Hindawi Publishing Corporation
Journal of Electrical and Computer Engineering
Volume 2012, Article ID 649057, 14 pages
doi:10.1155/2012/649057

Research Article

High-Level Synthesis: Productivity, Performance, and Software Constraints

Yun Liang,[1] Kyle Rupnow,[1] Yinan Li,[1] Dongbo Min,[1] Minh N. Do,[2] and Deming Chen[2]

[1] *Advanced Digital Sciences Center, 1 Fusionopolis Way, No. 08-10 Connexis North Tower, Singapore 138632*
[2] *Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA*

Correspondence should be addressed to Yun Liang, eric.liang@adsc.com.sg

Received 20 July 2011; Accepted 25 October 2011

Academic Editor: Kiyoung Choi

Copyright © 2012 Yun Liang et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

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 algorithm descriptions in languages such as C/C++ and 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 present an unbiased study of the performance, usability and productivity of HLS using AutoPilot (a state-of-the-art HLS tool). In particular, we first evaluate AutoPilot using the popular embedded benchmark kernels. Then, to evaluate the suitability of HLS on real-world applications, we perform a case study of stereo matching, an active area of computer vision research that uses techniques also common for image denoising, image retrieval, feature matching, and face recognition. Based on our study, we provide insights on current limitations of mapping general-purpose software to hardware using HLS and some future directions for HLS tool development. We also offer several guidelines for hardware-friendly software design. For popular embedded benchmark kernels, the designs produced by HLS achieve 4X to 126X speedup over the software version. The stereo matching algorithms achieve between 3.5X and 67.9X speedup over software (but still less than manual RTL design) with a fivefold reduction in design effort versus manual RTL design.

1. Introduction

Field programmable gate array (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 [1–3].

High-level synthesis (HLS) targets this problem: HLS tools synthesize algorithm descriptions written in a high level language (HLL) 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 both the breadth of HLS compatible input source code and quality of the output hardware designs. Ongoing development of HLS has led to numerous industry and academia-initiated HLS tools [4–25] 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 implementation. High level languages such as C have been shown as an effective means for capturing FPGA circuits when the C description is written specifically for HLS and hardware (e.g., C implementation is derived from manual FPGA implementation) [24]. However, although software

specifically written for HLS is sometimes available, the vast majority of software is not designed for synthesis.

There are many success stories of using HLS tools, but there is little systematic study of using HLS tools for hardware design, particularly when the original software is not written specifically for HLS. The code refinement process for HLS [25] and the benchmarks proposed by Hara et al. [26] bear similarity to our work. In [25], various coding guidelines are introduced to make code more HLS friendly and better performing. In [26], CHStone, a suite of benchmarks for HLS, is proposed, and the benchmarks are analysed in terms of source level characteristics, resource utilization, and so forth. In contrast, in this paper we present a systematic study of HLS including its productivity, performance, and software constraints. Without such a systematic study, the crucial insights into how to use current state of the art HLS tools are lacking, including the following.

(i) Performance of HLS-produced hardware on typical software.

(ii) Advantages and disadvantages of common code transformations.

(iii) Challenges and limitations of transforming code for HLS.

(iv) Coding style for hardware-friendly software design.

These limitations motivate us to investigate the abilities, limitations and techniques to use AutoPilot [9], one of the state of the art HLS tools. In this paper, we evaluate the achievable speedup over software design, the performance gap compared to manual design, coding constraints, required code optimizations, and development time to convert and optimize software for HLS. First, we evaluate AutoPilot using embedded benchmark kernels. Through evaluation of these kernels, we test the suitability of HLS on a wide range of applications as these kernels are widely used for various applications. However, these kernels are relatively small—real applications will contain multiple computation kernels that communicate, and more complex coding styles and data structures. Whereas a single small kernel can be easily transformed to correspond to a coding style well-suited for HLS, such transformations are more difficult for real applications. Thus, we select stereo matching [27] as a case study of HLS on complex real-world applications. Stereo matching is an important underlying technology for 3D video; the depth maps generated by stereo matching are used for interpolated video views and 3D video streams. Techniques employed for stereo matching algorithms include global energy minimization, filtering, and cost aggregation, which are used throughout image and video processing applications. In this study, stereo matching presents the additional challenge that software implementations are created by computer vision researchers unfamiliar with hardware design constraints. Thus, our stereo matching case study must optimize software not originally designed for HLS implementation.

HLS implementations of the embedded benchmark kernels achieve 4X to 126X speedup over the software implementation. In our stereo matching case study, we examine a variety of stereo matching algorithms, evaluate suitability of the software for AutoPilot compatibility, and convert four suitable software implementations not originally intended for HLS. Our experiments demonstrate that HLS achieves 3.5X to 67.9X speedup with 5X 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 the following.

(i) Evaluation of common barriers to HLS compatibility.

(ii) An effective HLS optimization process.

(iii) Evaluation of HLS on popular embedded benchmark kernels.

(iv) A case study of stereo matching algorithms for HLS suitability.

(v) Guidelines for mapping general-purpose SW to HW using HLS.

(vi) Directions for future study and enhancements of HLS tools.

The rest of this paper is organized as follows. Section 2 discusses the AutoPilot HLS tool and its supported features. Section 3 discusses the embedded benchmarks kernels, the stereo matching problem, and various stereo matching algorithms for HLS. Section 4 presents our HLS optimization process. Section 5 presents the experiments and results, and finally Section 6 presents our observations and insights on the productivity, usability, and software constraints to use HLS.

2. AutoPilot High Level Synthesis

AutoPilot is a commercial HLS tool developed by AutoESL (AutoESL was acquired by Xilinx in January 2011) [9] 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

Table 1: Communication interface directives.

Directive	Description
`protocol`	Region is a protocol—do not reschedule operations
`interface`	For a communication interface, use a specified protocol (among predefined list)

Table 2: 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—allow separate optimization of each “instantiated” function call
`inline`	Inline this function call (do not create separate level of RTL hierarchy)—allow resource sharing and optimization across hierarchy levels

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.

2.1. 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. Table 1 shows the details. For this work, we do not develop or use specialized communication protocols.

2.2. Function Calls. By default, AutoPilot generates RTL for each functional call as a separate module, and function execution is not overlapped. The directives (Table 2) 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.

2.3. 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, and combination of nested loops. When accessing data in arrays, loop unrolling is commonly performed together with data array partitioning (next subsection) to allow multiple parallel independent array accesses, and thus creating parallelism opportunity along with pipelining opportunity. In addition, the loop directives

Table 3: 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

Table 4: 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 subarrays 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

as shown in Table 3 can specify expression balancing for improved fine-grained parallelism, and pipelining of computation within a code section.

2.4. 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 an FIFO or ping-pong buffer, reducing total storage requirements. Table 4 lists the array directives.

2.5. Labeled Regions. Some of the directives (as denoted in Table 3) 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.

Table 5: Kernel characteristics.

Kernel Name	Description
`Matrix Multiply (MM)`	Computes multiplication of two arrays (used in many applications). Array sizes are 1024 × 1024 or 512 × 512. Data type is 32-bit integer.
`Blowfish encrypt/decrypt`	Blowfish is a symmetric block cipher with a variable length key. It is widely used for domestic and exportable encryption.
`Adpcm encode/decode`	Adaptive Differential Pulse Code Modulation (ADPCM). It is a variation of the standard Pulse Code Modulation (PCM).
`AES`	Advanced Encryption Standard (AES) is a block cipher with option of 128-, 192-, and 256-bit keys and blocks.
`TDES`	TDES applies the Data Encryption Standard (DES) cipher algorithm three times to each data block.
`SHA`	SHA is the secure hash algorithm. It is often used in the secure exchange of cryptographic keys and for generating digital signatures.

2.6. Other HLS Tools and Their Evaluation. CatapultC [13] and ImpulseC [17] are two widely used industry HLS tools. CatapultC and ImpulseC use transformations similar to AutoPilot 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 interfunction communication. LegUp is an academic-initiated open source HLS tool [23]. Given a C program, LegUp can automatically perform hardware software codesign, where some program segments are mapped to custom hardware (synthesized from the C-code) and the remaining code is mapped onto an FPGA-based soft processor. LegUp leverages the low-level virtual machine (LLVM) [28] infrastructure, which provides a variety of allocation, scheduling, and binding algorithms that can be applied to HLS.

Berkeley Design Technology, Inc. (BDTI) [29] offers HLS tool certification program which evaluates the capabilities of HLS tools in terms of quality of results and usability. However, the evaluation workload focuses exclusively on the digital signal processing applications. The participated HLS vendors perform a self-evaluation first and then BDTI certifies the results. In contrast, we evaluate AutoPilot using popular embedded benchmark kernels, and real-world stereo matching software. More importantly, we evaluate HLS using software not designed for HLS, and optimize ourselves (rather than allowing the tool vendor to find the best optimization), tracking design effort.

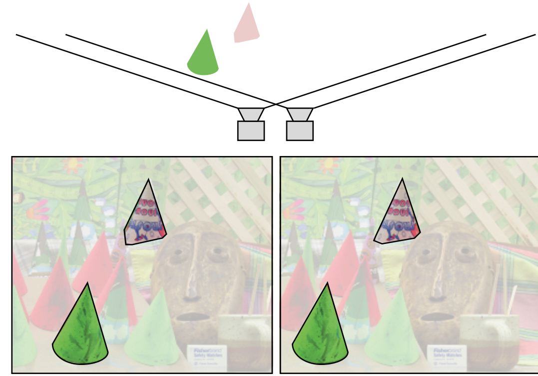


Figure 1: Example image capture for stereo matching. Two cameras physically offset capture an image of the same scene. The disparity between the objects in the left and right images infers information about object depth. One foreground and one background object are highlighted for clarity.

3. Benchmarking Applications

We evaluate HLS with AutoPilot using two categories of applications. We first use popular embedded benchmark kernels, which are widely used in various applications. However, these kernels are relatively small in code size and simple in data structure. Therefore, we also would like to evaluate HLS using real-world applications with complex code styles and data structures. We picked stereo matching for 3D image construction, which uses techniques also common for image de-noising, image retrieval, feature matching, and face recognition. In the following, we present the kernels and stereo matching algorithms in details.

3.1. Embedded Benchmark Kernel. To meet the stringent performance or energy constraints, embedded system designers often identify one or more computational intensive kernels (e.g., for signal processing, image processing, compression and decompression, cryptographic algorithms, etc.) for hardware acceleration. Although these computational-intensive kernels are often small in terms of code size, they play an important role for the overall performance and energy consumption as they account for significant execution time. Thus, efficient hardware implementation of these kernels could have significant impact on the overall system performance.

We choose a subset of applications from MiBench [30] including three common and representative cryptographic algorithms: AES, TDES, and SHA. The kernel description is shown in Table 5. Most of these kernels are implemented with a computational-intensive loop without dynamic memory allocation or complex pointer access.

Table 6: Common resolutions, frame rates, and disparity ranges (disparity range is computed with the same image sensor size (17.3 mm^2), focal length (2.5 mm), distance between cameras (12 mm) and image depth (0.5 m) for each image resolution. Parameters are based on the commercial cameras [31]. Computation scaling normalized to standard definition video).

Standard	Resolution	Maximum frame rate	Disparity range	Computation scaling
Middlebury Test Image	450 × 375	N/A	60	N/A
SD Video	640 × 480	30 fps	85	1
1080 p HD	1920 × 1080	120 fps	256	81
UHDTV	7680 × 4320	240 fps	1024	10000

3.2. Stereo Matching. Stereo matching is an extremely active area of research in computer vision, and an important underlying technology for 3D video. The depth maps generated by stereo matching are used for interpolated video views and 3D video streams. 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 [31]. 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.

Techniques employed for stereo matching algorithms include global energy minimization, filtering, and cost aggregation, which are used throughout image and video processing applications. In particular, these techniques are also employed for image de-noising, image retrieval, feature matching, and face recognition. The computational com-plexity 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 that may demand acceleration; 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). The depth map is used together with input color image(s) to produce synthesized views for 3D video applications. 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 6 shows the resolution, typical maximum frame rates, and disparity range in pixels for standard, high-definition, and next-generation high-definition video standards.

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.

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 [32–40] 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 unsuitable for HLS support. These include the following.

(i) Libraries for data structures (e.g., Standard Template Library).

(ii) OpenCV computer vision library for efficient implementations of common, basic vision algorithms.

(iii) Use of dynamic memory reallocation.

For example, as an effort to compare and evaluate many stereo matching algorithms, Scharstein et al. [32] 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 resize storage elements properly.

Stereo matching algorithms can be classified into global and local approaches. Global approaches use a complex optimization technique to simultaneously optimize the disparity matching costs for all pixels in the image. In contrast, local approaches compute the pixel matching costs individually for each pixel and concentrate on effective cost aggregation methods that use local image data as a likely source of semantic information. From the above set of algorithms, five of the twelve sets of source code can be transformed for high level synthesis compatibility. For each of the five algorithms, we perform transformations to improve suitability for HLS, but we do not redesign the algorithm implementation with HLS in mind.

We test two global approaches, Zitnick and Kanade [33], and constant-space belief propagation [34] and three local approaches, our internally developed stereo matching code, a scanline-optimized dynamic programming method, and cross-based local stereo matching [40]. Each of these algorithms uses differing underlying techniques to generate depth maps. We do not aim to judge the relative merits of different stereo matching approaches in terms of depth map accuracy. Rather, we discuss algorithm and implementation

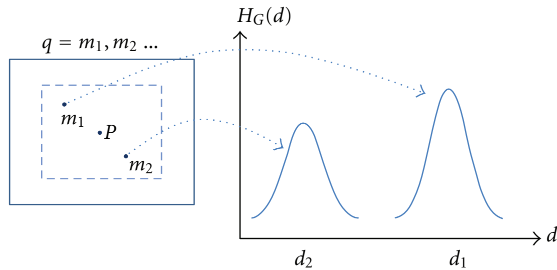


Figure 2: Calculation of the adaptive support function with Gaussian filters. Horizontal axis d represents disparity, $H_G(d)$ is the Gaussian filter amplitude. Point m_1 above has a larger $H_G(d)$ value because of closer color-space value.

details that make the algorithms more or less suitable for HLS.

3.2.1. Zitnick and Kanade (ZK). The algorithm proposed by Zitnick and Kanade [33] 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 bitwidth reductions are insufficient to reduce storage needs. Therefore, due to the infeasible storage requirements, we omit ZK from detailed synthesis results.

3.2.2. Constant Space Belief Propagation (CSBP). The constant space belief propagation algorithm [34] also generates dense depth maps based on a global energy minimization solution. In original belief propagation [35], 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.

3.2.3. 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 as a driver algorithm to study HLS capabilities. It consists of an initial estimation using absolute difference between pixel values, multiscale image downsampling [41] and the fast bilateral filtering method [42] 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 postprocessing 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.

3.2.4. Scanline Optimization (SO). Scanline optimization [32] (Software is an internally developed version of the algorithm in [32].) 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.

3.2.5. Cross-Based Local Matching (CLM). Cross-based local matching [40] 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.

4. Optimization Process

In total, we will evaluate the AutoPilot high level synthesis output using eight embedded benchmark kernels (matrix multiply, blowfish encrypt/decrypt, adpcm encode/decode, AES, TDES, SHA) and five stereo matching algorithms (BFAS, CSBP, SO without occlusion, SO with occlusion, and CLM). For all these benchmarks, we perform a five-step optimization process where applicable to convert for HLS compatibility and optimize the hardware implementation. The five optimization steps are baseline implementation, code restructuring, bitwidth 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.

4.1. Baseline—Minimum Modifications. For each benchmark, we generate a baseline implementation—the minimum code modifications so that the algorithm can be properly synthesized using AutoPilot. For all the embedded kernels, they are AutoPilot synthesizable without code modifications. However, for stereo matching algorithms, we have to perform some modifications including 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.

All the baseline implementations are AutoPilot compatible, and produce hardware designs that produce correct output. However, some of the benchmarks use significantly more BRAM resources than available in the FPGA chip we use (Xilinx Virtex-6 LX240T). In addition, the BFAS software versions also have complex computation, which causes overconstrained use of LUT, FF and DSP resources as well. For the kernels, all of them can fit in the FPGA except matrix multiplication. As for the stereo matching algorithms, 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 versus 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.

4.2. Code Restructuring. For the embedded benchmark kernels we perform various code restructuring transformations including data tiling and block merging. For matrix multiplication, we perform data tiling to the source code since the array size is too big to fit into the BRAM of FPGA. In other words, a tile of array A and a tile of array B are copied from off-chip memory to on-chip memory for multiplication and then the computed data are written back to the off-chip memory array C. For example, with an array size of 1024×1024 and a tile size of 64×64 elements the array is divided into 256 tiles. In our experiments, we explore different tile sizes (8×8, 16×16, 32×32, 64×64, and 128×128). For matrix multiplication, one of the stringent resources is on-chip BRAM. Before tiling, the estimated number of required BRAMs is 5595 for 1024×1024 arrays, which is too big to fit into the FPGA. After tiling, the designs of one tile can fit into FPGA with less BRAMs (e.g., only 48 BRAMs are required for tile size 64×64). More importantly, tiling enable us to run multiple tiles in parallel (Section 4.5).

To efficiently use memory bandwidth, we transform arrays to increase element size and match the system's bound on memory access bit-width. However, the computation is still performed on the original, smaller element. Finally, to enable a powerful AutoPilot optimization—array streaming (Section 4.3), we change parameters passed in function calls from arrays of data to pointers.

For the stereo matching algorithms, the most important code restructuring task is to partition the image into subimages 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, we also perform function merging, interchange nested loops to improve parallelism opportunity, and share internal memory buffers to reduce resource use if it is possible. 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.

4.3. Reduced Bitwidth and BRAMs. The bit-width optimization is mainly effective for stereo matching algorithms; their computation involves a lot of arrays and the array element bitwidth can be reduced for some of these arrays. The bitwidth reduction can improve both the latency and resource usage. As for the embedded kernels, we observe fewer opportunities to perform these optimizations.

Throughout all of the stereo 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 multistep process. First, for each data array in the design, we determine the operand range and redefine the array using AutoPilot's fixed-point integer (ap_fixed) variables 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.

BRAM optimizations are effective for the embedded benchmark kernels and stereo matching algorithms. For

```
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.

```
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.

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 used is much less than an 18 K BRAM. Therefore, we use the array_map pragma to map the arrays together into a single array that is stored in a single BRAM.

It is important to note that AutoPilot provides one powerful BRAM optimization—array streaming. 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 not only reduces BRAM usage but can have significant impact on performance improvement. However, it is not always feasible to apply this optimization as the data has to be written and read in FIFO order. Of all our benchmark applications, we only can apply the array stream optimization for the three cryptographic algorithms, AES, SHA, and TDES. These algorithms process streams of data in a sequential manner (e.g., block by block). More importantly, the data is written and read in FIFO order. Because of the requirement for this FIFO order, the array stream optimization is not available for the other embedded benchmark kernels or the stereo matching algorithms due to the complex data access order.

4.4. Pipelining and Loop Optimization. Through the previous three steps, we have reduced the amount of computation and memory resources. In this step, we examine the computation loops in the program and apply loop pipelining, loop merging, loop unrolling, 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.

In general, performance can be improved via loop optimizations at the expense of extra resource usage. For example, both loop unrolling and pipelining enable multiple loop iterations to be executed in parallel with more resource usage (e.g., registers or functional units), but different optimizations improve performance and use resources in different ways. More importantly, it is possible to apply multiple optimizations together, that is, loop pipelining unrolled loops. Furthermore, if the computation can be divided into different independent small parts, we can instantiate multiple computation pipelines to parallelize the computation by duplicating the logic (Section 4.5). Hence, for loop optimizations, it is important to consider resource use; loop optimizations may improve performance, but can limit flexibility of implementing multiple computation pipelines in the parallelization and resource duplication step.

```
#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.

4.5. 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 array 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, where the [FS_UNROLL] dimension of the data arrays is the added dimension that will be unrolled completely.

5. Experimental Results

In this section, we present the results of our proposed HLS optimization process on both embedded benchmark kernels and various stereo matching algorithms. For each benchmark, we use autocc (AutoPilot's C compiler) and autosim to verify correctness of the modified code. Then, we perform HLS using AutoPilot [9] 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 are obtained from ISE after placement and routing reports. After synthesis of AutoPilot's RTL, we measure the latency in clock cycles using ModelSim simulation. 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 3 GB of RAM.

5.1. Embedded Benchmark Kernels. Recall that we perform data tiling to matrix multiplication. In our experiments, we explore different tile sizes (8×8, 16×16, 32×32, 64×64, and 128×128). More importantly, tiling allows us to perform the subsequent optimizations (Sections 4.4 and 4.5). First, tiling enables us to do a complete loop unrolling for the inner most loop computation within one tile as the loop bounds are smaller compared to the original version. Second, tiling reduces the BRAM usage for one tile, thus we can parallelize the computations of multiple tiles by instantiating multiple pipelines. The exploration results of various tile sizes are shown in Figure 6. As shown, the runtime is very sensitive to the tile size and tile size 64×64 returns the best results. This is because larger tile size limits the degree of parallelization while smaller tile size limits the unrolling factor (e.g., loop iterations) within one tile. Hence, the best tile size is somewhere in between.

For embedded benchmark kernels, most of the arrays can be completely partitioned for better memory bandwidth because the arrays used by the kernels are relatively small and there are relatively few arrays in total. We observe that there are few opportunities to perform reduced bit-width optimization for the embedded benchmark kernels because the bit-width has typically been optimized already. The performance improvement is mainly from the last two optimization steps as multiple RTL blocks are allowed to execute in parallel through either loop pipelining or parallelization.

With a tile size of 64×64, there are 256 total tiles for an array size of 1024×1024. For the hardware design, we can fit 16 computation pipelines in the FPGA, therefore, we can execute 16 tiles in parallel. For each computation tile, we completely unroll the innermost loop. This solution gives us the best performance in our experiments. Blowfish and ADPCM cannot be parallelized with the resource duplication optimization due to data dependency. Blowfish and adpcm process data block by block, with data processing serialized between blocks. Hence, the improvement is from the loop optimization step only—loop unrolling and pipelining. Similarly, TDES, AES, and SHA process streams of data with required sequential dependence between neighbouring elements in the data stream. Thus, it precludes parallelization via duplication of computation pipelines. Fortunately, for these algorithms we can enable the array stream optimization which converts the arrays into FIFOs and then allows efficient fine-grained pipelining of the algorithm computation in the loop optimization step.

Figure 7 shows the speedup over original software for each kernel. As shown, AutoPilot produces high quality designs for all the kernels; 4X to 126X speedup is achieved. For matrix multiplication, we try two different input sizes, 1024 $\times$ 1024 and 512×512. For matrix multiplication, additional speedup is achieved at the parallelization step as multiple tiles computation can be run in parallel. However, for the rest of the kernels, the speedup is from loop optimization step

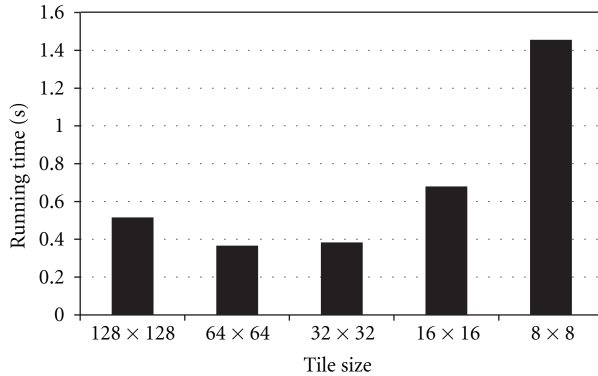


Figure 6: Comparison of different tile sizes for MM.

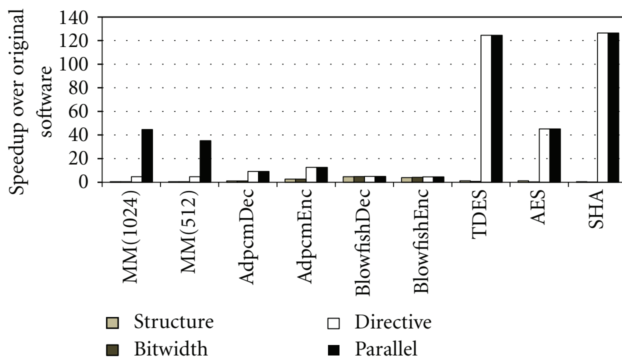


Figure 7: Speedup over original software for each kernel. Directive denotes the pipelining and loop optimization step.

only due to the sequential data dependency. We observe that TDES, AES, and SHA achieve significantly more speedup than the other kernels. Thanks to the powerful array stream optimization applied at the Reduced Bitwidth & BRAM step, we are able to do fine-grained loop and function pipelining in the loop optimization step (directive). For TDES and AES, we successfully achieve initiation interval (II) of 1. The resource utilization of the kernels are low due to their small size. The utilization for LUT is from 1% to 28%. FF utilization ranges from 1% to 20%. For all the kernels except MM, the required BRAM usage is very small because the array sizes are small and are successfully completely partitioned. MM requires 192 BRAMs for 1024×1024 size array and 160 BRAMS for 512×512 size array.

In summary, AutoPilot offers high-quality designs across a wide range of embedded benchmark kernels. Furthermore, HLS allows us to easily explore various optimizations at high level (C for AutoPilot). It only took a hardware engineer two weeks to complete the experiments for all the kernels.

5.2. Stereo Matching. Figure 9 shows the speedup over original software for each stereo matching algorithm and optimization step. Figure 8 shows the usage of various resources in combination with speedup for all the optimization steps. As shown, incremental improvement is achieved at each step.

First, for the code restructuring step, BFAS results in a 50% reduction in AutoPilot estimated LUT use, 70% fewer flip-flops, 60% fewer DSP blocks, and 95% fewer BRAMs. The other stereo matching algorithms do not employ computation resources as heavily, so there was less benefit for LUT, FF, or DSP resources, but all of them received at least 90% reduction in memory resources.

Second, for the reduced bitwidth and BRAMs step, 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 stereo matching 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.

Third, for the pipelining and loop optimization step, directive insertion is performed iteratively together with parallelization to find the best tradeoff of directives and parallelization. For the code eventually used with the next optimization step, BFAS achieved 1.5x speedup over the bitwidth and 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.

Finally, for the parallelization step, 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.

Overall, all of the stereo matching algorithms 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.

5.3. Discussion. We have demonstrated that HLS can produce high quality designs for embedded benchmark kernels: 4X to 126X speedup. In general, parallelization and loop pipelining are two effective optimizations. For the kernels without data dependency (e.g., MM), significant speedup is achieved via parallelization and resource duplication. For the benchmarks (e,g. AES, TDES, and SHA) available for array stream optimizations, significant speedup is achieved via fine grained pipelining. The benchmarked kernels are widely used in various applications, which indicate that HLS is suitable for a wide range of applications. The high speedup

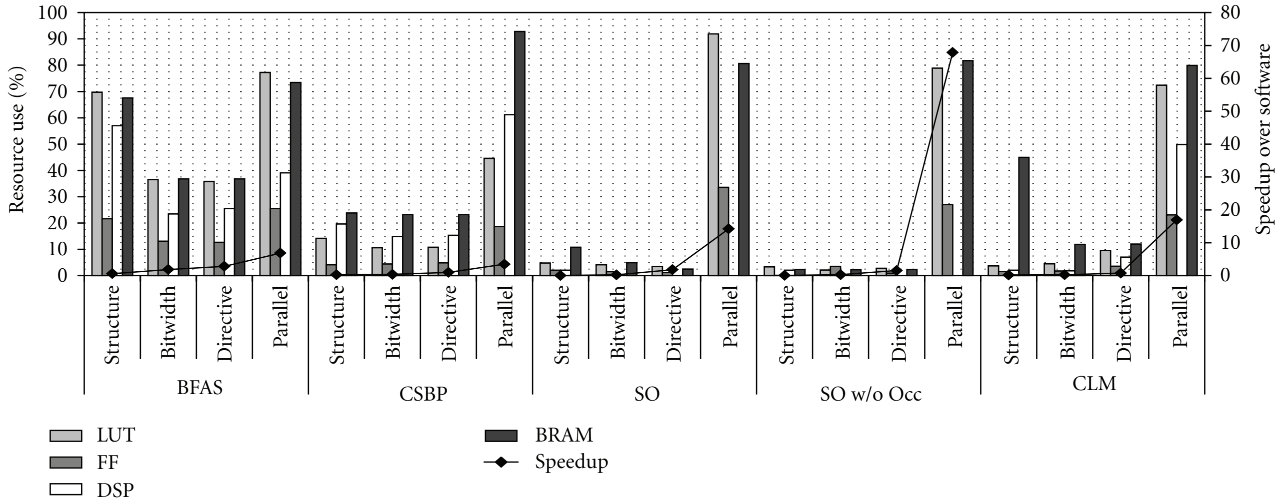


FIGURE 8: Resource use (left y-axis) and speedup (right y-axis) for each ISE synthesizable software version. Directive denotes the pipelining and loop optimization step.

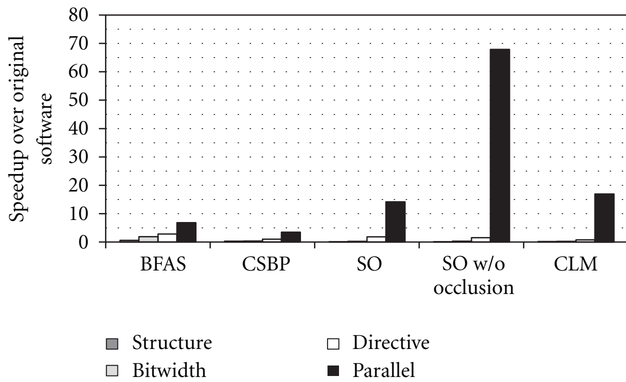


FIGURE 9: Speedup over original software for each algorithm for each ISE synthesizable optimization step.

also suggests that HLS can be used together with hardware software co-design tools for choosing the potential kernels for hardware acceleration.

We also perform a case study on stereo matching using HLS. Stereo matching contains multiple kernels related through data communication. Ideally, we want to convert the data arrays that are communicated through kernels into FIFOs and allow pipelining between kernels as mentioned earlier. However, this array stream optimization is only applicable to data written and read in FIFO order, which is not the case for any of the stereo matching algorithms. This certainly limits the speedup we achieve. Fortunately, for stereo matching, we can partition the image into sub-images that can independently compute disparity and depth information. The image partition allows us to process multiple sub-images in parallel. Thus, for the final solution, speedup of 3.5X to 67.9X is still achieved.

6. Observations and Insights

Through our experiments, we have demonstrated that HLS can achieve significant speedup for both embedded benchmark kernels and complex stereo matching algorithms. Also, because of the dual requirements of high-performance and fast develop cycle, HLS is an attractive platform for acceleration of these applications. More importantly, HLS synthesizes the kernels written at high level (C/C++ for AutoPilot). This allows us to easily explore various optimizations at high level. 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.

6.1. Productivity. It is important to evaluate the design effort required to achieve this level of speedup. Table 7 shows the development effort spent on embedded benchmark kernels and stereo matching algorithms, normalized to development time of a single hardware designer. All the experiments for embedded kernels are completed within 2 weeks as the kernels are relatively small. BFAS required longer than the others, as it was the first stereo matching algorithm implemented and some time was spent on learning the stereo matching problem. The manual stereo matching design presented in [3] 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 [1, 2]. Thus, compared to manual design, HLS achieves a fivefold reduction in design effort.

6.2. 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 the following.

(i) Convert loops using `break` or data-dependent loop bounds to static loop bounds to improve pipelining/parallelization.

(ii) Use FIFO data read & write order for dataflow optimizations.

(iii) Reduce operand size to minimize storage needs.

(iv) Use array_map to reduce storage by combining arrays.

(v) Use complete array partitioning to convert arrays to scalars.

(vi) Structure and parameterize likely parallelization locations to simplify parallelism search space.

(vii) Perfectly nest loops when possible—when not possible, consider parallelism on the innermost loop.

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

In many cases, these software constraints are easily achieved by software engineers familiar with optimization techniques. Although the goals of optimization are somewhat different because the code will correspond to hardware, the techniques are similar to typical optimization. However, these constraints sometimes conflict with "good" software engineering practices. Examples of "good" software practices include maximizing code reuse with heavily parameterized code; using variable loop bounds and early exit conditions to reduce worst-case paths in comparison to 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.

6.3. 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 completely 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 4.5 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.

Table 7: Development effort.

Algorithm	Development Effort
BFAS	5 weeks
CSBP	3 weeks
CLM	4 weeks
SO	2.5 weeks
Embedded kernels	2 weeks

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 demonstrated with the AES, TDES, and SHA benchmarks, 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.

6.4. Performance. We have demonstrated that AutoPilot can achieve high speedup over the software implementation: 4X to 126X speedup for embedded benchmark kernels and 3.5X to 67.9X speedup for stereo matching algorithms. It is important to consider the performance difference between HLS and manual hardware implementations. A manual implementation of CLM achieved speedup of ∼400X [3], similar in magnitude to other FPGA stereo matching solutions [43]. A GPU implementation of CLM [44] achieved 20X speedup over the original software, which is similar to the 17X speedup achieved in this work.

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 restructuring 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 [3], 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.

6.5. 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.

6.5.1. Usability

(i) Improved loop unrolling/pipelining for complicated loops that require temporary register and/or port duplication.

(ii) 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.

(iii) Automatic tradeoff analysis of loop pipelining and unrolling.

(iv) Automatic detection and conversion for common computation structures such as tree-based reductions.

(v) Improved robustness of dataflow transformations, streaming computation for 2D access patterns.

6.5.2. Performance Gap

(i) Detection of memory level dependence across multiple, independent loops and functions, automatic interleaving of computation between the loops and functions.

(ii) Automatic memory access reordering to allow partitioning, streaming, or improved pipelining.

(iii) Automatic temporary buffers for memory access reuse.

(iv) Iteration between array optimizations, pipelining, and parallelization for efficient search of design space.

7. 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 popular embedded benchmark kernels and several modern stereo matching software codes 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 4X to 126X speedup on the embedded benchmark kernels and 3.5X to 67.9X speedup on the stereo matching applications, 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

This paper is supported by the Advanced Digital Sciences Center (ADSC) under a grant from the Agency for Science, Technology, and Research of Singapore.

References

[1] J. Bodily, B. Nelson, Z. Wei, D.-J. Lee, and J. Chase, "Comparison study on implementing optical flow and digital communications on FPGAs and GPUs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 3, no. 2, p. 6, 2010.

[2] C. He, A. Papakonstantinou, and D. Chen, "A novel SoC architecture on FPGA for ultra fast face detection," in *Proceedings of the IEEE International Conference on Computer Design*, pp. 412–418, October 2009.

[3] L. Zhang, K. Zhang, T. S. Chang, G. Lafruit, G. K. Kuzmanov, and D. Verkest, "Real-time high-definition stereo matching on FPGA," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 55–64, 2011.

[4] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: hardware design in Haskell," in *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming*, vol. 3, pp. 174–184, September 1998.

[5] B. Bond, K. Hammil, L. Litchev, and S. Singh, "FPGA circuit synthesis of accelerator data-parallel programs," in *Proceedings of the 18th IEEE International Symposium on Field-Programmable Custom Computing Machines*, pp. 167–170, May 2010.

[6] S. S. Huang, A. Hormati, D. F. Bacon, and R. Rabbah, "Liquid metal: object-oriented programming across the hardware/software boundary," in *Proceedings of the European Conference on Object-Oriented Programming*, pp. 76–103, 2008.

[7] M. Lin, I. Lebedev, and J. Wawrzynek, "OpenRCL: low-power high-performance computing with reconfigurable devices," *Proceedings of the International Conference on Field Programmable Logic and Applications*, pp. 458–463, 2010.

[8] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W. M. W. Hwu, "FCUDA: enabling efficient compilation of CUDA kernels onto FPGAs," in *Proceedings of the 7th IEEE Symposium on Application Specific Processors*, pp. 35–42, July 2009.

[9] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong, "AutoPilot: a platform-based ESL synthesis system," in *High-Level Synthesis: From Algorithm to Digital Circuit*, P. Coussy and A. Morawiec, Eds., Springer, New York, NY, USA, 2008.

[10] Nallatech, "DIME-C," 2011, http://www.nallatech.com/Development-Tools/dime-c.html.

[11] Y Explorations, "eXCite," 2011, http://www.yxi.com/.

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

[13] "CatapultC Synthesis Datasheet".

[14] Synopsys, "Synthesizing Algorithms from MATLAB and Model-based Descriptions. Introduction to Synphony HLS White pape".

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

[16] G. Sandberg, "The Mitrion-C Development Environment for FPGAs".

[17] Impulse Accelerated Technologies, "ImpulseC Datasheet".

[18] P. Coussy, G. Corre, P. Bomel, E. Senn, and E. Martin, "High-level synthesis under I/O timing and memory constraints," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, pp. 680–683, May 2005.

[19] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: a high-lev l synthesis framework for applying parallelizing compiler transformations," in *Proceedings of the International Conference on VLSI Design*, p. 461, Los Alamitos, Calif, USA, 2003.

[20] R. Domer, A. Gerstlauer, and D. Gajski, "SpecC methodology for high-level modeling," in *Proceedings of the DATC Electronic Design Processes Workshop*, 2002.

[21] J. L. Tripp, M. B. Gokhale, and K. D. Peterson, "Trident: From high-level language to hardware circuitry," *Computer*, vol. 40, no. 3, pp. 28–37, 2007.

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

[23] A. Canis et al., "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," in *Proceedings of the Proceedings of the 19th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, p. 33, 2011.

[24] S. Sirowy, G. Stitt, and F. Vahid, "C is for circuits: capturing FPGA circuits as sequential code for portability," in *Proceedings of the 16th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 117–126, February 2008.

[25] G. Stitt, F. Vahid, and W. Najjar, "A code refinement methodology for performance- improved synthesis from C," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 716–723, November 2006.

[26] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis," *Journal of Information Processing*, vol. 17, pp. 242–254, 2009.

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

[28] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 75–86, March 2004.

[29] "BDTI High-Level Synthesis Tool Certification Program Results," http://www.bdti.com/Resources/BenchmarkResults/HLSTCP.

[30] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: a free, commercially representative embedded benchmark suite," in *Proceedings of the IEEE International Workshop on Workload Characterization*, pp. 3–14, 2001.

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

[32] D. Scharstein, R. Szeliski, and R. Zabih, "A taxonomy and evaluation of dense two-frame stereo correspondence algorithms," in *Proceedings of the IEEE Workshop on Stereo and Multi-Baseline Vision*, p. 0131, 2001.

[33] C. Lawrence Zitnick and T. Kanade, "A cooperative algorithm for stereo matching and occlusion detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 7, pp. 675–684, 2000.

[34] Q. Yang, L. Wang, and N. Ahuja, "A constant-space belief propagation algorithm for stereo matching," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 1458–1465, June 2010.

[35] T. Meltzer, C. Yanover, and Y. Weiss, "Globally optimal solutions for energy minimization in stereo vision using reweighted belief propagation," in *Proceedings of the 10th IEEE International Conference on Computer Vision*, vol. 1, pp. 428–435, October 2005.

[36] B. M. Smith, L. Zhang, and H. Jin, "Stereo matching with nonparametric smoothness priors in feature space," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pp. 485–492, June 2009.

[37] E. Tola, V. Lepetit, and P. Fua, "DAISY: An efficient dense descriptor applied to wide-baseline stereo," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 32, no. 5, Article ID 4815264, pp. 815–830, 2010.

[38] A. S. Ogale and Y. Aloimonos, "Shape and the stereo correspondence problem," *International Journal of Computer Vision*, vol. 65, no. 3, pp. 147–162, 2005.

[39] K. J. Yoon and I. S. Kweon, "Adaptive support-weight approach for correspondence search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 28, no. 4, pp. 650–656, 2006.

[40] K. Zhang, J. Lu, and G. Lafruit, "Cross-based local stereo matching using orthogonal integral images," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 19, no. 7, Article ID 4811952, pp. 1073–1079, 2009.

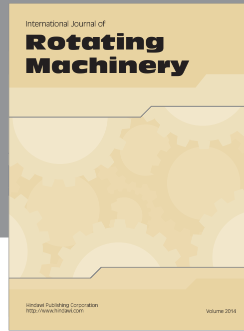
[41] D. Min and K. Sohn, "Cost aggregation and occlusion handling with WLS in stereo matching," *IEEE Transactions on Image Processing*, vol. 17, no. 8, pp. 1431–1442, 2008.

[42] S. Paris and F. Durand, "A fast approximation of the bilateral filter using a signal processing approach," in *Proeedings of the 9th European Conference on Computer Vision*, 2006.

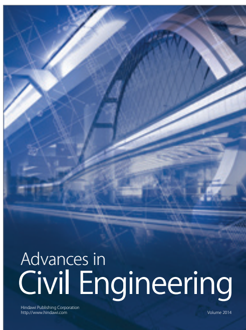
[43] S. Jin, J. Cho, X. D. Pham et al., "FPGA design and implementation of a real-time stereo vision system," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 20, no. 1, pp. 15–26, 2010.

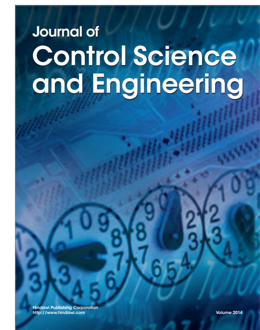
[44] K. Zhang, J. Lu, G. Lafruit, R. Lauwereins, and L. Van Gool, "Real-time accurate stereo with bitwise fast voting on CUDA," in *Proceedings of the 12th International Conference on Computer Vision Workshops*, pp. 794–800, October 2009.

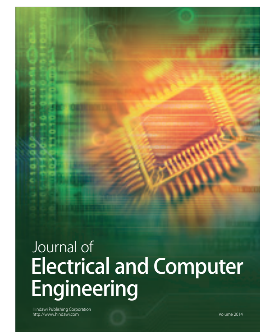
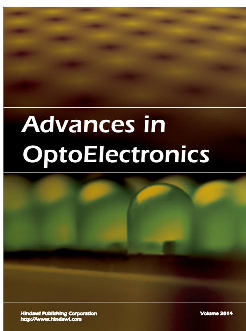
Journal of
Engineering

The Scientific
World Journal

International Journal of
Rotating
Machinery

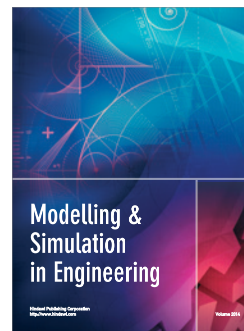
Journal of
Sensors

International Journal of
Distributed
Sensor Networks

Advances in
Civil Engineering

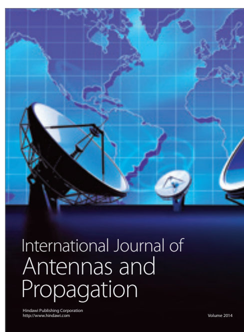
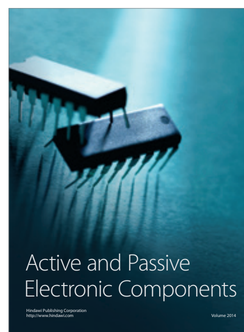
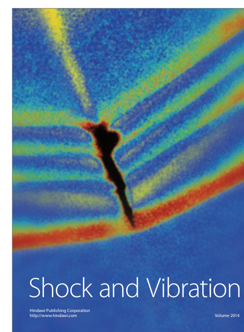
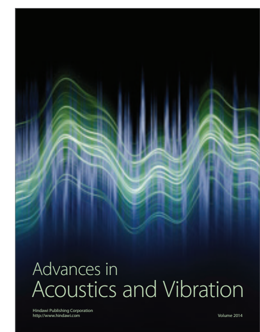
Hindawi
Submit your manuscripts at
http://www.hindawi.com

Journal of
Control Science
and Engineering

Journal of
Robotics

Journal of
Electrical and Computer
Engineering

Advances in
OptoElectronics

VLSI Design

International Journal of
Navigation and
Observation

Modelling &
Simulation
in Engineering

International Journal of
Aerospace
Engineering

International Journal of
Chemical Engineering

International Journal of
Antennas and
Propagation

Active and Passive
Electronic Components

Shock and Vibration

Advances in
Acoustics and Vibration