A Study of High-Level Synthesis: Promises and Challenges

Kyle Rupnow[1], Yun Liang[1], Yinan Li[1], Deming Chen[2]
[1] Advanced Digital Sciences Center
[2] University of Illinois at Urbana-Champaign
* Email: {k.rupnow,eric.liang,yinan.li}@adsc.com.sg, dchen@illinois.edu

A wide variety of application domains such as networking, computer vision, and cryptography target FPGA platforms to meet computation demand and energy consumption constraints. However, design effort for FPGA implementations in hardware description languages (HDLs) remains high – often an order of magnitude larger than design effort using high level languages (HLLs). Instead of development in HDLs, high level synthesis (HLS) tools generate hardware implementations from algorithm descriptions in HLLs such as C/C++/SystemC. HLS tools promise reduced design effort and hardware development without the detailed knowledge of the implementation platform. In this paper, we study AutoPilot, a state-of-the-art HLS tool, and examine the suitability of using HLS for a variety of application domains. Based on our study of application code not originally written for HLS, 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 examined applications, we demonstrate speedup from 4X to over 126X, with a five-fold reduction in design effort vs. manual design in HDLs.

1. Introduction

In many application domains, FPGAs are already the preferred implementation platform – the combination of performance and power/energy efficiency over CPUs makes FPGAs an attractive platform for a wide variety of applications in domains such as networking, computer vision, and cryptography. However, design effort for FPGAs remains a significant barrier to effectively accelerate applications. FPGA-based application design time is often an order of magnitude more than pure software implementations [1]. Furthermore, hardware design expertise is rare compared to software expertise, limiting FPGA design production to the comparatively small set of experienced hardware designers.

High-level synthesis (HLS) targets this problem: tools synthesize algorithm descriptions written in a HLL to register transfer level (RTL) implementations. HLL implementations can be written faster and more concisely, reducing susceptibility to programmer error, and improving code readability and reusability. Thus, HLS promises to allow software engineers to effectively produce high performance, energy efficient hardware.

There has been significant recent development in HLS tools, producing numerous industry and academia HLS tools that generate RTL from HLLs such as C, C++, SystemC, CUDA, OpenCL, Matlab, and Haskell, in addition to specialized languages and language subsets. Martin and Smith describe the evolution of HLS tools in terms of user experience and commercial viability [2]. An important aspect of the user experience is the two distinct markets HLS targets (and their expectations):

1. Experienced HW designers – parity performance with significant design effort reduction
2. Non-HW designers – 'good enough' performance without required hardware knowledge

However, there is little systematic study of using/evaluating HLS tools and the challenges of taking advantage of the HLS features. Therefore, we use AutoPilot for an HLS case study and investigate the techniques required to produce high-quality HW from software source not originally intended for HLS to evaluate the usability and suitability of HLS among software engineers. For this analysis, we examine embedded benchmarks from the MiBench [3], several stereo matching (computer vision) algorithms [4], and three cryptographic algorithms, AES, TDES and SHA.

2. AutoPilot High Level Synthesis

AutoPilot is a commercial HLS tool developed by AutoESL[5][1] that supports C/C++ and SystemC as input languages. AutoPilot supports integer and floating point data types as well as fixed point, reduced bitwidth variables. AutoPilot synthesis employs a wide range of standard compiler optimizations followed by directive-guided synthesis, which transforms computation blocks, data arrays and communications between them. By default, each function call produces a separate level of RTL hierarchy, and each data array is mapped to BRAMs. Directives guide function optimization, interleaving of computation, duplication of resources, and data array mapping to BRAMs among others. For a detailed description of all available AutoPilot directives, see [6], [7].

3. Applications

For the MiBench benchmark suite, we select matrix multiplication, adpcm encode/decode and blowfish encode/decode as simple, but representative embedded kernels for signals processing and communications applications. Stereo matching is an important computer

[1] AutoESL was acquired by Xilinx in January, 2011

978-1-61284-193-9/11/$26.00 ©2011 IEEE

vision problem [4] that uses techniques also employed for image de-noising, feature matching and face recognition. We select five stereo matching algorithms not originally written for HLS; for more details, see [7]. Finally, we select AES, TDES and SHA as three common and representative cryptographic algorithms that are employed throughout embedded systems.

4. HLS Optimization Flow for Software

Optimizing benchmarks for HLS involves two main goals: 1. reduce resource use (which can also improve resource utilization), and 2. increase pipelining and/or parallelism to take full advantage of available FPGA resources. To meet these goals, we perform a five-step optimization process: baseline implementation; code restructuring; data storage; pipelining; and parallelization via resource duplication. For some codes, with many independent data partitions, parallelization is critical for application performance, whereas other codes such as cryptographic algorithms require heavily pipelined hardware to achieve high performance.

4.1 Baseline implementation

The baseline implementation converts dynamic memory allocations to static declarations, memset/memcpy function calls to for loops, and arbitrary pointer indirections to static pointers to make the original source compatible with AutoPilot.

4.2 Code restructuring

Code restructuring includes manual data partitioning, function merging, loop merging, and interchanging of nested loops in order to reduce resource use. Although AutoPilot has directives that can perform some of these functions, manual restructuring is more flexible.

4.3 Data storage

We perform datatype conversion and use data array directives to optimize data storage efficiency and bandwidth. First, we convert floating-point computation to fixed-point or integer when possible. Then, we use AutoPilot's ap_int and ap_fixed datatypes to specify reduced bitwidth. Note, that AutoPilot can automatically reduce bitwidth, but not for data arrays that also use data array directives, so we manually change bitwidth using AutoPilot's datatypes. Then, we apply data array directives – small arrays are completely partitioned (converted to registers), other arrays use partitioning, mapping, or reshaping so that BRAM storage is used efficiently and the BRAM access bandwidth is not a computation bottleneck. Complex computation functions with limited input range are converted to lookup tables to reduce computation resource use and latency.

4.4 Pipelining & loop optimizations

After resource use optimization, we now pipeline the design to improve performance of the datapath. We target an initiation interval of 1 cycle, but may require more cycles if there are multiple reads and/or writes to the same BRAM in the inner loop. We also use loop flattening to reduce loop-traversal overhead, and complete loop unrolling with expression balancing to create tree-based computation of small loops.

4.5 Parallelization

If the algorithm can process multiple data items in parallel, we duplicate the entire pipelined datapath to use as much of the FPGA resources as possible. This step is critical for performance in the stereo matching algorithms, but not used for the MiBench embedded kernels or the cryptographic algorithms.

5. Experiments

For each benchmark application, we use autocc (AutoPilot's C compiler) and autosim to verify correctness of the modified code. Then, we perform HLS using AutoPilot version 2010.A.4 targeting a Xilinx Virtex-6 LX240T. 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 post-placement and routing reports. Using the post synthesis design, we measure the latency in clock cycles using ModelSim simulation – hardware latency in seconds is computed by multiplying the clock period by the measured clock cycles, and speedup is the ratio of hardware latency to the (unmodified) software latency. Software latency is measured on an Intel i5 2.67 GHz CPU with 3GB of RAM.

5.1 Results

The embedded kernels (Figure 1) are small and relatively simple sections of source code – therefore, there is no need to perform conversions for a baseline implementation of the algorithms, only the matrix multiplication kernel requires code restructuring, and only one kernel uses data array directives. For all but the matrix multiplication algorithms, the primary performance impact is due to effective pipelining – matrix multiplication achieves significant additional benefit from duplication of computation pipelines.

Each of the stereo matching algorithms (Figure 2) requires significant restructuring to partition the input image sufficiently to allow the data structures to fit within the FPGA. These designs use many temporary arrays and complex computation, requiring heavy use of bitwidth modification, and array directives to minimize resource use. The parallel stage has the greatest impact on overall performance; in total we achieve speedup from 3.5x to 67.9x, where most of the speedup is due to duplication of computation pipelines. For more details on these algorithms and the optimization steps, see [7].

The cryptographic algorithms, AES, SHA and TDES (Figure 3) are small and relatively simple. However, these codes represent a different test case than prior codes. In both stereo matching and the embedded kernels, the kernel computation is easily sub-divided and parallelized – there is large computation demand per element, but little sequential dependence. The cryptographic algorithms, however, process streams of data with required sequential dependence between neighboring elements in the data stream. In particular, these algorithms demonstrate the value of array streaming together with dataflow execution. For these algorithms, the sequential dependence precludes

parallelization via duplication of computation pipelines. Therefore, all speedup obtained is through detailed fine-grained pipelining of the algorithm computation, which is achieved through dataflow partitioning and array streaming, and the pipeline directives.

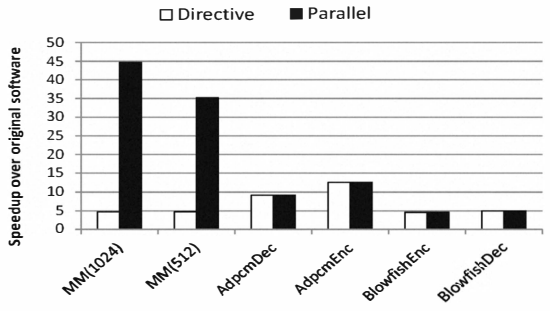

Figure 1 – Total speedup over software for the MiBench embedded kernels after each optimization step

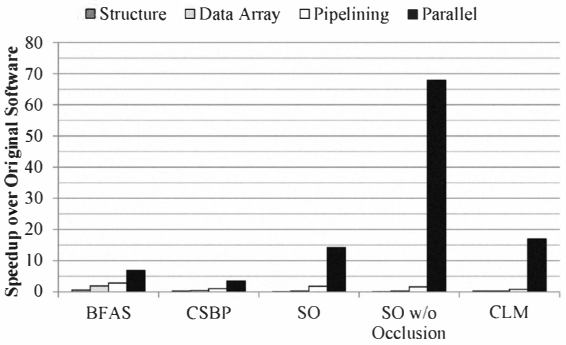

Figure 2 – Total speedup over software for five stereo matching algorithms after each optimization step

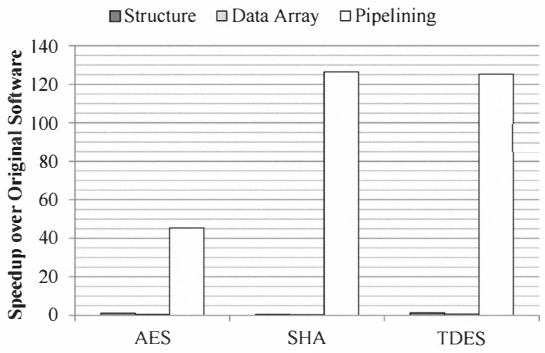

Figure 3 – Total speedup over software for AES, SHA, and TDES after each optimization step

5.2 Observations

HLS is an attractive tool for acceleration of a wide range of applications for high performance, fast development cycle, and development abstract for hardware platform knowledge. In the prior section, we demonstrated that a designer can achieve good performance through application of AutoPilot's directives. Now, we will reflect on the quality of HLS in terms of productivity, software constraints, usability, and performance of the tools.

5.2.1 Productivity

It is important to evaluate design effort required to achieve the speedup demonstrated. For all of the design effort data, we quote time in terms of a single engineer's effort. The MiBench embedded kernels required one week of design effort, as they are extremely simple and the process of converting and optimizing each kernel is quite straightforward. For the stereo matching algorithms, design effort varied more, depending on the complexity of the algorithms. The first algorithm (BFAS in Figure 2) converted required some additional effort to learn about the stereo matching problem, and required 5 total weeks of effort. The remaining algorithms required 2.5 weeks (SO), 3 weeks (CSBP), and 4 weeks (CLM), which roughly corresponds to relative complexity. Finally, the cryptographic algorithms required 1 week each to optimize. Although the cryptographic kernels are small, structure and data array modifications were critical in order to expose parallelism properly for the pipelining, array streaming and dataflow directives to work effectively. In comparison, typical design times are as much as an order of magnitude larger. The manual stereo matching design of the CLM algorithm [8] 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, and they achieved ~400x speedup as compared to the 16.9x speedup we achieve. The difference is primarily due to fine-grained interleaving of computation and communication.

5.2.2 Software constraints

HLS tools require statically declared memory, which also precludes the use of many standard libraries such as STL or OpenCV. Furthermore, we provide 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 to enable dataflow optimizations
- Reduce operand size to minimize storage needs
- Use array_map to reduce storage by combining arrays
- 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 (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. for complete unrolling a small to medium size inner loop may be best, for pipelining the largest loop may be best, etc.).

Typically, these software implementation constraints are easily achieved by software engineers familiar with optimization techniques. Although optimization goals are somewhat different to generate hardware, the techniques are similar to typical optimization. However, techniques to parallelize, pipeline, and interleave computation and communication 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. 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.

5.2.3 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 sufficient parallelism quite easily. Directives such as dataflow execution together with array streaming can quite easily allow fine-grained interleaving of computation and communication. 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, 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's optimizations are powerful, it is sometimes difficult to apply to code not designed for it. As demonstrated with the cryptographic algorithms, array streaming is extremely powerful, but because it can only be applied to arrays with FIFO read and write access orders, we cannot use array streaming in the stereo matching codes.

5.2.4 Future directions

Together, this study points to potential future enhancement in HLS tool development. In terms defined by Martin and Smith [2], the current 3rd generation (and burgeoning 4th generation) tools are meeting the constraints of the expected user groups, but there is still significant room for improvement to expand HLS use. The two general directions of future study also correspond roughly to the two different target markets for HLS: performance improvements to make HLS an attractive replacement (and design effort reduction) for experienced HW designs, and usability improvements to further reduce the difficulty for non-HW designers to use HLS to produce efficient HW designs. 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) Performance Gap

- Detection of memory level dependence across multiple, independent loops and functions, automatic interleaving of computation with communication
- Automatic memory access re-ordering to allow partitioning, streaming, or improved pipelining
- Automatic creation of temporary buffers to improve memory access re-use

2) Usability

- Improved loop unrolling/pipelining for complicated loops that require temporary register
- Support for port duplication directives that automatically duplicate BRAM resources to increase bandwidth to import data elements
- Automatic tradeoff analysis of loop pipelining and unrolling
- Improved robustness of dataflow transformations, streaming computation for 2D access patterns

6. 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 embedded benchmark kernels, stereo matching software codes and common cryptographic algorithms all not originally written for HLS to demonstrate the performance and design effort of using HLS to produce hardware. We present an unbiased study of the progress of HLS tools, guidelines for algorithm implementation, and an effective optimization process. We demonstrate that with short development time, HLS based design can achieve up to 126X speedup in the current generation of tools, and suggest areas of future study to improve future HLS tools.

7. References

[1] 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.

[2] G. Martin and G. Smith, "High-Level Synthesis: Past, Present, and Future," *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 18-25, Aug. 2009.

[3] M. R. Guthaus, et al., "MiBench: A free, commercially representative embedded benchmark suite," in *2001 IEEE International Workshop on Workload Characterization, 2001. WWC-4*, 2001, pp. 3- 14.

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

[5] 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.

[6] AutoESL Inc., "AutoPilot Reference Manual." .

[7] K. Rupnow, et al., "High Level Synthesis of Stereo Matching: Productivity, Performance, and Software Constraints," FPT, 2011.

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