Designing scalable FPGA architectures using high-level synthesis

Johannes de Fine Licht
ETH Zurich
definelicht@inf.ethz.ch

Michaela Blott
Xilinx Inc.
mblott@xilinx.com

Torsten Hoefler
ETH Zurich
htor@inf.ethz.ch

Abstract

Massive spatial parallelism at low energy gives FPGAs the potential to be core components in large scale high performance computing (HPC) systems. In this paper we present four major design steps that harness high-level synthesis (HLS) to implement scalable spatial FPGA algorithms. To aid productivity, we introduce the open source library *hlslib* to complement HLS. We evaluate kernels designed with our approach on an FPGA accelerator board, demonstrating high performance and board utilization with enhanced programmer productivity. By following our guidelines, programmers can use HLS to develop efficient parallel algorithms for FPGA, scaling their implementations with increased resources on future hardware.

CCS Concepts • **Computing methodologies** → **Parallel programming languages**; • **Hardware** → *High-level and register-transfer level synthesis*;

ACM Reference format:
Johannes de Fine Licht, Michaela Blott, and Torsten Hoefler. 2018. Designing scalable FPGA architectures using high-level synthesis. In *Proceedings of PPoPP '18: 23nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Vienna, Austria, February 24–28, 2018 (PPoPP '18),* 2 pages.
https://doi.org/10.1145/3178487.3178527

1 Motivation

Although FPGAs have started to see deployment in cloud and data center solutions, their use in HPC is still niche, owing partially to the software background of most HPC end users, and partially to the rapid development in GPU and many-core offerings. With exascale on the horizon, the available power budget will be a significant bottleneck in constructing large scale HPC systems. This has made energy efficiency emerge as a first class citizen, triggering a surge of interest in FPGAs as a way to scale up performance without a proportional increase in energy consumption.

PPoPP '18, February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-4982-6/18/02.
https://doi.org/10.1145/3178487.3178527

Traditionally, FPGA implementations are written in hardware description languages (HDLs) by hardware experts, who design circuitry solving the application on a register transfer level. With the increased adoptation of FPGAs in cloud solutions, vendors and third parties are attempting to raise the level of abstraction, by developing tools that allow software engineers to program FPGAs using popular languages such as OpenCL and C/C++.

```
1 void StreamingDataflow(Data_t *...) {
2   #pragma HLS PIPELINE DATAFLOW
3   Stream<Data_t> pipes[D+1];
4   ReadMemory(memory_in, pipes[0]);   // Writes head
5   for (int d = 0; d < D; ++d) {
6     #pragma HLS UNROLL
7     ProcessingElement(pipes[d], pipes[d+1]);
8   }
9   WriteMemory(pipes[D], memory_out); // Reads tail
10 }
```

Listing 1. Implementation of streaming dataflow in HLS.

2 Scalable FPGA design in HLS

We propose guidelines to implement massively parallel FPGA programs using high-level synthesis (HLS) from applications that exposes a scalable source of parallelism, exploiting fixed memory access patterns to efficiently pipeline computations to achieve a compute bound implementation. To achieve true scalability, we target designs that can fold a dimension of the target application by a variable factor D, such that D steps can be evaluated by D processing elements (PEs) in parallel, reducing the time to solution by a factor $\approx 1/D$. This is constrained only by the available logic and memory on the chip. We use the kernel structure shown in Listing 1 to implement such programs. Components are connected by FIFO *stream* (or *pipe*) objects, moving data between modules on the chip. To achieve a massively parallel design in HLS, we follow four major guidelines:

- **Pipelining and vectorization:** We exploit the immediately available spatial parallelism by pipelining and unrolling, instantiating every arithmetic operations separately in hardware, then multiply them by the maximum vector width supported by the memory bandwidth.
- **Buffering:** To reduce pressure on bandwidth to external memory, and to regularize the access pattern, we use on-chip buffering to maximize spatial reuse. We map variables

	Type	Stencil	Source	Device	Performance	Frequency	Power	Power efficiency
TPDS'17 [1]	CPU	Heat 2D	C	Xeon E5645	54 GOp/s	2400 MHz	(2 × 80 W)[a]	(0.34 GOp/J)[a]
PPoPP'17 [4]	GPU	Jacobi 2D	CUDA	Titan X	490 GOp/s	1417-1531 MHz	(250 W)[a]	(1.96 GOp/J)[a]
FPT'13 [3]	FPGA	RTM (3D)	Maxeler	MaxGenFD	131 GOp/s	100 MHz	142 W	0.92 GOp/J
TPDS'14 [5]	FPGA	Jacobi 2D	HDL	9× EP3SL150	260 GOp/s	133 MHz	201 W	1.29 GOp/J
TPDS'17 [6]	FPGA	Jacobi 2D	OpenCL	395-D8	238 GOp/s	230 MHz	(75 W)[a]	(3.17 GOp/J)[a]
Ours	FPGA	Jacobi 2D	C++	TUL KU115	228 GOp/s	160 MHz	31 W	7.36 GOp/J

Table 1. Performance comparison to related work for single precision. [a]Power not reported: Estimating with TDP.

to either on-chip RAM units or registers depending on the required access pattern.

- **Tiling:** To accommodate arbitrary domain sizes, we implement a tiling scheme to make the on-chip memory requirements independent of the domain size. By maximizing the tile size we can minimize the performance penalty of tiling overhead.
- **Streaming dataflow:** To overcome the memory bottleneck and routing delays, we replicate PEs and connect them in sequence, instantiated as an asynchronous dataflow architecture, allowing scaling with logic on the chip.

To demonstrate our approach we evaluate an iterative 4-point Jacobi 2D stencil algorithm. We can fold the number of timesteps T to T/D passes through a chain of D PEs arranged in a linear array. Each PE evaluates the full spatial dimension at a separate, consecutive timestep, effectively treating $\{t, t+1, ..., t+(D-1)\}$ in parallel (Fu and Clapp [2]). Because the memory bandwidth requirement is constant in the number of PEs connected in the deep pipeline (Sano et al. [5]), the amount of parallelism $\approx D$ can scale with logic and buffer resources on the chip. For the example presented here, this results in a ~36× increase in DSP usage and performance over a single vectorized PE, for a total of ~11 400× performance increase over a naive HLS design.

3 Experimental evaluation

We target the TUL KU115 accelerator board, which hosts a Xilinx XCKU115-2FLVB2104E FPGA and four DDR4 banks. We build kernels and interface with the host computer using the SDx 2017.1 toolflow. The domain size is fixed at 8192×8192. Power is measured as the difference between total system power during kernel execution and system power with no FPGA installed. Figure 1 shows board utilization, frequency, power, and performance results, and Table 1 compares the result of our single precision kernel to related work. We additionally plot the fraction of the *experimental* peak performance measured by building synthetic kernels of the corresponding set of floating point operations. The proposed guidelines allow us to reach 80% of the measured synthetic peak performance in HLS with the stencil kernel, at a 3.8× increase in energy efficiency over TDP for the best reported result on GPU.

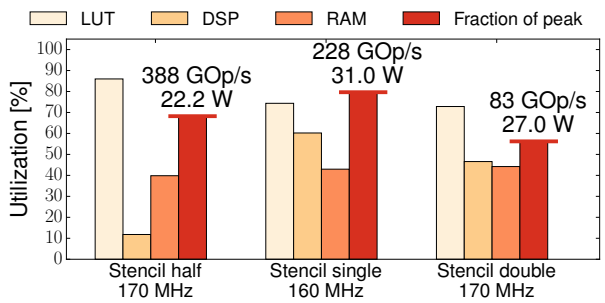

Figure 1. Performance and resource utilization.

4 Tool support

We provide the open source library *hlslib*[1] to aid development of HPC designs such as the ones described here. This includes useful primitives and plug-in hardware components that serve as productivity-enhancing abstractions, and a hardware emulation flow for designs with cycles in the data dependency graph. The library consists primarily of C++ header files, with the addition of scripts for CMake integration and hardware generation, interfacing with the Vivado HLS and SDAccel tools for Xilinx FPGAs, exploiting OpenCL on the host side to launch C++ kernels.

5 Acknowledgements

We thank Xilinx for donation of software, hardware and compute hours, and the Swiss National Supercomputing Centre (CSCS) for access to compute infrastructure.

References

[1] Uday Bondhugula, Vinayaka Bandishti, and Irshad Pananilath. 2017. Diamond Tiling: Tiling Techniques to Maximize Parallelism for Stencil Computations. *TPDS* 28, 5 (May 2017), 1285–1298.

[2] Haohuan Fu and Robert G. Clapp. 2011. Eliminating the Memory Bottleneck: An FPGA-based Solution for 3D Reverse Time Migration. *Proceedings of FPGA'11*, 65–74.

[3] Xinyu Niu, Jose G. F. Coutinho, Yu Wang, and Wayne Luk. 2013. Dynamic Stencil: Effective exploitation of run-time resources in reconfigurable clusters. *Proceedings of FPT'13*.

[4] Nirmal Prajapati, Waruna Ranasinghe, Sanjay Rajopadhye, et al. 2017. Simple, Accurate, Analytical Time Modeling and Optimal Tile Size Selection for GPGPU Stencils. *Proceedings of PPoPP'17*.

[5] Kentaro Sano, Yoshiaki Hatsuda, and Satoru Yamamoto. 2014. Multi-FPGA Accelerator for Scalable Stencil Computation with Constant Memory Bandwidth. *TPDS* 25, 3 (March 2014), 695–705.

[6] Hasitha M. Waidyasooriya, Yasuhiro Takei, et al. 2017. OpenCL-Based FPGA-Platform for Stencil Computation and Its Optimization Methodology. *TPDS* 28, 5 (May 2017), 1390–1402.

[1] https://github.com/definelicht/hlslib