Using C, C++ and C# For A Wider Consensus in High-Level Synthesis Compilers

Babar Khan
TU Darmstadt, Computer Science, Germany

***Abstract*—High-level synthesis (HLS) is an automated design process that brings a significant reduction in design cycles by pushing the design to higher levels of abstraction. What is more, HLS has received increasing attention because of its ability to handle machine learning matrices and iterative design efforts. Specifically, the low precision arithmetic and number systems have become the standard for performing deep learning inference. Recently it was shown that the data type-agnostic (DTA) programming methodology based on HLS empowers Xilinx Vivado HLS compiler to synthesize hardware architectures for a plethora of data types without modifications of a C++ source code. Especially, the very same code can be used for real-valued and complex-valued data paths by utilizing the C++ class and template library. This paper presents a novel implementation of the aforementioned methodology in additional languages namely C and C# to evaluate the improved performance for three different arithmetic formats namely floating-point, fixed-point and arbitrary-precision integer.**

I. Data Type Agnosticism Methodology

FPGAs are configurable integrated circuits that provide a good trade-off in terms of performance, power consumption, and flexibility with respect to other architectures. However, it is a challenging task to program FPGAs. The processes to design hardware at a high-level of abstraction can enhance the productivity of FPGA designers. High-level Synthesis (HLS) being an automated design process takes as input an algorithmic description at a higher-level of abstraction and generates as output a hardware implementation of a micro-architecture (e.g. in VHDL/Verilog) known as register transfer level (RTL). HLS is gaining more popularity because of its ability to handle machine learning matrices and iterative design efforts. Due to the high computational requirement for machine learning matrices and iterations, it is important for modern hardware to support a wide variety of data types.
Knoop *et al.* [1], [2] has proposed to extend the capabilities of design space exploration of HLS through a data type-agnostic programming methodology using Xilinx Vivado HLS. The results showed that the rapid digital architecture design flow based on HLS is a superior alternative to the traditional RTL design methodology.
This work shows an extension of data type-agnostic programming methodology to mainstream HLS compilers. To the best of our knowledge no prior work exists that benchmark custom arithematic in mainstream HLS compilers. Fig.1 illustrates the concept of data type-agnostic programming methodology. It shows the data types are conceptualized as variants and they are pushed through the standard HLS design flow. A test bench is a vital part of the flow to check the correctness of the design. Finally, the evaluated output is plotted in terms of resource utilization and accuracy.

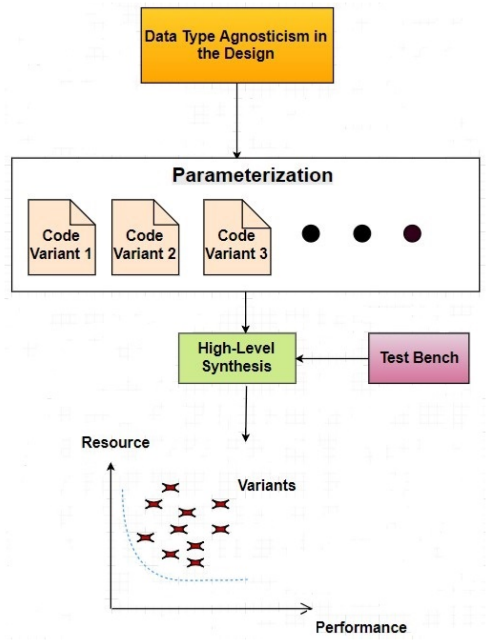

Figure 1: The data type-agnostic programming methodology integration in a design flow

Table I: Benchmarked High-Level Synthesis (HLS) Compilers

Compiler	Compiler	Input	Output
Vitis	LLVM	SysC/C/C++	VHDL/Verilog
Catapult	GCC	SysC/C/C++	VHDL/Verilog
Intel	LLVM	C/C++/OpenCL	Verilog
LegUp	LLVM	C	Verilog
Kiwi	.NET	C#	Verilog
Bambu	GCC	C	Verilog

As shown in Figure 2, adhering to the proposed methodology, the algorithmic description of the source code was kept generic and parametrisable in terms of data types. In C++ based HLS tools this was achieved with the help of C++ Templates and keyword *typedef*. For C based HLS tools this was achieved using the void pointer (void*) and keyword *typedef*. In a C# based HLS tool, C# generics were used. As FPGAs are ultimate pipeline processors, the loop optimizations were one of the major optimizations applied. Logic, DSP and memory usage were taken into considerations for the resource utilization. Hence the four main metrics to evaluate the resource utilization were LUTs/ALUTs, DSPs, BRAMs and FlipFlops. The target device used for LegUp and Intel HLS compiler was Altera Stratix V. Similarly, the target device

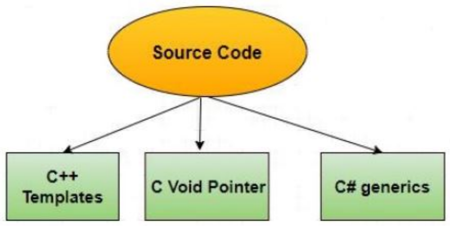

Figure 2: Data Type Agnosticism implementation in C based languages

Table II: First Design Architecture Results for Vitis,Catapult and Intel

Standard-Optimized Resource and Performance

Tool	**Data Type**	**LUT**	**DSP**	**BRAM**	**FF**	**Freq**	**Cyc**
Vivado	ap_fixed<28,8>	87	1	0	82	251	609
Catapult	ac_fixed<28,8>	137	1	0	68	248	801
Intel	ac_fixed<28,8>	156	1	0	107	248	768
Tool	**Data Type**	**LUT**	**DSP**	**BRAM**	**FF**	**Freq**	**Cyc**
Vivado	float	892	5	0	644	246	2001
Catapult	float	732	4	0	487	242	1456
Intel	float	817	4	0	621	233	2332

Table III: Design Architecture Results for Kiwi, LegUp and Bambu

Standard-Optimized Resource and Performance

Tool	**Data Type**	**LUT**	**DSP**	**BRAM**	**FF**	**Freq**	**Cyc**
Kiwi	int 14	68	1	0	103	279	502
LegUp	int 14	73	1	0	112	266	569
Bambu	int 14	129	2	0	186	233	1017
Tool	**Data Type**	**LUT**	**DSP**	**BRAM**	**FF**	**Freq**	**Cyc**
Kiwi	int 32	124	3	1	165	233	906
LegUp	int 32	202	4	1	258	231	1002
Bambu	int 32	180	2	1	209	208	1112
Post-Optimized Resource and Performance							
Tool	**Data Type**	**LUT**	**DSP**	**BRAM**	**FF**	**Freq**	**Cyc**
Kiwi	int 14	101	1	1	198	291	106
LegUp	int 14	194	1	1	209	277	378
Bambu	-	-	-	-	-	-	-
Tool	**Data Type**	**LUT**	**DSP**	**BRAM**	**FF**	**Freq**	**Cyc**
Kiwi	int 32	156	6	1	301	241	129
LegUp	int 32	236	4	1	278	238	456
Bambu	-	-	-	-	-	-	-

used for Vivado, Kiwi, Catapult and Bambu was Xilinx Virtex-7. Both Stratix V and Virtex-7 are SRAM-based island-style 28 nm state-of-the-art high-performance FPGAs containing many heterogeneous computational elements. The DSP units in Altera and Xilinx devices are almost equal.

A. Evaluation

For Vivado, Catapult and Intel HLS the two main data types used were fixed-point and floating-point. Considering a scenario where the range of floating-point input is from 0 to 1 and the desired precision is 3 decimal digits. Then the most feasible approach would be to have input arrays of a hardware kernel that can have a fixed-point data type with 11-bit variable input (1-bit integer value with 10 decimal places). Further, for the difference (subtraction) operation a signed fixed-point data type with 12-bit variable (2-bit integer with 10 decimal places) can be considered. The multiplication operation can have up to 20 decimal places of significance. Thus, an unsigned 21-bit variable (1-bit integer value with 20 decimal places) fixed-point data type can be used for the multiplication result. Regarding the accumulation operation, there will be the requirement of extra integer bits. For the vector size N = 100, the accumulator should be an unsigned a 28-bit variable (8-bit integer with 20 decimal places).
For Kiwi, LegUp and Bambu the data types considered were a 32-bit integer and arbitrary-precise integer of 14-bits. Trimming constant or unwanted bits can decrease the width of the computed values, enabling the HLS tool to use reduced width functional units. If the input arrays. are full 32-bit signed values then using an arbitrary precision integer with 32-bits will not be more efficient than using straight away a native integer of 32 bits. Considering a scenario where the input arrays have a range of values between 0 and 1 then using an arbitrary precision integer of 7 bits for input arrays is sufficient. For the difference and multiplication operations, arbitrary precise of 8-bits and 14-bits respectively can be used. Similarly, for the accumulation and output array, 14-bits is sufficient.

II. Results Summary

Through data type-agnostic programming methodology not only the designer can minimize the hardware cost while achieving the numerical accuracy, occasionally the designer can also relax the conventional requirements of precise equivalence. This falls in line with a computation technique called approximate computing. At the custom hardware level, there are many error-tolerant applications that have the inherent resilience to generate tolerable outputs despite some of its computations are executed imprecisely in the hardware.

References

[1] B. Knoop, K. Vinod, S. Schmale, D. Peters-Drolshagen, and S. Paul, "Fast digital design space exploration with high-level synthesis: A case study with approximate conjugate gradient pursuit," in *2016 50th Asilomar Conference on Signals, Systems and Computers*, Nov 2016, pp. 412–416.

[2] B. Knoop, J. Rust, S. Schmale, D. Peters-Drolshagen, and S. Paul, "Rapid digital architecture design of orthogonal matching pursuit," in *EUSIPCO*. IEEE, 2016, pp. 1857–1861.