High Level Synthesis of Stereo Matching: Productivity, Performance, and Software Constraints

Kyle Rupnow, Yun Liang, Yinan Li, Dongbo Min
Advanced Digital Sciences Center
{k.rupnow,eric.liang,yinan.li,dongbo}@adsc.com.sg

Minh Do, Deming Chen
University of Illinois at Urbana-Champaign
{minhdo,dchen}@illinois.edu

FPGAs are an attractive platform for applications with high computation demand and low energy consumption requirements. However, design effort for FPGA implementations remains high – often an order of magnitude larger than design effort using high level languages. Instead of this time-consuming process, high level synthesis (HLS) tools generate hardware implementations from high level languages (HLL) such as C/C++/SystemC. Such tools reduce design effort: high level descriptions are more compact and less error prone. HLS tools promise hardware development abstracted from software designer knowledge of the implementation platform.

In this paper, we examine several implementations of stereo matching, an active area of computer vision research that uses techniques also common for image de-noising, image retrieval, feature matching and face recognition. We present an unbiased evaluation of the suitability of using HLS for typical stereo matching software, usability and productivity of AutoPilot (a state of the art HLS tool), and the performance of designs produced by AutoPilot. Based on our study, we provide guidelines for software design, limitations of mapping general purpose software to hardware using HLS, and future directions for HLS tool development. For the stereo matching algorithms, we demonstrate between 3.5X and 67.9X speedup over software (but less than achievable by manual RTL design) with a five-fold reduction in design effort vs. manual hardware design.

I. Introduction

Stereo matching is an important underlying technology for 3D video; depth maps generated by stereo matching are used for interpolated video views and 3D video streams. Stereo matching is an extremely active area of research in computer vision – the main quality evaluation benchmark suite currently lists over 100 different algorithms [1]. Techniques employed for stereo matching algorithms include global energy minimization, filtering, and cost aggregation, which are for image de-noising, image retrieval, feature matching, and face recognition among other image & video processing. The computational complexity of computer vision applications in general and stereo matching applications specifically demands hardware acceleration to meet frame rate goals, and its rapid evolution demands a shorter development cycle. For these reasons, stereo matching is representative of many computer vision applications – it requires a fast development cycle; and the available software is representative of algorithms developed for CPU implementations (but not designed or optimized for HLS).

FPGA devices have long been an attractive option for energy efficient acceleration of applications with high computation demand. However, hardware development targeting FPGAs remains challenging and time-consuming – often requiring hardware design expertise and a register transfer level (RTL) algorithm description for efficient implementation. Manual design of RTL hardware often takes many months – an order of magnitude longer than software implementations even when using available hardware IP [2-4].

High level synthesis (HLS) targets this problem: HLS tools synthesize algorithm descriptions written in HLLs such as C/C++/SystemC. A HLL description can typically be implemented faster and more concisely, reducing design effort and susceptibility to programmer error. Thus, HLS provides an important bridging technology – enabling the speed and energy efficiency of hardware designs with significantly reduced design time. In recent years, HLS has made significant advances in the breadth of HLS compatible source code and quality of output hardware designs. Ongoing development has led to numerous industry and academia-initiated HLS tools [5-24] that can generate device-specific RTL descriptions from popular HLLs such as C, C++, SystemC, CUDA, OpenCL, Matlab, Haskell, and specialized languages or language subsets.

The advancements in language support for HLS mean that many implementations can be synthesized to hardware, but the original software design may not be suitable for hardware. HLS-produced hardware is most efficient when the HLL description is written specifically for HLS. Although such software is sometimes available, the vast majority of software is not designed for HLS. Among this software includes large classes of software such as computer vision that have high computation demand and high acceleration potential.

Although there are many success stories of using HLS tools, there is little systematic study of using HLS tools for hardware design, particularly when the original software is not written specifically for HLS. Without such a study, the crucial insights into how to use current state of the art HLS tools are lacking, including:

- Performance of HLS-produced hardware on typical software
- Advantages, disadvantages of common code transformations
- Challenges, limitations of transforming code for HLS
- Coding style for hardware-friendly software design

These limitations motivate us to investigate the abilities, limitations and techniques to use AutoPilot [10], one of the state of the art HLS tools. In this paper, we evaluate the performance gap compared to manual design, coding constraints, required code optimizations, and development time to convert and optimize software for HLS. In this study, the original software implementations are created by computer vision researchers unfamiliar with hardware design constraints, and subsequently converted and optimized by hardware designers. The original source code may contain poor programming for HLS, but it is representative of typical source. Thus, we study the feasibility of converting typical CPU code (potentially poor HLS code) to HLS-friendly code.

We examine a variety of stereo matching algorithms, evaluate suitability of the software for AutoPilot compatibility and convert six suitable software implementations not originally intended for HLS. Our experiments demonstrate HLS achieves 3.5x to 67.9x speedup with a five-fold reduction in design effort compared to manual RTL design, where manual RTL design is still faster than the HLS produced RTL. This paper contributes to the study of HLS with:

- Evaluation of common barriers to HLS compatibility
- An effective HLS optimization process
- A case study of stereo matching algorithms for HLS suitability
- Guidelines for mapping general purpose SW to HW using HLS
- Directions for future study and enhancements of HLS tools

The rest of this paper is organized as follows: Section II discusses the stereo-matching problem and Section III discusses the AutoPilot HLS tool and its supported features. Section IV discusses the suitability of the 12 open source algorithms for HLS, barriers to HLS compatibility, and algorithm details. Section V presents our optimization technique, the experiments and results of optimized high level synthesis of these open source stereo matching algorithms, and Section VI presents our observations and insights on the productivity, usability and software constraints to use HLS.

II. Stereo Matching

Stereo matching is an important underlying technology for 3D-video applications. It measures the disparity between corresponding points in an object between two or more time-synchronized but spatially separated images, captured by a multiple camera system [25]. Input images are rectified to make the problem easy and accurate, so corresponding pixels are assumed to be on the same horizontal line in the left and right images. Disparity measures distance in pixels between an object in one image and the same object in another image, which is inversely proportional to object depth, as depicted in Figure 1. The depth map is subsequently used to generate interpolated view angles and 3D video streams.

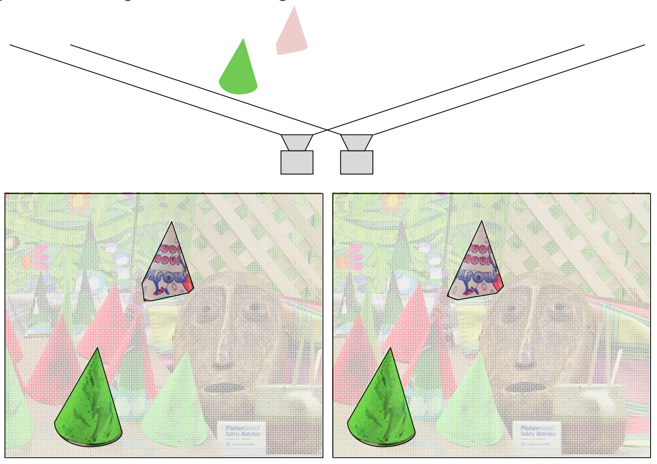

Figure 1 - Example image capture for stereo matching. Two cameras physically offset capture an image of the same scene. The disparity between the objects in the left and right images infers information about object depth. One foreground and one background object are highlighted for clarity.

The depth-map is used together with input color image(s) to produce synthesized views for 3D video applications. For example, in a stereoscopic (glasses-based) 3D-display, each viewer may have different optimal viewing conditions due to individual variation in depth perception. Auto-stereoscopic (glasses-free) 3D displays may require as many as 36 different views for the supported range of viewing angles and distances.

Computation complexity to measure pixel disparity has multiple scaling factors when we attempt to generate depth maps on successively higher resolution video. For a dense depth map, each pixel must be assigned a disparity, high definition video requires high frame rate, and increased image resolution also increases disparity range. Table 1 shows the resolution, typical maximum frame rates and disparity range in pixels for standard, high-definition, and next generation high-definition video standards.

Table 1 - Common Resolutions, Frame Rates and Disparity Ranges[1]

Standard	Resolution	Maximum Frame Rate	Disparity Range	Computation Scaling
Middlebury Test Image	450x375	N/A	60	N/A
SD Video	640x480	30fps	85	1
1080p HD	1920x1080	120fps	256	81
UHDTV	7680x4320	240fps	1024	10000

The large computation complexity of stereo matching requires hardware acceleration to meet performance goals, but stereo matching is also rapidly evolving, which demands reduced development time for hardware implementations. Thus, stereo matching is an attractive application for HLS based hardware design.

[1] Disparity range is computed with the same image sensor size (17.3 mm^2), focal length (2.5 mm), distance between cameras (12mm) and image depth (0.5m) for each image resolution. Parameters are based on the commercial cameras [25]. Computation scaling normalized to standard definition video.

III. AutoPilot High Level Synthesis

AutoPilot is a commercial HLS tool developed by AutoESL [10] that supports input languages of C, C++ and SystemC, which can be annotated with directives to guide the high level synthesis with respect to the hardware implementation. AutoPilot supports a subset of C/C++ – the main unsupported features are dynamic memory allocation and arbitrary indirection (pointers that are not static arrays). AutoPilot supports integer and floating point data types, as well as arbitrary precision fixed-point integer types. AutoPilot employs a wide range of standard compiler optimizations such as dead-code elimination, strength reduction and function inlining. After these code optimizations, synthesis is performed at the function level – producing RTL modules for each function. Each module has private datapath and FSM-based control logic. By default all data arrays are mapped to local BRAMs; scalar variables are mapped to registers.

AutoPilot can apply optimizations to five groups of software source code: communication interfaces, function calls, for loops, data arrays, and labeled regions (a named code section enclosed by curly brackets). AutoPilot performs some optimizations automatically including expression balancing, loop unrolling, loop flattening, and simple array partitioning. However, AutoPilot is conservative in applying these optimizations to allow the user flexibility in optimizing the design for area, clock speed, throughput or some combination of them. All of AutoPilot's optimizations are available as `#pragma` annotations and synthesis script directives.

After code optimizations, AutoPilot uses information about the implementation platform to further specialize the code to the particular platform. The hardware synthesis process then maps the optimized code to hardware, performing computation scheduling, resource binding, and pipelining. Finally, AutoPilot generates the interface code so that the synthesized code transparently maintains the same communication interface as the original implementation.

A. *Communication Interfaces*

Directives can specify that data accesses use particular communication interface protocols such as ACK, Valid, memory, or FIFO (among others). Additionally, users can define their own protocol and define a code region as a protocol so that code in that region is not rescheduled. For this work, we do not develop or use specialized communication protocols.

Table 2 - Communication Interface Directives

Directive	Description
`protocol`	Region is a protocol – do not reschedule operations
`interface`	Use a specified protocol (among predefined list)

B. *Function Calls*

By default, AutoPilot generates RTL for each functional call as a separate module, and function execution is not overlapped. Directives can specify that functions can use fine-grained communication and overlap computation of multiple functions. In addition, directives can inline functions to prevent extra levels of RTL hierarchy and guide AutoPilot's optimization.

Table 3 - Function Call Directives

Directive	Description
`dataflow`	dataflow optimization to overlap computation between multiple function calls (or loop, or regions) – used with ping-pong or FIFO buffers
`instantiate`	Create a separate implementation of this function call to separately optimize each 'instantiated' call
`inline`	Inline this function (do not create RTL hierarchy) to allow resource sharing and optimization

C. *For loops*

For loops are kept rolled by default to maintain the maximum opportunity for resource sharing. AutoPilot directives can specify full or partial unrolling of the loop body, combination of multiple loops, combination of nested loops. When accessing data in arrays, loop unrolling is commonly performed together with data array partitioning (next sub-section) to allow multiple parallel independent array accesses, and thus creating parallelism opportunity along with pipelining opportunity. In addition, directives can specify expression balancing for improved fine-grained parallelism, and pipelining of computation within a code section.

TABLE 4 - FOR LOOP DIRECTIVES

Directive	Description
`loop_flatten`	Combine multiple levels of perfectly nested loops to form a single loop with larger loop bounds
`loop_merge`	Combine two separate loops at the same hierarchy level into a single loop
`loop_unroll`	Duplicate computation inside the loop – increase computation resources, decrease number of iterations
`pipeline`	Pipeline computation within the loop (or region) scope – increase throughput and computation resources
`occurrence`	Specify that one operation occurs at a slower (integer divisor) rate than the outer loop – improve pipeline scheduling, resource use
`expression_balance`	Typically automatic – code in the loop (or region) is optimized via associative and commutative properties to create a balanced tree of computation

D. *Data arrays*

Data arrays may be transformed to improve parallelism, resource scheduling, and resource sharing. Arrays may be combined to form larger arrays (that fit in memory more efficiently) and/or divided into smaller arrays (that provide more bandwidth to the data). In addition, when the data is accessed in FIFO order, an array may be specified as streaming, which converts the array to a FIFO or ping-pong buffer, reducing total storage requirements.

TABLE 5 - DATA ARRAY DIRECTIVES

Directive	Description
`array_map`	Map an array into a larger array – allow multiple small arrays to be combined into a larger array that can share a single BRAM resource
`array_partition`	Separate an array into multiple smaller arrays – allow greater effective bandwidth by using multiple BRAMs in parallel
`array_reshape`	First partition an array, then map the sub-arrays together back into a single array – creates an array with same total storage, but with fewer, wider entries for more efficient use of resources
`array_stream`	If array access is in FIFO order, convert array from BRAM storage to a streaming buffer

E. *Labeled Regions*

Some of the directives (as denoted in Table 4) can also be applied to arbitrary sections of code labeled and enclosed by curly brackets. This allows the programmer to guide AutoPilot's pipelining and expression balancing optimizations to reduce the optimization space.

F. *Comparison to other HLS tools*

Two of the main competitors to AutoPilot, CatapultC and ImpulseC, use similar transformations but with fewer total features. CatapultC supports C++/SystemC, with data array and loop pragmas, but no function or dataflow transformations. Available pragmas include loop merging, unrolling and pipelining, and data array mapping, resource merging and width resizing.

ImpulseC uses a highly customized subset of the C language, with coding style restrictions to make the input more similar to HDL. As a result, ImpulseC supports a wide range of loop and data array transformations, again without function or dataflow pragmas (dataflow hardware is described explicitly). ImpulseC supports simultaneous loop optimization with automatic memory partitioning (using scalarization). Other ImpulseC pragmas are specifically related to the coding style, which requires explicit identification of certain variable types used for inter-function communication.

IV. INITIAL EVALUATION: ALGORITHMS

We examined twelve sets of freely available stereo matching source code including an internally developed stereo matching code. The available source includes both published research work [26-34] as well as unpublished stereo matching source code. As stated previously, these codes are developed for stereo matching research, not suitability for HLS. Thus, despite this seeming wealth of available source code, many of the source packages use common programming techniques that are only efficient (and wise) to use in software, but are unsuitable for HLS support. In this case, these are:

- Libraries for data structures (e.g. Standard Template Library)
- OpenCV computer vision library for efficient implementations of common, basic vision algorithms
- Use of dynamic memory re-allocation
- Use of arbitrary pointers for run-time indirection, pointer passing, pass-by-reference (when they cannot be converted)

For example, as an effort to compare and evaluate many stereo matching algorithms, Scharstein et al. [26] developed a framework that implements many algorithms within a single software infrastructure. However, the implementation employs heavy use of memory re-allocation to instantiate the correct combinations of computation blocks and re-size storage elements properly.

Stereo matching algorithms can be classified into global and local approaches. Global approaches use a complex technique to simultaneously optimize the disparity matching costs for all pixels. In contrast, local approaches compute the matching costs individually for each pixel and use cost aggregation methods that use local image data for semantic information. From the above algorithms, six of the thirteen codes can be transformed for HLS compatibility. For each algorithm, we perform transformations to improve suitability for HLS, but we do not re-implement the algorithm with HLS in mind.

We test two global approaches, Zitnick and Kanade [27], and constant-space belief propagation [28]; and three local approaches, our internally developed stereo matching code, a scanline-optimized dynamic programming method, and cross-based local stereo matching [34]. The algorithms use differing underlying techniques to generate depth maps, but the purpose of this paper is not to judge the relative merits of different approaches in terms of depth map accuracy. Rather, we discuss algorithm and implementation details that make the algorithms more or less suitable for HLS. In the next sub-sections, we discuss these algorithms.

A. *Zitnick and Kanade (ZK)*

The algorithm proposed by Zitnick and Kanade [27] generates dense depth maps under two global constraints: 1) *uniqueness* – each pixel in the depth map corresponds to one and only one depth (and thus disparity value), and 2) *smoothness* – in most portions of the depth map, the depth of adjacent pixels is continuous.

The implementation of the ZK algorithm is based on a 3D array, with one entry for each possible combination of pixel and disparity. The ZK algorithm uses a large, dense, 3D array of data for computation and storage; although the code is *compatible* with AutoPilot's language restrictions, the access order is not suitable for streaming (to reduce storage needs), and bit-width reductions are insufficient to reduce storage needs. Therefore, due to the infeasible storage requirements, we omit ZK from detailed synthesis results.

B. *Constant Space Belief Propagation (CSBP)*

The constant space belief propagation algorithm [28] also generates dense depth maps based on a global energy minimization solution. In original belief propagation [29], data cost is computed per-pixel and disparity value. Then, each pixel iteratively updates messages with its 4 neighbors based on the smoothness constraint, and the final disparity is estimated as the minimum cost. CSBP refines BP by reorganizing computation so memory use is independent of the maximum disparity (but scales with image size). Hierarchically, pixel messages are computed on down sampled versions of the image and successively refined as the image is scaled towards the original resolution. Thus, CSBP scales the computation hierarchy in order to limit the maximum memory consumption.

C. *Bilateral Filtering with Adaptive Support Function (BFAS)*

The BFAS algorithm is a new local stereo method developed by a team of our computer vision researchers for this paper as a driver algorithm to study HLS capabilities. It consists of an initial estimation using absolute difference between pixel values, multiscale image down-sampling [35] and the fast bilateral filtering method [36] for initial cost aggregation, and refinement using an adaptive support function The depth map is computed using winner-takes-all voting and occlusion via cross-checking left and right disparity maps.

Following the depth map computation, we can optionally refine the depth map quality in a post-processing step that uses an adaptive support function. In this aggregation step, each pixel's contribution to the aggregated choice is scaled based on the distance (within the support window) from the center of the window, and color-space distance as shown in Figure 2.

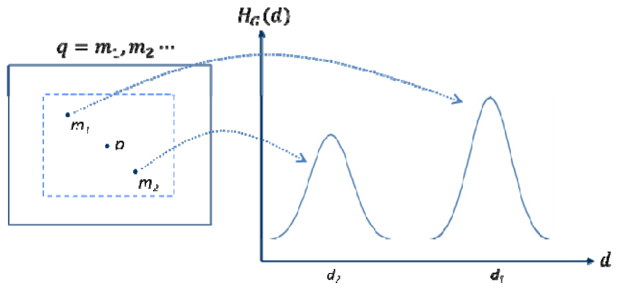

Figure 2 – Calculation of the adaptive support function with Gaussian filters. Horizontal axis d represents disparity, $H_G(d)$ is the Gaussian filter amplitude. Point m_1 above has a larger $H_G(d)$ value because of closer color-space value.

D. *Scanline Optimization (SO)*

Scanline optimization [26][2] is a simple 1D-optimization variant of a dynamic programming stereo matching formulation. Each individual row of the image is independent – pairwise matching costs are computed for each pixel and disparity, then the minimum cost path through the matrix of pairwise matching costs simultaneously optimizes for the matching cost function and the smoothness constraint (small disparity change between adjacent pixels). Optionally, SO can generate a single depth map, or generate left and right depth maps that can be used with a cross checking technique to compute pixel occlusion within the depth maps.

[2] Software is an internally developed version of the algorithm in [26].

E. *Cross-based Local Matching (CLM)*

Cross based local matching [34] is, like the BFAS algorithm, a local matching algorithm. However, whereas the BFAS algorithm uses a fixed window size and Gaussian filtering for aggregation, CLM uses an adaptive window shape and size to determine the pixels to aggregate. The support region consists of a region of contiguous pixels where the pixel's luminance value is within an empirically selected delta. Each pixel's support region is defined based on four parameters, pixel distance for ±horizontal and ±vertical, which forms a cross in the center of the region, and defines the bounds of the irregularly shaped region. Support regions are computed independently for the left and right images, and pixels that are in both the left and right support regions are used for cost aggregation.

V. Optimizing Stereo Matching Codes for AutoPilot

In total, we will evaluate the AutoPilot high level synthesis output for five software versions of the algorithms presented in section IV: BFAS, CSBP, SO without occlusion, SO with occlusion, and CLM. For each software version, we perform a five-step optimization process to convert for compatibility and optimize the hardware implementation. The five optimization steps are baseline implementation; code restructuring; bit-width reduction and ROM conversion; pipelining; and parallelization via resource duplication. These optimizations are intended to reduce resource use, improve resource utilization, expose software features that can be pipelined and/or parallelized, and take full advantage of available FPGA resources.

For each optimization step, we use autocc (AutoPilot's C compiler) and autosim to verify correctness of the modified code. Then, we perform HLS using AutoPilot [10] version 2010.A.4 targeting a Xilinx Virtex 6 LX240T. Then, if the AutoPilot-produced RTL can fit in the FPGA, we synthesize the RTL using Xilinx ISE 12.1. Area and clock period data is obtained from ISE post-placement and routing reports. After synthesis of AutoPilot's RTL, we verify a shorter subset (one partition of the image) via ModelSim simulation, and measure latency in clock cycles. Then, hardware latency in seconds is computed by multiplying the clock period by the measured clock cycles; speedup is the ratio of hardware latency to original (unmodified) software latency. The software execution is performed on an Intel i5 2.67 GHz CPU with 3GB of RAM.

A. *Baseline – minimum modifications*

For each software version, we generate a baseline implementation – the minimum code modifications so that the algorithm can be properly synthesized using AutoPilot. These modifications include conversion of dynamic memory to static declarations, conversion of memcpy and memset calls to for loops, and conversion of arbitrary (run-time changing) pointers to static pointers to memory locations.

Each of the baseline implementations is AutoPilot compatible, and produces hardware designs that produce correct output. However, each of the software versions uses significantly more BRAM resources than available in the LX240T – as much as 10x. In addition the BFAS software versions also have complex computation, which causes over-constrained use of LUT, FF and DSP resources as well. Of the 6 software sources, only SO without occlusion can fit in the FPGA after minimum modifications.

In addition to area inefficiency, these designs are always slower than the original software, sometimes by an order of magnitude. These results are expected, as AutoPilot's default settings do not unroll loops, pipeline computation, or transform data storage elements. In addition, the slowdown is exacerbated by several factors including reduced efficiency of for loops vs. memset/memcpy calls, limited pipeline and parallelism, inefficient datapath width, and the difference between the CPU clock period and the achievable FPGA clock period. All of these reasons will be eliminated or mitigated by the following optimization steps.

B. *Code restructuring*

For stereo matching algorithms, the most important code restructuring task is to partition the image into sub-images that can independently compute disparity and depth information. In all of the algorithms, the largest required data storage element(s) are directly proportional to the size of image processed. For the SO algorithm, this conversion is relatively simple: disparity computation is dependent within one row of pixels but independent between rows. However, the CSBP, BFAS, and CLM algorithms use windowed computation and a support window that spans both rows and columns of the image. These algorithms employ averaging, interpolation and filtering. Thus, we must partition the image into overlapping windows so that the computation result is correct.

In addition to image partitioning, we also perform function merging, loop merging, interchange nested loops to improve parallelism opportunity and share internal memory buffers to reduce resource use. At this stage in optimization, we perform these code transformations manually. Although AutoPilot has synthesis directives that can merge loops or eliminate memory buffers that are used in FIFO order, these directives are relatively limited compared to transformations we can perform manually.

For BFAS, this step results in a 50% reduction in AutoPilot estimated LUT use, 70% fewer flip-flops, 60% fewer DSP blocks and 95% fewer BRAMs. The other software versions do not employ computation resources as heavily, so there was less benefit for LUT, FF, or DSP resources, but all versions received at least 90% reduction in memory resources.

C. *Reduced bitwidth and ROM conversion*

Throughout all of the algorithms, full width integer data types are commonly used for convenience. However, based on operand range analysis, we can reduce the operand bit-width. Similarly, floating point computation is used for convenience, but in these applications all values are still in the range of 0-255 (8-bit RGB components), with constants accurate to within 0.001. This relatively low accuracy still requires 10 fractional binary digits, but the combination of bitwidth reduction and smaller, simpler functional units can offer significant gain in terms of both latency and resource.

Using constant-propagation, AutoPilot can sometimes determine that a set of variables have a fixed range and automatically reduce the bit-width of those variables to reduce storage requirements. However, this automatic bit-width reduction is not compatible with AutoPilot's array directives; when we wish to use array_map to share a BRAM between multiple data arrays, AutoPilot will not automatically reduce the variable size of the arrays. Therefore, this optimization step is a multi-step process. First, for each data array in the design, we manually determine the operand range and re-define the array using AutoPilot's integer and fixed-point datatypes, ap_int and ap_fixed, to reduce the per-element storage requirement. Then, we use the array size in number of elements and access order to determine what resources should be used.

For arrays with few total elements (in our case, less than 100) and statically determined access order, we use complete array partitioning which directs AutoPilot to use registers instead of BRAMs. For the other arrays, we search for combinations of arrays where the access order is synchronized or array access is entirely disjoint. For these access patterns, sharing the same physical BRAM does not result in additional read or write latency. Therefore, for such arrays, we use array_map to combine the arrays and share physical BRAMs. For example, in Figure 3, we show code extracted from BFAS; there are 4 parameter arrays with 101 entries each that are used in only one function in synchronized order. The default AutoPilot implementation uses 1 BRAM each for the arrays although the total storage bits is much less than an 18K BRAM. Therefore, we use the array_map pragma to map the arrays together into a single array that is stored in a single BRAM.

```
ap_fixed<20,3> m_bI_F[101] = {…};
ap_fixed<20,3> m_bI1_F[101] = {…};
ap_fixed<20,3> m_bI2_F[101] = {…};
ap_fixed<20,3> m_bI3_F[101] = {…};

#pragma      AP       array_map       instance=m_BI
variable=m_BI_F,m_bI1_F,m_bI2_F,m_bI3_F vertical

RecursiveGaussian_3D(…,              m_BI_F[NumOfI-1],
m_bI1_F[NumOfI-1],                  m_bI2_F[NumOfI-1],
m_bI3_F[NumOfI-1]);
```

Figure 3 - Code Example #1 - Array map directive to save BRAM use

It is important that one of AutoPilot's most powerful optimizations is unavailable due to data access order. The array stream pragma converts data arrays that are accessed in FIFO order into smaller, fixed size buffers (significantly reducing BRAM use), and also allows dataflow optimizations which overlap computation of multiple RTL blocks in the same manner as pipelining does on smaller computation units. This optimization can have significant impact, but is only available if data is written and read in FIFO order, which is not the case for any of the algorithms tested.

After bitwidth optimization and array directives, all of the algorithms except CSBP reduce BRAM consumption by 50% to 75%. CSBP primarily uses one large array of data, so there is no gain via the use of array directives, and the data element size was already chosen well for the storage. However, for all of the algorithms, the bitwidth optimization also reduced LUT, FF and DSP use due to reduced size datapath computation units.

Only BFAS made use of functions that were suitable for conversion into ROM tables; after ROM conversion, BFAS had an additional 8% reduction in LUT use, 5% fewer FFs and 7% fewer DSP units than the bitwidth reduced software. The BFAS algorithm uses exponential, square root, and logarithm floating point functions, but with small input ranges; the cost of a ROM is small compared to the cost of implementing floating point or integer functional units that perform these functions.

D. *Pipelining and loop optimization*

After restructuring and bitwidth reduction, we have reduced the amount of computation and memory resources that must be used per image partition. In this step, we examine the computation loops in the program and apply loop pipelining, loop merging, loop flattening and expression balancing to optimize performance. Because of the manual transformations in the code restructuring step, there are relatively few opportunities for loop merging, but it is used in a few cases to combine initialization loops with different loop bounds. When possible, we convert imperfectly nested loops to perfectly nested loops to allow loop flattening, which saves 1 cycle of latency for each traversal between loop levels.

For inner loops, we normally use pipelining to improve the throughput of computation. Using the pipeline directive, we set an initiation interval (II) of 1 as the target for all pipelined code. In most cases, AutoPilot can achieve this initiation interval. However, in some cases the computation on the inner loop requires multiple reads and/or writes from/to different addresses in the same BRAM. Thus, for these loops the initiation interval is longer to account for the latency of multiple independent BRAM reads/writes.

In some cases, when the inner loop has a small loop bound and loop content is performing a computation or search (rather than memory writes), we use complete unrolling and expression balancing instead of pipelining. For example, in Figure 4, we show a code section from CLM; instead of pipelining the inner loop computation, we fully unroll the inner loop (with DprRange=60), and then use expression balancing to perform the search for a maximum pDprCount value in parallel instead of sequentially.

For this step, the best benefit is available when computation loops use static loop bounds – a static loop bound allows AutoPilot to perform complete unrolling on the inner loop to increase pipeline efficiency, or partially unroll by a known factor of the upper bound. For this reason, CSBP is particularly poor for this step. CSBP consists of functions that almost universally use variable loop bounds with large variation in the number of iterations. Although this programming style for CPUs maximizes code and data structure re-use, it prevents AutoPilot from optimizing computation resources. With sufficient resources, we could instantiate each function call with different maximum loop bounds, but that also necessarily constrains the hardware search space and prevents resource sharing between the function instantiations. In general, when loop unrolling and parallelizing, it is important to consider resource use; loop unrolling may improve performance, but can limit flexibility of implementing multiple computation pipelines in the parallelization and resource duplication step.

```
VoteDpr = 0;
count   = pDprCount[0];
for(d = 1; d < DprRange; d++){
#pragma AP unroll complete
#pragma AP expression_balance
   if(pDprCount[d] > count){
      count   =  pDprCount[d];
      VoteDpr =  d;
   }
}
```

Figure 4 - Code Example #2 - Loop unroll and expression balancing for fine-grained parallelism

In practice, directive insertion is performed iteratively together with parallelization to find the best trade-off of directives and parallelization. For the software version eventually used with the next optimization step, BFAS achieved 1.5x speedup over the bitwidth & ROM step, CSBP achieved 2.5x improvement, CLM achieved 2.9x speedup, and the SO versions achieved 7.2x and 5.3x with and without occlusion, respectively.

E. *Parallelization and resource duplication*

At this step, we examine the synthesis result of the previous step and further parallelize the computation by duplicating logic within the computation pipeline to instantiate multiple, parallel computation pipelines and fit in the Virtex 6 LX240T. In AutoPilot, function parallelism is easily explored through a combination of array partitioning and loop unrolling. However, AutoPilot does not contain a directive or pragma to explicitly denote parallelism; all parallelism is created implicitly when AutoPilot detects that computations are independent. As a result, introducing parallelism can be sensitive to AutoPilot correctly detecting independence between computations.

To create a position where we can explore functional parallelism, we define an extra dimension on data arrays used within the inner loop. Then, we create a new inner loop and specify complete unrolling of the inner loop. Note that this seems logically identical to a partial partitioning of the data array and partial unrolling of the computation inner loop; however, this method more clearly signifies functional independence to expose the parallelism opportunity to AutoPilot. For example, in Figure 5 we show code extracted from BFAS that demonstrates a section where we can explore the amount of parallelism in the disparity computation by changing the FS_UNROLL parameter.

All of the software versions achieve speedup after parallelization and resource duplication: from 3.5x to 67.9x improvement over the original software. In general, each optimization step provides some incremental improvement, but the final step shows the greatest benefit. However, this is not to mean that the other steps are not important; rather, this emphasizes the importance of minimizing resource consumption in order to allow maximum flexibility in the parallelization step. Figure 6 shows the speedup over original software for each algorithm and optimization step. Figure 7 shows both resource usage and speedup for each optimization step. We can observe that although parallel optimization provides the maximum amount of speedup it also increases the resource usage significantly. However, smart optimization provides opportunities for achieving better speedup without much resource increase. This is the case for BFAS, where bitwidth, directive, and parallelization steps all together use similar resources as what the structure step uses but provide a 6.5X speedup.

Algorithms with larger resource use have relatively little flexibility to employ resource duplication – BFAS can duplicate 4 disparity computation pipelines; CSBP can also use 4 disparity pipelines. In contrast, the SO algorithm is significantly simpler – it can allow 20 parallel pipelines with occlusion and 34 without.

```
#define FS_UNROLL 2
ap_fixed<22,9> DispCost[FS_UNROLL][W* H];
ap_fixed<22,9> CostL[FS_UNROLL][W*H];
#pragma   AP   array   partition   complete   dim=1
variable=DispCost,CostL

for(int k=Min;k<=Max; k += FS_UNROLL){
   FL_00:for(int l=0; l< FS_UNROLL; l++){
   #pragma AP unroll complete
      SubSampling_Merge(…,DispCost[l],…, k+l);
      CostAgg(…,DispCost[l],CostL[l],…);
      Cost_WTA(…,CostL[l],…, k+l);
   }
}
```

Figure 5 - Code Example #3 - Array partition and complete loop unroll to expose functional parallelism

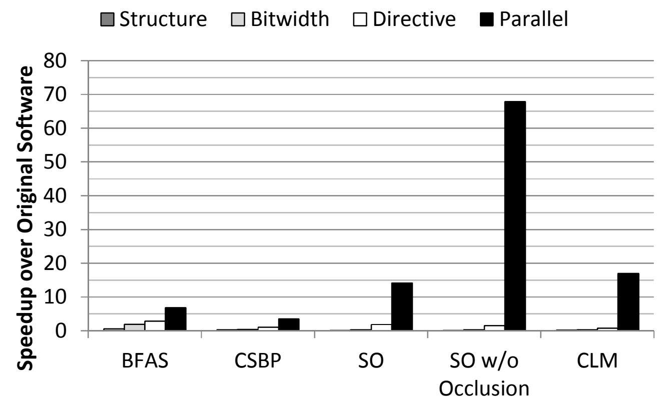

Figure 6 - Speedup over original software for each algorithm, ISE synthesizable optimization step

VI. OBSERVATIONS & INSIGHTS

HLS is an attractive platform for acceleration of stereo matching because of the dual requirements of high performance and fast development cycle. We demonstrated that HLS can achieve significant speedup for these stereo matching algorithms. Now, we evaluate HLS in terms of the productivity, software constraints, usability, and performance of the tools in order to reach the performance we have demonstrated.

A. *Productivity*

It is important to evaluate the design effort required to achieve this level of speedup. Table 6 shows the development effort spent on each algorithm, normalized to development time of a single hardware designer. BFAS required longer than the others, as it was the first algorithm implemented and some time was spent learning the stereo matching problem. The manual stereo matching design presented in [4] required 4-5 months of design effort for an experienced hardware designer to implement the design, plus additional time for a team to design the algorithm. Manual implementations for other application domains quote similar development efforts [2], [3].

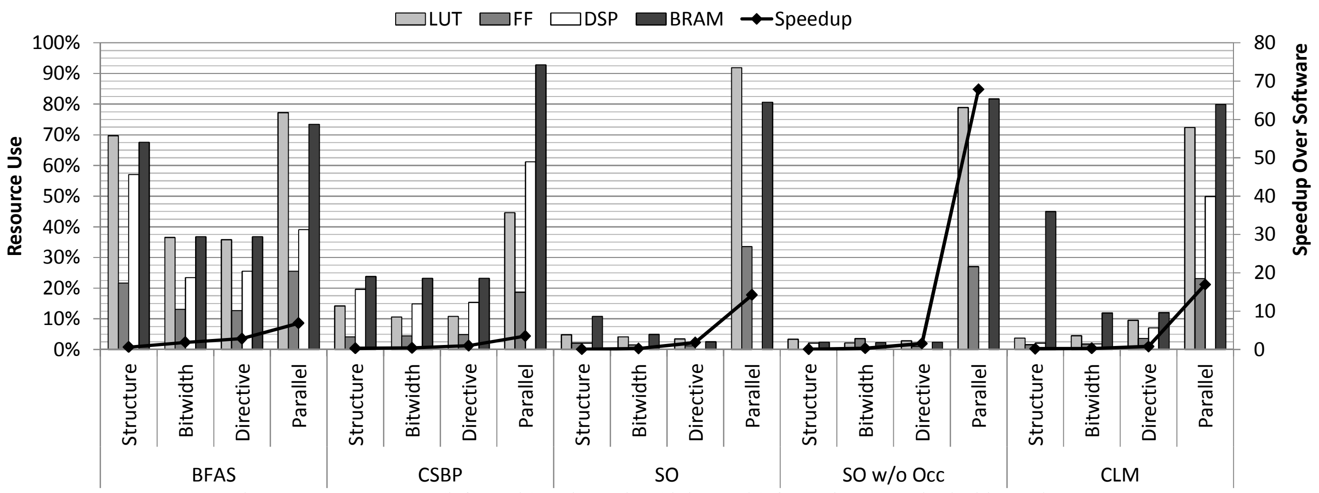

Figure 7 - Resource use (left y-axis) and speedup (right y-axis) for each ISE synthesizable version.

TABLE 6 - DEVELOPMENT EFFORT BY ALGORITHM

Algorithm	Development Effort
BFAS	5 Weeks
CSBP	3 Weeks
CLM	4 Weeks
SO	2.5 Weeks

B. *Software constraints*

As discussed earlier, typical software constraints of HLS tools require statically declared memory, which also precludes the use of many standard libraries such as STL or OpenCV. Furthermore, there are additional guidelines on *efficient* software for HLS. These include:

- Convert loops using `break` or data dependent loop bounds to static loop bounds to improve pipelining/parallelization
- Use FIFO data read & write order for dataflow optimizations
- Reduce operand size to minimize storage needs
- Use array_map to reduce storage by combining arrays
- Use complete array partitioning to convert arrays to scalars
- Structure and parameterize likely parallelization locations to simplify parallelism search space
- Perfectly nest loops when possible – when not possible, consider parallelism on the inner-most loop

The "best" loop to be the innermost loop is a tradeoff between multiple factors including the number of transitions between loop levels (1 cycle of latency per transition), data access order for computation, ability to unroll/parallelize, and ability to pipeline. These factors sometimes conflict (e.g. complete unrolling prefers a small to medium size inner loop, pipelining prefers the largest loop).

In many cases, these software constraints are easily achieved by software engineers familiar with optimization. Although the goals are somewhat different with code that will be hardware, the optimization techniques are similar. However, these constraints sometimes conflict with "good" software engineering practices that maximize code reuse with heavily parameterized code, variable loop bounds and early exit conditions to reduce worst-case paths, and FIFO ordered fine-grained interleaving of computation. These constraints suggest that HLS tools may also need to improve in ability to efficiently handle some such codes. For example, AutoPilot contains a loop_tripcount directive that is used for performance analysis, but not in the synthesis process. If also used during the synthesis process to specify bounds and typical iterations on variable loops, this could allow easier optimization of such code.

C. *Usability*

AutoPilot's optimizations are very powerful – array map and array partition can have significant impact on storage requirements, and together with loop unrolling, it is possible to explore possible parallelism points quite easily. However, automatic transformations sometimes make this task more difficult; by default AutoPilot will attempt to complete unroll an inner loop to expose parallelism when pipelining, but when the loop has variable bounds, this can result in significant overhead.

AutoPilot is conservative in applying optimizations, which prevents generation of incorrect hardware. However, this also can make exposing information about code independence (for parallelism) difficult. For example, the parallelism code shown in section V.E is required because anything except complete array partitioning does not guarantee that AutoPilot will assume partitioned arrays are independently accessed. Furthermore, because AutoPilot does not have a directive to explicitly denote parallelism, creating parallel hardware is sensitive to AutoPilot's ability to detect independence. This can be challenging in cases where by necessity code shares data resources, but the user knows (and could denote) that parallel function calls would not interfere.

Finally, although AutoPilot has powerful optimizations available, it is sometimes difficult to apply it to code that was not designed in advance to use the optimization. As mentioned earlier, array streaming to allow dataflow optimization is an extremely powerful optimization for data marshaling and computation overlapping, but it is only available if data is generated and used in FIFO order, which would require significant restructuring in 2D-windowed computation such as computer vision applications.

D. *Performance*

We have demonstrated that AutoPilot can achieve between 3.5x and 67.9x speedup for these software algorithms, but it is important to consider the performance difference between HLS and manual hardware implementations. A manual implementation of CLM achieved speedup of ~400x [4], similar in magnitude to other FPGA stereo matching solutions [37]. A GPU implementation achieved 20x speedup [38], which is similar to the 17x speedup achieved in this work. In total, we can achieve up to 67.9x speedup even without ability to use the array stream and dataflow optimizations.

It is important to emphasize that this performance gap is a gap between HLS hardware produced from software not designed for HLS and manually produced RTL. This is not to suggest that HLS-produced hardware cannot achieve performance near manual designs, but to point out that the current abilities of HLS cannot be easily used in general software not designed for HLS. We have achieved significant speedup on some of these algorithms without significant re-structuring of the original software, but a significant portion of the remaining gap is due to memory-level dependence and data marshalling. When we examine the hardware design in [4], a major portion of the performance is attained through fine-grained overlapping of computation throughout the design pipeline. Although AutoPilot has synthesis directives that can create this sort of hardware, the code must be designed in advance to use the correct data access order and code structure, and that software code structure is different from typical software structure that is used in CPU source.

E. *Future Directions*

Together, this study leads to two groups of future directions for HLS tools: one to improve the usability and accessibility of currently available HLS features, and the second to improve performance gap between HLS and manual design by adding new features. Some of these observations are specific to AutoPilot's optimization and code generation flow; however, the challenges of supporting a wider range of input source code are applicable to all of the state of the art HLS tools.

1) Usability

- Improved loop unrolling/pipelining for complicated loops that require temporary register and/or port duplication
- Support for port duplication directives to add extra read and/or write ports to BRAM-based storage through directives rather than manual data array duplication
- Automatic tradeoff analysis of loop pipelining and unrolling
- Automatic detection and conversion for common computation structures such as tree-based reductions
- Improved robustness of dataflow transformations, streaming computation for 2D access patterns

2) Performance Gap

- Detection of memory level dependence across multiple, independent loops and functions, automatic interleaving of computation between the loops and functions.
- Automatic memory access re-ordering to allow partitioning, streaming, or improved pipelining
- Automatic temporary buffers for memory access re-use
- Iteration between array optimizations, pipelining and parallelization for efficient search of design space

VII. CONCLUSIONS

High level synthesis tools offer an important bridging technology between the performance of manual RTL hardware implementations and the development time of software. This study uses several modern stereo matching software codes not originally written for HLS, optimizes them, and compares the performance of synthesized output as well as design effort. We present an unbiased study of the progress of HLS in usability, productivity, performance of produced design, software constraints, and commonly required code optimizations. Based on this study, we present both guidelines for algorithm implementation that will allow HLS compatibility and an effective optimization process for the algorithms. We demonstrate that with short development time, HLS based design can achieve 3.5X to 67.9X speedup on these stereo matching codes, but more in-depth, manual optimization of memory level dependence, data marshaling, and algorithmic transformations are required to achieve the larger speedups common in hand designed RTL.

ACKNOWLEDGMENTS

The Advanced Digital Sciences Center is funded by A*STAR under the Human Sixth Sense Project.

REFERENCES

[1] D. Scharstein and R. Szeliski, "Middlebury Stereo Vision Website." http://vision.middlebury.edu/stereo.

[2] J. Bodily, et al., "A Comparison Study on Implementing Optical Flow and Digital Communications on FPGAs and GPUs," *ACM TRETS*, vol. 3, pp. 6:1–6:22, May. 2010.

[3] C. He, A. Papakonstantinou, D. Chen, "A novel SoC architecture on FPGA for ultra fast face detection," in *ICCD*, 2009, pp. 412–418.

[4] L. Zhang, et al., "Real-time high-definition stereo matching on FPGA," in *FPGA*, 2011, pp. 55–64.

[5] P. Bjesse, et al., "Lava: hardware design in Haskell," in *ACM SIGPLAN Notices*, 1998, vol. 34, pp. 174–184.

[6] B. Bond, et. al, "FPGA Circuit Synthesis of Accelerator Data-Parallel Programs," in *FCCM*, 2010, pp. 167–170.

[7] S. S. Huang, et al., "Liquid Metal: Object-Oriented Programming across the Hardware/Software Boundary," *ECOOP*, 2008, pp. 76–103.

[8] M. Lin, et al., "OpenRCL: Low-Power High-Performance Computing with Reconfigurable Devices," in *FPL*, 2010, pp. 458-463.

[9] A. Papakonstantinou, et al., "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs," in *SASP*, 2009, pp. 35-42.

[10] Z. Zhang, et al., "AutoPilot: A Platform-Based ESL Synthesis System," in *High-Level Synthesis: From Algorithm to Digital Circuit*, Ed. P. Coussy, A. Morawiec., 2008.

[11] Nallatech, "DIME-C." http://www.nallatech.com/.

[12] Y Explorations, Inc., "eXCite." http://www.yxi.com/.

[13] Altium, Limited, "C-to-Hardware Compiler User Manual." .

[14] "CatapultC Synthesis Datasheet." .

[15] Synopsys, Inc., "Synthesizing Algorithms from MATLAB and Model-based Descriptions. Introduction to Synphony HLS." .

[16] Altera, Inc., "Nios II C-to-Hardware (C2H) Compiler." .

[17] G. Sandberg, "The Mitrion-C development environment for FPGAs." .

[18] Impulse Accelerated Technologies, Inc., "ImpulseC Datasheet." .

[19] P. Coussy, et al., "High-level synthesis under I/O Timing and Memory constraints." HAL - CCSD, 2005.

[20] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK□: A High-Lev l Synthesis Framework For Applying Parallelizing Compiler Transformations," in *VLSI Design*, 2003, vol. 0, p. 461.

[21] R. Domer, et al., "SpecC Methodology for High-Level Modeling," presented at the DATC Electronic Design Processes Workshop, 2002.

[22] J. L. Tripp, et al., "Trident: From High-Level Language to Hardware Circuitry," *Computer*, vol. 40, pp. 28–37, Mar. 2007.

[23] D. Gajski, "NISC: The Ultimate Reconfigurable Component." Center for Embedded Computer Systems, TR 03-28, Oct-2003.

[24] A. Papakonstantinou et al., "Multilevel Granularity Parallelism Synthesis on FPGAs," FCCM , 2011, pp. 178-185.

[25] Point Grey Research, "Point Grey Stereo Vision Cameras." http://www.ptgrey.com/products/stereo.asp.

[26] D. Scharstein, et al., "A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms," in *IEEE Workshop on Stereo and Multi-Baseline Vision*, 2001, p. 0131.

[27] C. L. Zitnick et al., "A cooperative algorithm for stereo matching and occlusion detection," *IEEE TPAMI*, vol. 22, no. 7, pp. 675-684, 2000.

[28] Q. Yang, L. Wang, N. Ahuja, "A constant-space belief propagation algorithm for stereo matching," in *CVPR*, 2010, pp. 1458-1465.

[29] T. Meltzer, C. Yanover, and Y. Weiss, "Globally Optimal Solutions for Energy Minimization in Stereo Vision Using Reweighted Belief Propagation," in *ICCV*, 2005, vol. 1, pp. 428-435.

[30] B. M. Smith, et al., "Stereo matching with nonparametric smoothness priors in feature space," in *CVPR*, 2009, pp. 485-492.

[31] E. Tola, et al., "DAISY: An efficient dense descriptor applied to wide-baseline stereo," *IEEE TPAMI*, vol. 32, no. 5, pp. 815-830, 2010.

[32] A. S. Ogale, Y. Aloimonos, "Shape and the Stereo Correspondence Problem," *IJCV*, vol. 65, no. 3, pp. 147-162, Dec. 2005.

[33] K.-J. Yoon, I. S. Kweon, "Adaptive support-weight approach for correspondence search," *IEEE TPAMI*, vol. 28, no. 4, pp. 650-656, 2006.

[34] K. Zhang, J. Lu, G. Lafruit, "Cross-based local stereo matching using orthogonal integral images," *IEEE TCSVT*, vol. 19, no. 7, pp. 1073–1079, Jul. 2009.

[35] D. Min, K. Sohn, "Cost aggregation and occlusion handling with WLS in stereo matching," *IEEE TIP*, vol. 17, no. 8, pp. 1431-1442, Aug. 2008.

[36] S. Paris and F. Durand, "A Fast Approximation of the Bilateral Filter Using a Signal Processing Approach," presented at the ECCV, 2006.

[37] S. Jin et al., "FPGA Design and Implementation of a Real-Time Stereo Vision System," *IEEE TCSVT*, vol. 20, no. 1, pp. 15-26, 2009.

[38] K. Zhang, et al., "Real-time accurate stereo with bitwise fast voting on CUDA," in *ICCVW*, 2009, pp. 794-800.