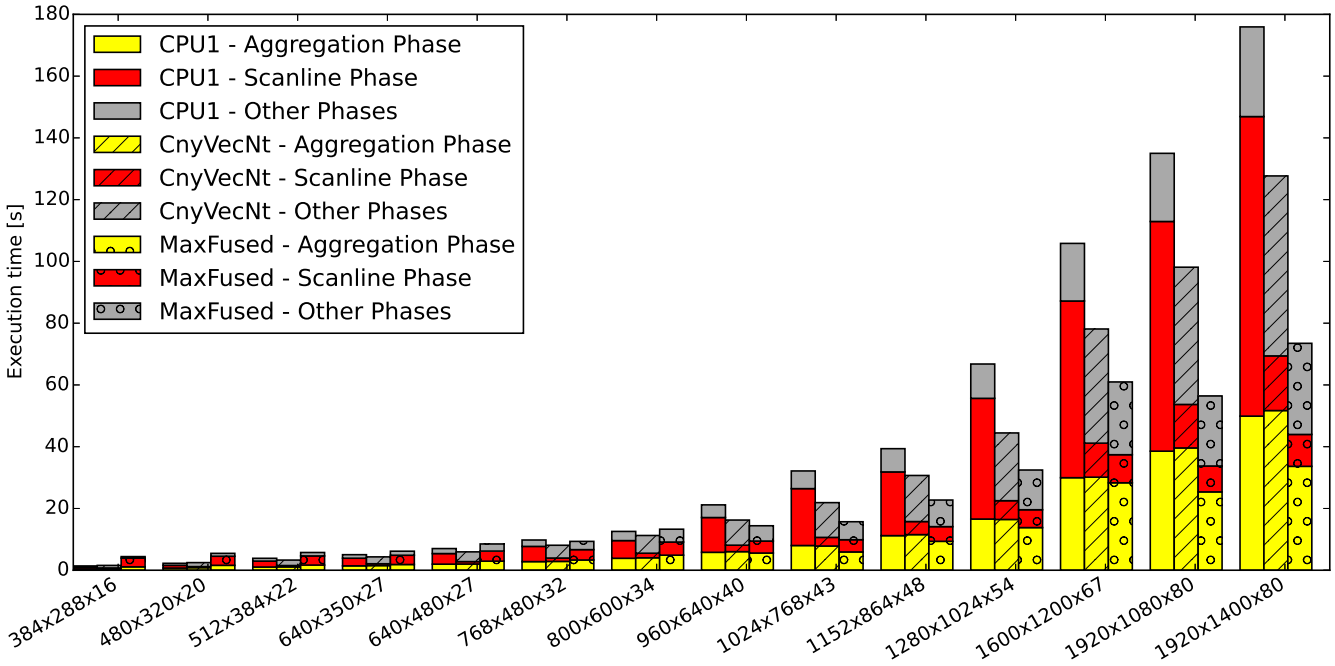

Figure 15. Low-disparity image series: Execution times with different configurations, stacked bars divided into aggregation phase (yellow), scanline phase (red) and all remaining parts on host (grey).

the respective host, *CnyVecNt* notably looses some of its earlier speedups compared to *CPU1*, because its host code is executed on the slower *CPU2*.

B. Platform Overheads

All further comparisons are only performed with regard to the faster *CPU1*. We proceed with the analysis of the two accelerated phases, in this subsection on the basis of results from 1920x1400x80, and compare *CPU1* to all accelerated designs. Figure 16 breaks down the total execution time of the aggregation phase, splitting the entire yellow blocks from Figure 15 into individual components. The first component is the raw execution time of the five described aggregation kernels, still summed up together. We see that this raw kernel execution time is significantly reduced on all accelerator platforms and configurations compared to *CPU1*, down from 47s to between 15s and 24s on the accelerators, with the design with highest parallelism, *MaxKern* executing fastest.

The next component is the total time of all data transfers between host and accelerator memory, that are initiated through our memory manager. For pure CPU execution, naturally no such transfers are needed. Here we see that part of the lower execution times observed on the Maxeler platform in comparison to the Convey platform, are caused by lower transfer times, either because of faster physical interconnect, or because of the overhead incurred for the realization of the shared memory space on Convey. Masking

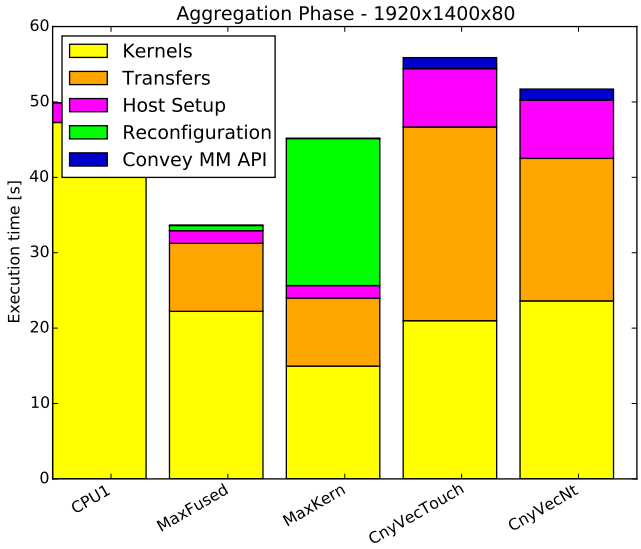

Figure 16. Breakdown of individual runtime components inside aggregation phase

some of this overhead in the *CnyVecNt* configuration overcompensates for the increased raw kernel runtimes compared to *CnyVecTouch*. As third component, we summarized the time spent outside of the five kernels selected for accelera-

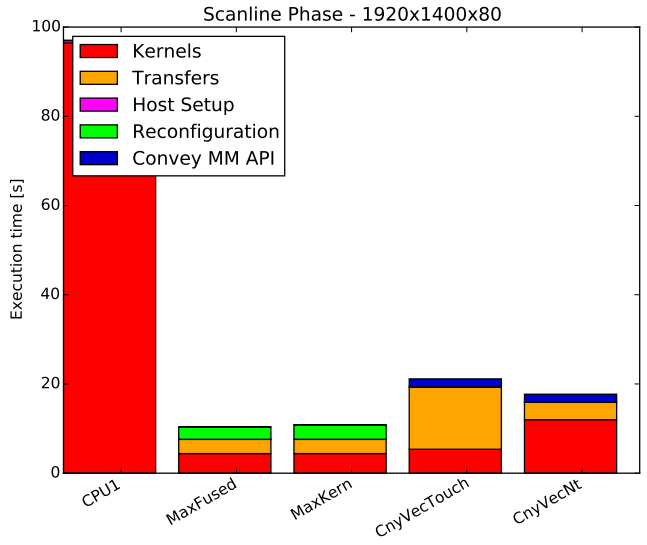

Figure 17. Breakdown of individual runtime components inside scanline phase

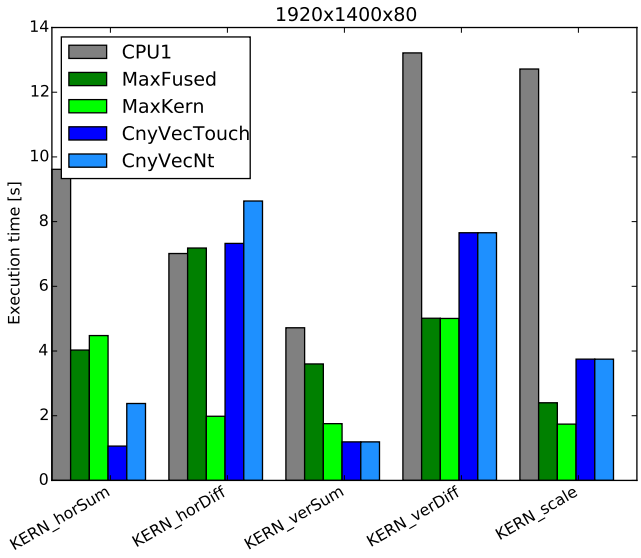

Figure 18. Execution times of individual aggregation kernels.

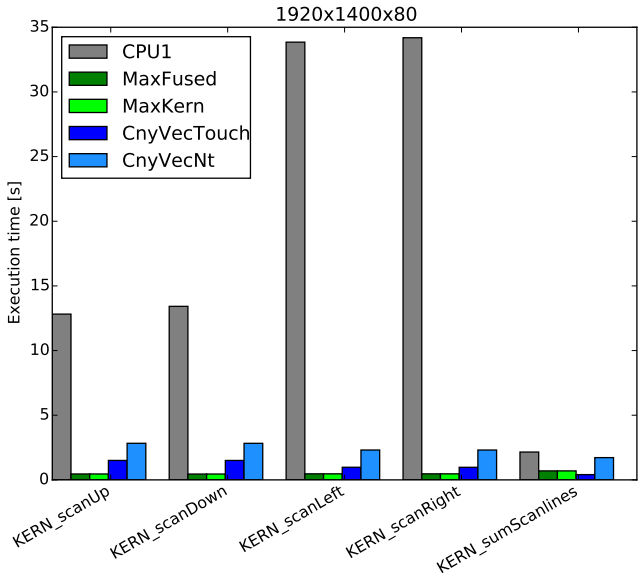

Figure 19. Execution times of individual scanline kernels.

tion. In the aggregation phase, this *Host Setup* time includes for example the time to initialize the aggregation regions needed for scaling. This phase is notably slower on the Convey platform again because of the slower host *CPU2*.

The fourth component, reconfiguration times, only occur on the Maxeler platform. We see that for the *MaxFused* design, this overhead is negligible as only one reconfiguration is performed, whereas for the individual aggregation kernels in *MaxKern*, it more than eats up the additional speedups achieved in raw kernel execution times. As final component, we measured the time spent in platform specific allocation and free API calls on Convey, which turns out to be relatively minor in the two configurations observed.

Figure 17 displays the same components for the scanline phase. In this phase, both Maxeler configurations execute the identical designs and thus perform identically. Due to higher computational intensity and higher data reuse, all accelerator platforms in all configurations reduce the raw kernel execution times much more than in the aggregation phase. Compared to its kernel execution times, *CnyVecTouch* incurs a huge overhead for data transfers, which *CnyVecNt* can again partially mask during kernel execution. Compared to the CPU execution times, these overheads are smaller than during the aggregation phase, thus allowing considerable overall speedups.

C. Kernel Performance

In order to compare the kernel specific dataflow design approach with the vector processor overlay in regard to their suitability for kernel-centric acceleration, we now look at individual kernel execution times and disregard the platform overheads discussed in the previous Subsection. Figures 18 and 19 show the raw kernel execution times of the aggregation and scanline phases, again for the largest image pair, now separated into individual kernels, but summing up the execution times of all invocations of the same kernel to be comparable to the previous plots.

We again compare the faster *CPU1* with all accelerated versions for completeness, however we don't consider the data of *CnyVecNt* very relevant when it comes to assessing

the potential of the vector overlay approach, since here the raw kernel execution times are just increased due to the partial masking of transfer overhead, which is to be excluded in this comparison anyway. Therefore we focus on *CnyVecTouch* for the evaluation of the overlay architecture. For the specialized dataflow designs on the other hand, we consider both design points, since both the existence and the absence of design trade-offs due to reconfiguration overheads can represent relevant real-world scenarios.

For the aggregation kernels in Figure 18, we see quite diverse results, with either *MaxKern* or *CnyVecTouch* achieving the best kernel runtimes. Furthermore, we see an unexpected artifact for `HorSum`, where *MaxFused* in spite of less compute parallelism is faster than *MaxKern*. Presumably both kernels are limited by effective memory bandwidth, with *MaxFused* generating a slightly more favorable memory access pattern. The `HorDiff` kernel, in turn, was already projected to be compute bound for the *MaxKern*. The *MaxFused* design with four times less compute parallelism is around 4x slower, supporting this assumption. The `verSum` and `scale` kernels seem to have become compute bound for the *MaxFused* design, showing smaller slowdowns compared to *MaxKern*.

A detailed discussion on the underlying effects for the comparison of dataflow and vectorized kernels needs to take into account the achieved compute parallelism, memory reuse properties as outlined in Section VI, bandwidth requirements and the impact of memory access patterns on the achieved bandwidths. Also, during our optimizations of the vector overlay kernels, we saw that performance can not be easily modeled as a function of compute throughput or of effective memory bandwidth, but also depends on latencies and sequence of dependent instructions. Thus, a detailed attribution of certain results to possibly dominating performance factors would be mostly speculation without additional measurements. However, the effects of sensitivity to latencies and those of different arithmetic intensities caused by data reuse and design of operations are attributable to the kernel design paradigm and thus form the actual subject of our comparison. On the other hand, effects of different amounts of available compute resources and memory bandwidths distort this comparison. Thus, in the following subsection, we try to extract the former design effects by compensating for the latter hardware effects. However, first we proceed with the comparison of kernel performance.

Comparing the runtimes of scanline kernels in Figure 19, we see a more homogeneous result pattern, with the most notable observations the difference in CPU performance in horizontal and vertical directions and that the specialized kernels dominate for the actual scanline computation whereas the vector overlay takes the lead for the subsequent summation step.

In our concrete stereo-matching application, the various kernels do have their individual, well defined contributions to the overall runtime. However, when comparing the two design methods in regard to their general suitability for kernel-centric acceleration, we want to abstract from these weights and just profit from the variety of compute and data-usage patterns represented by different kernels. Thus we consolidate these results into a single metric, the geometric mean of individual kernel speedup factors that each approach achieves over the reference CPU execution. We denote this metric as *Kernel-Ratio*, analogical to the similarly computed SPECRatio. This metric has the nice property that the choice of reference platform doesn't impact relative ratio between two other platforms.

Table VI
KERNEL-RATIOS (GEOMETRIC MEAN OF ALL KERNEL SPEEDUPS) RELATIVE TO CPU1 FOR THE SXGA IMAGE WITH HIGH DISPARITY AND FOR THE GEOMETRIC MEAN OVER ALL SIZES.

Architecture	All Sizes	SXGA-high
MaxKern	6.59	9.86
MaxFused	5.20	7.76
CnyVecTouch	5.80	8.37

Table VII
NORMALIZED KERNEL-RATIOS (GEOMETRIC MEAN OF ALL KERNEL SPEEDUPS, NORMALIZED WITH REGARD TO COMPUTE RESOURCES AND BANDWIDTHS) RELATIVE TO CNYVECTOUCH.

Architecture	All Sizes	SXGA-high
MaxKern	3.04	3.16
MaxFused	2.41	2.48

Table VI summarizes those Kernel-Ratios for three accelerated designs with reference to *CPU1* for the geometric mean of all input sizes and individually for largest problem size tested, SXGA with high disparity. The reference invariance of the Kernel-Ratios metric allows to directly derive additional ratios between the platforms in the list. So, considering the comparison of the two kernel design approaches, for all image sizes the Kernel-Ratios of *MaxKern* with reference to *CnyVecTouch* is computed as $\frac{MaxKern}{CnyVecTouch} = \frac{6.59}{5.80} = 1.14$. Similarly, for SXGA high-disparity test, it is computed as $\frac{MaxKern}{CnyVecTouch} = \frac{9.53}{8.37} = 1.18$.

The results, when comparing the Kernel-Ratios of the two specialized dataflow kernel approaches on Maxeler with the vector overlay on Convey, are surprising. In the geometric mean, the specialized kernels just marginally faster than the vector overlay. When trading off parallelism for the integration of several specialized kernels in *MaxFused*, the specialized kernels are even slightly slower than the overlay, for all image dimensions with $\frac{MaxKern}{CnyVecTouch} = \frac{5.20}{5.80} = 0.90$ and for high-disparity SXGA with $\frac{MaxKern}{CnyVecTouch} = \frac{7.76}{8.37} = 0.93$. However, as indicated above, these numbers do abstract away the data transfer and reconfiguration overheads, but still contain the mismatch in available compute resources and memory bandwidths.

D. Hardware-Normalized Kernel-Ratio

We try to extract the effects of different kernel design approaches on the two platforms by compensating for the effects of underlying hardware. For this, we need metrics to compare the hardware platforms and approach this by looking at compute resources and memory bandwidth. When we compare basic compute resources in terms of 6-input LUTs, which are common to Virtex-5 and Virtex-6 FPGAs, we can observe a ratio of Maxeler to Convey hardware of $\frac{297600}{829440} = 0.359$. Similarly, the ratio of theoretical peak memory bandwidth can be computed as $\frac{28.8\ GB/s}{74.4\ GB/s} = 0.387$. Now for a somewhat sophisticated compensation of hardware configurations, we would like to offset each observed kernel speedup with one of those factors, depending on whether the kernel is compute or bandwidth bound. However, since the factors are roughly similar, we just average those two ratios to 0.373. We multiply Maxeler to Convey Kernel-Ratios by the inverse of the combined hardware ratio in order to normalize performance to comparable hardware platform characteristics. This leads to a metric we denote as Normalized Kernel-ratio and present in Table VII. We can summarize these results as central contribution of this work as follows:

> In a diverse set of compute kernels with data parallelism, specialized dataflow kernel implementations on FPGAs are around 3x more efficient in terms of performance than a reusable vector processor overlay implemented on comparable hardware. In a concrete scenario, due to trade-offs between reconfiguration overheads and exposed parallelism, this advantage shrinks to around 2.5x.

After this bold statement, we need to discuss the circumstances and limitations that govern the general applicability of these results. First of all, the utilized method of normalizing for different hardware platforms by a single compensation factor depends on the similar ratios of compute resources and bandwidths. Once those differ considerably, such scaling needs to be done on a per-kernel basis after an analysis whether compute or bandwidth would be the limiting factor. For the dataflow kernels, the foundations for such work are present in [10], but for the vector overlay, the performance bounds hard to quantify since all our kernels are actually constrained by a combination of computation, latencies and bandwidth. Also, after migration from one hardware platform to the other, the performance bounds can be different, requiring a more elaborate compensation step.

Secondly, we need to discuss aspects of memory bandwidth. The peak bandwidth data we utilized for our normalization already incorporates two aspects from our practical results. On the Maxeler platform, the memory controller is part of the synthesized FPGA design. The theoretical bandwidth maximum can be achieved with the memory controller clocked at 400 MHz. Due to difficulties to meet the timing of this controller after synthesis, we targeted 300 MHz in our experiments and the peak bandwidth value used in our calculation reflects this. On the Convey platform, the memory controllers are implemented in separate FPGAs and their design is fixed, running at 300 MHz. As reported, we utilize a 31-31 interleaving scheme, which maximizes actual performance in our measurements, but technically reduces peak bandwidth to $\frac{31}{32}$ of the physical interface capabilities, which we also included in our numbers.

The practically realizable bandwidths of both memory interfaces depend, beyond those peak numbers, on additional influence factors, like burst sizes, strides and granularity, which are hard to quantify without extensive tests on both platforms. However, we can qualitatively state, that the efficient support for element-wise vector memory operations, in particular indexed ones, of the vector overlay depend on the capability to access individual 8-byte blocks enabled by the scatter-gather RAM modules of the Convey platform we use. So we need to constrain our Normalized Kernel-Ratio results for this design approach with:

> The vector processor overlay requires a memory interface with sufficiently fine access granularity in order to achieve the indicated performance efficiency.

Thirdly, we want to discuss the compute resources. Our scaling method depends on the implicit assumption, that performance scales linearly with available hardware. When it comes to parallel execution units that operate on unrolled data and are implemented primarily with LUTs, this assumption makes sense. However, other aspects of resource usage often doesn't scale linearly with compute throughput. On the one hand, some parts of the designs remain constant, e.g. in our experiments the control logic of the dataflow kernels and the scalar processing units of the vector overlay. On the other hand, some components grow more than linearly, e.g. some data reordering buffers or input selection multiplexers.

Also, the FPGAs of the two utilized platforms have different ratios of additional resources as BRAMs and DSP-blocks to LUTs, which the scaling in terms of raw logic resources neglects. In particular, as seen in Table III, the current designs of several of our dataflow kernels rely on the high ratio of BRAMs to LUTs on the Maxeler platform's Virtex-6 SX475T FPGA, which is $\frac{\#36Kb-blocks}{\#LUTs} = \frac{1064}{297600} \approx \frac{1}{280}$. On the Virtex-5 LX330 FPGAs of the Convey platform, this ratio is lower: $\frac{\#36Kb-blocks}{\#LUTs} = \frac{288}{207360} = \frac{1}{720}$. However, again as a qualitative statement form our design experience of the dataflow kernels, many of the BRAM resources are directly dedicated to buffering or reordering kernel inputs, outputs and intermediate results in order to properly utilize the burst-oriented memory interface of the Maxeler platform. So, the second addendum to our Normalized Kernel-Ratio result now states more precisely for the other design ap-

proach:

> Dataflow kernels can achieve the indicated performance efficiency even with a burst-oriented memory interface, but require FPGAs with a sufficiently high ratio of BRAMs to LUTs for this.

Finally, we need to discuss clock frequencies and low-level optimization. Our data-flow kernels are generated with the Maxeler design flow, which enhances design productivity by transparently applying a number of best-practice decisions, e.g. to pipelining or organization of buffers. Many of these can be modified manually, but in our designs such optimizations were mostly performed demand driven, in response to specific timing or resource problems. In order to relax the need for deep pipelining and along with it the need to very carefully optimize the balancing of pipeline stages and their physical layout, most compute paths of our dataflow kernels run at modest 100-130 MHz. For the much more widely distributed and reused vector overlay on the Convey platform on the other hand, common sense and anecdotal evidence suggest, that a huge amount of effort and expertise was invested into low-level optimizations. Consequently, this design runs at 300 MHz, which has a large impact on the performance we measured and compared in this work. We do consider this difference as characteristic for the relationship between reusable and problem-specific designs and as such not as a weakness of the comparison, but nevertheless want to state this in a third addendum to our overall findings:

> Our comparison premises that much more manual low-level optimization effort is put into a reusable overlay design than into problem-specific dataflow kernels.

E. Estimates on Design Efforts

As final step of our comparison, we want to present some empirical data about our experienced productivity when performing kernel-centric acceleration with two different design philosophies and targets. As we did not systematically track the design process and many factors which are hard to quantify impact the perceived productivity, these results need to be contemplated with at least a gain of salt. The design and implementation results presented in this work were done in several disjunct phases and with different levels of experience gained from other projects.

Overall we would describe the dataflow kernel design process as two phases, the first starting with some limited amount of experience in dataflow kernel design with the Maxeler toolflow, spanning the equivalent of 8-10 full-time developer weeks for conceptualization of kernels and their unrolling patterns, implementation and many stand-alone tests in simulation, along with early synthesis results to get a feeling the resource usage characteristics. The second phase, conduced with much additional background of the Maxeler platform, took another 6-8 weeks with focus on integration, synthesis and optimization.

This phase was in practice prolonged by the process of waiting for synthesis results, which we tried to exclude from the above reported time span, because it to some degree depends on the amount of parallel synthesis resources and to some degree can be covered organizationally, e.g. by running synthesis over night. As an illustrative number: the total tool runtime for the final design of *MaxFused* was reported as 22 hours, 5 mins, 11 secs. Within this time, for the place and route step, a total of 11 different cost tables were explored, with four parallel instances running concurrently. Another special challenge was posed by one kernel instance, where the Maxeler simulation tool was not able to reproduce a memory interface related error actually encountered in hardware.

The design of the vector coprocessor kernels was also performed in two major phases. An equivalent of 4-6 full-time developer weeks was spent for first concepts and prototypical implementations with no preliminary knowledge of the concrete vector ISA, but with some general background in assembly programming. With a lot more experience with the architecture, another 6-8 weeks was spent for the final kernel designs and optimizations, including a considerable fraction of the time that was spent in exploring performance impacts of memory settings and data transfer patterns triggered through our memory manager. On this platform, assembly of a kernel design and integration into an executable was completed within seconds, allowing for much faster optimization iterations. A special challenge was posed by repeated crashes of the accelerator hardware that occurred when using the debugger for the vector coprocessor.

We summarize our subjective characterization of the design process as follows:

> Designing specialized dataflow kernels with Maxeler's spatial programming language and design flow requires some more time and some more expertise than developing assembly code for a vector processor, but not a whole lot. However, the time-consuming synthesis can add some tedious waiting to the process.

IX. Related Work

We discuss related work in three different fields. First we give an overview of other approaches for stereo-matching on FPGAs, then we discuss the field of kernel-centric acceleration and finally present other approaches to design and evaluate overlay architectures.

Apart from our own previous work in [10] and [11], stereo-matching on FPGAs has been tackled with co-design of algorithm and hardware, typically implementing the entire processing pipeline without off-chip memory accesses. Different algorithmic approaches have been explored with different design goals in mind. For example Tippetts et al. [14] present a complete stereo-matching system with less than 10,000 LUTs and 30 BRAMs, at much lower result quality, but robust in respect to uncalibrated and unrectified images. Apart from simple pre- and post-processing steps, their approach employs an intensity profile shape matching algorithm, that directly works on row-local intensity data.

The FPGA implementations with highest matching accuracy reflect more of the matching patterns utilized in this work. Shan et al. [21] 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. [22] try to follow the algorithm of Mei et al. [16] in their FPGA implementation more closely. In addition to the aggregation technique of Shan et al. [21], 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 spatial 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.

Jin et al. [27], [28] use a similar single-pass aggregation phase and winner-takes-all disparity selection and combine it with a voting scheme, denoted fast locally consistent (FLC) [29], which is more sophisticated than the one utilized in the post-processing step we employ. Between these two phases, intermediate disparity results are actually buffered off-chip, but requiring much less bandwidth, since no volume data is stored.

These implementations come quite close to our results in terms of matching quality, with Wang et al. [22] reaching an average of 6.17% bad pixels and Jin et al. [28] of only 5.86% bad pixels. They are somewhat more limited in problem dimensions than our approach of working on blocks of memory, with Jin et al. [27], [28] projecting a design that supports our largest test inputs to exceed the LUT and BRAM resources of their and our current hardware platform, but suitable for large Virtex-7 FPGAs. In terms of performance, these co-designed implementations are orders of magnitude faster than our implementation, by executing less computation steps on volume data and by integrating the compute pipelines more tightly. Therefore, these approaches are superior when algorithmic trade-offs can be made, whereas our approach is justified, when exact reproduction of results or a simpler, structured design process are required.

Such a kernel-centric design approach is also coming along with the OpenCL-to-FPGA design flows, that are gaining traction, from academic initiatives [30] to FPGA vendor toolchains [31], [32]. Even though so far the actual synthesis of FPGA designs from OpenCL kernels was the focus of this research, a defining feature of the OpenCL approach is the distinction between data-parallel kernels that are to be executed on parallel resources and a host part of the application. This host code may run on a server CPU like in our work or in [31] or on a CPU inside a SoC implemented on FPGA as in [30]. Our memory management interface could be seen as easier, more abstract alternative to a subset of the OpenCL runtime feature set. But just like we used our interface to abstract away the underlying data transfer APIs implemented on the Maxeler and Convey platforms [8], [9], we could also add support for a target platform that internally uses the OpenCL API functionality through our interface.

FPGA overlays or architecture templates for such overlays have been researched as means to enable faster or easier manual design, faster synthesis or compilation toolflows and faster reconfiguration on top of a reusable overlay. Coole and Stitt propose intermediate fabrics (IFs) [33], an overlay architecture of coarse grained compute resources and a configurable interconnect implemented on top of an FPGA. Different IFs are designed first manually [33], [34], later also automatically based on OpenCL [35], to each support a group of kernels with similar compute demands. Their coarse grained abstraction allows orders of magnitude faster synthesis and reconfiguration than for the underlying FPGA fabric. Depending on the set of investigated kernels, the degree of specialization and the reconfiguration properties, the authors report area overheads from 1.4x [33] over 1.8x [35] to 4.4x [34] for the overlay and assume identical clock-speeds for overlay and specialized designs.

In the area of instruction programmable FPGA overlays, active academic research on vector processors [36], [37] is going on in the area of embedded computing devices as throughput-optimized alternatives to scalar soft processors. Ovtcharov et al. [38] add the concept of GPU-like multi-threading to hide latencies of functional units and memory accesses by pipelining the execution of different threads. As proposed by Kingyens and Steffan [39] and brought forward by Convey with CHOMP [40] as successor to the vector processor utilized in this work, such a GPU-like architecture may be a promising architecture template for acceleration of server- and datacenter-scale computing tasks. From the programmers perspective, it offers more transparent ways to exploit parallelism in multiple dimensions, than with the vector partitioning approach we had to specify explicitly. From an architectural point of view, this may come at higher resource consumption, e.g. for address calculations. On the other hand, the improved latency hiding promises higher performance. To the best of our knowledge, there are no published performance results for these architectures, in

particular not in comparison to custom datapaths.

X. Conclusion

In this work we have compared two design approaches for kernel-centric acceleration, specialized dataflow kernels versus an instruction programmed vector processor on FPGA with the example of a Stereo-Matching application. We have shown that given comparable FPGA and memory resources, the specialized dataflow kernels promise around 3x more performance than kernels executing on a fixed vector overlay, and we have analyzed three important preconditions for this result: 1) the vector processor needs a sufficiently fine grained memory interface, 2) the dataflow kernels need FPGA architectures with sufficient BRAMs for local buffers, and 3) a reusable overlay typically receives more low-level optimization than specialized kernels with a much more narrow usage scope. We have also elaborated, that in our concrete scenario, due to trade-offs between reconfiguration overheads and exposed parallelism, the advantage of specialized dataflow kernels is reduced to around 2.5x.

Looking forward, it will be interesting to extend such an analysis to other pre-synthesized or customizable overlay architectures, following GPU-like SIMT execution patterns or implementing CGRA structures on FPGA. Also, a careful analysis whether and were the spatial programming language paradigm or the utilized toolflow might add inefficiencies could help to contextualize our results.

We have motivated the kernel-centric acceleration approach used in this work with productivity considerations and by the desire to precisely retain the desired application behavior while using FPGA resources for acceleration. Our overview of the design process hints, that both the dataflow and the vector ISA abstraction may help for this process, but synthesis times are still an issue for specialized dataflow kernels. The similarly kernel-centric OpenCL design flows, that currently gain traction in the FPGA community promise even more abstraction, possibly along with reduced control over the designs, and speed-up the synthesis process by reusing the memory and PCIe interface as fully mapped and routed components on the FPGA. Thus they will represent another interesting design approach to compare to.

The analysis of data-transfer and platform overheads when looking at the entire application underlines, that the current trend of tighter integration of FPGAs, as well as other accelerators like GPUs, into the same SoC with CPUs and a shared memory subsystem may turn out very valuable for kernel-centric acceleration approaches.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgment

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.C. Herbordt, T. Van Court, Yongfeng Gu, B. Sukhwani, A Conti, J. Model, and D. DiSabello. Achieving high performance with FPGA-based computing. *IEEE Computer*, (3):50–57, March 2007.

[2] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Jim Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proc. Int. Symp. on Computer Architecture (ISCA)*, 2014.

[3] Rajesh K. Gupta and Giovanni de Micheli. A co-synthesis approach to embedded system design automation. *Design Automation for Embedded Systems*, (1-2):69–120, 1996.

[4] W. Wolf. A decade of hardware/software codesign. *IEEE Computer*, (4):38–43, April 2003.

[5] Daniel Lustig and Margaret Martonosi. Reducing GPU offload latency via fine-grained CPU-GPU synchronization. In *Proc. Int. Symp. on High-Performance Computer Architecture (HPCA)*, pages 354–365. IEEE Computer Society, 2013.

[6] Guohui Wang, Yingen Xiong, J. Yun, and J.R. Cavallaro. Accelerating computer vision algorithms using OpenCL framework on the mobile GPU - a case study. In *Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 2629–2633. IEEE, May 2013.

[7] Maxeler Technologies. MPC-C series. https://www.maxeler.com/products/mpc-cseries/.

[8] Maxeler Technologies. Programming MPC systems. Whitepaper, https://www.maxeler.com/media/documents/MaxelerWhitePaperProgramming.pdf, June 2013.

[9] T.M. Brewer. Instruction set innovations for the Convey HC-1 computer. *IEEE Micro*, (2):70 –79, March-April 2010.

[10] Tobias Kenter, Henning Schmitz, and Christian Plessl. Kernel-centric acceleration of high accuracy stereo-matching. In *Proc. Int. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8. IEEE Computer Society, December 2014.

[11] Tobias Kenter, Henning Schmitz, and Christian Plessl. Pragma based parallelization – trading hardware efficiency for ease of use? In *Proc. Int. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6. IEEE Computer Society, December 2012.

[12] Olga Veksler. Fast variable window for stereo correspondence using integral images. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 556–561. IEEE Computer Society, 2003.

[13] Daniel Scharstein and Richard Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *Int. Journal on Computer Vision*, (1-3):7–42, April 2002.

[14] Beau Tippetts, Dah Jye Lee, Kirt Lillywhite, and James K. Archibald. Hardware-efficient design of real-time profile shape matching stereo vision algorithm on FPGA. *Int. Journal of Reconfigurable Computing (IJRC)*, 2014.

[15] K. Wegner and O. Stankiewicz. Similarity measures for depth estimation. In *Proc. 3DTV Conference: The True Vision - Capture, Transmission and Display of 3D Video (3DTV-Con)*, pages 1–4. IEEE, May 2009.

[16] 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)*. IEEE, 2011.

[17] H. Hirschmuller and D. Scharstein. Evaluation of cost functions for stereo matching. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 1–8. IEEE Computer Society, June 2007.

[18] Andrea Fusiello, Vito Roberto, and Emanuele Trucco. Efficient stereo with multiple windowing. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 858–863. IEEE Computer Society, 1997.

[19] Heiko Hirschmüller. Stereo processing by semiglobal matching and mutual information. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, (2):328–341, February 2008.

[20] Ke Zhang, Jiangbo Lu, and Gauthier Lafruit. Cross-based local stereo matching using orthogonal integral images. *IEEE Transactions on Circuits and Systems for Video Technology (TCSVT)*, (7):1073–1079, July 2009.

[21] Yi Shan, Yuchen Hao, Wenqiang Wang, Yu Wang, Xu Chen, Huazhong Yang, and Wayne Luk. Hardware acceleration for an accurate stereo vision system using mini-census adaptive support region. *ACM Transactions on Embedded Computing Systems (TECS)*, (4s):132:1–132:24, April 2014.

[22] Wenqiang Wang, Jing Yan, Ningyi Xu, Yu Wang, and Feng-Hsiung Hsu. Real-time high-quality stereo vision system in FPGA. In *Proc. Int. Conf. on Field Programmable Technology (ICFPT)*, pages 358–361. IEEE Computer Society, December 2013.

[23] Heiko Hirschmüller and Daniel Scharstein. Evaluation of stereo matching costs on images with radiometric differences. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, (9):1582–1599, September 2009.

[24] Xilinx. Virtex-6 family overview, DS150 (v2.4). http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf, January 2012.

[25] Xilinx. Virtex-5 family overview, DS100 (v5.0). http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf, February 2009.

[26] Tobias Kenter, Gavin Vaz, and Christian Plessl. Partitioning and vectorizing binary applications for a reconfigurable vector computer. In *Proc. Int. Symp. on Reconfigurable Computing: Architectures, Tools, and Applications (ARC)*. Springer, April 2014.

[27] Minxi Jin and T. Maruyama. A fast and high quality stereo matching algorithm on FPGA. In *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 507–510. IEEE Computer Society, August 2012.

[28] Minxi Jin and Tsutomu Maruyama. Fast and accurate stereo vision system on FPGA. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, (1):3:1–3:24, February 2014.

[29] S. Mattoccia. Fast locally consistent dense stereo on multicore. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 69–76. IEEE Computer Society, June 2010.

[30] M. Owaida, N. Bellas, K. Daloukas, and C.D. Antonopoulos. Synthesis of platform architectures from OpenCL programs. In *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 186–193. IEEE Computer Society, May 2011.

[31] T.S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D.P. Singh. From OpenCL to high-performance hardware on FPGAs. In *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 531–534. IEEE Computer Society, August 2012.

[32] Mohamed S. Abdelfattah, Andrei Hagiescu, and Deshanand Singh. Gzip on a chip: High performance lossless data compression on FPGAs using OpenCL. In *Proc. Int. Workshop on OpenCL (IWOCL)*, pages 4:1–4:9. ACM, 2014.

[33] James Coole and Greg Stitt. Intermediate fabrics: virtual architectures for circuit portability and fast placement and routing. In *Proc. Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES/ISSS)*, pages 13–22. ACM, 2010.

[34] Greg Stitt and James Coole. Intermediate fabrics: Virtual architectures for near-instant FPGA compilation. *IEEE Embedded Systems Letters (ESL)*, (3):81–84, September 2011.

[35] J. Coole and G. Stitt. Fast, flexible high-level synthesis from OpenCL using reconfiguration contexts. *IEEE Micro*, (1):42–53, January 2014.

[36] Aaron Severance and Guy Lemieux. VENICE: a compact vector processor for FPGA applications. In *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, page 245. IEEE Computer Society, 2012.

[37] Christopher H. Chou, Aaron Severance, Alex D. Brant, Zhiduo Liu, Saurabh Sant, and Guy G.F. Lemieux. VEGAS: soft vector processor with scratchpad memory. In *Proc. Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 15–24. ACM, 2011.

[38] K. Ovtcharov, I. Tili, and J.G. Steffan. TILT: a multithreaded VLIW soft processor family. In *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE Computer Society, September 2013.

[39] Jeffrey Kingyens and J. Gregory Steffan. The potential for a GPU-like overlay architecture for FPGAs. *Int. Journal of Reconfigurable Computing (IJRC)*, 2011.

[40] John D. Leidel, Kevin Wadleigh, Joe Bolding, Tony Brewer, and Dean Walker. CHOMP: a framework and instruction set for latency tolerant, massively multithreaded processors. In *Proc. SC Companion: High Performance Computing, Networking Storage and Analysis (SCC)*, pages 232–239. IEEE Computer Society, 2012.