Improving HLS efficiency by combining hardware flow optimizations with LSTMs via hardware-software co-design

Harisankar Sadasivan, Fan Lai, Hasan Al Muraf, Sheng Chong
Department of Computer Science and Engineering, University of Michigan Ann Arbor, MI 48109, USA
Email: hariss@umich.edu (H.S), fanlai@umich.edu (F.L), hasanal@umich.edu (H.M), chonshen@umich.edu (S.C)

Abstract

The translation of C programs to Verilog can present significant challenges for programmers aiming to synthesize hardware. To address these challenges, several High-Level Synthesis (HLS) tools have been developed, focusing on minimizing human efforts. In this research paper, we conduct an extensive study to gain a comprehensive understanding of the HLS landscape, with a particular emphasis on evaluating the performance of LegUp, a popular open-source HLS tool. Moreover, we explore the possibilities of hardware-specific optimizations within the HLS flow to enhance area utilization and run-time efficiency. To achieve this, we augment LegUp's static range analysis with dynamic range analysis, resulting in reduced FPGA area usage. Additionally, we demonstrate the benefits of combining bit-width optimization and operation chaining, leading to a decrease in the number of FPGA slices required. Expanding our investigation, we delve into hardware-software co-design, showcasing the positive impact of cache miss event-based profiling and cache pre-fetching on overall system performance. Through this research, we aim to provide valuable insights into the capabilities and potential enhancements of LegUp while offering novel approaches for optimizing hardware synthesis and co-design processes.

Introduction

Field Programmable Gate Arrays (FPGAs) are increasingly being utilized for the development of customized application-specific accelerators [1, 2, 3]. While FPGAs offer fine-grained parallelism that can outperform general-purpose CPUs, they are not yet the preferred choice for algorithm developers seeking high performance platforms.

One of the primary challenges in FPGA adoption lies in the translation of programs from High-Level Languages (HLLs) such as C or Python to Hardware Description Languages (HDLs) suitable for hardware synthesis. Achieving precise control over timing sequences while maintaining programmability at the high level is difficult. Previous studies

[14,15,16] have focused on designing intelligent tools for automating this translation process, known as High-Level

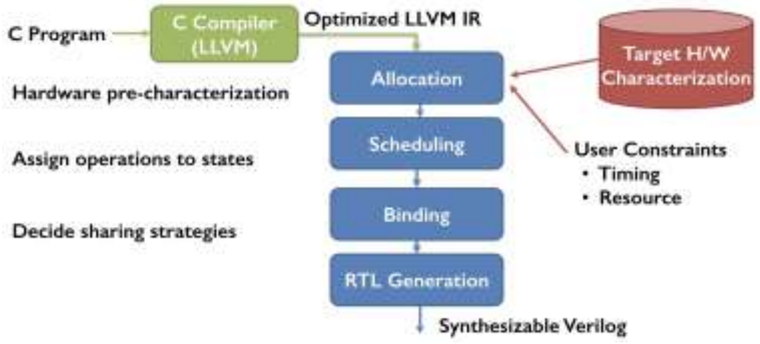

Figure 1: High-Level Synthesis Flow

focused on High Level Synthesis (HLS) which refers to translation from HLLs to HDLs. These works primarily aim to reduce turnaround time and enhance efficiency.

In this context, prior focus [6,7,8] is directed towards intelligent program translation from HLLs to HDLs, coupled with hardware-specific optimizations using commercial HLS tools such as Xilinx Vivado HLS. However, most popular HLS tools, including Xilinx Vivado HLS [15], are closed sourced and are only commercially available, restricting users from implementing custom LLVM passes to optimize the HLS workflow and observe hardware performance improvements. Among these state-of-the-art HLS tools, LegUp [6] stands out due to its applicability and flexibility. Notably, LegUp utilizes the LLVM infrastructure, allowing for the addition of custom back-end compiler passes.

However, the flexibility of LegUp comes at the cost of abstracting away most of the optimization details, while the underlying optimizations in LegUp remain incomplete. In this research paper, we first evaluate the performance of LegUp using various benchmarks. Based on these evaluations, we incorporate hardware optimizations inspired by Shang [14], another open-source HLS tool, into LegUp. Our evaluation results demonstrate superior performance compared to the baseline LegUp.

Advancing into the realm of hardware-software codesign, prior studies have aimed to improve overall performance by minimizing communication between the CPU and FPGA through low-speed buses. To this end, we analyze memory access traces using PinTool [17] and explore the potential benefits of machine learning heuristics for memory prefetching. Building upon these empirical findings, we present the following contributions:

- We conduct an evaluation of LegUp using diverse benchmarks and provide valuable insights for software-hardware co-design.
- We implement optimizations in LegUp and evaluate their performance on a real FPGA platform.

- We analyze memory access traces and employ LSTM techniques to demonstrate the advantages of prefetching, which can enhance both software execution and hardware operation.

Flow Optimizations

Bitwidth optimization

The C programming language defines a standard set of data types with fixed lengths, typically 8, 16, 32, or 64 bits. This coarse quantization inherently results in the under-utilization of bits. Traditionally, there has been limited incentive for programmers to optimize variable bit widths, as processor data-paths are constrained to fixed widths. However, when a program is synthesized into hardware using High-Level Synthesis (HLS), optimizing the bit-level representation directly translates to circuit area and power reductions [8]. In the case of LegUp 4.0, it solely supports static range analysis, claiming a maximum slice usage reduction of up to 30% on FPGA platforms.

Static range analysis relies entirely on compile-time information, such as constants in the code. Bit-width reductions achieved through static range analysis preserve program correctness, ensuring accurate execution for all input datasets. Conversely, dynamic range analysis infers ranges based on runtime information specific to a particular input dataset, enabling more significant bit-width reductions. However, it comes with the caveat that the bit-width-reduced program is no longer guaranteed to work correctly for all inputs. We propose that combining static and dynamic range analysis within LegUp 4.0 would yield notable improvements in hardware area utilization.

Operation Chaining

Operation chaining is a technique defined by LegUp 4.0 that diminishes the cycles in a design by enabling the result of an operation to be utilized in the same cycle [8]. Here, the time consumed for both the operations is ensured to be less than the critical path so that the clock frequency is not affected.

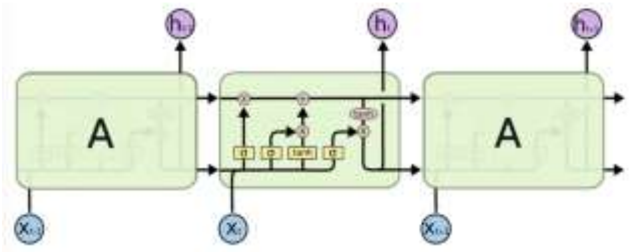

Figure 2: Structure of a LSTM network [9]

We propose that utilizing the combination of these two optimizations- bit-width optimizations and operation chaining- would yield us significant observable results on a synthesized hardware.

Memory Prefetching

Although FPGAs have been widely used in real production, they often come with limited communication resources, especially when they connect to other components with a low-speed bus. Noting the high cache miss rates from our LegUp profile and the hardware constraints in our FPGA setting, we make some efforts to explore the benefits of prefetching with some commonly practiced prefetch algorithms, a solution we designed based on spatial and temporal locality [18, 19], as well as a LSTM-based machine learning heuristic [20].

Prefetching for locality

A common practice of prefetching is to fetch adjacent next-K [18] memory addresses along with the current address. For example, when page A is being accessed, the prefetcher assumes that page A+1, A+2,..., A+K are also going to be accessed in the near future and brings those pages into the page table proactively. This approach is very helpful for accessing long regular streams. However, if memory addresses are not accessed sequentially, then this next-K page prefetching approach is not very beneficial.
Another common trend in prefetching is to correlate events in the memory access history [20] (usually by PC or Load Address), which relies on data access history repeating itself. However, spatial locality can be addressed by temporal correlation-based algorithms only if the sequential accesses are being frequently repeated. If non-repetitive sequential accesses cover a significant portion of the memory access trace, then this temporal prefetching technique shows low accuracy.

LSTM Pre-fetcher

In LSTM based prefetchers [20,21,22], the sequence of accessed pages over a period is fed to the LSTM optimizer and the weights and biases are trained as shown in Figure 2. Once the training is complete, it is tested using a dataset which is a holdout sample of prefetch access sequences. The algorithm takes in n input number of elements and predicts the next page to be accessed. The corresponding page is then stored in the prefetch queue if not present, or its access time is updated otherwise. If the queue size is full, the least recently used (LRU) element is switched out of the queue. In this way, the LSTM prefetcher attempts to keep the most likely pages in the prefetch queue for quicker access. In this paper, our LSTM prefetcher was trained on a sample of page access sequences generated using the tools discussed. Due to the huge size of data generated, only a representative set of sequences was used for training and testing the LSTM model. The time it takes for the model to train varies with input size. At the same time, testing returns a prediction for every sequence of length equal to number-of-inputs in the test sequence file.

Evaluation Setup

Pure hardware flow

Pure hardware flow means that the entire HLL code is converted to Verilog and synthesized to hardware on an FPGA. There is no soft-processor or CPU involved. We chose this operating mode mainly based on two observations. First, the CPU-FPGA hybrid system is still a concept product, there are no such boards we can get from market. Second, the soft processor in the original implementation of LegUp stalls itself whenever the hardware accelerator is running. In that case, simulation for hybrid mode seems inappropriate as implementing more functions on FPGA will always improve parallelism. In our project, we used a Spartan-7 FPGA on XiIinx Arty S7-50 development board.
With our custom pass for dynamic range analysis added, LegUp generated Verilog Register Transfer Level (RTL) code with 4 different HLS configurations:

- Operation chaining enabled
- Bit-width optimization enabled.
- Bit-width optimization and chaining enabled.
- None of the above enabled.

Prefetching for co-design

Workloads: To get an idea about how memory addresses are being accessed by data-intensive applications, we run three unmodified memory-intensive applications using industrial benchmarks and production workloads.

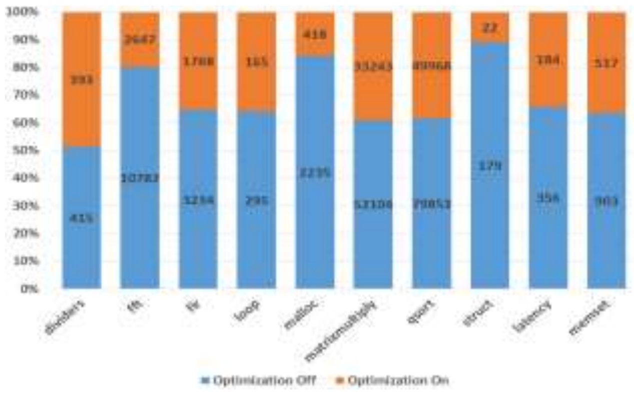

Figure 3: Running cycles for benchmarks with HLS optimization 'ON'and 'OFF'

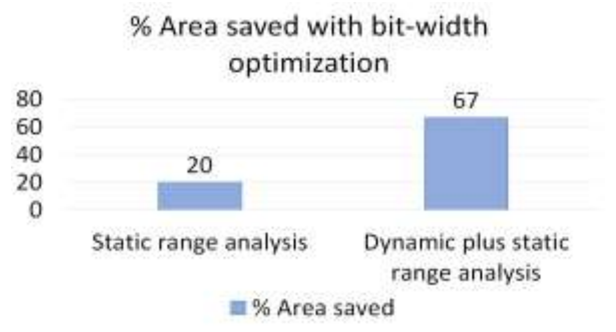

Figure 4: Percentage area saved on FPGA with static and dynamic range analysis for bit-width optimization

- TunkRank algorithm over Twitter dataset on PowerGraph [23];
- Facebook workloads (ETC) [24] on Memcached;
- PC-C benchmark on VoltDB [25];

Evaluation Results

Pure hardware flow optimization

We ran a set of benchmarks provided by LegUp on a Verilog simulator. We get the run cycles for both the cases where we disabled and enabled HLS optimizations, as shown in Figure. 3. From Figure. 3, we can see that with the HLS optimizations 'ON', the simulation cycles for most benchmarks are significantly decreased. However, the divider benchmark is seen to have less improvement even after optimization because the LegUp-generated RTL for Verilog divider module is not an optimum design. This is the downside of automating C to RTL design translation.

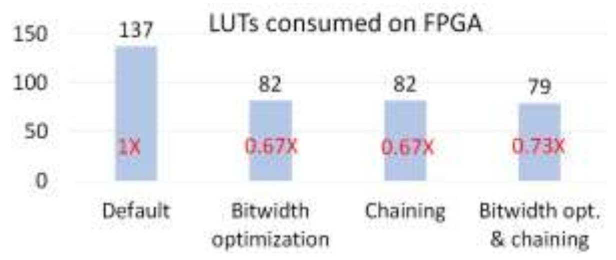

Figure 5: LUT consumption on FPGA for default HLS configuration, with bit-width optimization (static and dynamic range analysis) enabled, with chaining enabled and a combination of bit-wdith optimization and chaining enabled, in this order from left to right.

From Figure. 4, it is evident that with dynamic range analysis functionality added to LegUp 4.0, the savings on area went up from 20% to 67%. This is not surprising as static range analysis can only save so little on area. On adding dynamic analysis on run-time, we compromise the correctness of the program for various inputs to gain improvement in terms of area.

From 5, we can see that the LUT usage (area) on FPGA is reduced by 67% on using bit-width optimization or chaining but is improved by as high as 73% on combining bit-width optimization (dynamic and static range analysis) with operation chaining. This significant improvement when we club both optimizations is because bit-width optimization frees up LUTs and chaining enables related operations to be scheduled together on the same cycle. It is important to note that we are constrained by an upper bound on the maximum LUTs that can be used (1%) by the synthesized code because our constraint file fed to the allocation step of LegUp originally does not support our Spartan-7 FPGA. A video comparing the run-times on FPGA is uploaded here: https://youtu.be/wt0gY-ss48w. Here, we can see that with both chaining and bitwidth optimization enabled (static and dynamic range analysis), we save 60% of run-time for our selected loop benchmark. This is something which we could evaluate on all benchmarks in the future.

Pre-fetching for co-design

We used PIN [11], a runtime binary instrumentation tool to capture the memory and instruction pointer traces for each of the above mentioned applications. For each of the applications, at the very beginning of the run, an initialization period is required to build network connection and then read from disk to generate the workload. After the initialization step, the applications perform computation over the workloads. During the preparation and computation time, the temporal prefetching algorithm gives better results than the spatial one. But, during the large read/write operation, the performance of spatial prefetcher tends to improve while the temporal one shows a negative trend.

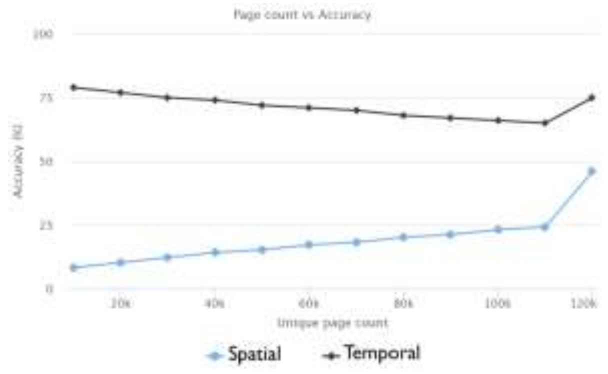

Figure 6: Prefetch accuracy over number of pages being accessed for Facebook ETC workload (queue size = 512)

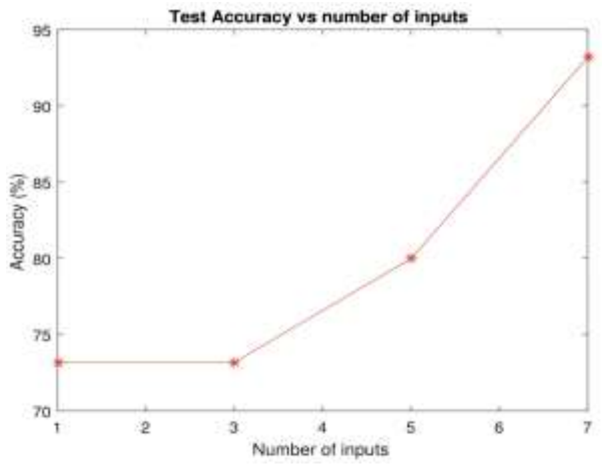

Figure 7: Trend of test accuracy with number of inputs

Figure 6 shows the above-mentioned scenario for MemCached trace. Over time, accuracy of the spatial prefetching techniques rises from 8% to 46% and while the accuracy of temporal prefetcher drops from around 75%to 65%. For VoltDB and PowerGraph traces, the accuracy of the spatial prefetcher is around 54% and 65% respectively. One may note that for temporal prefetchers the accuracy is around 72% and 78% respectively.
Figure 7 reports the evaluation results of LSTM, where testing was carried out using prefetch queues of various lengths such as 1, 2, 4, 8, 16, 32 and longer. In this evaluation, we used a training page sequence of length of 900,000 and tested it with 500 cases from a page sequence of length 100,000. It was found that the prediction quality of the LSTM predictor was high enough that the length of the prefetch queue became insignificant for testing accuracy. In most cases, the predictor is able to accurately predict the next page to be accessed, which means that having a queue larger than 1 does not lead to any performance enhancements. In cases where the sequence deviates from the 'memory' held by the LSTM model and leads to incorrect predictions, the required page was last accessed long enough in the past that it no longer remains in the queue.

Discussions and Takeaways

LegUp supports only hardware-only flow on many FPGAs. There was no pre-characterization file recommended from LegUp as the input to allocations in the HLS tool chain flow for our Xilinx Spartan-7 and hence, by default, LegUp does not generate RTLs which are ready-to-be synthesized on Arty S7.

Pure HW-flow optimizations on LegUp are far from complete. Although LegUp can provide user-defined optimizations during its synthesis, most of these optimizations greatly depend on the users' experience, which makes the co-design intractable. For example, users may find it difficult to specify which function to offload and when to offload. In this context, we can expect an Integer Linear Programming solution for this problem.

Machine learning is promising, but presents some overhead in training and deployment. Although we have shown an encouraging trend to applying machine learning-based heuristics on memory prefetching, they often introduce noticeable performance overhead. For example, the training accuracy greatly depends on the dataset, and varies with different features. Meanwhile, a simple prediction inference can take O(100) ms, which make the real-time interaction with system become impossible. One future direction is to explore convolution based networks to reduce inference times.

Conclusion

LegUp, an open-source LLVM based HLS tool, has been widely recognized to achieve desirable performance with LLVM-IR optimizations and its inherent allocation, scheduling and binding stages. In this work, we show that the area usage and the execution time for a pure hardware flow on an FPGA may be further optimized by orchestrating a bit-width optimization scheme in dynamic range analysis within LegUp. Also, it is observed that the bit-width optimization makes it possible for operation chaining to further improve the area usage. We learn that the hardware-software co-design can be a future direction when we consider cache miss rates, which can be improved by heading toward machine learning based approaches for memory prefetching. However, a desirable machine learning-based solution should carefully achieve the trade-off between performance and potential overhead.

References

[1] E. Fernandez, W. Najjar, and S. Lonardi. String matching in hardware using the fm-index. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 218–225, May 2011.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, and et.al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16),* pages 265–283, Savannah, GA, 2016. USENIX Association.

[3] Chris Margiolas. Heterogenous execution engine for llvm, hexe - a workload ir extractor, 2015.

[4] Chao Li, Yanjing Bi, Yannick Benezeth, Dominique Ginhac, and Fan Yang. High-level synthesis for fpgas: code optimization strategies for real-time image processing. *Journal of Real-Time Image Processing*, 14(3):701–712, Mar 2018.

[5] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 30, 2011.

[6] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason Anderson, Stephen Brown, and Tomasz Czajkowsk. Legup: High-level synthesis for fpga-based processor/accelerator systems. in FPGA '11.

[7] Marcel Gort and Jason H. Anderson. Range and bitmask analysis for hardware optimization in highlevel synthesis. in ASP-DAC '13.

[8] David C. Zaretsky, Gaurav Mittal, Robert P. Dick, and Prith Banerjee. Balanced scheduling and operation chaining in high-level synthesis for fpga designs. in ISQED '07.

[9] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. in SIGMETRICS '12.

[10] Sepp Hochreiter and Jurgen Schmidhuber. Long short-term memory. in Neural Computation '97.

[11] V. J. Reddi, A. Settle, D. A. Connors, , and R. S. Cohnr. Pin: A binary instrumentation tool for computer architecture research and education. in WCAE '04.

[12] Gupta, Rajesh, and Forrest Brewer. "High-level synthesis: A retrospective." *High-Level Synthesis: From Algorithm to Digital Circuit* (2008): 13-28.

[13] O'Loughlin, Declan, et al. "Xilinx vivado high level synthesis: Case studies." (2014): 352-356.

[14] "Shang." *GitHub*,github.com/etherzhhb/Shang. Accessed 1 Dec 2019.

[15] Lahti, Sakari, et al. "Are we there yet? A study on the state of high-level synthesis." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.5 (2018): 898-911.

[16] Nane, Razvan, et al. "A survey and evaluation of FPGA high-level synthesis tools." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.10 (2015): 1591-1604.

[17] Luk, Chi-Keung, et al. "Pin: building customized program analysis tools with dynamic instrumentation." *Acm sigplan notices* 40.6 (2005): 190-200.

[18] Mittal, Sparsh. "A survey of recent prefetching techniques for processor caches." *ACM Computing Surveys (CSUR)* 49.2 (2016): 1-35.

[19] Mittal, Sparsh. "A survey of recent prefetching techniques for processor caches." *ACM Computing Surveys (CSUR)* 49.2 (2016): 1-35.

[20] Braun, Peter, and Heiner Litz. "Understanding memory access patterns for prefetching." *International Workshop on AI-assisted Design for Architecture (AIDArc), held in conjunction with ISCA*. 2019.

[21] Braun, Peter, and Heiner Litz. "Understanding memory access patterns for prefetching." *International Workshop on AI-assisted Design for Architecture (AIDArc), held in conjunction with ISCA*. 2019.

[22] Peled, Leeor, Uri Weiser, and Yoav Etsion. "A neural network prefetcher for arbitrary memory access patterns." *ACM Transactions on Architecture and Code Optimization (TACO)* 16.4 (2019): 1-27.

[23] Gu, Juncheng, et al. "Efficient memory disaggregation with infiniswap." *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 2017.

[24] Zhang, Wei, et al. "Load balancing of heterogeneous workloads in memcached clusters." *9th International Workshop on Feedback Computing (Feedback Computing 14)*. 2014.

[25] Molema, Karabo Omphile. *The conflict of interest between data sharing and data privacy: a middleware approach*. Diss. Cape Peninsula University of Technology, 2016.