On the employment of approximate multipliers in high-level synthesis toolkits

Ratko Pilipović[1], Patricio Bulić[1], Uroš Lotrič[1]

[1] *University of Ljubljana, Faculty of Computer and Information Science*
E-mail: ratko.pilipovic@fri.uni-lj.si

Rising demands for processing complex and large data volumes stretch the capabilities of modern high-performance computing centres. As an answer, computing centres started to employ FPGAs, which offer comparable performance to GPUs with significantly lower power consumption. Moreover, high-level synthesis toolkits offer easy-to-use design flows which allow programming FPGAs in high-level languages like C, C++ and Python. However, although the HLS toolkits have significantly improved in last years, it is usually beneficial to prepare the core modules in more efficient hardware-description languages. In this paper, we illustrate the employment of an approximate multiplier in the high-level synthesis of the Sobel edge detector. First, we overview the Intel FPGA SDK programming flow and go through the steps for integrating custom-made modules in high-level synthesis. Then, in the experiments, we compare the synthesis of two implementations of the Sobel filter, one with an exact and one with an approximate multiplier. The results show that the Sobel filter with an approximate multiplier offers noticeably smaller FPGA resource utilisation than the Sobel filter with the exact multiplier.

1 Introduction

HPC systems mainly rely on the general-purpose graphic processing units (GPUs) to boost the performance of typical HPC workloads, such as machine learning, ODE solvers, and N-body problems. GPUs owe their success to parallel structure and excellent floating point performance. However, for specific applications, such as multimedia processing and artificial intelligence, field programmable gate arrays (FPGAs) offer a viable alternative [1]. Due to their reconfigurable nature, the FPGA can overcome common bottlenecks in processing AI algorithms and multimedia content. Moreover, FPGAs offer lower energy consumption in data centres and edge devices than GPUs.

Traditionally, FPGAs are programmed using hardware description languages (HDLs), such as VHDL or Verilog, which offer unique features to design combinational or sequential logic. However, HDLs have many drawbacks, like program verbosity, ridged and error-prone syntax, and longer development time [2]. Contrary to HDLs, the high-level synthesis (HLS) toolkits utilise high-level programming languages (C, C++, SystemC) to describe hardware designs. HLS tools parse and translate the program written in a high-level language into a corresponding register-transfer level (RTL) representation that meets certain user-specified design constraints. Compared to HDLs, HLS offers faster development, design reuse, rapid search of the design space state, and many more.

The reconfigurable nature of FPGAs enables the employment of custom computing units specially tailored for different applications. Such dedicated circuits are approximate arithmetic circuits, which facilitate energy-efficient processing in error-tolerant applications. Among approximate arithmetic circuits, the approximate multipliers deliver significant gains in area utilisation and energy consumption with negligible impact in various applications, e.g. machine learning and multimedia processing. Unfortunately, employing HLS toolkits to synthesise approximate multipliers is an unfruitful task, as they cannot achieve the same design efficiency as HDLs. However, with the recent development of HLS toolkits, it is possible to integrate HDL modules into more complex designs described by high-level languages and synthesise them together using HLS toolkits.

In this paper, we illustrate the employment of an approximate multiplier in the HLS design flow and assess its influence on synthesised design. We first implement the approximate multiplier as an RTL module written in Verilog language. Next, we integrate the RTL module in the Sobel filter implemented as the OpenCL kernel and perform high-level synthesis using the Intel FPGA SDK HLS toolkit. Finally, to assess the benefits behind the employment of an approximate multiplier, we compare resource utilisation of the Sobel filter with the exact and approximate multiplier.

The remainder of this paper is organised as follows. In Section 2 we describe existing HLS toolkits, while Section 3 concentrates on the Intel's HLS solution. Section 4 described the tested applications employing an approximate multiplier. The synthesis results are presented in Section 5 and in the Section 6 we conclude the paper.

2 Related work

The need for highly efficient HLS flow led to the development several HLS toolkits. This section describes some of the commonly used HLS toolkits in the academic world and industry.

2.1 Academic HLS toolkits

Bambu [3] is a modular HLS tool developed at Politecnico di Milano. It preserves the semantics of a program given in the C language without requiring any code manipulation. Furthermore, *Bambu*'s modularity offers easy customisation and extension with new HLS algorithms or flows. Lastly, it offers standalone verification of generated design on the given dataset.

LegUp [4] is a HLS compiler developed at the University of Toronto. It leverages a low-level virtual machine (LLVM) framework, which enables *LegUp* to synthesise most of the C commands. *LegUp* offers a complete and partial synthesis of C code to the hardware. Here the microprocessor executes one part of code while the rest is synthesised as a hardware accelerator. *LegUp* support threads and OpenMP to automatically convert the parallel code to parallel-operating hardware. It also supports automatic datapath pruning and bitmask analysis.

2.2 Industry HLS toolkits

Stratus HLS [5], provided by Cadence, offers synthesis of SystemC, C and C++. The key benefit of *Stratus HLS* is physically aware HLS, which produces smaller delays and enables low-power optimizations. *Stratus HLS* provides a rich library of intellectual property (IP) building blocks for easier and faster development. For verification purposes, it provides an automated verification flow.

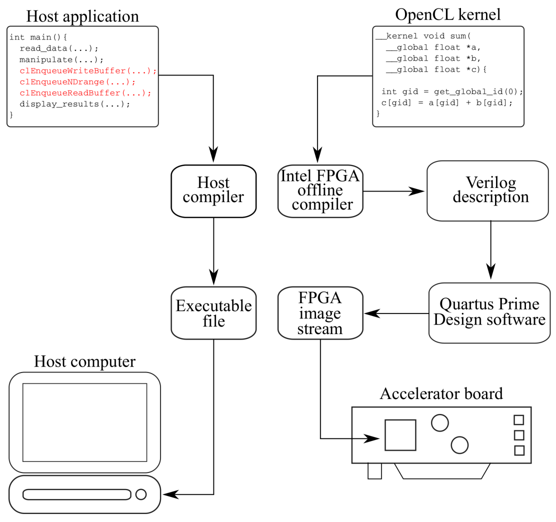
Vivado HLS, developed by Xilinx, can compile almost any C/C++ program except for dynamic language constructs. For the operation synthesis, *Vivado HLS* employs the operation chaining technique that performs operation scheduling within the clock period. For conditional statements, *Vivado HLS* generates circuits for all conditional branches. Therefore, runtime execution involves the selection between all possible results. To improve loop executions, it uses loop unrolling and pipeline execution. In *Vivado HLS* functions are mapped into modules capable of concurrent execution and self-synchronization. *Vivado HLS* serves for programming Xilinx FPGA chips, supporting Xilinx on-chip memories, DSP elements and floating-point operations.

The Intel FPGA SDK toolkit [6] employs OpenCL kernels to describe a hardware accelerator. The Intel FPGA SDK translates every command of the OpenCL kernel into custom hardware that may provide more power-efficient and flexible use than CPU and GPU architecture would allow. Besides design synthesis, the Intel FPGA SDK offers valuable tools to validate the design functionality. Also, a profiler is available to evaluate system performance and reveal the architecture bottlenecks.

3 Intel FPGA SDK toolkit

3.1 Elements and features

Figure 1 illustrates the programming flow for Intel FPGA SDK. The main SDK components that participate in programming FPGA are:



```
int main(){
  read_data(...);
  manipulate(...);
  clEnqueueWriteBuffer(...);
  clEnqueueNDrange(...);
  clEnqueueReadBuffer(...);
  display_results(...);
}
```

```
__kernel void sum(
  __global float *a,
  __global float *b,
  __global float *c){

 int gid = get_global_id(0);
 c[gid] = a[gid] + b[gid];
}
```

Figure 1: Intel FPGA SDK flow

- the OpenCL kernel and the Intel FPGA offline compiler,
- the host application and host compiler.

Compilation of the OpenCL kernel consists of two phases. In the first phase, the Intel FPGA offline compiler translates the OpenCL kernel into the Verilog module. Then, it invokes the Quartus Prime software [7], which synthesises the obtained Verilog modules and generates the FPGA image stream. The FPGA image contains the host's data to create a program object for the targeted FPGA. Compiling an OpenCL kernel is time-consuming, so Quartus must synthesise the OpenCL kernel before compiling the host application.

The host compiler compiles the host application, which manages the execution of OpenCL kernels on FPGAs. First, the host application discovers existing accelerator boards and allocates the buffers accessible both from the device and the device. Finally, the host application runs the OpenCL kernel on the accelerator device and controls its execution.

3.2 Employment of custom-made RTL modules in OpenCL kernels

Although the Intel FPGA SDK offers efficient high-level synthesis, in some cases, the OpenCL kernels need to be expanded with the custom-made RTL modules. For example, we want to use optimized RTL modules inside OpenCL code or implement some features we cannot describe in OpenCL code.

Figure 2 depicts the integration of the RTL modules in OpenCL kernel. First, we must create an OpenCL library which includes RTL modules, header files, emulation kernel, and meta files. We need to extend the RTL module with additional features to employ it inside the OpenCL kernel. The employed module must be synchronized with the rest of the design and needs to support the Avalon interfaces [8] to communicate with the system. Next, we need to add the header file, which acts as an interface between RTL modules and OpenCL kernels. In

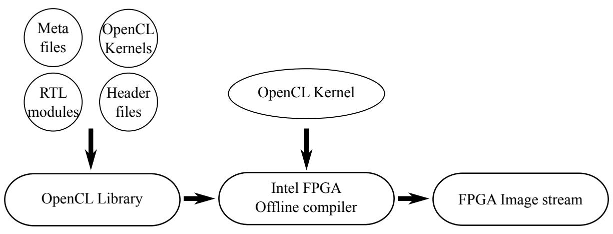


Figure 2: Intel FPGA SDK for OpenCL's Library support

addition to the header file, we need to provide a C emulation kernel of the RTL module during software simulations. Finally, we define the properties of RTL modules with metafiles, describing the number of pipeline stages, expected latency, processing stall, etc. After the OpenCL library is built, the Intel FPGA offline compiler can use the generated OpenCL library to link the RTL modules with the OpenCL kernel and generate the FPGA image stream.

4 Edge detection with an approximate multiplier

The Sobel operator [9] is the most commonly used edge detector. It relies on the spatial change in brightness to detect edges. The operator uses kernels

$$\mathbf{K}_h = \mathbf{K}_v^T = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad , \tag{1}$$

where $\mathbf{K}_h$ calculates the changes in brightness in the horizontal direction, and $\mathbf{K}_v$ computes brightness change in the vertical direction. By convolving both filters with the original image $\mathbf{I}$, we get the spatial brightness change

$$\mathbf{E} = |\mathbf{K}_h * \mathbf{I} + \mathbf{K}_v * \mathbf{I}| \tag{2}$$

In the final step, the algorithm compares the brightness change $\mathbf{E}$ of each pixel to a predetermined threshold. Pixels whose value is greater than the threshold constitute edges; otherwise, they belong to the background.

Edge detection and other image processing operations represent error-tolerant applications. Therefore, we replace the exact multiplier in the Sobel edge detector with an approximate one to achieve more energy-efficient processing. For an approximate multiplier, we chose the iterative logarithmic multiplier (ILM) [10], which represents a simple and efficient multiplier that achieves arbitrary accuracy through an iterative procedure. Unlike the Mitchell multiplier, ILM employs a simpler approximation term. Basic ILM has a relatively high mean relative error of around 10%. However, when the procedure is iteratively applied, the accuracy of the multiplier increases. With only one iteration for error correction, the ILM design delivers the mean relative error below 1%. A good feature of ILM is the ability to perform each iteration concurrently in a pipelined fashion which made the ILM multiplier especially useful in hardware neural networks [11].

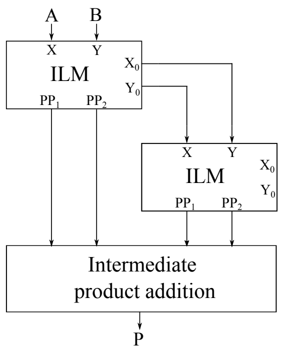


Figure 3: Proposed ILM design with one iteration

Let X and Y be two unsigned input numbers. ILM approximates the product of X and Y as

$$X \cdot Y \approx 2^{k_x+k_y} + 2^{k_y} \cdot X_0 + 2^{k_x} \cdot Y_0 \quad , \tag{3}$$

where $k_x = \lfloor \log_2 X \rfloor$, $k_y = \lfloor \log_2 Y \rfloor$, $X_0 = X - 2^{k_x}$, and $Y_0 = Y - 2^{k_y}$. Eq. 3 can be rewritten by adding the first and third summand to

$$X \cdot Y \approx 2^{k_x} \cdot Y + 2^{k_y} \cdot X_0 = PP_1 + PP_2 \quad . \tag{4}$$

Figure 3 shows the overall design of ILM with two iterations. The ILM block has four outputs PP_1, PP_2, X_0, and Y_0. PP_1 and PP_2 represent intermediate products which are obtained by Eq. 4. On the other hand, X_0 and Y_0 represent remainders from the first stage. The Intermediate product addition block adds the intermediate products from both stages using Wallace tree [12].

5 Results

In this section, we present the synthesis results for the Sobel filter. We implemented the Sobel filter with the exact and the approximate multiplier and synthesised it using the Intel FPGA SDK tool. The exact multiplier is synthesised using the multiplication operator in C, while the ILM is integrated into the kernel as a custom RTL module. The synthesis results are reported for Cyclone V FPGA chip present on the C5P development kit.

Table 1 shows the resource utilisation and latency for the Sobel operator when the exact multiplier and ILM are employed. We can see that the employment of ILM delivers up to 12% savings in look-up tables (LUTs) utilisation and around 5% savings in flip-flops (FFs) utilisation. Relatively small resource utilisation savings can be attributed to the synthesis of the exact multiplier. The Intel FPGA compiler is optimised to map standard arithmetic operations onto FPGA efficiently. Besides multiplication, the synthesised contains additional logic, e.g. memory access, addition, and flow control, that significantly affects resource utilisation. Concerning the latency, we can see that the employment of ILM increases latency by one cycle. Moreover, the HLS toolkit fails to schedule the integrated RTL module efficiently.

6 Conclusion

This paper illustrates the employment of an approximate multiplier in the HLS toolkits to improve synthesised de-

Table 1: Resource utilization and latency of Sobel filter when different multipliers are employed

Multiplier	Lookup tables	Flipflops	Latency (No. of cycles)
Exact multiplier	7079	8924	28
ILM	6203	8473	29

signs' resource consumption. We employ ILM with one iteration for error correction for the approximate multiplier. We describe the employed multiplier with Verilog language and its integration into the Sobel edge detector. The Sobel edge detector is implemented as an OpenCL kernel and synthesised using the Intel FPGA SDK toolkit. This way, we harness the power of both HDL and HLS: HDLs for efficient implementation of small and straightforward arithmetic circuits and HLSs for synthesising complex designs. With the help of the Intel FPGA SDK toolkit, we have successfully integrated the employed multiplier into the Sobel edge filter. The synthesis results show that approximate multiplier offers noticeable savings in resource utilisation – LUTs and FFs utilisation. However, a slight increase in latency presents a drawback of the proposed design. In future into the topic, we should investigate the influence of approximate multipliers on the HLS of neural networks. As multiplication dominates in the neural network's inference and training, we anticipate that our approach will significantly improve resource utilisation in neural network processing.

Acknowledgements

This research was supported by Slovenian Research Agency under Grants P2-0359 (National research program Pervasive computing), P2-0241 (Synergy of the technological systems and processes) and by Slovenian Research Agency and Ministry of Civil Affairs, Bosnia and Herzegovina, under Grant BI-BA/19-20-047 (Bilateral Collaboration Project).

References

[1] Y. Sano, R. Kobayashi, N. Fujita, and T. Boku, "Performance evaluation on gpu-fpga accelerated computing considering interconnections between accelerators," in *International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, ser. HEART2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 10–16. [Online]. Available: https://doi.org/10.1145/3535044.3535046

[2] R. Millón, E. Frati, and E. Rucci, "A comparative study between hls and hdl on soc for image processing applications," *arXiv preprint arXiv:2012.08320*, 2020.

[3] C. Pilato and F. Ferrandi, "Bambu: A modular framework for the high level synthesis of memory-intensive applications," in *2013 23rd International Conference on Field programmable Logic and Applications*. IEEE, 2013, pp. 1–4.

[4] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: high-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2011, pp. 33–36.

[5] Cadence, "Stratus HLS," https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/digital-design-signoff/stratus-ds.pdf, 2019, [Online; accessed 4-July-2019].

[6] "Intel® fpga sdk for opencl™ software technology." [Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html?wapkw=intel+fpga+sdk+for+opencl

[7] "Fpga design software - intel® quartus® prime." [Online]. Available: https://www.intel.com/content/www/us/en/products/details/fpga/development-tools/quartus-prime.html

[8] Intel, "Avalon Streaming Interface," https://www.intel.co.jp/content/dam/altera-www/global/ja_JP/pdfs/literature/fs/fs_avalon_streaming.pdf, 2019, [Online; accessed 10-July-2019].

[9] I. Sobel, "A 3x3 isotropic gradient operator for image processing," *Stanford Artificial Intelligence Project*, 1968.

[10] Z. Babić, A. Avramović, and P. Bulić, "An iterative logarithmic multiplier," *Microprocessors and Microsystems*, vol. 35, no. 1, pp. 23–33, 2011.

[11] U. Lotrič and P. Bulić, "Applicability of approximate multipliers in hardware neural networks," *Neurocomputing*, vol. 96, pp. 57–65, 2012.

[12] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Transactions on electronic Computers*, no. 1, pp. 14–17, 1964.