Sphery vs. shapes, the first raytraced game that is not software

Victor Suarez Rovere and Julian Kemmerer

We present *Sphery vs. shapes*, a raytraced game that's written in C and translated to a circuit that doesn't have any CPU, requiring few silicon resources that run at an outstanding power efficiency.

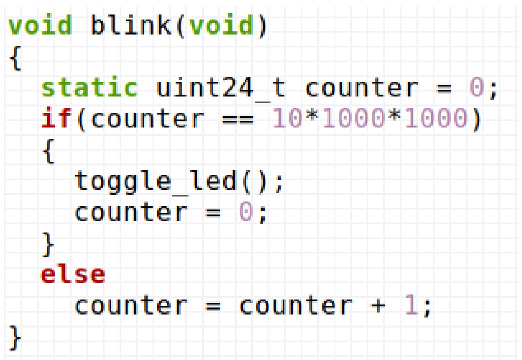
Background

When CPUs were invented, one of the main problems to solve was how to do complex data processing, such as math operations, in replacement of analog counterparts. CPU design evolved from the Turing machine, using a stored program with progressively more complex integrated circuit design techniques to accelerate the operating rate, but always as a sequence of instructions. On the other hand, digital electronic circuit components themselves are able to do some data processing, indeed it's known that enough "NAND" gates are able to implement any kind of digital data processing, a full CPU design can be expressed just as a network of such gates.

Data processing based on CPUs and stored programs became the first choice for general purpose computation, even if dedicated hardware would perform better. We believe that a main cause of that is this: the available tools to design software are more widespread, faster, and easier to use than the tools required to do complex data processing without a CPU.

We present here a workflow to achieve complex data processing using just digital circuits expressed in a known language that's a subset of C/C++. We show as an example of a complex design, a realtime raytraced game. The game's pixel rendering and animation logic is based on floating point and vector math operations. All of the game code is expressed using a clean syntax that translates directly to a digital circuit. The current target of this design is a FPGA board with Full HD digital video output, and the workflow also allows running the game in realtime on a regular PC using the unmodified source. This allows for much faster development-test iterations than with traditional hardware design tools. For the same workload, the computing efficiency resulted in more than 50X better than using a modern CPU, in a chip an order of magnitude smaller.

Blinky: the "*Hello, world!*" of hardware design



```c
void blink(void)
{
  static uint24_t counter = 0;
  if(counter == 10*1000*1000)
  {
    toggle_led();
    counter = 0;
  }
  else
    counter = counter + 1;
}
```

As one of the most simple systems that can be implemented in hardware, we have the blinking led.

One interesting property of the *blinky* code is that it can not only be converted to a circuit, “is synthesizable”, but it can also be compiled with a regular compiler. If we repeatedly run the code in a microcontroller, by implementing the `toggle_led` function for inverting the state of a pin, the same code will accomplish the intended function. Then for example, in an FPGA, the function will run in a faster and more time-predictable way as a digital circuit. Another advantage of using an FPGA is that multiple of such cores can run in parallel, so the designer can add as many as needed.

More complex designs can also be implemented. For example a UART could also run in a microcontroller, working as a “software-only” implementation. We provide a working example:

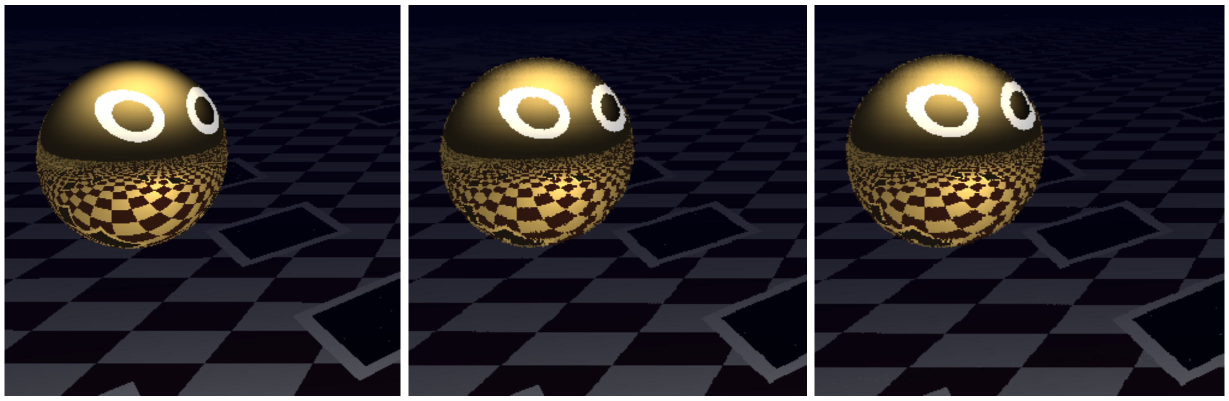
```
uint8_t receive_byte()
{
  // Wait for start bit transition
  // First wait for UART_IDLE
  uint1_t the_bit = !UART_IDLE;
  while(the_bit != UART_IDLE)
  {
    the_bit = get_uart_input(); // Read pin
    wait_clk();
  }
  // Then wait for the start bit start
  while(the_bit != UART_START)
  {
    the_bit = get_uart_input(); // Read pin
    wait_clk();
  }

  // Wait for 1.5 bit periods to align to center of first data bit
  wait_clks(UART_CLKS_PER_BIT+UART_CLKS_PER_BIT_DIV2);

  // Loop sampling each data bit into 8b shift register
  uint8_t the_byte = 0;
  uint16_t i;
  for(i=0;i<8;i+=1)
  {
    the_bit = get_uart_input(); // Read pin
    // Shift buffer down to make room for next bit
    the_byte = the_byte >> 1;
    // Save sampled bit at top of shift reg [7]
    the_byte |= ( (uint8_t)the_bit << 7 );
    // Wait a full bit period so next bit is ready to receive next
    wait_clks(UART_CLKS_PER_BIT);
  }

  // Don't need to wait for stop bit
  return the_byte;
}
```

Full source at https://github.com/JulianKemmerer/PipelineC-Periph/blob/main/uart_bit_bang/pipelinec_app.c

Current workflow

The traditional languages to do hardware design are Verilog and VHDL. They are widely supported by FPGA vendors and also used in the chip design industry. We find many drawbacks to using such languages: they're quite low level, not very widespread and known like the C language, and the related tools are usually not lightweight nor open source. Additionally, the tools are not as fast as an advanced C compiler. For such reasons, when facing a complex design, it's usual in the hardware design industry to first make a model of the required processing in C, and then manually translate it to VHDL or Verilog. With the additional time required in translation, the chance of incorporating human errors increases and makes it almost impracticable to keep both versions in sync. This is solved by an automatic C to Verilog translation.

The user is able to just compile and run the program with a regular compiler, as a first evaluation to test design ideas and fix bugs. Data types are selected as compilation options to use the fastest available for the CPU, so running time is minimized. In the example project, we do this as a simple "`make sim`" command, in which case all data types are of type float and vectors of such. It takes just a second to compile until the game is started and playable. When running "`make gen`", the source code is parsed and a C file is generated, where some data types are implemented as fixed points, and floating point types use a mattissa truncated to fewer bits, to more accurately represent the data types used for the hardware implementation. Hardware resources are limited and a good balance between accuracy and resource usage is needed. In about 4 seconds after the code is compiled, the game starts and runs at 60 FPS at 640x480 or 34 FPS in Full HD with the truncations active.


Simulation stages. Leftmost: CPU, all floats. Center: CPU with fixed types and truncated floats. Right: Verilator

In the above image, it can be appreciated that the full float version has smoother edges in the ball and eyes than when using lower width data types: some distortions (quantization noise) appear as a result of the fewer amount of bits. The tool greatly helps in finding precisely how many bits are needed; in the game project, the amount of bits were selected in a way that when playing the game, the distortions are not noticeable. Being able to run and play the game during development is something that is not possible with traditional simulation tools for hardware design since they are orders of magnitudes slower than using a compiled source. Thus, it would have taken an nearly impracticable amount of time to do a Verilator simulation for all the frames, and then try to deduct how the distortions would be perceived in real time as the ball is moving. Instead, it was easy and fast to test with the presented simulator capabilities, and with no changes to the code needed.

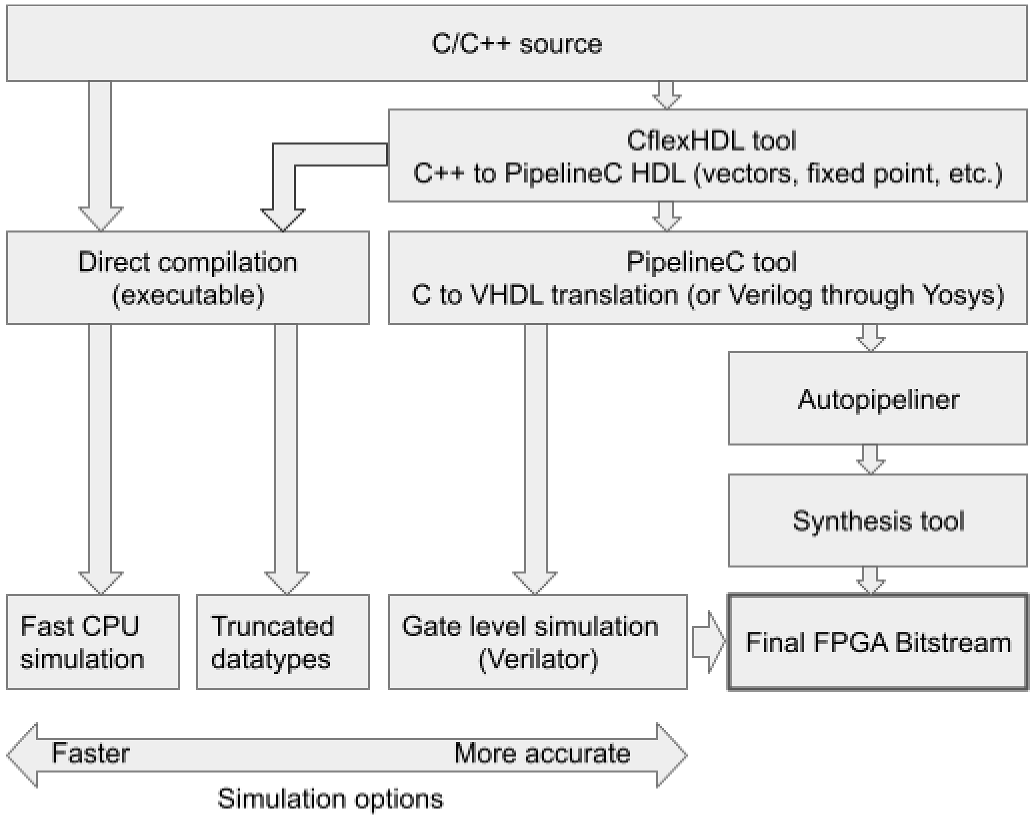
After the compiled programs are run and results accepted, the sources can be converted to Verilog using one of the two possible options: to do a lower level simulation, or to do a synthesis to the target silicon (currently a FPGA device). The advantage of the first option is that it's offered as a second opportunity of having a preview on how the circuit will behave, in a faster way than doing "full synthesis and auto-pipelining", and so a quicker way of discovering any issues. We do that by executing "`make verilator`", which takes ~2 minutes to do the translation to Verilog and until the first complete frame is simulated, then it's able to simulate a new frame every ~9 seconds at 640x480 resolution.

After the Verilator simulation is run, the results obtained should accurately match how the hardware performs. The next step is to do a "full synthesis" that is able to split long operations like a floating point addition in smaller steps using a pipeline: the user specifies a target clock frequency (148.5MHz for full HD video) and the tool splits calculations into the minimal amount of pipeline stages needed to meet the set speed. For this, we run "`make fullsynth`" that includes gathering timing information from the FPGA-specific synthesis tools, and iteratively refining where to split operations until an optimal point is met. This is fully automatic, and for the raytracer project it takes ~3 hours on a modern PC, leaving the final binary configuration (bitstream) ready to be uploaded to the FPGA.

The following summarizes all the options that the workflow offers to get faster and/or more accurate results while in development:

	Build command	**Build time**	**Speed @1080p**
Fast CPU simulation	*make sim*	1s	60-86 FPS
Precise CPU simulation	*make gen*	5s	40 FPS
Gate level simulation	*make verilator*	1min 50s	50s per frame

To run in hardware, the command is `make [board-name]`. It takes about 60-90 minutes (depending on clock frequency) to do the auto-pipelining and bitstream generation. As hardware designers know, those times are usual for most synthesis tools so to accelerate development we offer the faster above described options. In the graph below, the workflow details are depicted:


C/C++ source
CflexHDL tool
C++ to PipelineC HDL (vectors, fixed point, etc.)
Direct compilation (executable)
PipelineC tool
C to VHDL translation (or Verilog through Yosys)
Autopipeliner
Synthesis tool
Fast CPU simulation
Truncated datatypes
Gate level simulation (Verilator)
Final FPGA Bitstream
Faster
More accurate
Simulation options

Data types and algorithms

The following data types are natively supported by the tool:

- Signed and unsigned integers of arbitrary widths (1 to 64 bits, 128 bits TBD)
- Fixed point types (up to 32 total bits)
- Floating point types (8 bit exponent and 1 to 23 bits mantissa, other configurations TBD)
- 2D, 3D, and 4D vectors of fixed and floating point types
- Structures using any supported data type

Simple syntax of the like "`d=a*b+c`" is possible between vector and scalar types. An example of this is in the raytracer code:

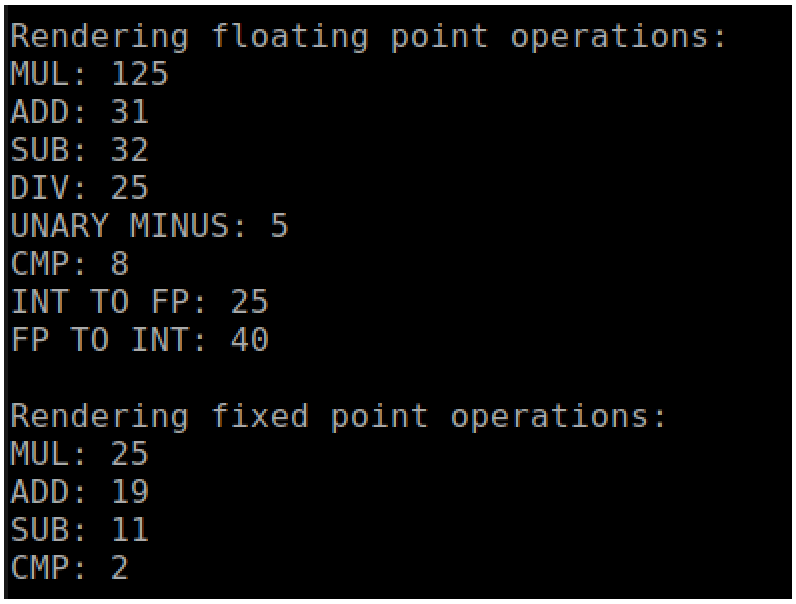
```
hit_out ray_sphere_intersect(vec3 center, point_and_dir hitin)
{
  vec3 rc = hitin.orig - center;
  float b =  dot(rc, hitin.dir);
  float c =  dot(rc, rc) - SPHERE_RADIUS*SPHERE_RADIUS;
  float diff = b*b - c;
```

Full source at https://github.com/JulianKemmerer/PipelineC-Graphics/blob/main/tr.cpp

As shown in the function prototype, a 3D float vector data type 1st argument is used, then structures as the 2nd argument and return type. In the first line of the function a subtraction operation is done between 3D float vector types, and in the following line a call to a function taking two 3D float vectors and returning a float scalar is used. This kind of flexibility and clean syntax makes for easy development of math-intensive algorithms used in graphics or general DSP.

Note how the arguments are "passed by value". For performance reasons, an 'IN' macro can be applied to the datatype intended for passing by reference in case of running a regular compiler, but remains passed by value in case of targeting hardware.

The simulator can count number of operations done with fixed and floating point types and report them, so when the program changes the develop can know about more or less resources used

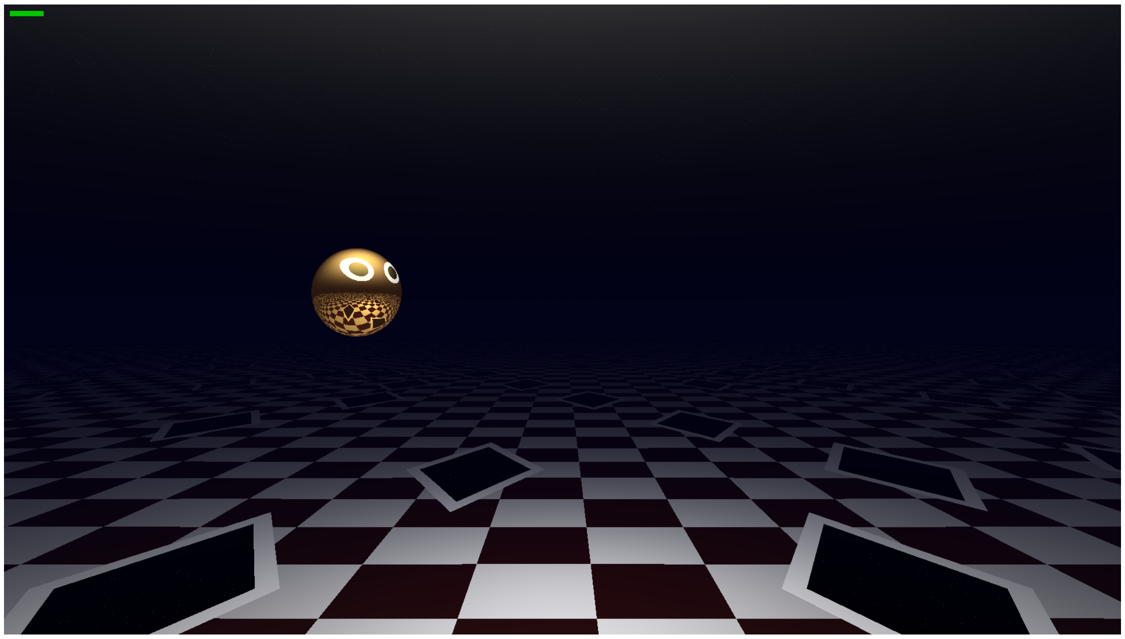


```
Rendering floating point operations:
MUL: 125
ADD: 31
SUB: 32
DIV: 25
UNARY MINUS: 5
CMP: 8
INT TO FP: 25
FP TO INT: 40

Rendering fixed point operations:
MUL: 25
ADD: 19
SUB: 11
CMP: 2
```

Float and fixed point types usage report

Sphery vs. shapes: game specifics

The most complex project we decided to implement was a raytraced game, since it demands many math operations using non-integer data types and some 3D vector math, plus the need to do data processing in realtime, reactive to user input.



Sphery vs. Shapes game video: https://youtu.be/LaSvbPOP9Nc

Some math functions were needed, including: floating point addition, subtraction, multiplication, division, reciprocals, square root, inverse square roots, vector dot products, vector normalization, etc. Fixed point counterparts were also used for performance reasons and to make the design easier to fit in the target FPGA, with the corresponding conversions to and from other types (integer and floats).

The design was done in a “chasing the beam” fashion, that is, a pixel color is calculated at the same rate that the monitor needs it for proper displaying, so no memory is required as a framebuffer. That means a 148.5MHz pixel clock rate at the selected resolution, so a new pixel calculation needs to be done in less than 7ns from the previous one, in a scene that includes: a ball that reflects a checkerboard floor, a floor that also reflects the ball, background with stars at pseudorandom positions, colors that depends on distance to the a light source, fog effect and holes in the floor (each one with a solid border) with probability of appearing with increasing difficulty as the game progress. Additionally, game *physics* emulates gravity and elastic bouncing with the floor. The ball includes animated eyes that move with height, blink in a non-periodic fashion, and slightly wide open in response to speed. The ball itself has a “temperature” property that represents friction with a change in color in response to the amount of time on the floor, and slowly cools while in the air. A score bar guides the player in its progress who is able to both lose and win the game, with a visual effect for both cases. A camera following algorithm is active at all moments to determine a good perspective to view the scene. All those features are implemented, as the whole project, just as logic that can be directly translated to a circuit with no CPU or software program, using a medium-sized, off-the-shelf FPGA chip and board.

The target resolution is Full HD 1920x1080 at 60Hz refresh rate, and the game itself is able to run compiled in a CPU at the same frame rate by using multiple concurrent CPU cores, to ease development and real-time testing.

A long pipeline of 482 stages (automatically calculated) is used to achieve the target performance of one new pixel output every ~7ns. The pipeline works in a way that when an output pixel is ready, the previous calculation stage is working on the next pixel, and successively the same scheme applies for all previous stages, computing for 482 pixels simultaneously. In that way, each individual stage of calculations runs in parallel so processing efficiency is very high, with few parts of the circuit being idle. Indeed, all parts of the circuits are active at all times. This is in contrast to a typical CPU, for example a 32-bit microcontroller, where each instruction that is executed utilizes large parts of the chip just to generate *one single 32-bit result* at each clock, to be stored on a general purpose register or memory. In the pipeline implemented in this example, about 37.4Kbits of registers are all updated at each clock cycle with an associated circuit doing useful work for the next pixel calculation.

The system is hard realtime too, there's basically no jitter in the output nor a way to stop it as could happen with a CPU doing a failed memory access to code or data.

Comparing the pipelined system with a CPU architecture

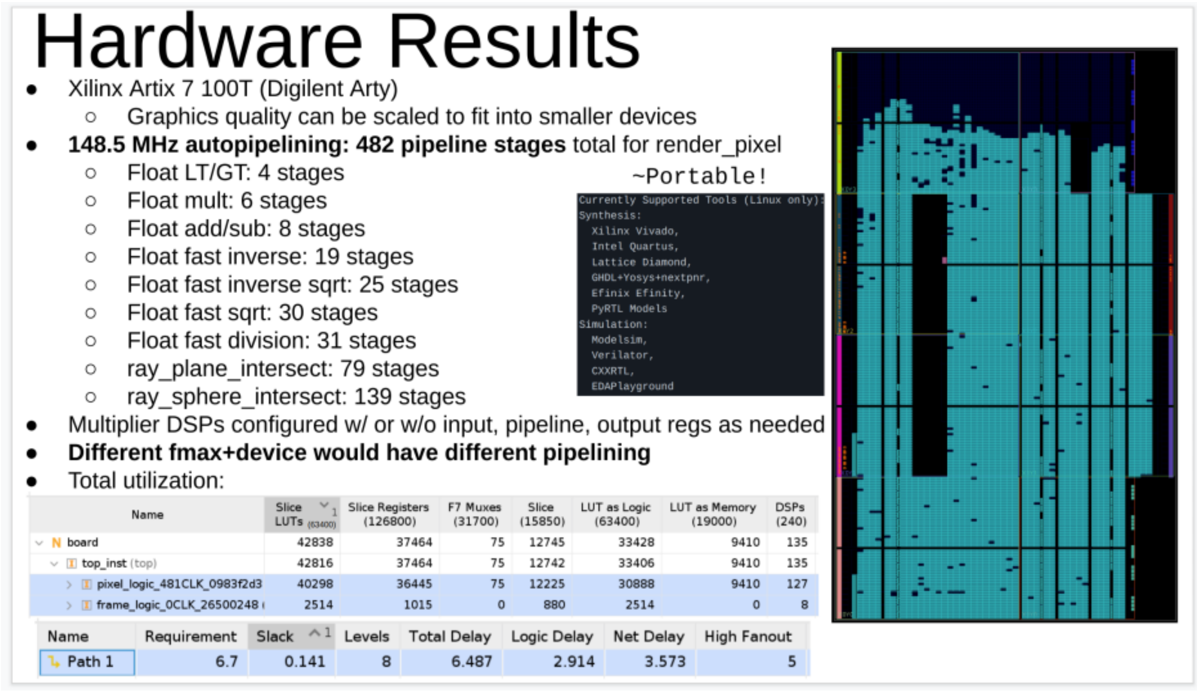
The workload required to raytrace the 1920x1080 pixels image was on the order of 300 arithmetic and logic operations over integers and about 150 floating point operations per pixel (including comparisons and optimized operations on just the exponent), all running at 148.5MHz rate, so about 67 Gop/s not counting data movements.

We compared our design with a quite powerful and modern CPU-based platform: a Ryzen™ 4900H 8-core/16 threads 64-bit CPU @ up to 4.4GHz clock, fabricated on a 7nm silicon process, at 45W TDP (running Linux in a desktop PC). Using the SIMD vector float processing offered by the CPU we see performance doubled over using some calculations in fixed points, as used in the FPGA. Indeed, the vector extensions and all-floating point processing was required to reach 60 FPS at the same resolution as in the FPGA. The C code for both targets was unchanged, since we just use a different *typedef* for data types to select between float or fixed types, while using the same syntax for math operations by using C++ operator overloading. So the processing source code remains exactly the same for running it interchangeably on the CPU or in the FPGA.

Comparing a FPGA with a CPU is not an easy task: you can always use a bigger and more modern FPGA than the one we used, but it was selected not to be too large on purpose to make the test platform more accessible. Also, you can also always use a CPU with more cores, bigger caches, faster clocks, smaller transistors, etc. But each time you add more silicon real estate to a computing platform in order to achieve higher performance, that naturally makes the system require more power. Because of that, we think that evaluating the power consumption per operation is a good way of trying to make a fair comparison.

When running the game on the CPU for measurements, the system uses 97.5% of the CPU (all cores/threads are active). It ran at 88ºC with the fans at their highest speed, and consumed 33W as reported by our simulator, which accesses and reports the energy meters internal to the CPU. For any excess time after the frame is rendered, the energy metering was stopped. The average clock rate of the 16 threads were automatically set at 4.220GHz (96% of max clock), thus equivalent to 67.5GHz of a theoretical single-thread CPU. We remark that the power required for the external DDR4 memory bank is not taken into account, nor the energy needed to run fans, but those devices usually take several watts combined.

On the other hand there's our FPGA platform: a medium-sized (~100K logic elements) and low cost chip from same vendor as the CPU, not requiring active cooling, and fabricated on a much older silicon process of 28nm instead of the 7nm of the CPU (packing just about 13% transistor per mm2). Considering a die size of 50% for the FPGA case we estimate 15X less transistors (and other silicon features) on the FPGA. The clock rate was set to 148.5MHz matching the requirements for the video generation, and the pipeline resulted in 482 stages. A total of 135 native multipliers run in parallel, used by floating point or integer/fixed point operations. This is detailed on the following figure:



Hardware Results

- Xilinx Artix 7 100T (Digilent Arty)
  - Graphics quality can be scaled to fit into smaller devices
- **148.5 MHz autopipelining: 482 pipeline stages** total for render_pixel
  - Float LT/GT: 4 stages
  - Float mult: 6 stages
  - Float add/sub: 8 stages
  - Float fast inverse: 19 stages
  - Float fast inverse sqrt: 25 stages
  - Float fast sqrt: 30 stages
  - Float fast division: 31 stages
  - ray_plane_intersect: 79 stages
  - ray_sphere_intersect: 139 stages
- Multiplier DSPs configured w/ or w/o input, pipeline, output regs as needed
- **Different fmax+device would have different pipelining**
- Total utilization:

~Portable!

```
Currently Supported Tools (Linux only):
Synthesis:
  Xilinx Vivado,
  Intel Quartus,
  Lattice Diamond,
  GHDL+Yosys+nextpnr,
  Efinix Efinity,
  PyRTL Models
Simulation:
  Modelsim,
  Verilator,
  CXXRTL,
  EDAPlayground
```

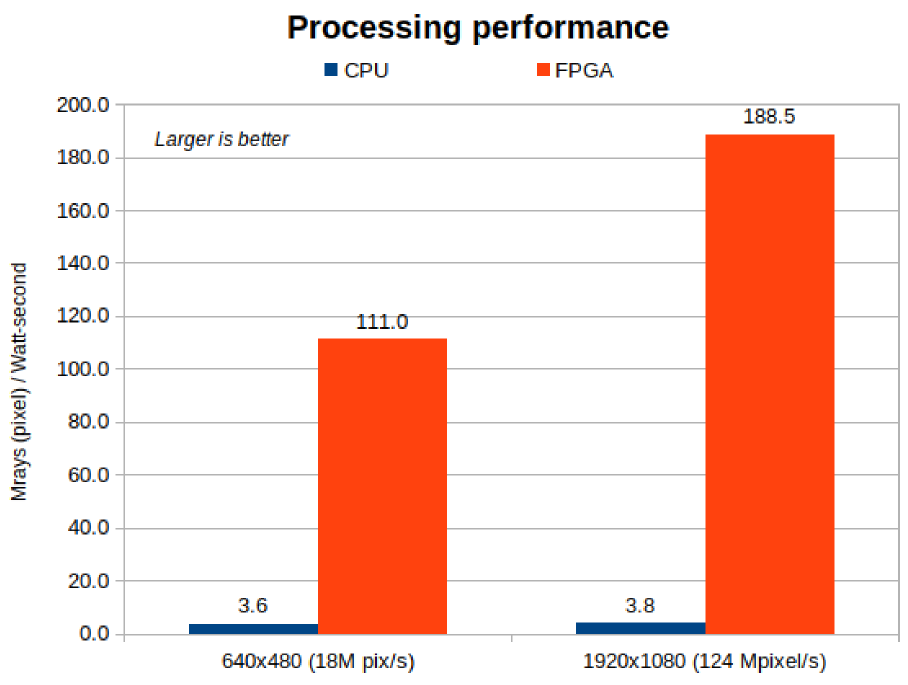
Name	Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	Slice (15850)	LUT as Logic (63400)	LUT as Memory (19000)	DSPs (240)
board	42838	37464	75	12745	33428	9410	135
top_inst (top)	42816	37464	75	12742	33406	9410	135
pixel_logic_481CLK_0983f2d3	40298	36445	75	12225	30888	9410	127
frame_logic_0CLK_26500248	2514	1015	0	880	2514	0	8

Name	Requirement	Slack	Levels	Total Delay	Logic Delay	Net Delay	High Fanout
Path 1	6.7	0.141	8	6.487	2.914	3.573	5

Synthesis results

As shown, the full pipeline required 37Kbits of flip flops, corresponding to an average of 83 bits per operation, constituting a theoretical peak internal bandwidth of 5.4Tbit/s. Computing the 482 stages/~operations simultaneously at 148.5MHz is roughly equivalent to 71.5GHz clock rate in serial form, a number that closely matches the theoretical CPU case.

Even with all these realtime compute demands, the power required by the FPGA core was just 660mW, and the chip stayed barely warm. So, our FPGA system, with an order of magnitude less silicon resources, resulted in 50X less power consumption than a modern CPU for the same workload, or equivalently a processing performance improvement at same power, as depicted in the graphics below:



We expect the efficiency gains to improve up to 6X when using a more modern FPGA at 7nm (where Dennard's law seems to hold at a set speed [1]), and even higher if the digital design were to be implemented in an ASIC, something that we plan to test.

[1] https://bjpcjp.github.io/pdfs/chips/SEMICON_Taiwan_2016_collaert.pdf

Software that constitutes the toolchain and demo

To complete our tool workflow, we use the following: `PipelineC`, a C-like HDL that provides C- to VHDL conversion and auto-pipelining, it's based on the `pycparser` parser. We also use the `CflexHDL` project, which provides a C++ to C parser/generator, fixed point and vector types using clean syntax based on operator overloading, plus some optimized algorithms like division. The implemented game is called `Sphery vs. Shapes`, it's written in C++ and carefully optimized for being compiled and/or synthesized with the mentioned tools.

We also use common developer tools like the `Clang` and `GCC` compilers, the `makefile` tool, the `LiteX` project for FPGA SoC generation and board support, `Yosys` and `GHDL` for Verilog/VHDL processing, the proprietary FPGA synthesis tools for generation of the bitstreams, and the `openFPGA` tool for uploading the bitstream to the board.

Applications and benefits

Our design and proposal has many valuable features, including:

It meets a hard realtime requirement, since the jitter on the timing of results being generated is in the order of picoseconds. The data processing is very fast thanks to the parallel processing provided by the automatic pipelining.

Power requirements are quite low when compared to a desktop CPU architecture. FPGAs like the ones used here are of smaller size and weight, having fewer thermal design requirements too, avoiding the space and power required to run fans.

Finally, we provide the easy syntax of a known language like C and the possibility to run algorithms compiled fast, saving on developer time and resources. Since no manual translation of models to VHDL/Verilog is required, the code is easier to update and maintain.

Many industries will benefit from some or all those features, as we detail below:

Let's take the aerospace field: they need to keep the power budgets really tight, weights low, and space small, as opposed to optimizing for raw processing speed. The technology would be of interest too for always-on sensor systems where high power efficiency is a must.

For industrial control, the ability to do closed loop processing in hard realtime makes for more reliable systems. Indeed as there are no CPU interpreting instructions, there is no execution to be interrupted, jitter is very low and constant. Access to math operations are provided in a simple way, and as such, control algorithms are easier to develop and adapt. The technology provides significant benefits for applications that need to implement DSP algorithms, for example: software defined radio, sonar, radar, and audio processing. In these fields there are plenty of algorithms already written in C that would be easy to adapt, if not capable of being run as-is.

For the case of Virtual Reality, a wireless device allows user movements but needs to operate on battery power alone, where low power is a must. Similarly to aerospace applications, low weight and small form factors are highly appreciated too. It also benefits Augmented Reality applications since realtime input images can be processed at low latency alongside the video output.

The technology also sees a good fit for networking applications, where packet filters are routinely adapted from C implementations for CPUs. Once the code is working as required, it can be run more efficiently on a FPGA with lower latency and energy costs. Overall, any datacenter could see benefits from the lower power usage, as it puts less demands on cooling requirements.

In the case of information security and cryptography, it's important to note that the system is fixed latency, in that all operations take the same amount of time, this makes timing based attacks infeasible. Additionally, since no stored instructions are executed there’s no requirement of an addressable memory at all, and the attack surface on internal data becomes negligible. For expensive cryptographic operations, getting faster results and using lower power also makes the technology attractive to be run in FPGA equipped datacenters, at a lower cost.

Future work

Our continued development includes some ongoing work and planned tasks like: Implementing examples for all the above referenced industries, and a RISC-V CPU + simulator from the same code. We also will continue adding C++ template syntax to better define widths of types and operations. Additionally we are testing other FPGA chips to measure efficiency in more cases. This includes chips that support open source synthesis and bitstream generation tools. In each case trying to push the clock frequency to the maximum while keeping resource usage at a minimum - as we are accustomed to doing all along.

For the demos presented here (game and peripherals) we plan to do an ASIC design, with open source tools and do a tape out and using open source silicon IP, too.

Conclusions

We show a ready-to-use toolchain for hardware design that greatly accelerates development time by using fast simulators at different stages, based on a known programming language syntax. The code can be translated to a logic circuit, run on a off-the-shelf CPU, or on a microcontroller to develop hardware/software peripherals without changes to the code. We showed an example of complex processing by writing a game that implements the usual math operations for raytracing applications, with a clean syntax. Since we apply an automatically calculated -and possibly long- pipeline, the system is capable of performing very well even compared to powerful modern CPUs, but using smaller and embeddable chips, at low power.

The results we obtained are readily reproducible, as materials are easy to obtain and not expensive. All the tools and sources are easily accessible online for recreating and validating this work.

About the authors

This work is a result of the tight interactions between Julian Kemmerer (@pipelinec_hdl) and Victor Suarez Rovere (@suarezvictor) during almost a year.

Victor Suarez Rovere is the author of CflexHDL tool used in this project (parser/generator and math types library) and of the Sphery vs. Shapes game. He's a software and hardware developer and consultant experienced in Digital Signal Processing, mainly in the medical field. Victor was awarded the first prize in the Argentine National Technology contest, a gold medal from WIPO as "Best young inventor" and some patents related to a multitouch technology based on tomography techniques.

Julian Kemmerer is the author of the PipelineC tool (C-like HDL w/ auto-pipelining) used in this work. He earned a Masters degree in Computer Engineering from Drexel University in Philadelphia where his work focused on EDA tooling. Julian currently works as an FPGA engineer at an AI focused SDR company called Deepwave Digital. He is a highly experienced digital logic designer looking to increase the usability of FPGAs by moving problems from hardware design into a familiar C language look.