Improved High-Level Synthesis for Complex CellML Models

Björn Liebig[1] ✉, Julian Oppermann[1], Oliver Sinnen[2], and Andreas Koch[1]

[1] Embedded Systems and Applications, Technische Universität Darmstadt, Germany
{liebig,oppermann,koch}@esa.informatik.tu-darmstadt.de
[2] Parallel and Reconfigurable Computing, University of Auckland, New Zealand
o.sinnen@auckland.ac.nz

Abstract. In this work, we present the use of a new high-level synthesis engine capable of generating resource-shared compute accelerators, even from very complex double-precision codes, for cell biology simulations. From the domain-specific CellML description, the compilation pipeline is able to generate hardware that is shown to achieve a performance similar to or exceeding current generation desktop CPUs, and has energy savings of up to 96% even for a single accelerator, which requires just 25-30% area on a mid-sized FPGA.

Keywords: High-Level Synthesis, CellML, FPGA, Floating-Point

1 Introduction

For the simulation of biological systems at the cell level, CellML [1] has proven to be a useful domain-specific representation, from which simulation models for a number of execution platforms can be created. Since experiments commonly require a multitude of simulation runs with different input data, achieving energy efficiency has become an objective in addition to raw simulation performance. To this end, automatic generation of FPGA-based simulation accelerators from CellML descriptions has already been successfully investigated [2, 3]. However, the purely throughput-oriented spatial architectures of prior art cannot compile more complex cell descriptions due to chip size constraints, and then only provide single-precision floating point accuracy.

We present the use of a new high-level synthesis engine capable of generating resource-shared compute accelerators even for very complex double-precision simulation codes, for use in a domain-specific CellML-to-accelerator compilation pipeline.

The key contributions presented in this paper are 1) a source-to-source transformation that enables more efficient high-level synthesis of the intermediate C used in the CellML compilation pipeline, 2) adaptions of a high-level synthesis engine for CellML compilation, and 3) a case study using the developed tool flow to accelerate five of the largest CellML models on the the Xilinx Zynq platform in double-precision. These models could not be translated using prior approaches,

Listing 1.1. Exerpt from CCGS output for [5], reformatted for readability

```
void computeRates(double VOI, double* CONSTANTS, double* RATES, double* STATES, double* ALGEBRAIC) {
  RATES[0]     = CONSTANTS[2] * STATES[2] * STATES[3] * (1.0 - STATES[0]) - CONSTANTS[3] * STATES[0];
  ALGEBRAIC[0] = 1.0 / (1.0 + exp(CONSTANTS[5] * (STATES[1] - CONSTANTS[6])));
  RATES[2]     = (ALGEBRAIC[0] - STATES[2]) / CONSTANTS[687];
  ALGEBRAIC[9] = CONSTANTS[116] / (1.0 + pow(CONSTANTS[119] / STATES[10], CONSTANTS[117]));
  ...
}
```

as their fully spatial realization exceeds the FPGA size. We demonstrate significant performance gains compared to CPU execution, and better energy efficiency than a GPU implementation.

2 Related Work

2.1 CellML-based Simulation

A CellML model is an XML-based description of a cell comprised of interconnected components, and expressed as a system of ordinary differential equations (ODE). OpenCMISS [4] is a simulation workbench where instances of cell models are used as the data points in larger grids (or other spatial arrangements), e.g., to simulate a piece of ventricular tissue. Roughly, the simulation approach is as follows: The interactions between the neighboring grid cells are computed at discrete *macro* time steps. In order to progress the state of each cell from the current to the next macro time step, numerical integration of the ODE system is used. The simulation accuracy depends on the granularity of the integration steps: the more *micro* time steps are used to discretize the time between two macro time steps, the better the approximation. As the integration phase is done for each cell independently, the resulting potential for acceleration on parallel architectures is huge [4, 2]. Note that this observation has a direct impact on us aiming for smaller accelerators: Even though we concentrate here on single-accelerator performance, we could tile the entire FPGA with independent processing elements, as the accelerators are not bottlenecked by memory bandwidth. This use of MIMD computation structures also allows FPGA-based systems to scale beyond the SIMD/SIMT-approach used by GPUs.

While it would be possible to build a front-end to parse and interpret CellML directly, we instead rely on the "C Code Generation Service" (CCGS) [6], which infers a sequential execution order for the underlying initial-value problem. We use the generated highly idiomatic C code as a domain-specific IR for the rest of the compile flow. Listing 1.1 shows an excerpt from the translation of a model by Faville et al. [5]. One execution of the function `computeRates` corresponds to one micro time step in the numerical integration. `VOI` is the "variable of integration", which is usually the time. `CONSTANTS` is a read-only array that contains model parameters. The current state of each component in the model is passed as the read-only array `STATES`. Intermediate values are stored in the array `ALGEBRAIC`. Its elements are always written before read. Finally, the values in the output-only array `RATES` represent the rates of change of the components for the next micro

time step. All arrays use the C type `double`, have statically known sizes and are accessed by literal indices. Each statement in the equation-evaluation code is an assignment to one of the aforementioned arrays.

2.2 CellML-specific HLS with ODoST

The vast potential for parallel processing motivated ODoST [2, 3], a high-level synthesis system custom-made to construct accelerators for the numerical integration phase. ODoST reads the C code derived from a CellML model, generates a *fully-spatial* datapath for the `computeRates` function and handles hardware synthesis for the accelerator. Internally, floating-point (FP) operators generated by FloPoCo [7] are used.

The proposed architecture is deeply pipelined and favors throughput instead of latency: while the computation of subsequent micro-time steps for a *single* cell cannot be pipelined, the accelerator can begin to compute a micro-time step for a *different* cell in every cycle. The fully-spatial design does achieve the optimum throughput of one result per clock cycle, but may have excessive hardware requirements for larger models, even after applying additional standard and domain-specific compiler optimizations [8].

In contrast, our approach constructs latency-optimized, non-pipelined micro-time step accelerators for even the largest CellML models. It leverages the intelligent resource-sharing and fast FP operators proposed by Liebig and Koch [9], and allows us to flexibly trade-off parallelism (number of function units) with area limits.

2.3 Generic HLS with Nymble

As the base for our CellML-specific work described here, we build on the Nymble [10] HLS framework with the Nymble-RS extensions [9] for resource-shared micro-architectures.

Nymble [10] itself uses the LLVM compiler framework as front-end and for target-independent optimization. It can translate just parts of functions to hardware, and even exclude sections within that area, moving them back to software, as required. The generated hardware kernels and software parts execute together in a shared memory architecture in the same address space (allowing transparent passing of pointers between software and hardware). Nymble has already been used as base for research in domain-specific compilation [11] and the synthesis of advanced micro-architectures [12, 13].

Nymble-RS allows the area-efficient high-level synthesis of complex irregular codes having unstructured non-vectorizable FP computations in large loop bodies. It uses numerous techniques, such as hierarchical microcode, schedule-sensitive tree height optimization, and multiple mechanisms for intermediate value storage, to tightly control the area of the generated accelerator. As it targets scientific HPC, it also includes highly optimized double precision operators faster than the ones offered by FloPoCo [7].

2.4 Industrial and academic HLS systems

Many other academic and industrial HLS tools are capable of translating (often differing) subsets of C into synthesizable RTL HDL code [14].

Most industrial tools such as Vivado HLS [15] or Mentor Catapult[16] focus only on the hardware synthesis itself. They do not support hardware/software-co-execution and interface synthesis in a heterogeneous system. This restriction also applies to the academic compilers Bambu [17] and DWARV [18]. For FP, DWARV does not support typical maths functions [18], while Synphony C and Catapult do not support FP at all [14]. Bambu does supports FP operations using the FloPoCo-Library[7]. However, it lacks the floor operation which is required in many CellML models.

The academic tool LegUp [19] has seen widespread use and much development, which led to its recent commercialization as an industrial software product. LegUp also leverages the LLVM compiler framework and supports a large C subset as input, including FP operations. The tool supports hardware-software co-synthesis in a heterogeneous system, which makes it another interesting candidate for CellML synthesis.

3 Proposed Compilation Flow

Figure 1 shows the compilation flow from an XML-based CellML model to the FPGA hardware design. CCGS [6] is used to translate the actual CellML descriptions from the CellML repository [20] to idiomatic C code. This code, representing the equation systems, is optimized by CellML-opt [8] using the "z" flow, which leads to the optimizations of LLVM's aggressive size optimization preset "-Oz" being applied before the actual hardware synthesis. At this stage, no unsafe FP transformations are performed.

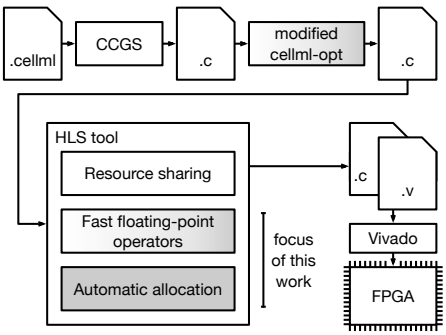

Fig. 1. CellML-to-accelerator compilation flow

In addition, CellML-opt was modified in this work to use actual local variables for intermediate values, instead of storing them in arrays. This removal of memory operations allows the actual HLS to more flexibly select between multiple storage mechanisms for intermediate values, and also enables, e.g., schedule-sensitive tree height optimization.

The output of the HLS tool consists of Verilog RTL, which is then fed for actual logic synthesis into the vendor tools (Xilinx Vivado, in our case). As the Nymble framework is a true hardware-software-co-compilation system, it also performs interface synthesis to

access the newly generated accelerator from the software. For evaluation purposes, we use a short software program here that performs only the numerical integration step using the accelerators, instead of the full-scale OpenCMISS [4] simulation framework.

3.1 Additional FP Operators for CellML Models

While Nymble-RS does already support a number of high-performance double precision FP operations (dmul, dadd, dsub, ddiv), it needed to be extended with the additional operations log, exp, floor and pow required by CellML simulation models. In this initial prototype of the flow, these have not yet been optimized to the same degree as the four basic operations, but instead are based on existing implementations. In all cases, we aim for an $f_{\max}$ of 200 MHz on Xilinx Virtex 7-level technology.

The *log* function is realized by an instance of the Xilinx CoreGen log core. For the target frequency, the core was configured to use 34 cycles, which ensures operation above 200 MHz. For the *exp* function, we use the FloPoCo exp core [21]. However, since FloPoCo is using a non-IEEE754 conformant number format, input and output must be converted from and to IEEE754 format. 200 MHz operation thus requires a pipeline depth of 26, with an additional two cycles required for the format conversion.

Since no realization for the *floor/-ceil* functions was available to us, a custom implementation was developed. As shown in Figure 2, it operates as follows: The number of *mantissa* bits to the right of the binary point is computed from the exponent in the first cycle. The second cycle clears the required number of mantissa bits, but tracks if any of these were '1' before. This latter information is used in the third cycle to conditionally increment the mantissa. Each of these operations has only a small delay and allows operation faster than 200 MHz. The first stage has an even shorter delay, and can be chained with a preceding multiplexer in resource-shared architectures.

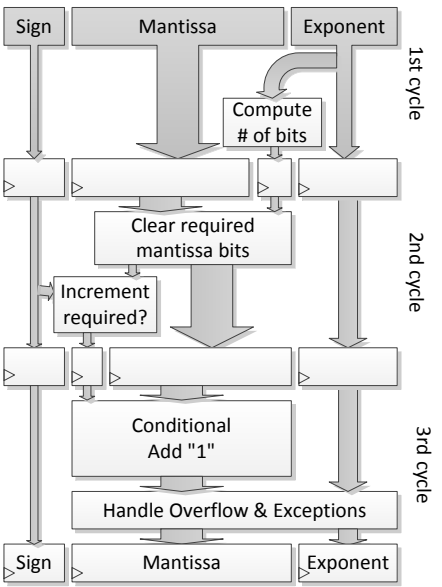

Fig. 2. Newly developed *floor* operator

The final function required for CellML models, *pow(x,y)*, is realized as *exp(log(x) * y)*. While this introduces a larger error (discussed in Section 4.3) and is thus not fully IEEE754-compliant, it allows resource-sharing with other exp, log and dmul operations and serves to fulfill our aim of generating area-efficient hardware.

3.2 Heuristic for Automatic Allocation

Despite being generated from CellML descriptions in an idiomatic manner, the actual operation mix varies wildly between different simulation models and should be reflected in the accelerator micro-architecture, specifically the number of FP operators for each operation type.

We extended Nymble-RS with the following heuristic to enable the HLS tool to automatically make a sensible, though not necessarily optimal choice. The only user-set constraint is the total number of FP units to be used, which serves as a limit on the hardware area required by the accelerator.

The simple heuristic aims to reflect the mix of FP operations in the model with the mix FP operators in the hardware, subject to the constraint that at least one FP operator is available for the required operation types. The initial minimum number of FP operators of each type is computed by determining the fraction for each operation type of the total user-set number of FP operators, based on the relative frequency these operation types occur in the model, and rounding down. The initial solution is then incrementally refined by determining the operators farthest away from their ideal ratio, and adding another operator of this type. This happens until the user-set upper bound on the total number of FP operators is reached.

While this heuristic already delivers good performance and energy efficiency (see Section 4), it could serve as a initial solution for more intelligent iterative refinement, e.g., by simulated annealing, which could then also consider individual operator areas.

4 Experimental Results

4.1 Test Setup

Table 1. Examples from CellML repository [20]

Model	# operations									Source
	+ -	*	/	pow	exp	log	⌊ ⌋	other	total	
A	615	687	290	160	36	20	0	11	1819	[5]
B	310	386	133	42	57	6	1	29	964	[22]
C	452	411	0	0	0	0	0	30	893	[23]
D	342	456	106	11	45	3	1	12	976	[24]
E	330	448	98	8	45	3	1	11	944	[25]

For ease of integration, we target the Xilinx Zynq XC7Z045 FPGA on the ZC706 evaluation board, running Xilinx "bare metal" software environment. The generated accelerators are accessed from the ARM Cores by using the AXI GP0 port, while the hardware can access the main memory and second level cache using the AXI ACP port. We used a version of the Nymble-RS tools based on LLVM 3.3 as front-end and for machine-independent optimization.

As our approach targets the translation of large CellML problems, five of the largest models from the CellML repository [20] (at the time of writing) are used as test cases in this evaluation. Table 1 lists the models with the number of their FP operations.

In all tests and for all compilers, we allow inexact mathematical optimizations. For Nymble-RS, this includes the use of faithful rounding FP units. If

not stated otherwise, simulations are run on 10 cells, with 1 million micro-step iterations each.

4.2 Design Space Evaluation

Table 2 shows the results when generating accelerators with an increasing number of FP operator units (and thus growing hardware area). Since it gave better results than more recent versions, we used Vivado 2014.1 for synthesis. All timing and area results shown are post-place-and-route.

Table 2. Design space exploration. II=Initiation interval, WCT=Wall Clock Time

Test Case	# FP Units	# FFs	# LUTs	# BRAMs	# DSPs	Schedule Length	II	f_{max} (MHz)	WCT (s)
A	12	27647 (6%)	46335 (21%)	36 (3%)	92 (10%)	319	305	193	15.803
	16	33973 (8%)	56610 (26%)	50 (5%)	134 (15%)	238	229	184	12.446
	20	36620 (8%)	65826 (30%)	56 (5%)	144 (16%)	231	204	164	12.439
	24	38834 (9%)	68752 (31%)	60 (6%)	169 (19%)	226	215	169	12.722
B	12	27713 (6%)	42383 (19%)	27 (2%)	77 (9%)	224	202	202	10.000
	16	30310 (7%)	47743 (22%)	31 (3%)	102 (11%)	212	192	199	9.648
	20	33615 (8%)	52579 (24%)	38 (3%)	121 (13%)	191	179	188	9.521
	24	33532 (8%)	57144 (26%)	37 (3%)	131 (15%)	191	179	182	9.835
C	12	21895 (5%)	37186 (17%)	21 (2%)	30 (3%)	202	186	203	9.163
	16	24101 (6%)	43513 (20%)	24 (2%)	40 (4%)	177	161	204	7.892
	20	27207 (6%)	52324 (24%)	25 (2%)	45 (5%)	158	142	191	7.435
	24	28671 (7%)	55580 (25%)	32 (3%)	55 (6%)	148	132	191	6.911
D	12	27763 (6%)	42814 (20%)	28 (3%)	77 (9%)	179	167	200	8.350
	16	29360 (7%)	46705 (21%)	30 (3%)	87 (10%)	169	158	202	7.822
	20	31269 (7%)	51069 (23%)	29 (3%)	97 (11%)	161	150	190	7.895
	24	33798 (8%)	56588 (26%)	32 (3%)	127 (14%)	158	149	173	8.613
E	12	27650 (6%)	42393 (19%)	34 (3%)	77 (9%)	179	163	201	8.109
	16	29264 (7%)	47269 (22%)	29 (3%)	87 (10%)	166	155	196	7.908
	20	31188 (7%)	50596 (23%)	32 (3%)	97 (11%)	157	147	188	7.819
	24	33586 (8%)	54845 (25%)	32 (3%)	127 (14%)	156	148	188	7.872

As can be seen, increasing the number of FP units generally carries an $f_{\max}$ penalty, often due to slower wiring. Thus, the fastest accelerator will not necessarily be the largest one.

In general, using ca. 25…30% of the Zynq Z7045 device (a mid-sized chip) per accelerator achieves the best execution time. As discussed in Section 2.1, the total simulation performance will ideally scale linearly by employing multiple accelerators, so the remaining FPGA area can be put to good use.

4.3 Computation Accuracy

Table 3 compares the average and maximum relative error introduced by single precision arithmetic (as used in ODoST) to the error introduced by our approach. The errors shown here are computed relative to a IEEE745-compliant reference software execution using double precision and compiled without `-ffast-math`.

The error introduced by single precision becomes rather large when one million (or more) iterations are used for integration. Depending on the required simulation accuracy, single precision may not suffice, whereas our double-precision based approach carries a much smaller average error.

Table 3. Average relative error

	Average Relative Error		Maximum Relative Error	
	single-precision	**our approach**	**single-precision**	**our approach**
A	5.67E-02	2.22E-14	1.37E-00	6.74e-13
B	2.62E-04	1.16E-14	4.78E-03	1.46E-13
C	3.09E-03	1.78E-14	1.75E-02	6.96E-14
E	6.06E-03	3.04E-14	4.09E-02	1.82E-13
D	7.12E-02	9.43E-15	2.93E-01	4.79E-14

4.4 Comparison to State-of-the-Art HLS Tools

In order to evaluate the performance of our tool, we attempted to compare it against other state-of-the-art HLS systems, both academic and industrial.

LegUp: With LegUp being the most prominent academic C-based HLS tool in recent years, it is a natural reference. However, with its recent commercialization, the license terms for LegUp 5.1 prohibit comparative benchmarking, thus limiting our evaluation to the latest open-source version 4.0, which does not carry that restriction. Since LegUp 4.0 does not fully support Xilinx devices, we target the Altera Cyclone V family of chips for comparison.

Unfortunately, a number of issues caused this attempt to fail: The use of the *floor* function often leads to crashes during HLS. To get around this, we temporarily removed the *floor*-calls and were able to proceed to logic synthesis using the Altera Quartus II vendor tools. Here we ran into the problem that Quartus prohibits the use of exp as a module name, which was present in the Verilog RTL generated by LegUp. Manually renaming then leads to multiple "module signcopy not found" error messages. As this module does not seem to have been distributed with the virtual machine image the LegUp authors kindly have made available for experimentation, we gave up here.

Industrial tool: We cannot publish the name of the industrial HLS tool we used as a reference here due to license restrictions. For these experiments, we employed a recent version of a C-based HLS tool targeting Xilinx devices. Logic synthesis was performed using Xilinx Vivado 2017.2, all numbers reported are post-place-and-route.

As before, we manually explored multiple combinations of inlining, unrolling, and pipelining for the input C-code, all expressed using the tool's directives, and report only the best (=fastest) results here. For comparability, we set the upper bound on the number of FP units for each operation type identically to that our own tool uses, computed as discussed in Section 3.2, with an upper bound of a total of 20 FP units. In addition, we permitted the industrial tool the use of one dedicated pow operator, which our tool emulates using log and exp. However, for

unknown reasons, the industrial tool exceeded this restriction (by using 2 or 3 pow units instead) for the test cases marked in Table 4 with an asterisk.

In contrast to LegUp, the industrial tool was actually able to compile the code for all of the models. However, for the largest model (case A, Faville), place-and-route failed due to congestion. Attempts to alleviate this by using specific anti-congestion settings in the Vivado tools failed. When the mapping process did succeed, the generated accelerators where all larger and slower (sometimes by an order of magnitude) than those of our system.

Table 4. Accelerators created by industrial HLS tool, relative to our approach

	FFs	LUTs	DSPs	BRAM	II	f_{max}	WCT (s)
A	85K	134K	494	22	413	P+R failed	
	233%	204%	343%	39%	202%		
B*	80K	84K	578	68	121	84.2	14.3
	237%	160%	478%	179%	68%	45%	151%
C	55K	76K	135	0	169	153.3	11.0
	201%	144%	300%	0%	119%	80%	148%
D*	69K	80K	404	23	1933	84.6	228.5
	222%	156%	416%	79%	1289%	45%	2894%
E*	70K	79K	404	23	1230	83.1	148.0
	224%	156%	416%	72%	837%	44%	1893%

4.5 Performance / Energy relative to CPU

To evaluate the performance and energy efficiency of our approach relative to a fast CPU, we proceeded as follows: We use a fast desktop-class Intel Core i7 6700K CPU running at 4.2 GHz and compile with gcc 5.3.1 with -O3 -ffast-math. On the FPGA side, we employ a Xilinx ZC706 Zynq 7045-based prototyping board. We concentrate on single-core performance, since both the CPU and the FPGA could easily scale to run multiple threads / accelerators in parallel.

We measure the execution time of simulating 10 cells with 1 million iterations each. For the FPGA, we use the 20 FP unit variant of each accelerator, executing at its specific maximum $f_{\max}$. As before, we report wall-clock time for the complete execution, including overhead for HW-SW interfaces.

Power measurements on the Intel CPU are performed using the Running Average Power Limit (RAPL) performance counters. On the FPGA, voltage and current measurements are obtained by directly querying Channel 1 of the on-board Voltage Regulator Module (VRM), which covers the Zynq's programmable logic and the ARM cores (both of which sleep here). In both cases, I/O power consumption is not included. Afterwards, the total amount of energy used for the computation is calculated by multiplying run time with average power consumption. The results are shown in Table 5.

Our FPGA-based accelerators are significantly more energy-efficient than the CPU, in the largest model A saving 96% of energy. On the performance side, the FPGA-based accelerators are generally faster than the CPU, for the large model

Table 5. Execution Wall-Clock-Time (10 Cells, 1M iterations each), Power and Energy Consumption (FPGA vs CPU)

Test Case	A	B	C	D	E
WCT i7 6700K (s)	60.17	10.50	1.45	9.67	8.10
WCT XC7Z045 (s)	12.44	9.52	6.91	7.82	7.82
Speed-Up	4.84x	1.10x	0.21x	1.24x	1.04x
Power i7 6700K (W)	19.1	20.4	22.4	20.0	20.2
Power XC7Z045 (W)	3.6	3.0	3.0	3.1	3.2
Energy i7 6700K (J)	1149.8	214.2	32.5	193.6	163.7
Energy XC7Z045 (J)	44.9	28.9	22.1	24.4	24.7
Energy Reduction	96%	87%	32%	87%	85%

A by almost 5x. The outlier here is model C, which is significantly faster on the CPU than on the FPGA. A closer examination of its code reveals an anomaly: Most of the simulation models have a very *irregular* code structure (e.g., large loop bodies with many different computations). But model C has very similar computations (three multiplications and a subtraction) in more than half of its lines of code. We assume that this code structure did allow the software compiler to perform autovectorization (which is included in -O3), and thus achieves a much higher performance.

4.6 Performance / Energy relative to GPU

We also compare our FPGA result to a GPU implementation. For that implementation, the C-code was used to create a CUDA Kernel. Note that we used the C code generated by cellml-opt, as these optimizations improve the CUDA performance as well. We simulate 100k cells to determine the average rate of cells-computed-per-second on a single NVidia Tesla K80 GK210 GPU. The power is measured using nvidia-smi. These results are shown in Table 6. While our approach has better latency, the Tesla-GPU produces more results per second (as expected for a throughput-architecture such as a GPU). However, the FPGA is more energy efficient (in terms of Joules per cell).

Table 6. Single FPGA kernel vs GK210 GPU

Test Case	Latency [s] for one cell	Throughput [Cells per Second]	Power [W]	Energy per Cell [J]
A on FPGA	1.2	0.81	3.6	4.48
A on GPU	322.3	12.1	138.4	11.46
B on FPGA	1.0	1.05	3.0	2.89
B on GPU	91.3	38.69	131.6	3.41
C on FPGA	0.7	1.35	3.0	2.21
C on GPU	23.9	34.60	132.5	3.83
D on FPGA	7.9	1.27	3.1	2.44
D on GPU	72.0	39.64	145.7	3.67
E on FPGA	7.8	1.28	3.2	2.47
E on GPU	75.3	42.98	147.0	3.42

5 Conclusion and Future Work

We presented a new approach for hardware synthesis of larger CellML models that offers superior latency and energy efficiency compared to CPU and GPU. Furthermore, our specialized HLS tool significantly exceeds the quality-of-results of a state-of-the-art industrial HLS system. The performance and size of the accelerators created by our approach can be flexibly scaled, achieving significant speed-ups in most cases even when dedicating just a quarter of a mid-size FPGA to the accelerator circuit.

To extrapolate the power of our approach beyond the Virtex 7-class devices, which were introduced in 2010, to current generation FPGAs, we have performed an initial experiment compiling and mapping model A to a modern XCVU13P-3 UltraScale+ FPGA. We used a total 16 FP units and achieved an $f_{\max}$ of 316 MHz, which would yield a speed-up of *8.3x* relative to the desktop class CPU in single-accelerator performance. As each accelerator requires only 2.9% of that FPGA's area, an *additional* speed-up could be achieved by tiling accelerators, e.g. 8 accelerators implemented in parallel still reach 282 MHz. This huge potential makes further research on reconfigurable computing for cell simulation highly promising.

References

1. Cuellar, A.A., Lloyd, C.M., Nielsen, P.M.F., et al.: An overview of cellml 1.1, a biological model description language. Simulation **79**(12) (2003) 740–747
2. Yu, T., Bradley, C., Sinnen, O.: Odost: Automatic hardware acceleration for biomedical model integration. TRETS **9**(4) (2016) 27:1–27:24
3. Yu, T., Oppermann, J., Bradley, C., Sinnen, O.: Performance optimisation strategies for automatically generated FPGA accelerators for biomedical models. Concurrency and Computation: Practice and Experience **28**(5) (2016) 1480–1506
4. Bradley, C., Bowery, A., Britten, R., et al.: Opencmiss: A multi-physics & multi-scale computational infrastructure for the vph/physiome project. Progress in Biophysics and Molecular Biology **107**(1) (2011) 32 – 47 Experimental and Computational Model Interactions in Bio-Research: State of the Art.
5. Faville, R.A., Pullan, A.J., Sanders, K.M., et al.: Biophysically based mathematical modeling of interstitial cells of Cajal slow wave activity generated from a discrete unitary potential basis (2009) CellML file: faville_model_2008.cellml (Catherine Lloyd).
6. Miller, A.K., Marsh, J., Reeve, A., et al.: An overview of the cellml API and its implementation. BMC Bioinformatics **11** (2010) 178
7. de Dinechin, F., Pasca, B.: Designing custom arithmetic data paths with flopoco. IEEE Design & Test of Computers **28**(4) (2011) 18–27
8. Oppermann, J., Koch, A., Yu, T., Sinnen, O.: Domain-specific optimisation for the high-level synthesis of cellml-based simulation accelerators. In: 25th Intl. Conf. on Field Programmable Logic and Applications, FPL 2015, London, United Kingdom, September 2-4, 2015, IEEE (2015) 1–7
9. Liebig, B., Koch, A.: High-level synthesis of resource-shared microarchitectures from irregular complex c-code. In: Field-Programmable Technology (FPT), 2016 Intl. Conf. on, IEEE (2016) 133–140

10. Huthmann, J., Liebig, B., Oppermann, J., Koch, A.: Hardware/software co-compilation with the Nymble system. In: 8th Intl. Workshop on Reconfigurable and Communication-Centric Systems-on-Chip, IEEE (July 2013) 1–8
11. Huthmann, J., Mller, P., Stock, F., Hildenbrand, D., Koch, A.: Accelerating high-level engineering computations by automatic compilation of geometric algebra to hardware accelerators. In: 2010 Intl. Conf. on Embedded Computer Systems: Architectures, Modeling and Simulation. (July 2010) 216–222
12. Thielmann, B., Huthmann, J., Koch, A.: Precore - a token-based speculation architecture for high-level language to hardware compilation. In: 2011 21st Intl. Conf. on Field Programmable Logic and Applications. (Sept 2011) 123–129
13. Huthmann, J., Oppermann, J., Koch, A.: Automatic high-level synthesis of multi-threaded hardware accelerators. In: 2014 24th Intl. Conf. on Field Programmable Logic and Applications (FPL). (Sept 2014) 1–4
14. Nane, R., Sima, V.M., Pilato, C., et al.: A survey and evaluation of fpga high-level synthesis tools. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems **35**(10) (Oct 2016) 1591–1604
15. Xilinx, Inc: Vivado Design Suite User Guide – High-Level Synthesis. (2012)
16. Fingeroff, M., Bollaert, T.: High-Level Synthesis Blue Book. Mentor Graphics Corp. (2010)
17. Pilato, C., Ferrandi, F.: Bambu: A modular framework for the high level synthesis of memory-intensive applications. In: Field Programmable Logic and Applications (FPL), 2013 23rd Intl. Conf. on, IEEE (2013) 1–4
18. Nane, R., Sima, V.M., Olivier, B., et al.: Dwarv 2.0: A cosy-based c-to-vhdl hardware compiler. In: Field Programmable Logic and Applications (FPL), 2012 22nd Intl. Conf. on, IEEE (2012) 619–622
19. Canis, A., Choi, J., Aldham, M., et al.: LegUp: High-Level Synthesis for FPGA-based Processor/Accelerator Systems. In: Proc. Intl. Symp. on Field Programmable Gate Arrays (FPGA). (2011) 33–36
20. Lloyd, C.M., Lawson, J.R., Hunter, P.J., et al.: The cellml model repository. Bioinformatics **24**(18) (2008) 2122–2123
21. Detrey, J., de Dinechin, F.: Parameterized floating-point logarithm and exponential functions for FPGAs. Microprocessors and Microsystems, Special Issue on FPGA-based Reconfigurable Computing **31**(8) (2007) 537–545
22. Grandi, E., Pasqualini, F.S., Bers, D.M.: A novel computational model of the human ventricular action potential and Ca transient (2010) CellML file: grandi_pasqualini_bers_2010_flat.cellml (Geoffrey Nunns).
23. Hornberg, J.J., Binder, B., Bruggeman, F.J., et al.: Control of MAPK signalling: from complexity to what really matters (2005) CellML file: hornberg_binder_brugge-man_schoeberl_heinrich_westerhoff_2005.cellml (Catherine Lloyd).
24. Iyer, V., Hajjar, R.J., Armoundas, A.A.: Mechanisms of Abnormal Calcium Homeostasis in Mutations Responsible for Catecholaminergic Polymorphic Ventricular Tachycardia (2007) CellML file: iyer_2007_ss.cellml (Penny Noble).
25. Iyer, V., Mazhari, R., Winslow, R.L.: A computational model of the human left-ventricular epicardial myocyte (2004) CellML file: iyer_mazhari_winslow_2004.cellml (Steven Niederer).