A case for adopting FPGA development in games: accelerating a position-based cloth simulation with Vitis HLS C

Claudio Cambra

January 12, 2022

Abstract

This paper outlines the dual implementation of a position-based cloth simulation accelerator using standard C++ and Vitis HLS. We explore how an initial implementation in a common language in the development of interactive mediums can be transferred into Vitis HLS and be synthesised for use in, and be accelerated by, an FPGA.

The proposed synthesised design should help offload physics calculations from the CPU to the dedicated accelerator, helping make the case for similar use of accelerators in increasing the number and fidelity of physics-enabled objects in virtual worlds. The accelerator implemented in this paper accelerates a cloth simulation which employs the Position-Based Dynamics algorithm proposed by Matthias Muller et. al.

Despite the inherent inefficiencies of such an approach, synthesised latency data from this implementation shows that one can expect a halving of physics-processing latencies across a variety of mesh sizes, with the benefits increasing with larger mesh sizes. The conclusion is that despite the performance being left on the table by a development approach that is not hardware-first, developers can expect significant performance gains from even inefficient FPGA acceleration.

1 Introduction

With the growing demand for realistic virtual representations of the real world, and with the advent of VR as an emerging market for consumers, physics simulations have become increasingly sophisticated in interactive entertainment products.

This has meant two things. The first, is that more and more elements in a virtual world must react convincingly to direct and indirect interaction. Users expect a virtual world to react like the real world, and as they do in the real world, they would like to pick up, throw, push, and pull virtual objects. The second is that the push for realism has also meant an increasing sophistication in other aspects of these virtual worlds – particularly in terms of lighting effects, texture quality, model details, and more. This presents a potential problem for the status quo, which at the moment relies on the GPU being saturated by rendering-related tasks, with most physics processing being done by traditional CPUs.

Traditional CPUs are not particularly well suited to such a task, however. Some efforts since the late 2000s and early 2010s have attempted to focus physics processing on the GPU. Earlier still there were physics processing cards, such as AGEIA's PhysX cards, that promised to accelerate physics processing in real-time 3D environments. Unfortunately, these acceleration methods did not gain significant traction due to a lack of consumer demand for advanced physics simulations, steep pricing, and the requirement for specialised development tools to take advantage of this hardware.[1]

This paper proposes that acceleration of physics processing should be performed by flexible, affordable, and standardised Field Programmable Gate Array (FPGA) devices programmed with High Level Synthesis (HLS) tools. FPGAs are integrated circuits that can be reconfigured during runtime, allowing them to be repurposed for different functionality as needed. FPGAs can be used to accelerate algorithms, such as those used in physics calculations that are traditionally done by the CPU. High level synthesis tools make the task of implementing said algo-

[1] Wilson, *Exclusive: Asus debuts Ageia PhysX Hardware.*

rithms on an FPGA much more straightforward as the hardware design that is mapped to the FPGA is synthesised from C/C++ code, which is significantly easier to read and write than Hardware Definition Languages (HDLs) like Verilog.

To exemplify how a game's physics simulation might be accelerated on such a device, and how it might be developed, this paper explores an implementation of a cloth simulation using a position-based approach. The position-based dynamics algorithm proposed by Muller et. al. is an alternative to traditional force-based physics calculations in computer graphics, instead working directly on the positions of objects' vertices. This algorithm has high potential for parallelisation and for pipelining, thanks to its structure of several loops which sequentially modify the position of objects' vertices.[2]

Contributions: This appears to be the first HLS-based implementation of a position-based simulation. This is likely due to the fact that the majority of physics-based simulations today are force-based – existing accelerators focus on accelerating parts of these force-based calculations. Additionally, this seems to also be the first paper exploring the development process for non-hardware-developers, starting from an existing C++ implementation and converting it to a synthesised HLS design. Such an approach has not been taken before, likely due to the performance penalty incurred by not following an approach that best fits the hardware.

Outline: In Section II, we explore the position-based dynamics algorithm in more detail. In Section III, we explore previous works that have tackled physics acceleration using FPGAs. In Section IV, we provide detail about how the position-based dynamics algorithm is implemented in HLS C/C++ and how it differs from a parallel implementation in standard C++. In Section V, we present experimental results and analysis of the design of our physics accelerator as well as the advantages and disadvantages presented by the use of HLS in this project.

A copy of all the relevant source code can be found on GitHub.[3]

2 Background

The cloth simulation implemented in this paper uses a position-based algorithm to process the movement and reaction to collision of the cloth mesh. The position-based dynamics algorithm is defined by Muller et. al. as a 17-line pseudocode algorithm. This algorithm includes each of the steps needed for initiating and performing a position-based physics simulation.[4]

Figure 1: From Muller et. al. Position based dynamics, 2007

(1) **forall** vertices i
(2) initialize $\mathbf{x}_i = \mathbf{x}_i^0, \mathbf{v}_i = \mathbf{v}_i^0, w_i = 1/m_i$
(3) **endfor**
(4) **loop**
(5) **forall** vertices i **do** $\mathbf{v}_i \leftarrow \mathbf{v}_i + \Delta t w_i \mathbf{f}_{ext}(\mathbf{x}_i)$
(6) dampVelocities$(\mathbf{v}_1, \ldots, \mathbf{v}_N)$
(7) **forall** vertices i **do** $\mathbf{p}_i \leftarrow \mathbf{x}_i + \Delta t \mathbf{v}_i$
(8) **forall** vertices i **do** generateCollisionConstraints$(\mathbf{x}_i \rightarrow \mathbf{p}_i)$
(9) **loop** solverIterations **times**
(10) projectConstraints$(C_1, \ldots, C_{M+M_{coll}}, \mathbf{p}_1, \ldots, \mathbf{p}_N)$
(11) **endloop**
(12) **forall** vertices i
(13) $\mathbf{v}_i \leftarrow (\mathbf{p}_i - \mathbf{x}_i)/\Delta t$
(14) $\mathbf{x}_i \leftarrow \mathbf{p}_i$
(15) **endfor**
(16) velocityUpdate$(\mathbf{v}_1, \ldots, \mathbf{v}_N)$
(17) **endloop**

The key idea behind position-based dynamics can be summarised into three of the algorithm's sections. Line 7 of the algorithm involves the estimation of the new positions for each of an object's vertices. This is calculated by taking the current position of a vertex and then changing its position according to the vertex' current velocity. This estimate is set as the tentative new position for the vertex.

Lines 9 to 11 then modify these positions in an iterative fashion, ensuring that they satisfy the constraints between vertices. Constraints are the rules that dictate the distance between vertices, and they can be relaxed or tightened depending on the type of material we are seeking to emulate in the simulation. This is important for collisions, as the rules set by our constraints will affect the manner in which the object being simulated will react to a collision.

These constraints and estimated positions are then used in lines 13 and 14 in order to correct the estimated positions of vertices to conform to

[2] Müller et al., "Position based dynamics".
[3] https://github.com/claucambra/PositionBased-ClothSim
[4] Müller et al., "Position based dynamics", p. 3.

the constraints that we have established. This step is crucial for preventing two objects from penetrating each other, as when penetrations are detected between objects the penetrating vertices are adjusted to prevent this. It is also in this step that the vertices' velocities are adjusted.

The algorithm also includes steps to handle external forces such as gravity, damping velocities, and so on.

These steps are performed iteratively on the vertices of the object, progressively changing the positions of each of its vertices and thus simulation the motion and collision of said object.

3 Related works

There are some studies that explore the utility of FPGAs in accelerating certain game-related workloads, but few that focus specifically on real-time physics simulations. Woulfe and Manzke provide one of the most specific studies, exploring the application of a numerical integrator accelerator which could potentially be incorporated into a broader FPGA physics processing pipeline. However, this numerical integrator is only one part of the physics processing. Their implementation did not attain the performance achievable on a CPU, though the authors claimed that this would eventually be possible through greater parallelisation. Fundamentally, Woulfe and Manzke's approach is different from this paper's; the authors examined a component from a physics simulation and sought to create an optimised version, writte in RTL, accelerated by an FPGA; this paper seeks to accelerate a complete and existing simulation in C++ to an FPGA design synthesised from HLS C.[5]

A very similar approach was taken more recently by Toupas, Brokalakis and Papaefstathiou, who also implemented specific components of a physics engine – in this case, the Projected Gauss Siedel (PGS) and Integration algorithms which they put forward as the two most demanding parts in the simulation loop of the engines they studied. Unlike Woulfe and Manzke, their implementation was done using the Xilinx HLS tools (like this study). Their results showed that their proposed architecture outperformed a GPU implementation of these algorithms by up to 2.2x, while being 44x more energy efficient.[6]

Nery and Sena presented a study exploring the implementation of a co-processor to be used for AI path-finding algorithms using the same flavour of Xilinx HLS as this study. Conceptually, the approach of a co-processor is similar to ours, though their design adheres to a lower-level hardware-first philosophy much more strictly and with significant manual adjustments to enforce pipelining, loop unrolling, and other such optimisations that the synthesiser did not automatically apply. Their results were positive, with a 4x increase over a baseline ARM CPU.[7]

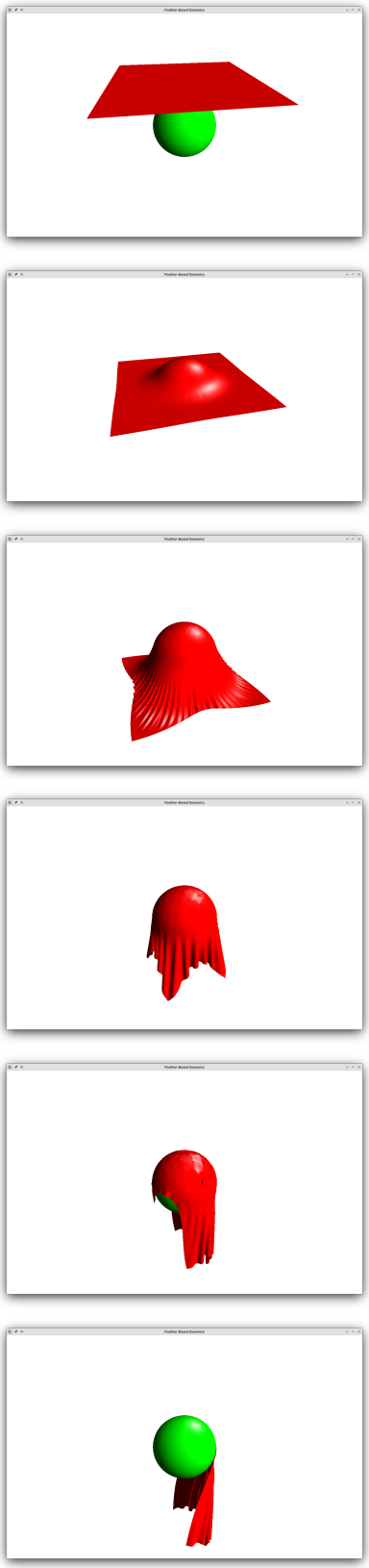
While examples of FPGAs being used to accelerate particle physics simulations are plentiful, these stray outside of the scope of our research; the focus of such simulations (accuracy, reproducibility) is fundamentally different from those in a computer graphics physics simulation (visual believability).

4 This design

Our simulation is a simple implementation of a position-based cloth simulation. This simulation involves a piece of cloth, modelled as a square grid of vertices, falling on a sphere and draping over it, before slipping off and falling. The physics calculations are performed in real time, and therefore the latency of each iteration of the position-based algorithm is crucial to the fluidity and smoothness of the simulation rendering.

[5] Woulfe and Manzke, "Towards a Field-Programmable Physics Processor (FP3)".
[6] Toupas, Brokalakis, and Papaefstathiou, "Accelerating Physics Engine Components with Embedded FPGAs".
[7] Nery and Sena, "Efficient A* Co-processor for Reconfigurable Gaming Devices".

Figure 2: Screenshots from the visualisation of our complete real-time cloth simulation



4.1 Baseline C++ implementation

First, this simulation was implemented in standard C++, akin to how a traditional OpenGL render would be implemented. The cloth simulation is instantiated as an object containing the relevant methods and variables needed to evaluate the desired positions for each vertex.

4.1.1 Data structure

The vertex is defined as a C struct ClothVertex with members for each of the needed properties of the vertex. The first property is the position property, which defined where the vertex is located as a 3-element array of floating point values corresponding to the X, Y, and Z axes. We have a similar member for the velocity of the vertex. Additionally, we use a member with the same layout for the normal of the vertex, used by OpenGL for lighting purposes, and a 4-element float array for colour, again used by OpenGL for rendering the mesh. In this struct we also have two integer properties, row and column, which are used by some of the simulation class methods to locate adjacent vertices.

4.1.2 Initialisation

The class constructor initialises all of the required vertices. During the initialisation, the starting positions of the vertices are decided according to the row and column in which the vertex will be placed. In this case, X, Y, and Z are set to create a flat mesh above the sphere which the cloth will fall on and collide with. The velocity is initialised to 0 on all axes so that the vertices are stationary upon initialisation. The values relevant to OpenGL rendering are also set (colour and normal). The vertex is added to a vector member of the class where they are stored during the execution of the simulation.

The class constructor also handles the initialisation of the constraints. In this case, we use two constraint values. The first (constraint_two) is calculated as the distance between the bottom-left vertex and the central vertex in the cloth mesh. This value is also used to calculate the hypotenuse of the square formed by the vertices, which is the second constraint value (constraint_diagonal) used by the simulation.

The constructor finally sets the correct values for the vertex normals and the indices that will be used by OpenGL to correctly place vertices.

4.1.3 Position-based physics processing

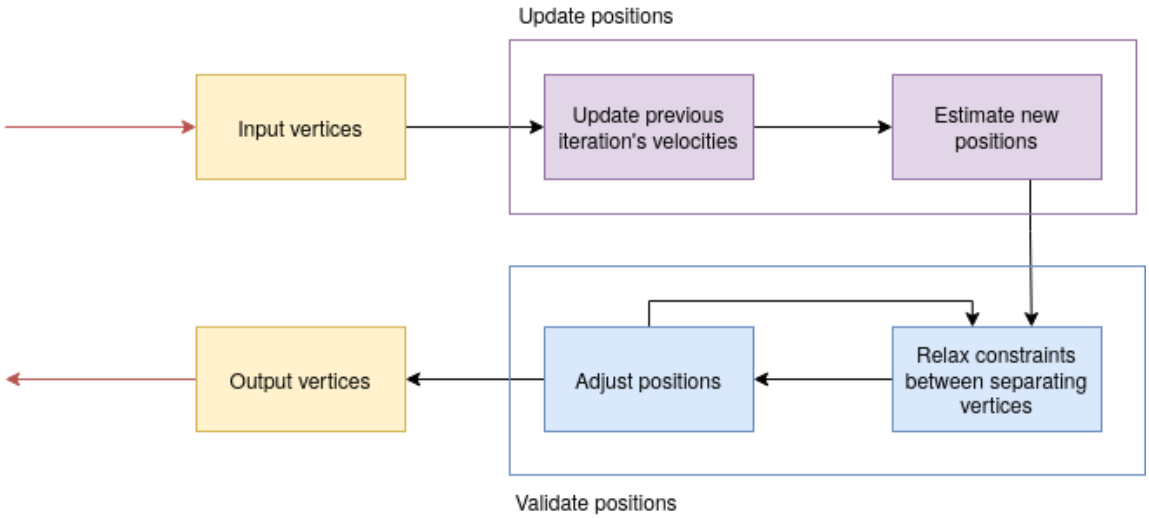
The class has three key functions that handle the main processing of the simulation. The method update_positions iteratively generates the estimated new positions for each of the vertices in the mesh using the current velocities of the vertices to calculate this estimate. It also updates the velocities of each vertex according to the calculations performed in the previous iteration of the simulation loop.

The method validate_positions handles the correction of these estimates based on our constraint values, performing several passes to ensure correctness. Each pass iterates over every vertex several times, calling the relax_constraint function which relaxes the rigid constraints between each vertex and its adjacent vertices, allowing the mesh to bend like cloth. relax_constraint is applied to the positions of the supplied vertex, the adjacent vertices in the next two rows as well as the vertices in the next two columns. The constraint relaxed here is constraint_two. The same process is undertaken for the vertices which are diagonal to the subject vertex, only this time constraint_diagonal is used to decide the factor by which the vertices are allowed to separate from each other. The last part of an iteration of validate_positions finally adjusts the estimated positions of the vertices to ensure that any penetration of the cloth mesh into the sphere is corrected. Higher vertex counts tend to require more validation iterations.

The C++ implementation also employs an update_normals method to update the normals required for OpenGL to correctly compute the lighting of the cloth mesh. While this function (and its helper functions) are not relevant to the physics calculations, they allow us to implement a rendering of the simulation using OpenGL that can let us visually verify the output of the simulation.

Additionally, there are several public members of the class corresponding to significant variables in the simulation. The aforementioned iteration count is a member variable, as is the number of vertices, the time step (which regulates the magnitude of the velocities applied to the vertices' updated positions), and a bias variable which is used for regulating the relaxation of the constraints. This allows for the aforementioned update_positions, validate_positions and update_normals to be called without needing to keep track of required arguments, making the simulation easier to implement in an event loop.

Figure 3: Control flow of cloth simulation.



4.2 Implementing the simulation in HLS

To test how transferable a traditional C++ design is to HLS, the HLS C implementation was designed to resemble the C++ design. However, some modifications were required to make the simulation work as desired. The final implementation used in testing is similar to the one

described in the prior section, but makes some structural changes. Most importantly, most of the features of the C++ implementation's object-oriented design were eliminated. It was instead more effective to treat each of the methods in the ClothSimulation class as a separate kernel that could be accelerated. Data can be fed into the accelerated kernel and data can be quickly received as an output, mapped to hardware inputs and outputs.

Therefore, instead of the ClothSimulation class, a redesigned ClothKernel class is used which only contains static methods. This class does not initialise any vertices since these are delivered to the functions of the ClothKernel class for processing. These static methods take as arguments input and output arrays, and the variables (e.g. iterations, constraint values, etc.) required to perform the relevant computations. The important change for update_positions and validate_positions is that they no longer operate on a single array structure of ClothVertex elements, but now read from an input array and write to an output array which must be passed to the following functions.

This ClothKernel therefore retains the update_positions and validate_positions functions, as well as their relevant helper functions. We have opted to eliminate the methods calculating normals as we are focusing on utilising the FPGA purely as an accelerator for the physics calculations. Additionally, we have also eliminated most of the members of the ClothVertex struct to only retain the position and velocity arrays needed for the physics calculations.

HLS makes use of a top-function, which is the function that calls the relevant kernels. In this case, the update_positions and validate_positions functions are called in the top-function process_cloth. This top-function also includes a simple memory structure, two arrays of ClothVertex elements which are used as buffers in the calls to the two key kernel functions. In the first step of the physics processing, where estimates of the vertex positions are generated, the input stream is read from directly and the modified vertices are written with their new position estimates to the first buffer. WThis first buffer is then read from in the validation stage and the adjusted positions are written to the second buffer. This two buffer layout provided significantly lower latencies in the validate_positions kernel than writing directly to the process_cloth output array. Finally, the data from the second buffer is copied to the output array. In some of the helper functions invoked by update_positions and validate_positions copies of the vertex structs are created, so as to avoid reading and writing simultaneously on the same struct.

During the synthesis process, the Vitis HLS synthesiser applies several optimisations to this code. For instance, loops are pipelined where it is considered to be safe to do so, helping improve the efficiency of the scheduling of each task. Where it is deemed possible, loops are also unrolled. While it is also possible to manually specify such things to be done using preprocessor directives, the automation of this task takes the burden off of the developer. However, Vitis tends to be very conservative and will not automatically pipeline loops in several cases where such a thing is safe.

Since the C++ implementation specifically avoided the use of any external libraries in the simulation loop, this conversion process was relatively easy – however, this would have been significantly more complex and time-consuming if the reimplementation of classes, data structures, and so on were required. Since these are often used in current game development, even at the engine development level, it is likely that the more complex and dynamic simulations used in modern game engines would require additional work to implement in HLS C.

C++ class and vertex

```
struct ClothVertex {
    std::array<float, 3> position;
    std::array<float, 3> normal;
    std::array<float, 3> velocity;
    std::array<float, 4> color;
    int row;
    int column;
};

class ClothSimulation
{
public:
    ClothSimulation();

    void reset();

    static float inv_sqrt(float number);
    void update_normal(ClothVertex &vertex
    , int row_difference, int
    column_difference) const;
    void normalise_normal(ClothVertex &
    vertex) const;
    void adjust_positions(ClothVertex &
    vertex) const;
    void relax_constraint(std::vector<
    ClothVertex> &buffer, int a, int b,
    float constraint) const;

    void update_normals();
    void update_positions();
    void validate_positions();

    std::vector<ClothVertex> vertices;
    std::vector<int> indices;

    static constexpr int resolution = 32;
    static constexpr float time_step =
    0.025;
    static constexpr int iterations = 96;
    static constexpr float bias = 0.17;

private:
    float m_constraint_two;
    float m_constraint_diagonal;
};
```

HLS class and vertex

```
struct ClothVertex {
    float position[3];
    float velocity[3];
};

class ClothKernel {
public:
  static float inv_sqrt(float number);

  static void adjust_position(ClothVertex
    &vertex);

  static void relax_constraint(ClothVertex
    &vertex_a, ClothVertex &vertex_b,
    float constraint_in, float bias);

  static void update_positions(ClothVertex
    vertices_in[], ClothVertex
    vertices_out[], float time_step);

  static void validate_positions(
    ClothVertex vertices_in[], ClothVertex
    vertices_out[], float constraint_dia,
    float constraint_two, float bias);
};
```

C++ key helper functions

```
void ClothSimulation::update_positions()
{
    for(auto &vertex : vertices) {
        vertex.velocity[2] -= (vertex.position[0] * vertex.position[0] + vertex.position[1] * vertex.position[1] + vertex.position[2] * vertex.position[2] > 1) ? time_step : 0;

        vertex.position[0] += time_step * vertex.velocity[0];
        vertex.position[1] += time_step * vertex.velocity[1];
        vertex.position[2] += time_step * vertex.velocity[2];
    }
}

void ClothSimulation::relax_constraint(std::vector<ClothVertex> &buffer, int row, int column, float constraint) const
{
    // This is the displacement vector between the two vertices
    std::array<float, 3> delta = {vertices[row].position[0] - vertices[column].position[0], vertices[row].position[1] - vertices[column].position[1], vertices[row].position[2] - vertices[column].position[2]};

    const float invlen = inv_sqrt(delta[0] * delta[0] + delta[1] * delta[1] + delta[2] * delta[2]);
    const float factor = (1.0f - constraint * invlen) * bias;

    buffer[row].position[0] -= delta[0] * factor;
    buffer[row].position[1] -= delta[1] * factor;
    buffer[row].position[2] -= delta[2] * factor;

    buffer[column].position[0] += delta[0] * factor;
    buffer[column].position[1] += delta[1] * factor;
    buffer[column].position[2] += delta[2] * factor;
}


void ClothSimulation::adjust_positions(ClothVertex &vertex) const
{
    const float number = (vertex.position[0] * vertex.position[0] + vertex.position[1] * vertex.position[1] + vertex.position[2] * vertex.position[2]);

    const float invrho = inv_sqrt(number);

    // Move vertex to surface of sphere if inside
    vertex.position[0] *= invrho < 1 ? 1 : invrho;
    vertex.position[1] *= invrho < 1 ? 1 : invrho;
    vertex.position[2] *= invrho < 1 ? 1 : invrho;
}
```

HLS key helper functions

```
void ClothKernel::update_positions(ClothVertex vertices_in[], ClothVertex vertices_out[], float time_step)
{
    for(int i = 0; i < 1024; i++) {
      ClothVertex vertex = vertices_in[i];
        vertex.velocity[2] -= (vertex.position[0] * vertex.position[0] + vertex.position[1] * vertex.position[1] + vertex.position[2] * vertex.position[2] > 2) ? time_step : 0;

        vertex.position[0] += time_step * vertex.velocity[0];
        vertex.position[1] += time_step * vertex.velocity[1];
        vertex.position[2] += time_step * vertex.velocity[2];
        vertices_out[i] = vertex;
    }
}
```

```
13
14 void ClothKernel::relax_constraint(ClothVertex &vertex_a, ClothVertex &vertex_b,
       float constraint_in, float bias)
15 {
16 #pragma HLS DEPENDENCE variable=vertex_a inter false
17 #pragma HLS DEPENDENCE variable=vertex_b inter false
18     // displacement vector
19     std::array<float, 3> delta = {vertex_a.position[0] - vertex_b.position[0],
       vertex_a.position[1] - vertex_b.position[1], vertex_a.position[2] - vertex_b.
       position[2]};
20
21     const float invlen = inv_sqrt(delta[0] * delta[0] + delta[1] * delta[1] + delta
       [2] * delta[2]);
22
23     const float factor = (1.0f - constraint_in * invlen) * bias;
24
25     float a_copy[3] = {vertex_a.position[0], vertex_a.position[1], vertex_a.position
       [2]};
26
27     vertex_a.position[0] = a_copy[0] - delta[0] * factor;
28     vertex_a.position[1] = a_copy[1] - delta[1] * factor;
29     vertex_a.position[2] = a_copy[0] - delta[2] * factor;
30
31     float b_copy[3] = {vertex_b.position[0], vertex_b.position[1], vertex_b.position
       [2]};
32
33     vertex_b.position[0] += b_copy[0] + delta[0] * factor;
34     vertex_b.position[1] += b_copy[0] + delta[1] * factor;
35     vertex_b.position[2] += b_copy[0] + delta[2] * factor;
36 }
37
38
39 void ClothKernel::adjust_position(ClothVertex &vertex)
40 {
41     const float number = (vertex.position[0] * vertex.position[0] + vertex.position
       [1] * vertex.position[1] + vertex.position[2] * vertex.position[2]);
42     const float invrho = inv_sqrt(number);
43
44     // Move vertex to surface of sphere if inside
45     vertex.position[0] *= invrho < 1 ? 1 : invrho;
46     vertex.position[1] *= invrho < 1 ? 1 : invrho;
47     vertex.position[2] *= invrho < 1 ? 1 : invrho;
48 }
```

4.3 Validation

The most important aspect of computer graphics simulation is not precision, but visual believability. Accordingly, validating the simulation implementation was first done by rendering the mesh using OpenGL and the C++ implementation, visually checking the behaviour of the mesh during the simulation.

Since Vitis HLS uses a subset of C/C++ that can be conventionally compiled and run, a software testbench was created that replicated the simulation parameters and process of the cloth simulation of the C++ simulation in HLS code. This testbench initialises an array of vertices (in this case, 1024), the constraint values, and then iteratively runs the top function for 1000 iterations. Verification of the correctness of the HLS implementation was then performed by comparing the vertex positions recorded by the processed vertices after each cycle in both C++ and HLS. To aid in this process, an `id` member was added to the vertex structs in both implementations which was set to the current index of the loop that instantiated the vertex; since both the testbench and the C++ simulation generated vertices in the same order, these identifiers would correspond to the same vertex.

5 Evaluation

An evaluation of this design is now presented through analysis of the synthesis data provided

by Vitis. It is expected that the synthesis report generated by Vitis will accurately represent the actual latency of the design with an acceptable synthesis time. For the purposes of getting accurate results from this HLS design, array iteration bounds and validation iterations were hardcoded in the HLS code.

5.1 Synthesis results

The synthesis of the cloth simulation kernel design provided mixed results. Vitis' conservative optimisation approach, particularly regarding pipeline hazards, prevented important optimisations from taking place in the synthesised validate_positions, which is unfortunate as the three-dimensional loop is by far the most expensive part of the physics processing pipeline. Attempts to partition buffer arrays were met with extremely long synthesis times, and attempts to manually unroll arrays were unsuccessful, with each of the synthesis attempts eventually being killed by the host OS due to massive resource consumption.

The synthesis results for the standard 32x32 mesh and 96-iteration validation implementation shows that there is no slack between each kernel function, and that even accounting for timing uncertainty we fall below the 10 nanosecond mark (7.072 estimated, 2.70 uncertainty). However, latency is high, particularly due to the validate_positions function. The relative lack of optimisations are likely due to the way in which vertices are stored and accessed – a single-dimensional array with data accesses that can be very far apart when needing to fetch vertices from the next row in the mesh is very inefficient.

Table 1: Timing summary for all designs

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	7.072 ns	2.70 ns

Table 2: Latency report for 32x32 mesh design, 96 validation iterations

Instance	Module	Latency (cycles)	Latency (absolute)	Interval (cycles)
grp_ process_ cloth_ Pipeline_ 1_ fu_1 56	process_ cloth_ Pipeline_ 1	1026	10.260 us	1026
grp_ process_ cloth_ Pipeline_ VITIS_ LOOP_ 30_ 1_ fu_ 178	process_ cloth_ Pipeline_ VITIS_ LOOP_ 30_ 1	1038	10.380 us	1038
grp_ validate_ positions_ fu_ 196	validate_ positions	45741	0.457 ms	45741
grp_ process_ cloth_ Pipeline_ 3_ fu_ 216	process_ cloth_ Pipeline_ 3	2050	20.500 us	2050

Table 3: Utilization estimate report for 32x32 mesh design, 96 validation iterations

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	-	-	-
FIFO	-	-	-	-	-
Instance	-	198	29567	25410	-
Memory	37	-	0	0	0
Multiplexer	-	-	-	900	-
Register	-	-	204	-	-
Total	37	198	29771	26310	0
Available	4320	6840	2364480	1182240	960

5.2 Benchmark results

To compare latencies, synthesised designs hardcoded to a range of mesh-sizes were tested and compared to equivalent C++ simulations. First, a single iteration of the position-based cloth simulation was run with the default parameters – a

32x32 mesh and 96 iterations of validation. The mesh size was then progressively increased from 32x32 to 1024x1024, and testing focused on the latencies that each of these designs generated. To make the latencies being analysed reasonable, the validation process was limited to a single iteration; even in a single-iteration validation of a 1024x1024 mesh, 94% of processing time is taken by just one validation iteration.

Testing is focused on latency as it is important in an interactive world to receive immediate feedback. The synthesised FPGA results are compared with those acquired from an Intel Core i7-9750H running the C++ simulation.

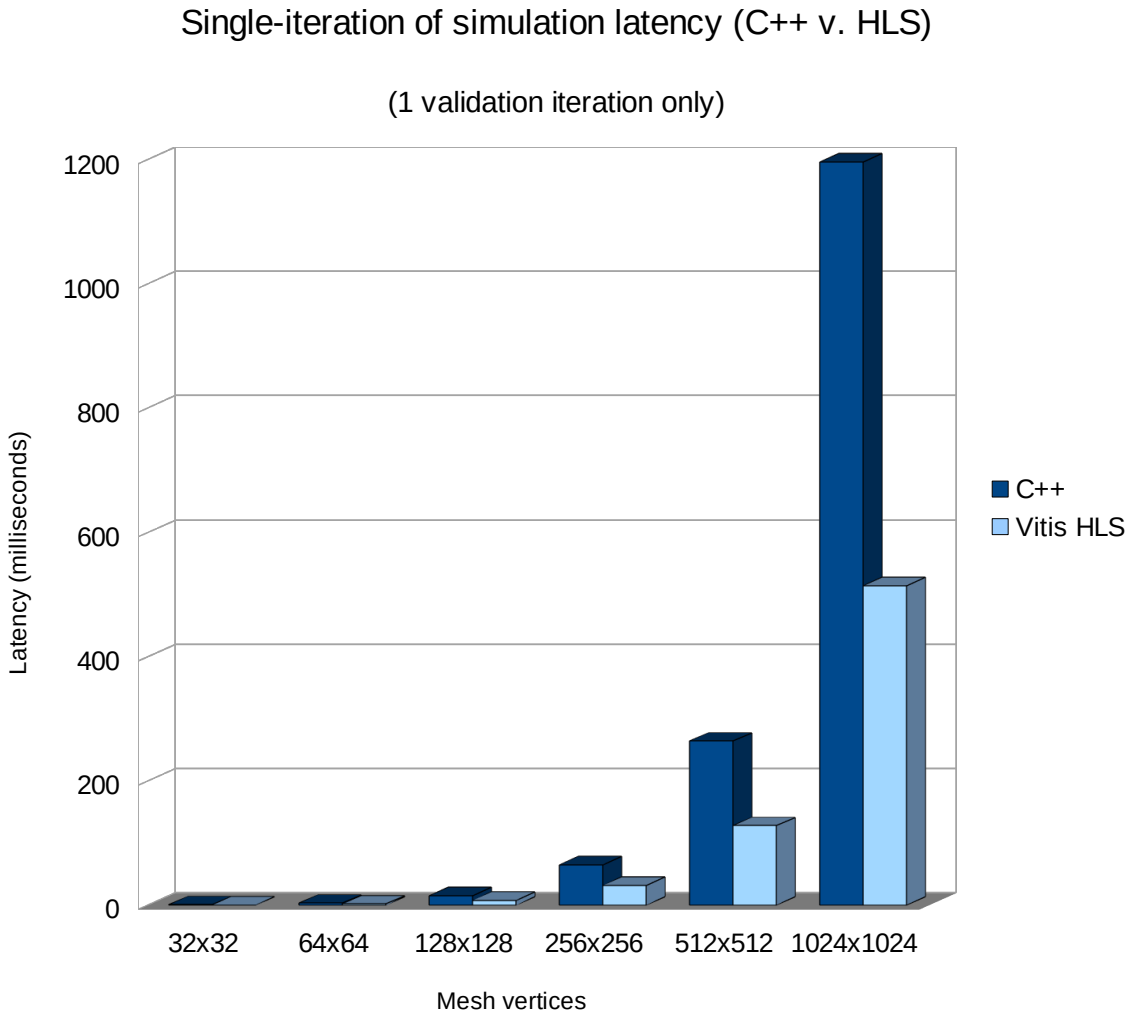
Though in isolation it is clear that synthesis latencies are quite high, it is also clear that the latency at each of the mesh sizes is being halved, even on a single iteration of the validation. The results for each mesh size can be seen below.

It is presumed that more favourable results would have been achieved with a design that more closely adapted to the layout of the hardware. Additional fine-tuning of latencies could have also likely been achieved with use of Vitis HLS' preprocessor directives, such as pragma HLS PIPELINE, especially in the expensive iterative validation process. Furthermore, non-HLS designs are almost universally faster than HLS designs, sometimes by several orders of magnitude. Other HLS languages, such as SystemC, are often faster than Vitis HLS too (at the detriment of demanding more specific knowledge of said language).

This analysis also obviates power or area concerns. While power may not be as significant an issue for development on future stationary systems (consoles, PC), it certainly is a concern for mobile applications of such acceleration (standalone VR headsets, portable consoles, phones).

Single-iteration of simulation latency (1 validation iteration)

	Latency (milliseconds)	
Vertices	**C++**	**Vitis HLS**
32x32	0.946	0.499
64x64	3.546	1.998
128x128	15.3	8.006
256x256	65.5	32.065
512x512	264.5	128
1024x1024	1196.083	514



Single-iteration of standard simulation (96 validation iteration, 36x36 vertex mesh)

Latency (milliseconds)	
C++	**Vitis HLS**
129.2	42.004

5.3 HLS for software developers

Despite the mixed results and the lack of optimisation, there was still a 2x improvement across the board in the benchmarks above. Combined with the ease of porting the cloth simulation algorithm to Vitis HLS' subset of C/C++, it is clear that there is potential here for the use of HLS in accelerating certain aspects of video-game physics processing.

Commercial attempts at physics processing units, such as Ageia's, failed due to a combination of high pricing, niche utility, and a specific software library being required to employ the acceleration afforded by their PPU. In contrast FPGAs can be acquired by consumers today at a range of prices and capabilities, with rumblings about the direct implementation of FPGA-like unconventional cores achieving broader adoption in consumer-facing chips. With the increasing growth of the VR market and its demand for ever-more interactive game worlds, the breadth and depth of physics simulation in interactive game worlds is growing, ignoring the fact that FPGAs are designed to be repurposed to accelerate varied tasks. Finally, Vitis HLS provides a more accessible entry point and a shallower learning curve to FPGA development for non-hardware developers than RTL languages such as Verilog.

Though most game developers would never touch HLS – most interact with core APIs and do not develop game backends – it seems that it would be a good idea for those who do to integrate some HLS into the software stack of game engines.

6 Conclusion

While the transition from a standard C++ implementation of the simulation to the HLS design was not direct, it was still relatively straightforward to implement a working accelerator kernel for a position-based cloth simulation. While the results provided by the synthesised kernel are not staggering, they nevertheless represent a substantial improvement over the CPU-bound implementation the FPGA design is based on.

The data suggests that there is a case for the inclusion of FPGA acceleration, and the use of high-level synthesis tools, in the game engine software stack. HLS' advantage specifically lies in the ease with which one can transfer a C++ program into a synthesisable design, even if this might come at a performance cost versus a bespoke design in a more domain-specific high-level synthesis language, or directly in RTL code. The accessibility of HLS could make it feasible for software-focused backend game developers to consider adopting tools such as Vitis HLS, and to accelerate certain tasks in applications where the GPU is likely to already be saturated by other 3D rendering or basic physics workloads. This will be especially relevant in VR simulations, where an increased level of interactivity (requiring believable physics simulations for all in-game objects) is required, and where it is already intensive to render the game world at a sufficient framerate.

References

Müller, Matthias et al. "Position based dynamics". In: *Journal of Visual Communication and Image Representation* 18.2 (2007), 109–118. ISSN: 1047-3203. DOI: 10.1016/j.jvcir.2007.01.005. URL: https://www.sciencedirect.com/science/article/pii/S1047320307000065.

Nery, Alexandre S. and Alexandre C. Sena. "Efficient A* Co-processor for Reconfigurable Gaming Devices". In: *2018 17th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*. 2018, pp. 97–9709. DOI: 10.1109/SBGAMES.2018.00021.

Toupas, Petros, Andreas Brokalakis, and Ioannis Papaefstathiou. "Accelerating Physics Engine Components with Embedded FPGAs". In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 2019, pp. 88–94. DOI: 10.1109/FPL.2019.00023.

Wilson, Derek. *Exclusive: Asus debuts Ageia PhysX Hardware*. 2006. URL: https://www.anandtech.com/show/2001/5.

Woulfe, M. and M. Manzke. "Towards a Field-Programmable Physics Processor (FP3)". In: 2006.