A Scalable Latency-Insensitive Architecture for FPGA-Accelerated Semi-Global Matching in Stereo Vision Applications

Jaco Hofmann, Jens Korinth, and Andreas Koch

Embedded Systems and Applications Group (ESA)
Technische Universität Darmstadt, Germany
{jah,jk,ahk}@esa.tu-darmstadt.de

***Abstract*—Semi-Global Matching (SGM) is a high-performance method for computing high-quality disparity maps from stereo camera images in machine vision applications. It is also suitable for direct hardware execution, e.g., in ASICs or reconfigurable logic devices. We present a highly parametrized FPGA implementation, scalable from simple low-resolution low-power use-cases, up to complex real-time full-HD multi-camera scenarios. By using a latency-insensitive design style, high-level synthesis from the Bluespec SystemVerilog next-generation hardware description language, and an automated design-space exploration flow, many implementation alternatives could be examined with high productivity. The use of the ThreadpoolComposer system-on-chip assembly tool allows the portable mapping of the SGM accelerator to different hardware platforms. The accelerator performance exceeds that of prior fixed-architecture approaches.**

I. Introduction

Using two cameras to derive depth information is an important step in computer vision. In the simplest case, it consists of performing computations on image streams from two cameras located at the same height, but separated horizontally. The captured images from the two cameras are ideally displaced from each other only in the horizontal direction, by a pixel offset (called a *disparity*) limited by the physical distance of the cameras. Using triangulation, the depth of each pixel (distance from the camera plane) can then be computed based on determining the per-pixel disparity values (pixels closer to the camera plane will have larger disparities). Note that a real stereovision pipeline would perform additional steps to compensate for lens distortion and inaccurate positioning. However, as this work focuses on the disparity computation (Fig. 1), pre-filtering techniques (as well as the actual camera interfaces) will not be discussed here.

Our hardware architecture implements a Semi-Global (Block) Matching (SGM / SGBM) method. SGM/SGBM algorithms have the advantages of both being fast, but also scoring much better on accuracy (e.g., on benchmarks such as KITTI [1] and Middlebury [2]) than the simpler Block Matching approaches.

The key contributions over prior implementations presented in this paper are 1) improvements in the low-level input scheduling scheme, which allowed performance gains by up to

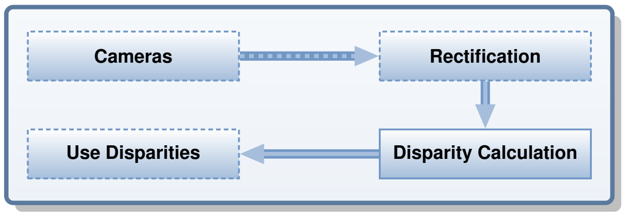


Fig. 1. Typical stereo vision system. This paper focuses on the disparity calculation step.

66%, and 2) the use of a multi-target design-space exploration flow using the ThreadPoolComposer (TPC) [3] system. We will also introduce the concept of design *variants*, an extended capability that allows TPC an even finer-grained exploration of the implementation space.

II. Semi-Global Matching Algorithm

SGM [4] offers not just a good speed vs. accuracy trade-off, it is also robust with regard to choices for configuration parameters [5]. This section will give a short overview over the underlying computations, as these have driven the design of the hardware architecture.

The function $C(\mathbf{p}, d) \in \mathbb{N}_0$ is used to denote the cost of matching a pixel $\mathbf{p} = (x, y)$ at coordinates (x, y) in the base image at an assumed disparity (offset) of d at coordinates $(x - d, y)$ in the matching image. We use a C based on a parametric rank transform [6], specifically the differences in the counts of pixels darker than the center pixel (see Eq. (3)). To determine the disparity resulting in the best match between the two images, these costs are calculated for *all* potential disparities $d < D_{\text{max}}$, where D_{max} is the upper limit of the potential disparity (due to the physical mounting distance of the two cameras). Initially, it is assumed that the match with the lowest cost indicates the true disparity $\arg\min_{d<D_{\text{max}}} C(\mathbf{p}, d)$ between base and match images for an individual pixel $\mathbf{p}$. However, this will be refined later by imposing additional constraints.

A key feature of (semi) global approaches is that the costs of potential matches are computed not just between individual (or neighborhoods) of pixels, but along multi-pixel *paths* stretching

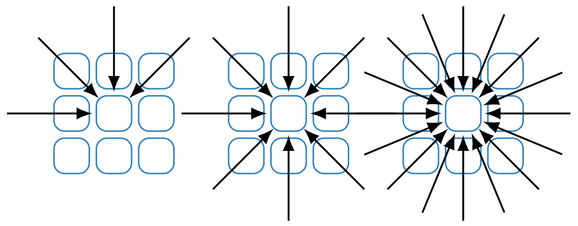


Fig. 2. (a) Four, (b) Eight, and (c) 16 directions used in Semi-Global Block Matching.

across the *entire* image. Paths are described by the relative offset vector $\mathbf{r} = (\Delta x, \Delta y)$ between successive path elements. Given an assumed (potential) disparity d, we denote the cost of matching along an entire path as $L_\mathbf{r}(\mathbf{p}, d)$. For a global view of the matches, these paths are distributed evenly over the image (e.g., as in see Fig. 2). At least eight evenly distributed paths are typically used (Fig. 2.b), with 16 being suggested for optimal coverage. Note that the number and arrangement of paths has a direct impact not only on the matching accuracy, but also on the computational effort and, in our case, on the actual architecture of the SGM hardware accelerator.

As a compromise between speed and accuracy, we will employ just the four paths 0° ($\mathbf{r} = (1, 0)$), 45° ($\mathbf{r} = (1, 1)$), 90° ($\mathbf{r} = (0, 1)$) and 135° ($\mathbf{r} = (-1, 1)$) (Fig. 2.a), which results in an accuracy loss (an increase in the count of mislabeled disparities) of just 1.7% in the Middlebury benchmark [7]. However, picking these specific four paths permits an extremely efficient hardware implementation capable of computing the $L_\mathbf{r}$ for *all* of these paths in *parallel*. This is enabled by the lack of forward data dependencies, as paths mostly include pixels that have already been read.

We first examine computing the cost $L'_\mathbf{r}(\mathbf{p}, d)$ for matching the pixels $\mathbf{p}$ along a path $\mathbf{r}$ for an assumed disparity of d. We call this the *raw* cost, which will be refined later to allow a more efficient hardware implementation.

$$L'_\mathbf{r}(\mathbf{p}, d) = C(\mathbf{p}, d) + \min \begin{cases} L'_\mathbf{r}(\mathbf{p} - \mathbf{r}, d) & \mathbf{1.a} \\ L'_\mathbf{r}(\mathbf{p} - \mathbf{r}, d-1) + P_1 & \mathbf{1.b} \\ L'_\mathbf{r}(\mathbf{p} - \mathbf{r}, d+1) + P_1 & \mathbf{1.c} \\ \min_i L'_\mathbf{r}(\mathbf{p} - \mathbf{r}, i) + P_2 & \mathbf{1.d} \end{cases} \tag{1}$$

Raw path costs are not just calculated for all of the selected paths, but also for all potential disparity values d up to the limit D_{max}. The calculation includes both the local cost C, as well as the addition of a semi-global component. The latter is the minimum of four alternatives: The first, expressed in (Eq. 1.**a**), is the cost of the *prior* pixel along the path, the second and third components penalize small disparity changes of $|\Delta d| = 1$ by P_1 (Eq. 1.**b** and **.c**), while the final term (Eq. 1.**d**) penalizes larger disparity changes (so-called *discontinuities*) by P_2. P_1 is usually determined off-line experimentally by analyzing representative input images typical for the actual stereo vision use-case. On the other hand, P_2 is adjusted dynamically at run-time: As disparity discontinuities are often also represented as pixel *intensity* changes, computing $P_2 = \frac{P'_2}{|I_\mathbf{p} - I_{\mathbf{p}-\mathbf{r}}|}$ compensates for different pixel intensities $I_\mathbf{p}$ and $I_{\mathbf{p}-\mathbf{r}}$ along the path $\mathbf{r}$. Similar to P_1, P'_2 is a constant determined experimentally based on representative sample images off-line. More details on the reasoning behind this approach is given in [4]. For determining the (semi) global matching cost, an initial approach would just accumulate the path costs across all paths. However, a more efficient hardware implementation can be achieved by proceeding in a different manner.

Accumulating the elements of (potentially long) paths running across the entire image can result in large values, which require correspondingly wide words for computation and storage. We can counter this by subtracting from the *raw* path costs for a pixel $L'_\mathbf{r}(\mathbf{p}, d)$ the *minimum* of the path costs for all assumed disparities d for the *prior* pixel $\mathbf{p} - \mathbf{r}$ along the path $\mathbf{r}$. This will result in just the *differences* between the prior and current pixels being encoded. These will (due to their smaller magnitude) require correspondingly narrower data words for storage and computation. The final path cost computation thus becomes

$$L_\mathbf{r}(\mathbf{p}, d) = C(\mathbf{p}, d) + \min \begin{cases} L_\mathbf{r}(\mathbf{p} - \mathbf{r}, d) \\ L_\mathbf{r}(\mathbf{p} - \mathbf{r}, d-1) + P_1 \\ L_\mathbf{r}(\mathbf{p} - \mathbf{r}, d+1) + P_1 \\ \min_i L_\mathbf{r}(\mathbf{p} - \mathbf{r}, i) + P_2 \end{cases} - \min_j L_\mathbf{r}(\mathbf{p} - \mathbf{r}, j). \tag{2}$$

Accordingly, The upper limit of the word width is given as $\lceil \log_2 R(C_{max} + P_2) \rceil$ [4] with R being the number of paths used.

We sum up the paths costs $L_\mathbf{r}$ along all paths $\mathbf{r}$ as $S(\mathbf{p}, d) = \sum_\mathbf{r} L_\mathbf{r}(\mathbf{p}, d)$. From these, the most likely (winning) disparity d for the pixel $\mathbf{p}$ is computed based on the minimal matching cost as $\arg\min_d S(\mathbf{p}, d)$. The winning disparity for each pixel in the input images form the output stream of the accelerator. The can be used later to derive the actual depth (Z-axis position, not discussed here).

In practice additional constraints need to be imposed to clean-up the output data. They remove outlier values and discover pixels for which the disparity could not be reliably determined. For the latter, the result of $\arg\min_d S(\mathbf{p}, d)$ might be a *multi-element* set, meaning that the minimal matching cost for a pixel $\mathbf{p}$ occurs for multiple *different* potential disparities d. Thus, the algorithm cannot decide on a single winning disparity for the pixel, and instead registers the disparity for this pixel as "invalid".

A second check compares the results of the algorithm when running it with swapped roles of base and match images. This so-called *left/right check* can also be implemented efficiently (avoiding recalculating all disparities for the former match image now used as base). This is achieved by re-using the previously computed $S(\mathbf{p}, d)$ along an epipolar line as $\arg\min_d S((x(\mathbf{p}) + d, y(\mathbf{p})), d)$ to select the winning disparity d for the second image. The left/right check sets the disparity to "invalid" if the corresponding disparities of the original and role-swapped passes differ by more than one. This check eliminates phantom disparities, resulting from occluded surfaces that are visible in one image, but hidden in the other.

In a final step, a basic 3×3 median filter is applied to the disparity map to suppress outliers.

III. Related Work on High-Performance SGM Implementations

With its attractive tradeoffs, SGM has been used in a number of implementations. However, due to its semi-global nature, it still remains a compute-intensive algorithm that makes reaching performance and energy efficiency goals challenging for implementers.

A recent software implementation [8] achieved just 16 frames-per-second (FPS) for VGA-resolution images and $D_{\max} = 128$. And this implementation already employed a fast Intel i7-4960HQ processor (rated to draw 47 W under load), including the use of the AVX2 vector extensions (SIMD).

The SGM algorithm is not well suited for GPU execution. It achieved just 11.7 FPS for VGA images, even when restricting the disparity search space to $D_{\max} = 64$ [9]. The NVidia GTX480 GPU used for this implementation is rated to draw between $250\,\mathrm{W}$ and $300\,\mathrm{W}$ of power when loaded.

For more power-constrained environments, an implementation [10] for the P4080 embedded eight-core processor manages to stay under 30 W, even when clocked at $1.2\,\mathrm{GHz}$. However, it achieves just 0.5 FPS for VGA images, also at $D_{\max} = 64$.

Heterogeneous systems employing FPGAs as compute elements do significantly better. Combining a mobile Core2Duo-based system with an embedded OMAP3530 ARM processor and a Xilinx Spartan 6 FPGA yielded 14.6 FPS [11], but the inter-processor communication imposed a latency of 250 ms for processing a single image of 1024×508 pixels and $D_{\max} = 128$. This latency will be too high for many real-time use-cases.

More focused implementations with fewer different processors do better: A solution using an Altera EP4SGX230 device achieved 31.79 FPS for 1024×768 resolutions at $D_{\max} = 96$ [12]. C-to-hardware compilation (high-level synthesis) to the Xilinx Zynq Z7020 SoC generated an accelerator reaching 30 FPS at VGA resolution, but only with a very narrow $D_{\max} = 16$ [13]. A combination of FPGA and CPU processing is used in [14] to achieve 60 FPS for images of 752×480 pixels on a Xilinx Artix 7 FPGA. While exact power numbers are not given for these cases, generally FPGAs draw far less than 10 W for computing purposes (when the power-hungry high-speed serial transceivers are not used).

The approach we employ here is most similar to [7], which achieved a maximum of 167 FPS at VGA on a Virtex 5 LX220T FPGA with $D_{\max} = 64$, and [15], which yielded 300 FPS at VGA on a Virtex 7 X690T device at the same $D_{\max}$. Compared to [7], the designs presented here add an extra level of fine-grained parallelization to improve performance. With regard to [15], we use an improved data buffering scheme and a refined design-space exploration flow to increase peak performance by 25% to more than 400 FPS on the same platform. Our approach also reduces the LUT counts by 50% compared to designs of similar FPS in [7].

IV. Architecture

Our architecture has not only been enhanced by introducing an extra level of fine-grained parallelism (e.g., parallel disparity computation and sorting) relative to [7], it has also been implemented using a state-of-the-art latency insensitive design style in Bluespec, a next-generation hardware description language. As a result, it is significantly more scalable, easier to extend, yet also faster than the original work (even when compensating for the differences in FPGA target technologies, see Section V-F). The algorithm described in Section II is mapped to the structure shown in Fig. 3.

The SGM accelerator can operate in two modes: It can be interfaced directly with external cameras to directly stream image data into the accelerator without host intervention, or it can employ DMA engines to transfer image data from host memory. In both cases, DMA is used to transfer the disparity data back to the host.

As a first step, the *rank* (see Section IV-A) of pixels from the input images is computed in parallel for the left and right images. Also in parallel, P_2 is computed for the pixels in the base image. These operations, which all realize functions of a subset of neighboring pixels (sometimes also called a kernel or stencil), exploit the capabilities of Bluespec SystemVerilog to express higher-order functions: A single generic module description realizes all of the low-level operations (e.g., pixel I/O, handshaking), and accepts the actual filter *function* as a *parameter*. This allows for very concise architecture descriptions, that can still be synthesized to high-performance hardware. Improvements to the generic module (e.g., the improvement of queuing schedules performed in this work) are immediately reflected in all of the operations using it.

The filtered input images are forwarded to the units computing the disparity values, called Row Processors, which are explained in detail later in this section. Using the same generic filter module as before, a 3×3 median filter is realized between the Row Processors and the output FIFO to suppress outliers.

As shown in Fig. 4, the Row Processors implement the actual SGM computation. Coarse grained parallelism is exploited by employing multiple Row Processors to process subsequent rows of the input images in parallel. An arbiter distributes the input lines to the Row Processors in a round-robin scheme. Each Row Processor is able to store an entire row of the input image in its input FIFO. The Row Processors are connected by FIFOs to realize a systolic array, with a wraparound connection to allow data to flow across the partition boundaries (horizontal slices of the images processed in parallel).

Inside the Row Processor the costs for all disparities up to $D_{\max}$ are calculated and buffered for usage in the disparity calculation stage. The disparity calculation stage is responsible for collecting all necessary data to compute Eq. (2). With the Row Processors forming a systolic array, this task is as simple as reading from a FIFO. Based on this semi-global data, and the costs calculated earlier, the actual disparity is calculated. In contrast to [7], a second, finer-grained level of parallelism is introduced by allowing multiple disparities to be processed in parallel inside a Row Processor (discussed in greater detail in [15]). In a final step, the winning disparities are checked for validity. The disparity stream, which was split through the arbiter, is then merged and filtered before being stored to the

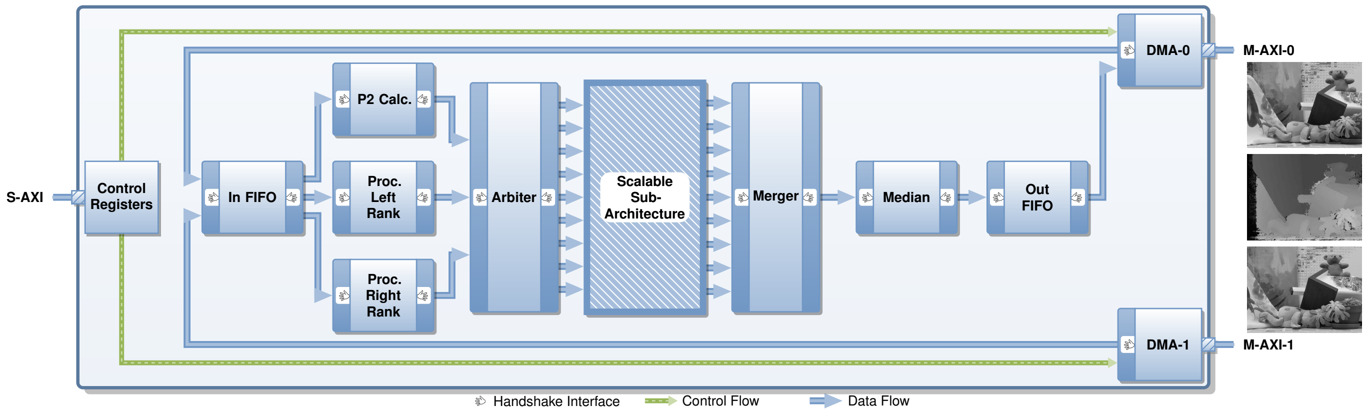


Fig. 3. Operation of the proposed architecture. The images are read from host memory by two DMA engines. A control interface is used to alter the behavior of the algorithm. The base images are processed by rank transform and P_2 calculation cores. The results are then fed into a parallel sub-architecture which performs the actual SGM calculations. The resulting disparities are filtered to suppress outliers, and forwarded to one of the DMA engines for transfer back to the host.

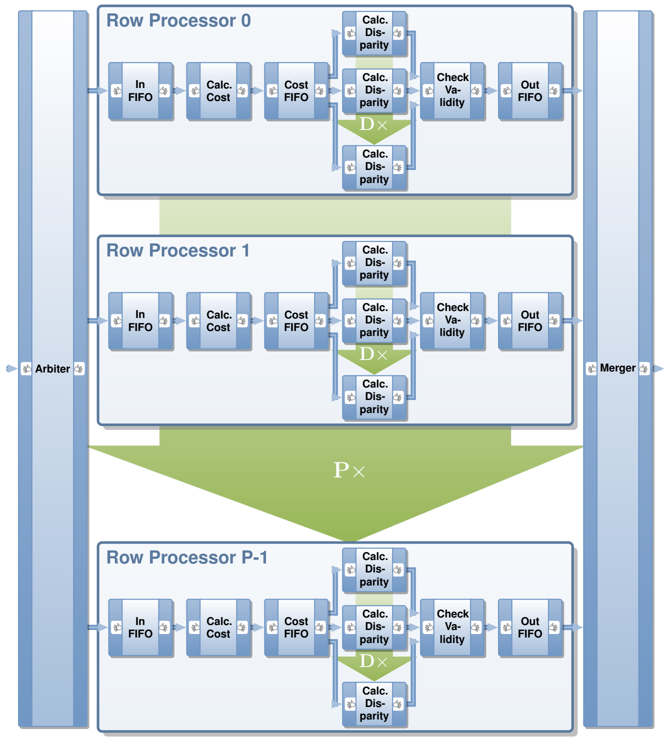


Fig. 4. Detailed view of the SGM calculation sub-architecture. Each row processor receives a line of the rank-transformed input image in sequential order. The row processor calculates all disparities for its specific row. The semi-global data from other input lines, as shown in Eq. (2), is distributed in a systolic array-pattern. Two methods of parallelization are employed: 1) Using multiple row processors, sequential lines of the input image can be processed in parallel. 2) Finer grained parallelism is employed inside of each row processor, where multiple disparities are calculated in parallel.

host memory.

A. Implementation Details

For the rank transform discussed in Section II, we use a non-parametric variant called Census[6]. It consists of counting the *number* of pixels in a neighborhood that have a *lower intensity* than the center pixel:

$$R(\mathbf{p}) = \|\{\mathbf{p}' \in N_s(\mathbf{p}) | I(\mathbf{p}') < I(\mathbf{p})\}\| \,, \tag{3}$$

Here, $R(\mathbf{p})$ is the rank transform of pixel $\mathbf{p}$, $N_s(\mathbf{p})$ the neighborhood around pixel $\mathbf{p}$ encompassing all pixels with a distance less or equal to $(s-1)/2$ from the center, and $I(\mathbf{p})$ the intensity of pixel $\mathbf{p}$. The neighborhood, also called a kernel, is square with a total size of s^2 pixels. Each rank-transformed value is encoded in $\lceil \log_2 (s^2) \rceil$ bit.

$R(\mathbf{p})$ is computed in parallel for both input images, yielding streams of $R_b(\mathbf{p})$ and $R_m(\mathbf{p})$ for the base and matching images, both of which are fed to the cost calculation stage together with a stream of P_2 penalty values. If multiple Row Processors are available, the inputs will be distributed by an arbiter in a round-robin fashion at the granularity of single rows. This leads to the processing proceeding as a downward moving stripe of rows. To this end, all Row Processors require input FIFOs which can store a full row to enable operation as a wavefront array.

Based on the rank transform R, the actual per-pixel matching cost is computed as

$$C(\mathbf{p}, d) = \|R(\mathbf{p}) - R((x(\mathbf{p}) - d, y(\mathbf{p})))\| \,. \tag{4}$$

Note that the wrap-around buffer from the *last* to the *first* Row Processor is larger than the normal inter-Row Processor buffers, as it has to hold all $D_{\max}$ intermediate results for *every pixel* in the last row.

The maximum value of any $L_{\mathbf{r}}$ (Eq. 2) has been shown to be always less than $C_{\max} + P_2$ [4]. This used to narrow the word widths for data storage and arithmetic operators in our hardware implementations.

A major bottleneck of the implementation presented in [15] is the data forwarding between the arbiter stage and the Row Processors. Larger input FIFOs and improvements in the arbitration strategy have achieved significantly better Row Processor in the designs presented here. These advances have the greatest impact on small (low-power) configurations with only few Row Processors, as these suffered the most from the idle periods introduced by non-optimal arbitration.

V. Evaluation

We evaluate our approach at three levels: First, we consider the accuracy, than the simulated target-independent performance of the hardware architecture in terms of clock cycles, and finally the wall-clock performance on three actual FPGA platforms, encompassing embedded system and data center use.

A. Accuracy

Since we use the same core algorithm as [7], we also achieve the same disparity computation accuracy. On the Middlebury[2], [16] benchmark, an average of only $8.4\,\%$ disparities has an error larger than one pixel.

B. Platform-independent performance

Cycle-accurate simulation of the architecture (described in highly parametrized Bluespec SystemVerilog, BSV) was used to determine the run-time characteristics of sample implementations. Comparison to the actual hardware implementations in Section V-E shows that these simulations are actually representative of final performance.

The core is evaluated at three image resolutions: 640×480 pixels (VGA), 1280×720 pixels (720p) and 1920×1080 pixels (1080p). VGA resolution images are evaluated at $D_{\text{max}} = 64$, the higher resolutions have $D_{max} = 128$. For all resolutions, the number of clock cycles needed to complete a single frame are determined through simulation. Note that we examine a *single* accelerator core here. Section V-E will extend this evaluation to *multi-core* architectures.

We have examined different compositions of coarse- and fine-grained parallelism. An implementation is described by the pair (**#p**, **#d**), which indicates the use of **#p** Row Processors, each computing **#d** assumed disparities in parallel. For each of the resolutions, we have used automatic design space exploration to generate 250 implementation alternatives, shown on the X-axis in order of increasing area or performance. Due to space constraints, only a subset of the alternatives could be labeled here with (#p,#d).

As shown in Fig. 5 for VGA images, the architecture scales well with increasing the number of Row Processors, down to a lower bound of 628871 cycles, at which point a single pixel is calculated in 2.04 cycles and a single disparity requires 0.032 cycles. This design is limited by the speed the input buffers can be filled, which could be increased even more by also applying fine-grained parallelization techniques to the Stage 1 computations (see Section VI).

At an assumed clock rate of $200\,\text{MHz}$ (which is realistic for a single core) the architecture would reach up to 306 fps as shown in Fig. 6. Such large and fast systems could be used to calculate the disparities over multiple cameras to produce a surround-view of a scene.

More interesting for low-power applications is the *minimal* frequency necessary to achieve 30 frames per second (a typical requirement for real-time processing). As shown in Fig. 7, the architecture is able to fulfill this requirement at a frequency as low as $18\,\text{MHz}$ for VGA images. Note that this is an improvement of 66% over [15], which required $30\,\text{MHz}$ for the same performance.

C. Building on-chip architectures with ThreadPoolComposer

The true performance of our approach can be measured only when actually mapping an implementation of the architecture

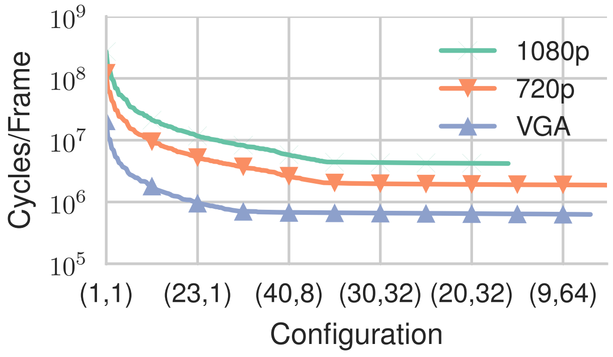


Fig. 5. Cycles needed to process a single disparity map for varying degrees of parallelism. The Configurations are ordered by performance.

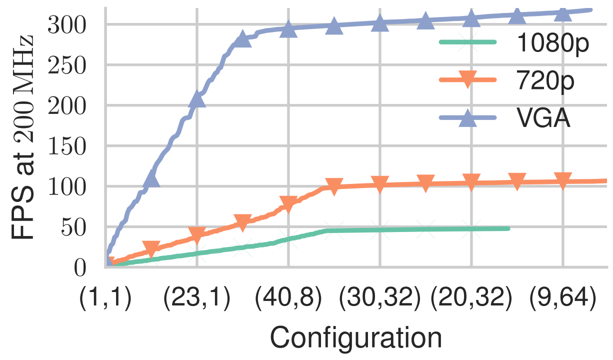


Fig. 6. Frames per second achieved by a single accelerator with the proposed architecture at a clock frequency of $200\,\text{MHz}$. The Configurations are ordered by performance.

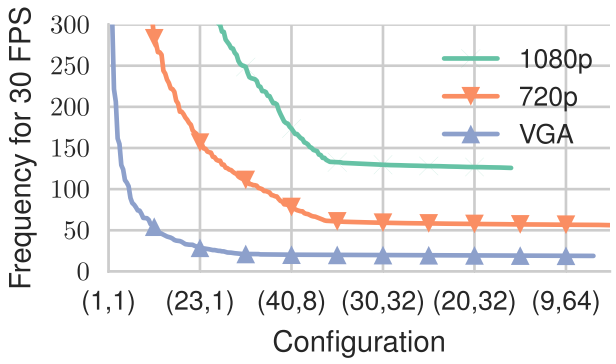


Fig. 7. Frequency needed to achieve 30 frames per second on the proposed architecture for varying degrees of parallelism. The Configurations are ordered by performance.

to a real hardware platform, of which we consider three cases: ZedBoard (XC7Z020), Xilinx ZC706 (XC7Z045) and VC709 (XC7VX690T) development boards. While the two Zynq-based platforms can execute stand-alone, the VC709 is installed in a host computer having an eight core Intel Xeon E5-1620v2 CPU running at 3.7 GHz. We used version 2016.04 of the ThreadPoolComposer toolflow to automatically generate designs for all three platforms, and the Xilinx Vivado Design Suite 2015.2 to perform hardware synthesis, place and route.

ThreadPoolComposer [3] is an automatic toolflow based on Scala/SBT and the Xilinx Vivado IP Integrator which yields a complete bitstream design as well as hardware/software interfaces to allow access to the accelerators using TPC API, the thin unified C/C++ programming interface provided by ThreadPoolComposer.

The key computing abstraction used by ThreadPoolComposer are *Thread Pools*. These realize one or more different computations (called *functions*) in hardware, with each function being implemented by one or more underlying Processing Elements (PEs). A Composition describes how many PEs are present for each function, and thus defines the possibly heterogeneous pool of concurrent accelerator units that may be used in multi-threaded execution. For the stereo vision application, we will build a homogeneous pool using SGM accelerator instances as PEs. The ThreadPoolComposer toolflow creates not only the pool itself, but also the generates the platform-specific system-on-chip (including host and memory interfaces) around it.

This approach gives control over the coarse-grained parallelism of the architecture, as well as the dynamic scaling of the design to different target platforms. TPC API hides the pool-internal parallelism from the user by providing an efficient interface to schedule jobs on the pools, regardless of the pool size or implementation platform.

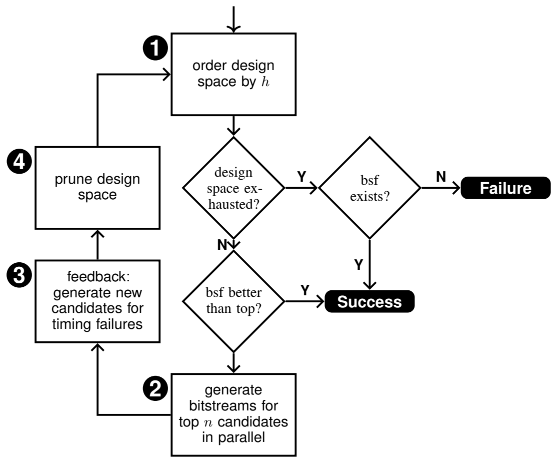
D. Design Space Exploration with ThreadPoolComposer

Highly parameterized hardware descriptions, which are the hallmark of hardware description languages such as Bluespec or Chisel, allow the generation of a wide range of hardware implementations from a single description. However, this design space grows exponentially with the number of parameters, and it quickly becomes unwieldy to explore manually (e.g., due to the many non-obvious effects of interacting parameters).

Version 2016.04 of ThreadPoolComposer provides a design space exploration mechanism, which automates most steps between the definition of the pool Composition and a fully routed bitstream. Note that an actual pool implementation is characterized not only by the Composition, but also by the clock frequency it can operate at.

The design space S of these implementations is ordered by a heuristic h, shown in Eq. (5), that estimates the job throughput for the given Configuration and clock frequency.

$$h(c, f) = \sum_{(k,n) \in c} \frac{n}{r_k \cdot f^{-1} + t_{data} + t_{irq}} \qquad (5)$$



❶
order design space by h
❹
prune design space
design space exhausted?
Y
bsf exists?
N
Failure
N
Y
❸
feedback: generate new candidates for timing failures
bsf better than top?
Y
Success
❷
generate bitstreams for top n candidates in parallel

Fig. 8. Novel DSE Algorithm in ThreadPoolComposer: bsf (best-so-far) denotes the solution with the highest heuristic value for which a bitstream has been built that achieved timing closure; top denotes the candidate with the highest heuristic value in the unexplored design space.

In Eq. (5), C is the set of all feasible compositions, a Composition $c \in C$ is a set of tuples (k, n) where k identifies a PE and n is the number of desired instances, f is the target design frequency, r_k is the average execution time of accelerator k in clock cycles, t_{data} is the time required for the data transfers and t_{irq} is the average interrupt latency. Feasible compositions are all compositions that would theoretically fit into the target device; ThreadPoolComposer uses area estimates for each core obtained by out-of-context synthesis to get a lower bound on the area utilization. Feasible target frequencies are all f where $50\,\text{MHz} \leq f \leq 450\,\text{Mhz}$, with the upper bound usually being tightened (lowered) during exploration based on timing estimates obtained by out-of-context synthesis. The values t_{data} and t_{irq} are characteristics of the target platform and can be obtained using an automatic benchmarking tool provided by ThreadPoolComposer. Finally, the average runtime of the core in clock cycles must be obtained by the user during simulation. The basic algorithm of ThreadPoolComposer is depicted in Fig. 8:

In the following, *candidate* will denote an element of the design space S, i.e., a pair (c, f) of a Composition $c \in C$ and a target design frequency f. A *solution* is a bitstream for a candidate which could be successfully placed, routed and achieved timing closure. A *failure* is a bitstream for a candidate that either could not be placed, did not achieve timing closure, or failed due to some other error. The *best-so-far solution*, i.e., the bitstream with the highest heuristic value that achieved timing closure, is denoted by bsf and the best candidate, i.e., the candidate with the highest heuristic value in the remaining design space, is denoted by top. Initially, S is enumerated by generating all feasible candidates and ordered according to h. If S is empty in ❶ and bsf does not exist, the process failed, otherwise it succeeded with bsf as its result. If S is not empty, top is compared to bsf, if it exists; if bsf is better than top, the process also succeeded. Otherwise the bitstream generation of a batch of n candidates is started in parallel in ❷. In ❸ the batch results are evaluated: bsf is updated (if any succeeded) and all timing failures *produce a feedback candidate*. ThreadPoolComposer analyzes the timing report of the design and extracts the *worst negative slack (WNS)*, from which a new minimal clock period for this specific Composition is computed and added as a new candidate. To improve the convergence rate of the DSE, aggressive *pruning of the design space* is applied in ❹: All candidates with the same Composition as a failure which had a placer error (usually: being too large for the target device) are removed entirely, since the area constraints cannot be satisfied for the Composition regardless of the frequency. If the failure was due to timing constraints being violated (usually: failing to meet the target clock frequency), all candidates with the same Composition and frequency exceeding that of the new upper frequency bound (lowered by the WNS of the failed attempt) are also removed in ❹, since they are unlikely to succeed. Finally, if bsf exists, all lower-quality candidates (having a smaller heuristic value than bsf) are removed, shrinking S.

Prior versions of ThreadPoolComposer explored the design space by altering pool compositions and frequency targets. Version 2016.04, used for this work, adds a more powerful capability: For a finer-grained approach, the tool can now also influence the characteristics of *each kind of PE* in the pool by altering PE-internal architecture parameters. Each choice of parameter values (for SGM: the tuple of D and P values) made by ThreadPoolComposer is called a *core variant*.

For example, a user could specify $\{(\text{SGM}(P=1, D=1), 1)\}$ as the starting Composition. With the core variants capability enabled, ThreadPoolComposer will consider, e.g., $\{(\text{SGM}(P=2, D=1), 1)\}$ and $\{(\text{SGM}(P=4, D=2), 1)\}$ as alternative Compositions and also include them in the design space S, ordered by their heuristic value. This is orthogonal to the exploration of pool compositions by ThreadPoolComposer. E.g., the tool can also examine a candidate with $\{(\text{SGM}(P=4, D=2), 4)\}$ (having four PEs instead of one), if it is promising according to its h-value. In this manner, excellent solutions that might be overlooked even by an experienced designer can be achieved (see Section V-E).

E. Results

Using the design space exploration described in the previous section, we explored the best SGM core for the three image resolutions (VGA, 720p and 1080p) for each of three target devices. Table I shows the results for each device and resolution, i.e., the designs with the highest heuristic score that achieved timing closure. The column **N** denotes the number of parallel instances of the core in the design, the column **F** the achieved design frequency in MHz. Using these designs, we then evaluated the performance on real hardware: A C++ program using TPC API, which can be compiled for all three platforms without changes to the source code, uses the accelerator pool to compute disparity maps for random images. The actual throughput achieved here (in frames per second) is shown in the last column **FPS** and corresponds closely to the throughput predicted as **h**-value by the heuristic. While

TABLE I
DSE RESULTS FOR SGM

Platform	Resolution	P	D	N	F	h	FPS
zedboard	640×480	5	1	1	110	26.6	26.6
zc706	640×480	21	1	1	205	198.2	197.0
vc709	640×480	12	2	3	131	426.9	410.0
zedboard	1280×720						
zc706	1280×720	27	2	1	145	59.4	59.3
vc709	1280×720	20	2	2	121	75.8	74.9
zedboard	1920×1080						
zc706	1920×1080	17	4	1	140	23.3	23.3
vc709	1920×1080	13	8	1	122	28.6	28.4

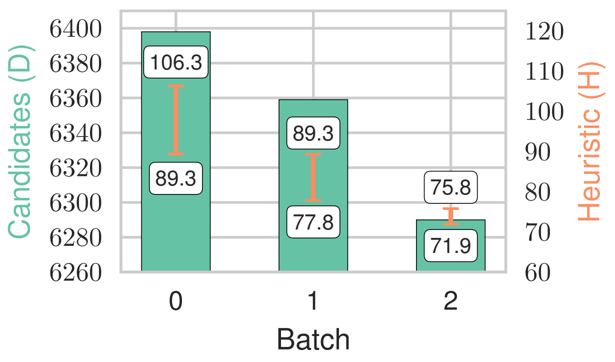


Fig. 9. Sample DSE run for 720p on VC709: 6398 candidates were initially in S, DSE succeeded after examining (synthesizing) three batches of 16 candidates each. The first 16 all had timing failures, which generated 16 new feedback candidates (with reduced target f), while pruning 39 candidates. S contained 6359 candidates at the second batch, of which 14 were timing failures generating feedback candidates, two had placer errors and 42 candidates were pruned. In the last batch, 6290 candidates were left in S and top succeeded, yielding the solution. The quality of that solution was so good that all other candidates (with smaller h-values) were removed from S and DSE terminated.

accelerators operating at HD resolutions could not be placed on the ZedBoard, even that small platform can handle real-time (> 25 fps) stereo vision computations at VGA resolution. The large VC709 board allows live processing of up to three independent 720p video stream pairs, or live processing of a single full HD 1080p stream pair.

Some of the optimal solutions found by DSE might easily have been overlooked in a manual approach: The best design for the VC709 and 1080p resolution was the result of the target clock frequency being *lowered* after a timing failure. Also, casual experimentation might lead a designer to go for higher values of P and lower values of D, as that leads to higher clock frequencies (e.g., 205 MHz for VGA resolution the ZC706 board). However, the best solutions for 720p and 1080p actually increase D and (for 1080p) lower P, and achieve the best throughputs despite running at lower clock rates.

F. Comparison with Original Architecture

By a combination of architectural refinements as well as a modern latency-insensitive design style, our implementation exceeds the performance of the original architecture [7] not only in absolute terms, but also when eliminating the effect of improvements in FPGA technology (we use 7th generation devices such as Artix/Kintex/Virtex-7 fabrics, while the original work by Banz et al. targeted Virtex-5 FPGAs). At the original clock frequency of 133 MHz, Banz' core achieves 167 VGA fps in a (30,1) configuration, while our approach already hits 174 fps in an identical configuration. However, by exploiting fine-grained parallelism and core variants, the design-space exploration step is able to discover even better configurations. Specifically, a smaller (10,4) configuration is able to achieve the *same* 174 fps, but requires only 50 % of the LUT area (which in turn allows the use of smaller devices and a corresponding reduction in energy consumed).

VI. Conclusion and Future Work

The proposed architecture to compute semi-global matching on FPGAs performs well over a wide range of scenarios. Low-power VGA configurations run at 30 FPS with a clock as low as 28 MHz on mid-range boards like the ZC706. For higher performance needs, the architecture offers multiple levels of parallelization, and can be tuned by TPC using automatic design space exploration to discover optimal configurations. Peak performance of the design exceeds 400 FPS (up from 300 FPS in [15]) at VGA resolution, which would enable real-time multi-camera surround vision using a single FPGA chip for processing. Using TPC for integrating the SGM accelerator into a complete system-on-chip, the design is easily portable between small embedded and high-performance data center-grade platforms.

Our introduction of fine-grained parallelism into the Row Processors allows a much better adaptation of the accelerators to the needs of the individual use-case, as just increasing the number of Row Processors (as originally done in [7]) does not always result in the most efficient implementation.

Areas for future work include extending the use of fine-grained parallelism to the initial processing steps of the architecture, namely the per-pixel cost computation (including the rank transform) and the P_2 calculation, as well as reducing the inter-cycle dependencies inside a single Row processor.

Acknowledgment

This work was partially funded by the EU in the FP7 research project *REPARA* (ICT-609666). The authors would also like to thank Xilinx, Inc. for supporting their work by hardware and software donations; as well as Bluespec, Inc. for making their design tools available for education and research.

References

[1] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[2] D. Scharstein and R. Szeliski, "High-accuracy stereo depth maps using structured light," in *IEEE CVPR*, vol. 1, Jun. 2003.

[3] J. Korinth, D. d. l. Chevallerie, and A. Koch, "An Open-Source Tool Flow for the Composition of Reconfigurable Hardware Thread Pool Architectures," in *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual Int. Symposium on*, May 2015.

[4] H. Hirschmüller, "Accurate and efficient stereo processing by semi-global matching and mutual information," in *IEEE CVPR*, vol. 2, Jun. 2005.

[5] ——, "Stereo processing by semiglobal matching and mutual information," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 30, no. 2, pp. 328–341, 2008.

[6] R. Zabih and J. Woodfill, "Non-parametric local transforms for computing visual correspondence," in *European Conf. on Comp. Visi. (ECCV) 94*, J.-O. Eklundh, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1994.

[7] C. Banz, S. Hesselbarth, H. Flatt, *et al.*, "Real-time stereo vision system using semi-global matching disparity estimation: Architecture and fpga-implementation," in *Emb. Comp. Sys. (SAMOS), 2010 Int. Conf. on*, Jul. 2010.

[8] R. Spangenberg, T. Langner, S. Adfeldt, *et al.*, "Large scale semi-global matching on the cpu," in *Intelligent Vehicles Symposium Proc., 2014 IEEE*, Jun. 2014.

[9] M. Michael, J. Salmen, J. Stallkamp, *et al.*, "Real-time stereo vision: Optimizing semi-global matching," in *Intelligent Vehicles Symposium (IV), 2013 IEEE*, Jun. 2013.

[10] O. J. Arndt, D. Becker, C. Banz, *et al.*, "Parallel implementation of real-time semi-global matching on embedded multi-core architectures," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 Int. Conf. on*, Jul. 2013.

[11] K. Schmid and H. Hirschmüller, "Stereo vision and imu based real-time ego-motion and depth image computation on a handheld device," in *Robotics and Automation (ICRA), 2013 IEEE Int. Conf. on*, May 2013.

[12] W. Wang, J. Yan, N. Xu, *et al.*, "Real-time high-quality stereo vision system in fpga," in *Field-Programmable Technology (FPT), 2013 Int. Conf. on*, Dec. 2013.

[13] A. Qamar, C. Passerone, L. Lavagno, *et al.*, "Design space exploration of a stereo vision system using high-level synthesis," in *Mediterranean Electrotechnical Conf. (MELECON), 2014 17th IEEE*, Apr. 2014.

[14] D. Honegger, H. Oleynikova, and M. Pollefeys, "Real-time and low latency embedded computer vision hardware based on a combination of fpga and mobile cpu," in *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ Int. Conf. on*, Sep. 2014.

[15] J. Hofmann, J. Korinth, and A. Koch, "A scalable high-performance hardware architecture for real-time stereo vision by semi-global matching," in *Computer Vision and Pattern Recognition Workshops, 2016. CVPRW '16. Conf. on*, Jul. 2016.

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