Technical University of Crete

Diploma Thesis

Modeling and Design of "Projected Gauss-Seidel" Algorithm in FPGA

Author:
Petros Toupas

Committee:
Prof. Apostolos Dollas
Prof. Dionisios Pnevmatikatos
Assoc. Prof. Ioannis Papaefstathiou (AUTH)

A thesis submitted in fulfillment of the requirements for the Diploma of Electrical and Computer Engineering in the

School or Electrical and Computer Engineering
Microprocessor and Hardware Lab

October 10, 2018

TECHNICAL UNIVERSITY OF CRETE

School or Electrical and Computer Engineering

Abstract

Modeling and Design of "Projected Gauss-Seidel" Algorithm in FPGA

by Petros TOUPAS

In recent years there has been a continuous increase in the use of physics engines, which are widely used in the industry of video games, scientific simulations, computer graphics, and films. Their main goal is to simulate the motions of objects based on physics rules of real-world. The input to a physics engine is a collection of objects (rigid bodies) with their properties and a collection of forces acting on those bodies. Rigid bodies are objects that do not deform when they collide and rigid body dynamics is the study of their motion. This input is being processed by performing a certain number of simulation steps which produce the updated properties for each object in the output. Rigid body simulation along with rigid body dynamics are used to simulate the real world physics. The complexity of the modern games is rapidly increasing, so does the computational cost of the simulation the physics engines must accomplish. The use of GPGPU (General-Purpose computing on Graphics Processing Units) can help us overcome the need for high computational requirements by exploiting the parallel processing a GPU can provide. In this thesis, we are going to use FPGA instead of GPU since we can still exploit the parallel processing and furthermore we can achieve much better power consumption in comparison with GPU. The physics engine we are working on is Bullet which has already implemented a high computational cost game scene with many rigid bodies (38880) in GPU with the use of the OpenCL library. Our implementation of the same game scene in Xilinx UltraScale+ ZCU102 with the use of Vivado HLS and C++, achieving 7.08× to 8.5× speedups over CPU, while these numbers change to 1.82× to 2.19× speedups and 35.72× to 43.67× power efficiency when compared to NVIDIA GeForce GTX 980.

Acknowledgements

I would like to thank my supervisor, Assoc. Prof. Ioannis Papaefstathiou (AUTH) for his guidance during the course of this thesis, as well as for the opportunity he gave me to delve into the field of gaming engines combined with re-configurable hardware and expand my knowledge on hardware and FPGA designing.

I would also like to thank all the members of Microprocessor and Hardware Lab (MHL) and especially Andreas Brokalakis for his valuable guidance and the amount of time he dedicated to help me complete this thesis and Pavlos Malakonakis for the valuable technical advice whenever I needed them and for pointing me in the right direction every time I had a dilemma.

I would like to deeply thank my friend and studying partner Giorgos-Antonios Pitsis for standing always beside me and helping me when I needed his support the most.

I also want to express my sincere thanks to my good friend and roommate Loukas-Rafael Nomikos for everything he has done for me in the last months while hosting me.

Last but not least I would like to thank from the bottom of my heart my parents and my sister for supporting me all these years and for giving me the opportunity to achieve my goals and dreams. Without them, I wouldn't be able to accomplish any of these.

Petros Toupas
Chania 2018

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

CPU	**C**entral **P**rocessing **U**nit
GPU	**G**raphics **P**rocessing **U**nit
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
DDR	**D**ouble **D**ata **R**ate
SDRAM	**S**ynchronous **D**ynamic **R**andom **A**ccess **M**emory
BRAM	**B**lock **R**andom **A**ccess **M**emory
LUT	**L**ook- **U**p **T**ables
FF	**F**lip **F**lops
DSP	**D**igital **S**ignal **P**rocessor
HLS	**H**igh **L**evel **S**ynthesis
OpenCL	**Open C**omputing **L**anguage
GPGPU	**G**eneral **P**urpose computing on **G**raphics **P**rocessing **U**nit
3D	**3 D**imensional space
AABB	**A**xis **A**ligned **B**ounding **B**ox
OBB	**O**riented **B**ounding **B**ox
SAP	**S**weep **A**nd **P**rune
SAT	**S**eparating **A**xis **T**heorem
GJK	**G**ilbert **J**ohnson **K**eerthi
PGS	**P**rojected **G**auss **S**eidel
LCP	**L**inear **C**omplementarity **P**roblem
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
LSB	**L**east **S**ignificant **B**it
MSB	**M**ost **S**ignificant **B**it
RTL	**R**egister **T**ransfer **L**evel
I/O	**I**nput / **O**utput
DMA	**D**irect **M**emory **A**ccess

This thesis is dedicated to my beloved parents and my dear sister for the faith they always showed me and the unlimited support they provided me...

Chapter 1

Introduction

The use of numerical simulation for every kind of physical effects has been well-developed in the last few years. Many applications such as computer animation, video games, visual effects in movies, and robotic simulation need these simulations to operate. The use of this type of applications is constantly increasing so does the need to perform the physics simulation faster and with the best possible precision.

Real-world systems are modeled by rigid bodies. There are some cases where the deformation of an object is an important factor in the physical behavior of the object (for example in a cloth or jelly). In these cases, the rigid bodies are not acceptable. We are going to work on Bullet, a physics engine which has been used in the development of many video games like Grand Theft Auto V and Rocket League as well as in films like Sherlock Holmes and Bolt for the simulation of special effects and in many 3D authoring tools like Blender, Cinema 4D and in many other cases[10]. This thesis aims to develop a rigid body simulation in ZCU102 FPGA, where many objects (38880) are colliding with each other.

1.1 Motivation

Physics engines in video games have always been CPU-bound tasks, which means that the CPU is responsible for every processing done inside the physics engine, apart from the rendering part which is performed on GPU. As the physics simulation is becoming more and more computationally demanding, the CPU is reaching its limits, hence there is a need to use a more suitable processing unit for this kind of demanding task. This processing unit will be practically a powerful, massively parallel co-processor alongside CPU. The calculations a physics simulation needs to perform are generally independent computations implemented in a software pipeline on a per-object basis. This data-level parallel nature of the physics simulation can exploit the parallel processing capabilities

of GPU/FPGA so that each stage of pipeline executes on different data sets in parallel. Accelerating the physics simulation on a physics engine is really important when it comes to video games since they are real-time applications. The faster the simulation steps are performed, the more times we can render the scene, which leads to higher FPS.

The Bullet's implementation of the game scene we are going to work on uses CPU alongside with GPU[11], where all the computationally intensive parts (huge loops with cross products, dot products, etc.) of the algorithm are being processed. In this thesis, we aim to achieve speedup and power efficiency better than GPU by using FPGA.

1.2 Contribution

This thesis presents two different FPGA architectures which accelerate the physics simulation of a game scene with many rigid bodies (38880). An extensive analysis was conducted on input data in order to understand in depth the way these data are used and to reduce their size as much as possible. This pre-processing procedure has been used in both architectures. The nature of the algorithm requires a large amount of data to be able to operate a simulation step. For this reason, we exploit the full bandwidth from the DDR SDRAM by using streams to transfer the data to and from the FPGA. Furthermore, we have executed a conversion of the original code from OpenCL to C++ and at the same time, we have modified the source code to meet the specifications of the FPGA.

The first architecture was implemented in floating point precision. We have used many features of Vivado HLS alongside with streaming to achieve the desired speedup, such as pipeline, array partitioning etc. The second architecture was implemented with half-precision floating point, which led us to have half the volume of data at the input, as well as half of the resources in the FPGA. It also led us to reduce the latency of the simulation step, hence resulted in a better speedup. The first architecture led us to a speedup 7.08× over CPU and 1.82× over GPU, as also to 35.72× power efficiency over GPU. When it comes to the second architecture we have achieved a speedup 8.5× over CPU and 2.19× over GPU and 43.67× power efficiency over GPU.

1.3 Thesis Outline

The remainder of this work is organized as follows.

- **Chapter 2:** We perform a theoretical background analysis of the physics engines as well as the heterogeneous systems.
- **Chapter 3:** We review related work on acceleration of physics simulations on physics engines with GPGPU systems and FPGAs.
- **Chapter 4:** We present the analysis and pre-processing we have performed on the Projected Gauss Seidel and Integration algorithms to extract valuable information about any potential data redundancy and optimization.
- **Chapter 5:** We describe the two different architectures we have designed.
- **Chapter 6:** We show the results of those architectures and we compare them with the corresponding implementations on GPU and CPU.
- **Chapter 7:** We entail the conclusions from this work, as well as the lessons learned and proposals for improvements.

Chapter 2

Theoretical Background

Bullet physics is a professional open source collision detection, rigid body, and soft body dynamics library. It targets real-time and interactive use in games, visual effects in movies, and robotic simulation. In this chapter, we will provide all the necessary information on how a physics engine works, the sub-steps it performs during a single simulation step and the algorithms it uses to perform each one of them. We will also analyze how Bullet implements all these tasks and procedures based on the [9] and [11]. Lastly, we will analyze the heterogeneous systems and explain why they fit perfectly and why they can be used by physics engines.

2.1 Physics Engines

A physics engine is computer software that provides an approximate simulation of certain physical systems, such as rigid body dynamics, soft body dynamics, and fluid dynamics, of use in the domains of computer graphics, video games and film. Their main uses are in video games (typically as middleware), in which case the simulations are in real-time. The term is sometimes used more generally to describe any software system for simulating physical phenomena, such as high-performance scientific simulation.

2.1.1 General Concepts of Physics Engines

Simulation Loop

We can think of a rigid body simulation as a continuous loop, which is like the heartbeat of every physics engine. The complexity of the simulation process leads us to fragment it into smaller, well-defined pieces. Each one of them is responsible for solving a simpler task. All pieces are tied together in a simulation loop as shown in Figure 2.1.

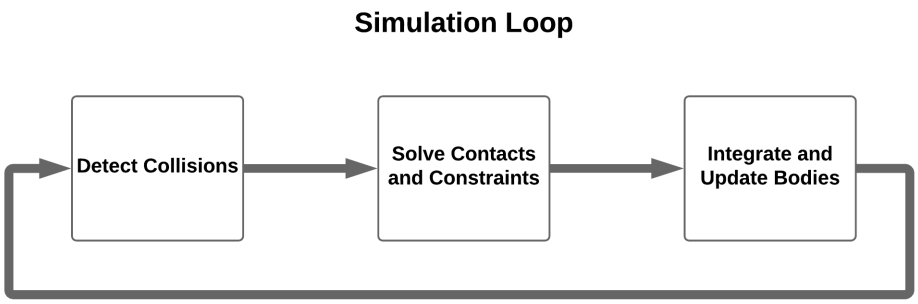


FIGURE 2.1: The simulation loop of a physics engine

The loop begins with a collision detection algorithm to find the possible collisions and contact points between all various bodies. These points are necessary in order to apply the physical laws of motion on the bodies, and finally determine collision forces which provide proper collision friction effects and prevent bodies from interpenetrating [18]. The above procedure is termed as *"collision response"*. After computing all the collision forces, the positions and velocities of the bodies are integrated forward in a time step dt before a new iteration of the simulation loop starts. Several iterations of the loop might be performed before a frame is rendered.

Simulation Time-Step

We can easily understand that choosing the right value for the time step dt is really important since it affects how the bodies behave during the loop. Especially when targeting real-time applications like video games, the choice of dt also affects how many times the simulation will be executed in one second. The simulation consists of two parts, updating the physics of the scene(game-step) and rendering the scene(render-step). In order to achieve a stable and smooth visual representation over time, we target at 60 FPS, i.e to perform 60 render-steps in one second. Ideally, we want to have one game-step for every render-step. In some computational intensive scenes, it's not possible to calculate so many game-steps in one second. However render-steps should always be at 60 times per second. To overcome this problem we take advantage of the interpolation technique [3] [19]. For instance, if we can calculate 30 game-steps in a second but we want to have 60 render-steps we are going to predict the intermediate game-steps using interpolation so we can still achieve the goal of 60 render-steps in a second. Interpolation is very important in order to help us achieve the desired FPS especially in computational intensive applications.

This way we can calculate less game-steps per second and still have a smooth and stable visual result. In the Figure 2.2 below, which we found at [16], we can see the speed-accuracy trade-off in physics engines that depends on the value of time-step (dt).

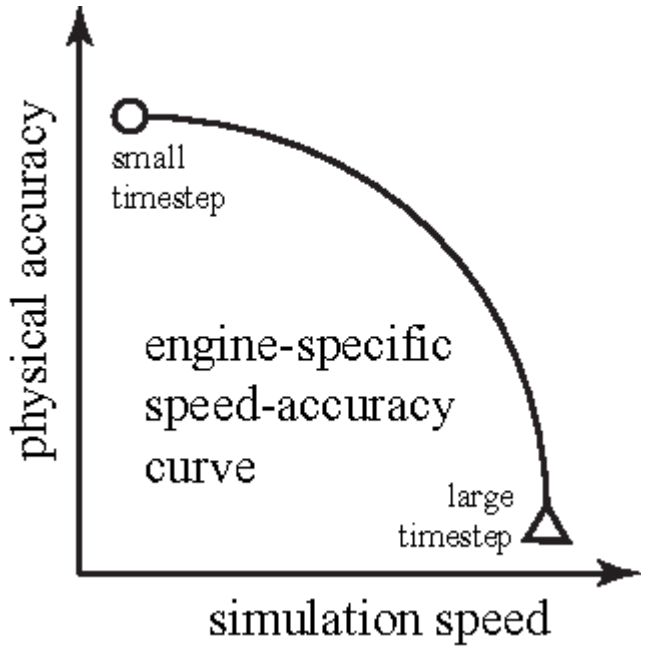


FIGURE 2.2: The speed-accuracy trade-off in physics engines depending on time-step(dt)

Geometry Types of Shapes

The geometry type of simulated rigid bodies is important for collision detection and contact point generation because it influences the performance and the complexity of the simulation. It is common that objects are represented by primitives shapes such as boxes, spheres or cylinders [2]. Convex polygonal models are also a very common geometry type in the field of interactive rigid body simulations. Thanks to the properties of these models, very fast algorithms have been created for collision detection and this is a reason why they are widely used. In a case where non-convex (concave) shapes exist, they have to be decomposed into convex shapes.

If each of the interior angles of a polygon is less than 180°, then it is called convex polygon. On the other hand, if at least one angle of a polygon is more than 180°, then it is called a concave polygon.

2.1.2 Collision Detection

The first big step the simulation loop needs to perform is collision detection. The goal of collision detection is to report if two objects have collided. However, this operation is extremely time-consuming. In case where the scene is composed of n bodies, the complexity to test all the possible pairs of bodies is $O(n^2)$ [17]. This can easily become a computational bottleneck so it is divided into 2 phases, Broad-Phase and Narrow-Phase.

Broad-Phase Collision Detection

Broad-phase detection is typically a computationally low cost operation that quickly answers the question, "Which objects have a strong possibility of colliding?". Most physics engines use the concept of bounding primitives to simplify and speed-up the broad-phase detection. There are many types of bounding primitives but the most common are, bounding spheres, Axis-Aligned Bounding Boxes, Oriented Bounding Boxes, and Convex Hulls [22]. The most commonly used is AABB since its faster and quite accurate at this point of the algorithm. In AABBs, each object is covered by a box as illustrated in 2D by Figure 2.3. In 3D world, this box is drawn aligning with each axis in coordinate system (X,Y,Z). Hence, it is called Axis Aligned Bounding Box [15].

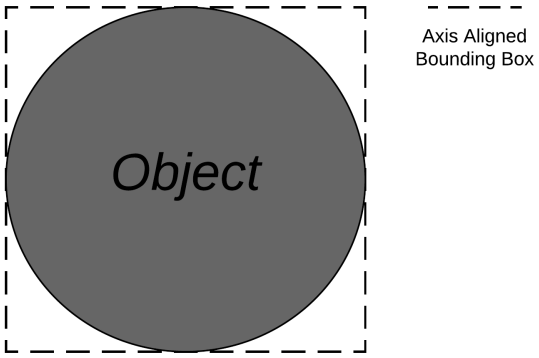


FIGURE 2.3: Axis Aligned Bounding Box of a shpere

We project each box onto each axis, to detect possible object collision. If projection intervals of both objects in X-axis do not overlap, the two corresponding objects do not collide (as shown in 2.4 where $K_1 - L_1$ and $K_2 - L_2$ intervals do not overlap).

On the other hand, Figure 2.5 shows two colliding objects because projection intervals overlap on both axis (X, Y). Therefore, two objects are called colliding when projection intervals overlap on any axis (X, Y, Z).

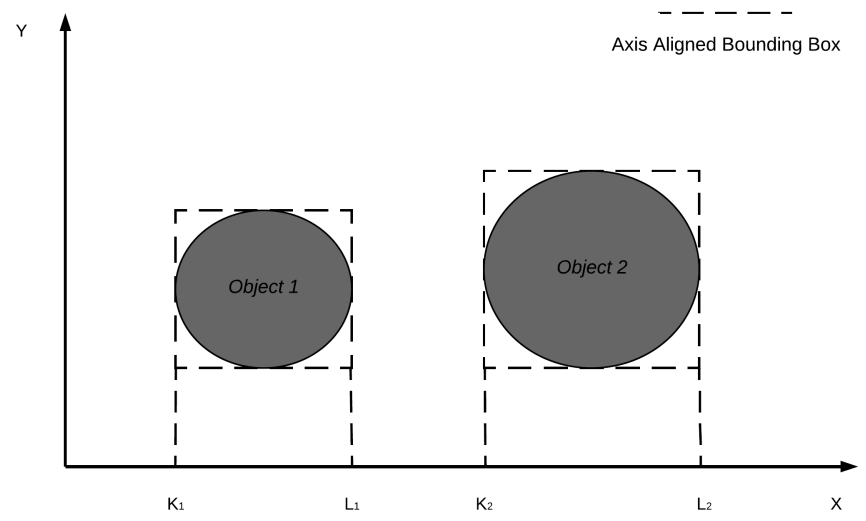


FIGURE 2.4: Two objects with their AABBs that don't collide

There have been developed many methods to solve the broad-phase collision step, like Sweep and Prune (SAP) [13], Spatial Hash and many more. Some of them will be discussed later in this chapter.

Narrow-Phase Collision Detection

In the narrow-phase we are using the detailed geometries of bodies instead of bounding primitives to determine with certainty whether the objects are penetrating or disjoint or to determine the distance between them [24]. Narrow-phase algorithms are slower because the calculations have to be much more accurate. Because of the complexity of this phase, it is preferable to use convex shapes in the real-time physics simulation. In case of non-convex (concave) shapes, the convex hull of the shape will be used for collision detection. A convex set is a set A where for all x and y in A and all t in the interval $(0, 1)$, the point $((1-t)\cdot x)+t\cdot y$ also belongs to A. The convex hull of a set of points S in n dimensions is the intersection of all convex sets containing S. For N points $p_1, ..., p_N$, the convex hull C is then given by the expression:

$$C = \{\sum_{j=1}^{N} \lambda_j \cdot p_j : \lambda_j >= 0 \text{ for all j and } \sum_{j=1}^{N} \lambda_j = 1\} \tag{2.1}$$

Some of the many algorithms for narrow-phase collision detection are Separate Axis Theorem (SAT) [7], Gilbert-Johnson-Keerthi (GJK) [31], and Distance Grid.

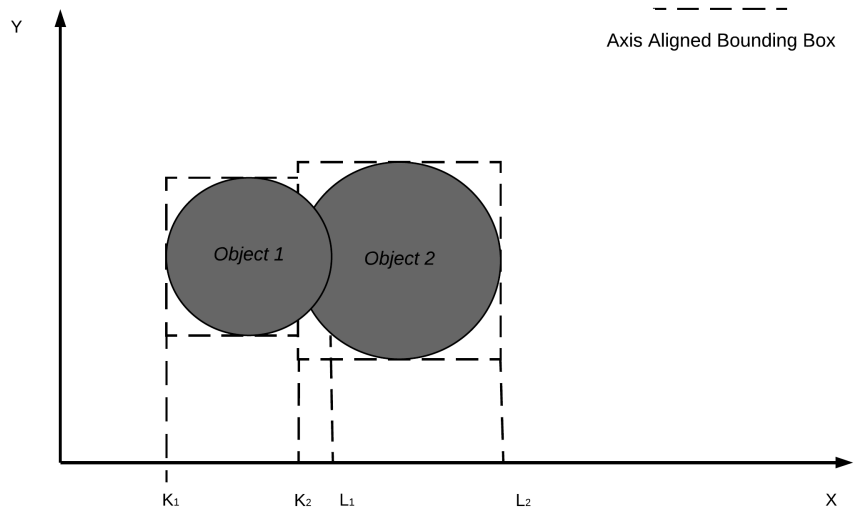


FIGURE 2.5: Two objects with their AABBs that collide

2.1.3 Equations of Motion and Integration

Numerical Integration

The main task of a physics engine is to provide as well as possible an approximate simulation of real world in a virtual environment. The desirable simulation works by making many small predictions based on the laws of physics. These predictions are performed by using a mathematical technique called numerical integration, which is a method to calculate the numerical value of a definite integral. This technique is also used to describe the numerical solution of differential equations. Let's recall the **Newton's second law:**

$$\vec{F} = m \times \vec{a} \tag{2.2}$$

It is also well known that acceleration is the rate of change in velocity over time and that velocity is the rate of change in position over time:

$$\vec{a} = \frac{d\vec{v}}{dt} = \frac{\vec{F}}{m} \tag{2.3}$$

$$\vec{v} = \frac{dx}{dt} \tag{2.4}$$

From the equations 2.2, 2.3, 2.4 we can easily conclude that if we know the current position and velocity of an object, as well as the forces that will be applied to it, we can integrate to find its position and velocity at some point in the future.

Explicit Euler Integration

The physics engines do not analytically solve the differential equations 2.3, 2.4. Instead, they are implementing a numerical integrate. Let us consider that the initial position and velocity of a body are equal to zero, and the acceleration is constant and equal to $10m/s^2$. We are going to take a time step (dt) forward to find the velocity and position at the end of this time step. Then we are going to repeat this, moving forward in more time steps (dt), using the result of the previous calculation as the starting point for the next. We can find the values of position and velocity of a body in a future time by solving the following equations:

$$x_t = x_{t-dt} + v_t \cdot dt \tag{2.5}$$

$$v_t = v_{t-dt} + a \cdot dt \tag{2.6}$$

The above technique is called explicit Euler integration and it is the most basic numerical integration technique. However, some problems arise with this approach because it is accurate only when the rate of change (in our case the acceleration and velocity) is constant over the time step. Lets recall one more equation, to help us find the exact position of a body after time t have passed when the acceleration is constant:

$$s = v_0 \cdot t + \frac{1}{2} \cdot a \cdot t^2 \tag{2.7}$$

In our example the exact position of the body after time $t = 10s$ have passed will be $500m$ according to 2.7.

The integration of velocity on 2.6 is 100% accurate because acceleration is constant. However, we will have a minor error in the integration of position as this depends on the velocity which will changes on each step (it is not constant). If we use time step $dt = 1s$ the body will be at position $450m$ after 10 seconds have passed. We have a fairly large discrepancy in relation to the exact position of the body calculated by 2.7. By reducing our time step down to $dt = 1/100s$ we can reduce this discrepancy significantly. The position of the body after $10s$ will be $499.49m$. As a result, we have an acceptable discrepancy if we use a very small time step and this is why we can use the explicit Euler integration. The difference between the two time steps used is shown in Figure 2.6.

Semi-implicit/Symplectic Euler Integration

When acceleration isn't constant throughout the time-step the results of explicit Euler with small value for dt are not acceptable. The workaround in this

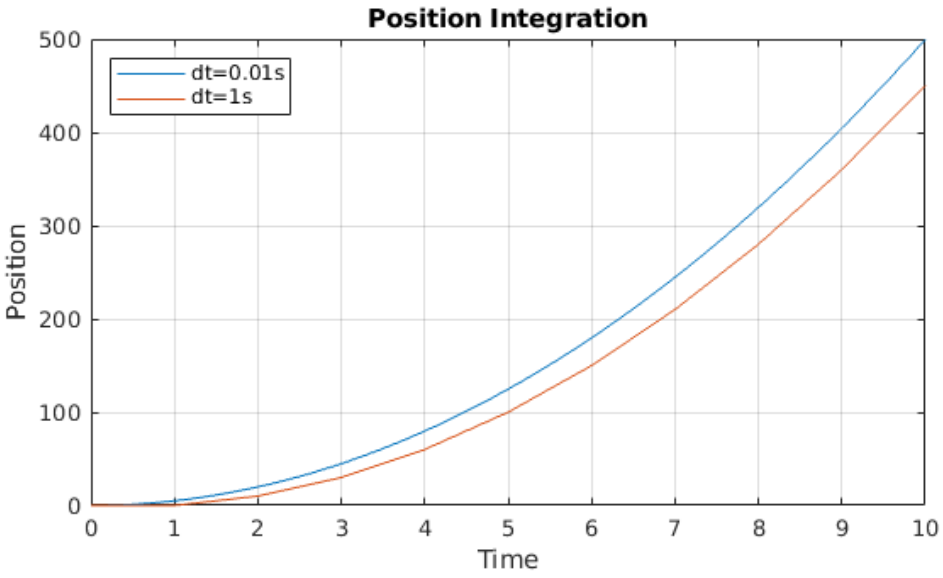


FIGURE 2.6: Position integration with different time steps

problem is to integrate the position and velocity of the bodies with the following equations:

$$a_t = a_{t-dt} + \frac{F}{m} \tag{2.8}$$

$$v_t = v_{t-dt} + a_t \cdot dt \tag{2.9}$$

$$x_t = x_{t-dt} + v_t \cdot dt \tag{2.10}$$

We will calculate the new acceleration with 2.8, then we will calculate the new velocity using 2.9, which we will use to calculate the new position on 2.10. This way the result is very close to the exact solution. This technique is called symplectic Euler. There are many more integration techniques like implicit Euler [20] [4], Runge-Kutta methods [35] [46], Verlet integration [14]etc, but among physics engines, the most commonly used is symplectic Euler.

Newton's Second Law for Rotation Physics engines are simulating scenes composed of rigid bodies, soft bodies, etc. Those objects have shape and size as well as mass. This means they should be able to rotate, so there is a need to represent this rotation in the physics simulation. The 3D environment, in which physics engines are based on, gives rigid bodies 6 degrees of freedom (6DoF), which means a translation in x, y, z axes and a rotation about x, y, z axes. We will extend Newton's second law 2.11 to rotation. We will update the variables in order to solve the equation for rotation. Instead of position x, velocity v, and acceleration a we now have rotation angle θ, angular velocity ω, and angular

acceleration a. We will also replace mass with moment of inertia I, which is the property of an object denoting the resistance to any change in velocity. Finally, we are going to replace force with torque τ, which is a force that has a tendency to rotate an object. We can now express the Newton's second law for rotation:

$$\vec{\tau} = I \times \vec{a} \tag{2.11}$$

We can integrate to calculate the rotation variables of a body over time, in the same way we did it for translation.

2.1.4 Collision Response

The collision response phase is the most complex part in a rigid body simulation loop which intends to be physically accurate. If there are many collisions in the system, the collision response becomes the limiting factor for the overall performance of the physics engine. Collisions are described by some properties that came up during the collision detection step. This step resolves both the interpenetration and velocities of both objects involved in every collision.

Forces on Collisions

The collisions involve two different bodies and the forces determined during the collision response which are acting on these collisions. Therefore, there has to be a way to map these forces to the bodies of each collision. A force $\vec{F}$ is a vector with a line of action [21]. A force produces a moment $\vec{\tau}$ or torque on each point which is not on the line of action of the force. This is represented by equation 2.12, where $\vec{r}$ is the vector from the center of mass of a body to the contact point.

$$\vec{\tau} = \vec{r} \times \vec{F} \tag{2.12}$$

Many forces exist in rigid body dynamics like wind, gravity, and electromagnetic force. However, the factors that are most difficult to deal with, but also critically important in interactive simulation, are the constraints and friction forces. Constraints are equations and inequalities that change the way the pairs of bodies are allowed to move in respect to each other, since they are kinematic restrictions [25]. Friction consists of dissipative forces, that act in collision interfaces to halt sliding (at sliding collisions) and to prevent sliding (at sticking and rolling collisions). For each collision, a single force acting on both bodies is

calculated. However depending on the collision properties this force acts positive on one and negative on the other body. This is determined by the way the local coordinate system of the collision is chosen.

Newton-Euler Equations

The Newton-Euler equations are the result of applying the **Newton's second law** twice, once for motion in translation and again for motion in rotation. In physics, the momentum can be defined as *"mass in motion"* and its represented in 2.13, while angular momentum is represented in 2.14.

$$\vec{p} = m \times \vec{v} \tag{2.13}$$

$$\vec{L} = I \times \vec{\omega} \tag{2.14}$$

Combining 2.2 with 2.13 and 2.11 with 2.14, we can extract the Newton-Euler equations above:

$$m \cdot \frac{dv}{dt} = \vec{F} \tag{2.15}$$

$$I \cdot \frac{d\omega}{dt} + \omega \times I \cdot \omega = \vec{\tau} \tag{2.16}$$

Impulse and Penalty Methods

When two bodies collide they experience very high forces for a short duration. Those forces are referred to as impulsive forces. We observe from equations 2.15 and 2.16, that these forces cause infinite accelerations, which makes direct numerical integration of the Newton-Euler equations impossible. One way to deal with this problem during simulation is to use a standard integration method until the time of impact, then use an impulse-momentum law to determine the new velocities, and finally restart the integrator.

There are algorithms that directly affect the velocities of the intersecting objects, known as Impulse Methods, and algorithms that directly affect the acceleration of the bodies, known as Penalty Methods [34].

The impulse methods allows us to directly affect the velocities of the simulated objects which have intersected. This is achieved through the application of an impulse, which can be thought as an immediate transfer of momentum between the two bodies. In classical physics, impulse is the accumulated force applied on a body over a specific amount of time. The impulse $\vec{J}$ is defined by the equation 2.17 above:

$$\vec{J} = \vec{F} \times dt \tag{2.17}$$

Combining 2.2, 2.3, and 2.17 we get the following:

$$\vec{J} = \vec{F} \times dt = m \cdot a \cdot dt = m \cdot \frac{dv}{dt} \cdot dt = m \cdot dv$$

$$\Rightarrow dv = \frac{\vec{J}}{m} \tag{2.18}$$

We want to calculate both linear and angular impulse to finally calculate the linear and angular velocity changes that have to be applied to the colliding objects. The goal is to give colliding objects a nudge, by changing their linear and angular velocity by an amount equal to 2.18.

The penalty methods for collision response in the physics simulation are more straightforward to implement than the impulse methods, and they have the advantage of directly utilizing the force-based movement implementation. They replace a constrained optimization problem by a series of unconstrained problems whose solutions ideally converge to the solution of the original constrained problem. Those unconstrained problems have a feature called penalty function that replaces, in some way, the constraints.

2.2 The Bullet Physics Engine

The Bullet physics engine makes use of all the methods and concepts discussed above. Of course, there are optimizations and parameterizations in all the techniques to suit its own specifications. A more detailed representation of the rigid body simulation loop, here is called rigid body pipeline, is shown in Figure 2.7.

The main algorithms Bullet uses for collision detection step, are *Sweep and Prune (SAP)* for broad phase collision detection and *Separating Axis Theorem (SAP)* for narrow phase collision detection. In collision response step Bullet uses the *Projected Gauss-Seidel (PGS)*, which belongs to the family of *Linear Complementarity Problems (LCP)* but is implemented in a iterative way to better match a real-time application, such as simulation of a rigid body.

Below we present some basic concepts used that are used by Bullet physics engines:

- **World:** A representation of a real-world scene in a virtual environment.
- **World space:** The coordinate system for the entire scene. Its origin is at the center of the scene.

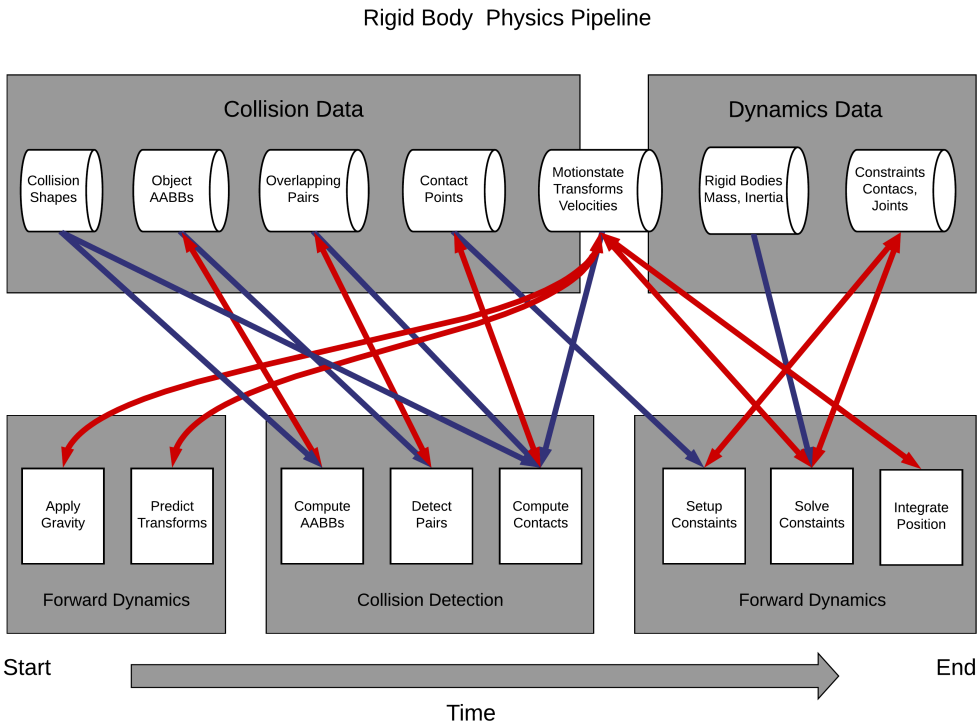


FIGURE 2.7: Bullet's rigid body pipeline

- **Object space:** The coordinate system from an object's point of view. The origin of object space is at the object's pivot point, and its axes are rotated with the object.
- **World transform:** It changes the coordinates from object space to world space. In essence, the world transform places an object into the world.
- **Dynamic Object:** They are rigid bodies whose transforms get updated by the Bullet Physics engine, as opposed to other objects that serve only as potential collision objects.

2.2.1 Sweep and Prune (SAP)

Naturally broad phase is a $O(n^2)$ problem, since in case of having n bodies we have to run n^2 collision tests between all bodies. To reduce the number of pairwise collision tests we make use of Sweep and Prune algorithm. Sweep and Prune actually eliminates group of pairs that are far apart. It is performed in three steps:

- **First Step:** Calculation of the bounding box for each body.

- **Second Step:** Sorting the minimum and maximum coordinated of the bounding boxes.
- **Third Step:** Sweeping through each list and determining which bounding boxes overlap.

The lists mentioned above are the minimum and maximum values of the coordinates of the bounding boxes on each axis (X, Y, Z). We create a list for each axis and we sort each list. There is a flag for each dimension that informs us about whether two bodies are overlapping in this dimension. When all three flags are set then we know that the pair overlap. The flags are modified during the sorting phase and they are toggled based on whether the coordinate values both refer to bounding box minimum value, maximum value or one refers to minimum and the other to maximum value. When a flag is toggled we can conclude one of the following three situations.

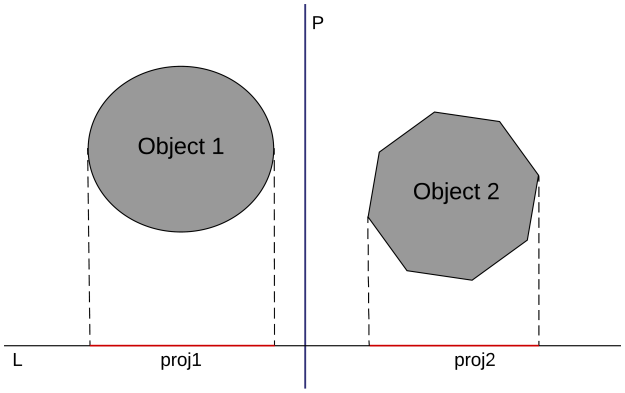
1. Bounding box of the given two bodies overlap in all three dimensions. We are going to add the corresponding pair to the list of active pairs.
2. Bounding box of the given two bodies overlapped in the previous time step. We are going to remove the corresponding pair to the list of active pairs.
3. Bounding box of the given two bodies did not overlap at the previous time step and does not overlap at the current either. We do nothing.

At the end of this process we will send the list of active pairs to the next algorithm i.e *Separating Axis Theorem* to find out which of them are colliding.

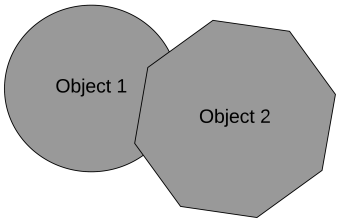
2.2.2 Separating Axis Theorem (SAT)

Bullet has more than one algorithms for the narrow phase collision detection. We are going to give some basic information about *Separating Axis Theorem* technique since it's the one we have used in our implementation. The *Separating Axis Theorem* [28] can determine if two convex shapes are intersecting. In some cases it can also be used to find the minimum penetration vector. Most physics simulations make use of convex shapes and that's what Bullet does, so this algorithm is perfectly suited for our application.

This technique is based on the state that, given two convex bodies, either the two bodies are intersecting or there exists a separating hyper-plane P such that one body is on one side and the other body is on the other side, as it is shown in Figure 2.8.



(A)



(B)

FIGURE 2.8: (A) Two convex bodies, separated by a hyper-plane P. (B) The same convex bodies are now intersecting and therefore there is no hyper-plane to separate them.

In Figure 2.8a we can also notice that the projections, in the L axis, of both objects do not overlap. The L axis is the perpendicular to P and its called *Separating Axis* hence the name of the technique. In Figure 2.8b the bodies are intersecting, its not possible to calculate a plane that separates them. If SAT find an axis where the projection of the shapes are not intersecting it can immediately exit, determining that the shapes are not intersecting. To conclude that two shapes are intersecting, the shape's projections must overlap in all three axes.

2.2.3 Projected Gauss-Seidel (PGS)

Linear Complementarity Problem

The Linear Complementarity Problem (LCP) of finding a vector $x \in R^n$ is defined in the following way:

$$M \cdot x + q \geq 0, x \geq 0, x \cdot M \cdot x + q \cdot x = 0 J \cdot V = 0 \tag{2.19}$$

where M is an $n \times n$ rational matrix and $q \in R^n$ is a rational vector. For given data M and q, the problem is denoted by $LCP(M, q)$[33]. One of the interesting aspects of LCP is its range of applications, from well understood and relatively easy problems such as linear and convex quadratic programming problems to NP-hard problem.

Gauss-Seidel Method

The Gauss-Seidel falls under the category of LCP's as we are going to describe above. Gauss-Seidel is an iterative method that is used to solve a linear system of equations of type $A \cdot x = \vec{b}$, where A is an $n \times n$ matrix, $\vec{b}$ is a vector of length n, and x is the vector of unknowns. The algorithm proceeds for a number of iterations. During an iteration each row of A is solved by adjusting the element of x corresponding to the diagonal element of A on the current row [6]. The algorithm is represented below.

Algorithm 1 Gauss-Seidel method

```
x ← x0                                   ▷ Initializing unknowns x with x0
for iter = 1 to iteration limit do
    for i = 1 to n do                    ▷ Where n = number of elements
        Δ_xi ← [b_i − Σ_{j=1}^{n} A_ij x_j] / A_ii
        x_i ← x_i + Δ_xi
    end for
end for
```

Iterations can be terminated using several different criteria, such as:

- Terminate after a fixed number of iterations.
- Terminate when $||Ax - b||$ falls below a tolerance.
- Terminate when the maximum $|x_i|$ falls below a tolerance.
- Terminate when each $|x_i|$ is less than some fraction of its value in the previous iteration.

In a physics simulation with n rigid bodies the linear and angular velocities of each body are stacked in a $6n$-by-1 vector V as shown below:

$$V = \begin{pmatrix} v_1 \\ \omega_1 \\ \vdots \\ v_n \\ \omega_n \end{pmatrix} \tag{2.20}$$

There are also pairwise constraints between rigid bodies. A single position constraint C_k is represented in the following expression:

$$C_k(x_i, q_i, x_j, q_j) = 0 \tag{2.21}$$

where x and q are the position and quaternion (orientation and rotation in 3D) of the rigid body. All of the constraints in a scene with rigid bodies are collected in a vector C with length s, with s denoting the total number of constraints. The time derivative of C yields the velocity constraint vector. By the chain rule of differentiation, the velocity constraint is guaranteed to be linear in velocity.

$$\frac{dC}{dt} = J \cdot V = 0 \tag{2.22}$$

Where J is the Jacobian. We can calculate the Jacobian in a system of linear equations by following the procedure below:

1. We determine each constraint equation as a function of body positions and rotations
2. We differentiate these constraint equations with respect to time
3. We identify the coefficient matrix of V, which is the J.

In general J is s-by-$6n$. Considering pairwise constraints, each row of J has at most, two non-zero blocks of length six (three scalars for position and three scalars for rotation). So we can define J_{sp} as a s-by-12 array.

$$J_{sp} = \begin{pmatrix} J_{11} & J_{12} \\ \vdots & \vdots \\ J_{s1} & J_{s2} \end{pmatrix} \tag{2.23}$$

Each block J_{ij} is a row vector of length six, where i denotes the constraint number and j the number of body (first or second) involved in the constraint.

Each one of the constraint has an internal reaction force f_c and a reaction torque τ_c. The final vector with the forces and torques of all bodies is represented below.

$$F_c = \begin{pmatrix} f_{c1} \\ \tau_{c1} \\ \vdots \\ f_{cn} \\ \tau_{cn} \end{pmatrix} \tag{2.24}$$

Because of the velocity constraint in 2.21 we can conclude that the velocity V is orthogonal to the rows of J, so do the constraint forces. So we get the following:

$$F_c = J \cdot \lambda = 0 \tag{2.25}$$

where λ is a vector of some multipliers that represent the signed magnitudes of the constraint forces.

Constraint equations are usually partitioned into equality and inequality constraints. In a rigid body simulation we have collision constraints and joint angle limits, which are inequality constraints. For each constraint a lower and upper bound on λ is specified as part of the constraint model.

$$\lambda_i^- \leq \lambda_i \leq \lambda_i^+, \forall i \in [1, s] \tag{2.26}$$

In case of inequality constraint we specify that $(\lambda^-, \lambda^+) = (0, \infty)$.

PGS Algorithm

The Projected Gauss-Seidel [32] [36] is an iterative algorithm based on matrix splitting. It extends the basic Gauss-Seidel algorithm to handle bounds on the unknowns. In our case, these are the bounds described as λ on equation 2.26. In **Newton-Euler** equations we have calculated in 2.15 and 2.16 we now add constraint forces and we get the following two equations:

$$m \cdot \frac{dv}{dt} = \vec{F_c} + \vec{F_{ext}} \tag{2.27}$$

$$I \cdot \frac{d\omega}{dt} + \omega \times I \cdot \omega = \vec{\tau_c} + \vec{\tau_{ext}} \tag{2.28}$$

We collect masses and rotational inertias along the diagonal of a mass matrix M as shown bellow:

$$M = \begin{pmatrix} m_1 E_{3\times3} & 0 & \dots & 0 & 0 \\ 0 & I_1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & m_n E_{3\times3} & 0 \\ 0 & 0 & \dots & 0 & I_n \end{pmatrix} \tag{2.29}$$

Where $E_{3\times3}$ is the identity matrix. Using 2.27 leads us to the constrained equations of motion:

$$M \cdot \frac{dv}{dt} = J \cdot \lambda + \vec{F_{ext}} \tag{2.30}$$

$$J \cdot V = \zeta \tag{2.31}$$

Since our rigid body simulation works with time steps we have to add this time steps to the motion equations we just calculated. Using a time step dt and considering that the volocities are changing from V_1 to V_2 the equation 2.30 is now described as:

$$M \cdot (V_2 - V_1) = dt \cdot (J \cdot \lambda + \vec{F_{ext}}) \tag{2.32}$$

Solving this equation and using the equation $J \cdot V^2 = \zeta$ we end up with:

$$J \cdot B \cdot \lambda = \eta \tag{2.33}$$

where $B = M^{-1} \cdot J$ and $\eta = (1/dt) \cdot \zeta - J((1/dt) \cdot V_1 + M^{-1} \cdot F_{ext})$. This way we reduce the problem to a linear equation in λ.

The algorithm of Projected Gauss-Seidel is shown below.

Algorithm 2 Projected Gauss-Seidel method

$\lambda \leftarrow \lambda 0$ ▷ Initializing bounds λ with $\lambda 0$
$\alpha \leftarrow B\lambda$
for $i = 1$ to s **do** ▷ Where s = number of constraints
    $d_i \leftarrow \mathrm{J}_{sp}(i,1) \cdot B_{sp}(1,i) + J_{sp}(i,2) \cdot B_{sp}(2,i)$
end for
for $iter = 1$ to iteration limit **do**
    for $i = 1$ to s **do**
        $b_1 \leftarrow \mathrm{J}_{map}(i,1)$
        $b_2 \leftarrow \mathrm{J}_{map}(i,2)$
        $d\lambda_i \leftarrow [\eta_i - J_{sp}(i,1) \cdot a(b_1) - J_{sp}(i,2) \cdot a(b_2)]/d_i$
        $\lambda 0_i \leftarrow \lambda_i$
        $\lambda_i \leftarrow \max(\lambda_i^-, \min(\lambda 0_i + d\lambda_i, \lambda_i^+))$
        $d\lambda_i \leftarrow \lambda_i - \lambda 0_i$
        $a(b_1) \leftarrow \mathrm{a}(\mathrm{b}_1) + d\lambda_i \cdot B_{sp}(1,i)$
        $a(b_2) \leftarrow \mathrm{a}(\mathrm{b}_2) + d\lambda_i \cdot B_{sp}(2,i)$
    end for
end for

Where J_{map} is the s-by-2 body map as representing below, with b_{ij} being the index of a rigid body and B_{sp} has been calculated just like J_{sp} in 2.23.

$$J_{map} = \begin{pmatrix} b_{11} & b_{12} \\ \vdots & \vdots \\ b_{s1} & b_{s2} \end{pmatrix} \tag{2.34}$$

Each b_{ij} is the index of a rigid body. By convention, if a constraint is between a single rigid body and ground, then $b_{i1} = 0$ and the corresponding J_{i1} is zero.

The cost of each iteration of PGS is O(s), where s is the number of constraints and n is the number of rigid bodies. An iteration involves simple vector operations on O(s + n) data. The performance of the algorithm is dominated by the number of constraints and the number of iterations used.

Chapter 3

Related Work

In this chapter, we will present a survey that we have done for related works that have been made over the subject that this thesis is dealing with.

3.1 Real-Time Physics Simulation Systems

The last few years have witnessed the continuous increase in physics engines used in the domains of computer graphics, video games and film. The main task of all physics engines is to solve the forward dynamics problem. The forward dynamics problem is to find the final motion of a system knowing the forces acting on it. There are many factors that need to be taken into account in a physics engine such as the simulation paradigm, collision detection and response to the type of numerical integrator, and whether air resistance or friction is considered. As a result each physics engine will provide quite different results despite stimulating the exact same system.

For a game developer many aspects come into consideration including available features, supported platforms, ease of use, and run-time performance. Researchers and simulation engineers are typically more concerned with the accuracy of a physics system. Most physics engines have a particular target application to which they are optimized. This results in different performance in each of the above categories, and often extra features are made available specifically included for the target application. Some of the most used physics engines are, AGEIA PhysX (also referred to as Novodex), Bullet Physics Library, Dynamechs, JigLib, Meqon, Newton Physics SDK, Open Dynamics Engine, OpenTissue Library, Tokamak, True Axis Physics SDK. In [5] is presented a research and comparison between these physics engines.

3.2 Bullet Physics Library

We will initially introduce the implementation of the Bullet library (the library we use as a basis for this work) for acceleration of its GPU physics engine using OpenCL [23]. Bullet is a physics simulation software, especially for rigid body dynamics and collision detection. The original Bullet 2.x is written in modular C ++ and its API was initially designed to be flexible and extendable rather than optimized for speed [11].

Bullet's acceleration approach was based on OpenCL to parallelize and program in GPU. In order to be able to achieve this, most of the code had to be rewritten to use C and structures instead of C ++ and classes with inheritance etc. A high-end desktop GPU has thousands of cores that can run in parallel, so you need to make effort keep all those cores busy. These cores are grouped into Compute Units with typically 32 or 64 cores each. The cores inside a single Compute Unit execute the same kernel code in the lock-step: they are all executing the same instruction, like a wide SIMD drive. The work that is performed by executing a kernel on a single core is called a Work Item in OpenCL. To make sure that multiple work items are executed on the same Compute Unit, you can group them into a Workgroup. The Workgroups to Compute Units, and this makes OpenCL scalable: if you add more Compute Units, the same program will run faster. The drawback is that there is no synchronization between Compute Units, so you need to design your algorithm around this. The host can wait until all work groups have finished, before starting new work.

This implementation has achieved the execution of rigid body and collision detection on the GPU using OpenCL, which has resulted in acceleration compared to original implementation and also the opportunity to simulate scenes with many rigid bodies (up to 100K).

3.3 Field-Programmable Physics Processor

In [29] it is proposed an alternative way to represent the concept of physics, through the creation of physics engine hardware, similar to the AGEIA PhysX. However, it proposes the use of Field-Programmable Gate Arrays (FPGAs), whose re-congurability should provide unique advantages. They have developed a numerical integrator which formed the basis of their FPGA-based physics engine. If a game performs many physics calculations, the FPGA could be used for accelerating these calculations. If, instead, a game performs many AI computations, the FPGA could be used to accelerate these AI routines. The

adaptability of FPGAs is illustrated by the way many diverse applications, such as ray tracing [40] and MATLAB computations [1], have already been accelerated using an FPGA. Through its re-congurability, the FPGA allows a multitude of tasks to be accelerated. Physics engines rely heavily on their implementation of numerical integration algorithms. Although as stated in [30] the simple Euler integrator is adequate for some kind of applications, such kind of traditional numerical integration algorithms do not perform well on FPGAs as they consist of a large number of dependent operations, leaving little scope for parallelism. In [29] there is a comparison between the more sophisticated Verlet[27] family of integrators, Runge-Kutta[35] integrators and Euler[20][4] integrators. It is concluded that Runge-Kutta integrators unlike the Euler and Verlet integrators are reversible in time, which is highly desirable for computer game applications.

3.4 FPGA Acceleration of Molecular Dynamics Simulation

There are many projects dealing with the acceleration of molecular dynamics simulation, which is a technique for modeling the motion and interaction of atoms or molecules using the equations of classical Newtonian mechanics[47]. Molecular dynamics (MD) simulation[27] is one of the most important tools for observing those critical biology phenomena. Basically, it simulates the motions of the molecular systems at an atomic level by 10^6 to 10^12 iterations for practical usage, which makes it very time consuming. At each iteration, it first calculates the forces applied for each atom and then updates the atom's motion. This process is very similar to the process PGS uses to update the rigid bodies position in a game scene. Besides that MD simulations and physics engines have more common concepts, such as numerical integration, Newtonian equations of motion and even more.

An FPGA acceleration of MD in high-level synthesis (HLS) is being analyzed in [26], so as to provide affordable programming cost. It demonstrates that HLS optimizations such as loop pipelining, module duplication and memory partitioning are essential to improve the performance ver CPU. Another research about MD can be found at [8], where some hardware structures are being implemented for computing the more time-consuming parts of molecular simulation. More specifically two types of hardware engines that compute the Lennard-Jones and Ewald Direct Space non-bonded interactions[37] have been

developed. One more work on MD has been developed in [47] where is given an overview on FPGA-based MD Simulations, and a way to explore the feasibility of FPGA-accelerated MD simulations.

Chapter 4

Modeling of PGS and Integration

In this thesis, we will focus on the collision response step of the rigid body simulation. In Bullet physics engine, as in many others, this simulation step is implemented by Projected Gauss Seidel algorithm, as it was presented in Chapter 2. In order to be able to achieve the best possible results, we have proceeded with an extensive analysis of the algorithm we are aiming to implement. Above we quote all of the analysis and pre-processing we have performed in order to achieve the best possible results.

4.1 OpenCL to C++ Conversion

Bullet's implementation of rigid body simulation uses OpenCL to accelerate certain parts of the algorithm in GPU. GPU is used as the device that OpenCL needs to execute the part of the algorithm that being accelerated. CPU is also needed as a host for initialization of the device memory and to start the execution of the program in the device. The portion of the code running on the device is called kernel.

In Bullet's case, PGS is implemented by three different kernels that communicate with each other in order to execute the algorithm. In our implementation on FPGA, OpenCL was not the best choice since we could not make use of the streaming feature. Firstly we had to reform every kernel in order to run in C++. Since OpenCL is very similar to C, this procedure was like converting C to C++. The only point that needed particular attention was the creation of the loops that each kernel "hid" internally. This stems from the way OpenCL works with work-items and work-groups. Work-groups contain multiple work-items that can be executed in parallel, while each work-item executes the algorithm with different input data. When a kernel is executed there are many work-items that being executed, so in order to convert into C++, we need a loop with a size equal to the number of work-items. Finally, we combined the three converted

kernels together and we formed one final function which implements the PGS algorithm in C++.

4.2 Isolating Processing from Rendering

The goal of a physics engine in to represent a real-world scene in a virtual environment. This cannot be achieved without rendering/drawing the desirable scene in the GPU. Rendering alongside with processing are the two main components of a physics engine. evertheless, these two parts are not completely isolated from each other, contrariwise there is a connection between them during the execution of a simulation loop. Their connection takes place when the processing part has made the necessary calculations and has to send the data to the rendering part so that the second implements the rendering/drawing of the current "snapshot" of the scene. Their connection takes place again when the rendering of the scene has finished and the data are being sent in again for processing. Bullet transfers the data between processing and rendering parts, using OpenGL and its features. We could not have the rendering part in the FPGA, so we had to isolate these two parts of the simulation loop in order to proceed with the implementation in FPGA. We did some research on how OpenGL works and how it transfers the data, and we finally have isolated the two parts from each other. In that point we were able to start analyzing the PGS and Integration algorithms we were aiming to implement.

4.3 Demo Description (Game Scene)

Bullet has already implemented a demo of a scene with many rigid bodies colliding with each other. The scene consists of a pile of 38880 cubes, which start from a certain height, end up falling over a terrain and colliding with each other until finally, they come to rest. This is the complete demo. We have isolated and implemented only the processing part of the demo, hence we are not going to render/draw the scene. In our implementation we are aiming to accelerate a certain part of the processing of the demo and this is the collision response and integration. Below, in Figure 4.1 below we present a "snapsot" of the demo as it is implemented by Bullet:

This demo uses two types of rigid bodies, dynamic and static. Dynamic rigid bodies are objects that can move during the simulation loop, have positive masses, and their world transform is being updated in every simulation frame. All of the cubes represented in our demo are considered dynamic rigid bodies.

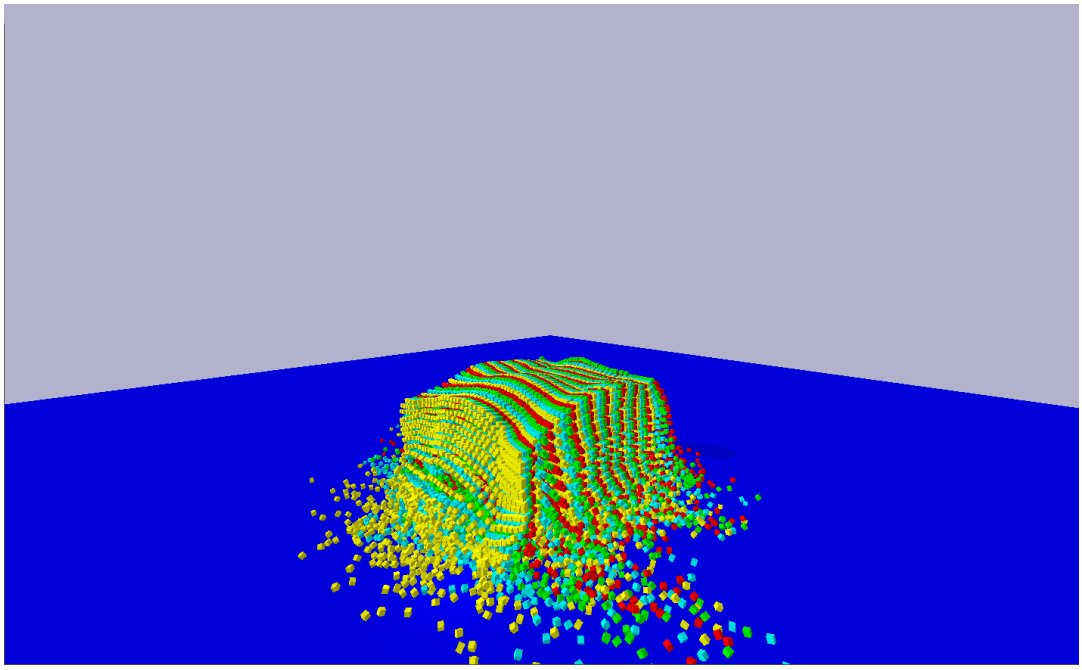


FIGURE 4.1: A snapshot of the demo we are going to work on

On the other hand, static rigid bodies have zero masses, and cannot move but can collide. We have only one static stiff body and this is the surface on which the dynamic rigid bodies strike.

4.4 Data Analysis of PGS and Integration

The demo we choose is a computational intensive process since it uses a large number of simulated bodies. The simulation loop has to find the pairs of contacting bodies and "solve" these collisions to finally move the bodies to their new positions. After counting the time it takes each step to execute, it became clear that the step that resolves the collisions between the bodies is the bottleneck of the whole process. More precisely the step which finds the collisions between the bodies consumes the 40% of the time, while the step which is responsible for the collision resolving consumes the 60% of the time. This is why this thesis has focused on accelerating the specific step of the algorithm. Before we start the implementation in Vivado HLS for the FPGA, we performed an extended analysis for our algorithms in MATLAB. This analysis was performed in order to fully understand all the aspects of the algorithms and to be able to take advantage of any possible data redundancy and algorithmic optimization.

4.4.1 Data Redundancies

PGS Data Reduction

The PGS algorithm gets as input the following:

- **Bodies:** Algorithm needs all the bodies and their properties in order to calculate the new velocities for each one of the them. More specifically it needs the linear and angular velocity, mass, position and inertia of each rigid body in the scene.

- **Collisions:** From previous steps in the simulation loop, all the collision pairs have been calculated. This algorithm needs this information as input in order to "solve" the collisions. A collision between two bodies consists of two indexes indicating the number of the two bodies colliding and the world position of bodies.

Since we are on 3D world we need 3 components to represent a body's position, velocity etc, one for each axis (x, y, z). Bullet by default uses a forth unused component for alignment and SIMD compatibility reasons. So, in our implementation we cut off this component since it has no practical information that is needed in processing.

As we mentioned above PGS needs inertia as input. By default this parameter is being sent into the algorithm as a 3×3 diagonal matrix. It is easily perceived that we should send only the diagonal of the table as all the remaining values of the matrix are zero. We also know from Chapter 2 that inertia is the resistance, of a body, to any change in its velocity and also a basic manifestation of the mass. Knowing that we have a scene full of cubes of the same mass and considering the definition of inertia we can calculate the inertia of a body inside the algorithm by passing inside only the type of shape and the mass of the body. These two parameters are only needed in order to calculate the inertia of a body according to Bullet's implementation. In our demo, the inertia of all bodies is the same so we can represent it with a constant value which is being sent once at the start of the algorithm.

This can easily generalized for all the primitive shapes in a physics simulation. All we have to do is to pass the necessary information for each shape one time at the start of the algorithm. Afterward, during the simulation loop, we will send only an index indicating in which primitive shape this body is included and its mass, and we will calculate the inertia of the rigid body.

The total data we need to send in the algorithm depends on the number of collisions. This changes in every iteration of the simulation loop. In our demo,

the most intensive iterations of the loop are having total collisions in a range of 90.000 to 130.000. We have implemented one iteration of the simulation loop with 127.000 collisions. In the Table 4.1 we present the data size we have to send into the algorithm in every iteration as also the total data size we have to pass in PGS as input pre and post-reduction.

TABLE 4.1: Data reduction on input of PGS

Input Data	#Data	Memory Footprint	Stream Reads
Pre-Reduction(Per Collision)	56	224(B)	14
Post-Reduction(Per Collision)	28	112(B)	7
Pre-Reduction(In Total)	7423040	28.32(MB)	1855760
Post-Reduction(In Total)	3662280	13.97(MB)	915570

The outpout data of the algorithm are the linear and angular velocities of the bodies. In order to send that data back to the simulation loop we are passing their values on output streams. The data reduction on output of PGS is shown in Table 4.2 The outpout data of the algorithm are the linear and angular velocities of the bodies, so we are reducing the output size by 2 floats in each iteration of the loop which writes on output streams, or we are reducing the output size by 77760 floats in total.

TABLE 4.2: Data reduction on output of PGS

Output Data	#Data	Memory Footprint	Stream Writes
Pre-Reduction(Per Collision)	8	32(B)	2
Post-Reduction(Per Collision)	6	24(B)	1.5
Pre-Reduction(In Total)	311040	1.19(MB)	77760
Post-Reduction(In Total)	233280	0.89(MB)	58320

Integration Data Reduction

The Integration part of the simulation loop gets as input the following:

- **Linear and Angular Velocity:** Integration needs linear and angular velocity of each body in order to calculate its new position and linear velocity.
- **Rotation and Position:** In also gets the rotation and position of each body in order to update them.

- **Mass** The mass of every rigid body is needed in order to calculate the final velocities.

We have performed the same data redundancies as in PGS algorithm but we have managed to reduce the input on Integration part a bit more. In our implementation we have incorporate the Integration and the PGS in a single function. By doing that we have avoided to send the velocities from one part to the other. We can take the velocities directly from BRAM. The rest of the data will be send over stream. In Table 4.3 we can see the data redundancies applied on Integration per iteration, and in total. The third row of the Table below is representing a projected result which could arise if we have also stored the positions of the bodies in BRAM, we will analyze this more in the next subsection.

Table 4.3: Data reduction on input of Integration

Input Data	#Data	Memory Footprint	Stream Reads
Pre-Reduction(Per Collision)	17	68(B)	4.25
Post-Reduction(Per Collision)	7	28(B)	1.75
Post-Reduction & Positions in B-RAM(Per Collision)	4	16(B)	1
Pre-Reduction(In Total)	660960	2.52(MB)	165240
Post-Reduction(In Total)	272160	1.03(MB)	68040
Post-Reduction & Positions in B-RAM(In Total)	155520	0.59(MB)	38880

The output data size of Integration pre and post-reduction are presented below in Table 4.4.

Table 4.4: Data reduction on output of Integration

Output Data	#Data	Memory Footprint	Stream Writes
Pre-Reduction(Per Collision)	8	32(B)	2
Post-Reduction(Per Collision)	6	24(B)	1.5
Pre-Reduction(In Total)	311040	1.19(MB)	77760
Post-Reduction(In Total)	233280	0.89(MB)	58320

PGS and Integration Combination

We have implemented both PGS and Integration in a single function instead of two different functions with the first passing the output data to the second.

This way we were able to reduce further the total size of input data in our function, as also the total size of the output data. In Tables 4.5 and 4.6 we can observe the data reduction we achieve by this process in combination with the data redundancies applied to our initial data. Pre-reduction rows are calculated by the hypothesis that the two parts, PGS and Integration, are being processed separately.

Table 4.5: Data reduction on input of PGS and Integration combination

Input Data	#Data	Memory Footprint	Stream Reads
Pre-Reduction(Per Iteration)	73	292(B)	18.25
Post-Reduction(Per Iteration)	35	140(B)	8.75
Post-Reduction & Positions in B-RAM(Per Iteration)	32	128(B)	8
Pre-Reduction(In Total)	8084000	30.84(MB)	2021000
Post-Reduction(In Total)	3934440	15.00(MB)	983610
Post-Reduction & Positions in B-RAM(In Total)	3817800	14.56(MB)	954450

Table 4.6: Data reduction on output of PGS and Integration combination

Output Data	#Data	Memory Footprint	Stream Writes
Pre-Reduction(Per Iteration)	16	64(B)	4
Post-Reduction(Per Iteration)	6	24(B)	1.5
Pre-Reduction(In Total)	622080	2.37(MB)	155520
Post-Reduction(In Total)	233280	0.89(MB)	58320

4.4.2 Memory Footprint

As we can see in Algorithm 2, PGS is executing the outer loop of processing more than one times in order to identify the active constraints and to be more accurate in the resolution of the impulse propagation [12]. We are also some components of the rigid bodies in BRAM to avoid passing the same values again and again with the stream.

In our implementation with floating point we need to keep in BRAM the data for the linear and angular velocity of the bodies as long as with a vector of length four, which is needed for every collision. This vector passes the information of

the solution of this collision from previous iteration of PGS outer loop to the present so that we are as accurate as possible. So the total size of the data we need to store in BRAM is 2.83 MegaBytes.

One would think that in the implementation with the half-precision floating point we would need exactly the half memory of the floating point implementation, and this thought is right. But in half-precision floating point approach we needed a copy of the velocities table to get better results in the latency. That's why we needed a total of 1.86 MegaBytes instead of 1.41 MegaBytes.

We could have also stored the position of each rigid-body in BRAM, in order to reduce the stream reads we have to do in every single iteration of the PGS inner loop. That would require 0.44 additional MegaBytes in floating point and 0.22 additional MegaBytes in half-precision floating point. In floating point the final 3.27 MegaBytes needed, is close to the upper limit in BRAM in ZCU102, thus its hard to implement this. In half-precision floating point we will have to create a copy of this array too but still there is no problem to tether BRAM equal to 2.30 MegaBytes.

4.5 Lack of Determinism in GPU Implementation of PGS

Pairwise constraints such as contact constraints between two bodies need to be solved sequentially in the PGS algorithm so that the most up-to-date velocity is available for each constraint. It is not possible to trivially update the velocities for bodies in each constraint in parallel because there are read-write conflicts. GPU, in order to "solve" all the collision constraints in parallel, creates independent groups of collisions (batches), where the constraints in each batch don't have read-write access to the same bodies.

However, even if these batches in a GPU are executed in parallel, the order in which collision constraints are being solved can be different each time. This non-determinism, or lack of consistency, can affect the results between different executions of the demo. If we have a different order of overlapping pairs, and contact points, we may also have a different order of collision constraints. The Projected Gauss Seidel algorithm produces different results, if the constraint rows are solved in a different order.

We executed our demo numerous times in the GPU with exactly the same input and kept a snapshot of the same physics step. We noticed that the positions of the bodies in these snapshots of the demo are not the same and

this is due to the lack of determinism. However, all of the demo executions are acceptable based on physics. The Figure 4.2 below shows the difference in position of the bodies between two different executions of the demo, expressed as deviation calculated by the following equation:

$$Deviation_{position} = \frac{|newValue - originalValue|}{|originalValue|} \cdot 100\% \tag{4.1}$$

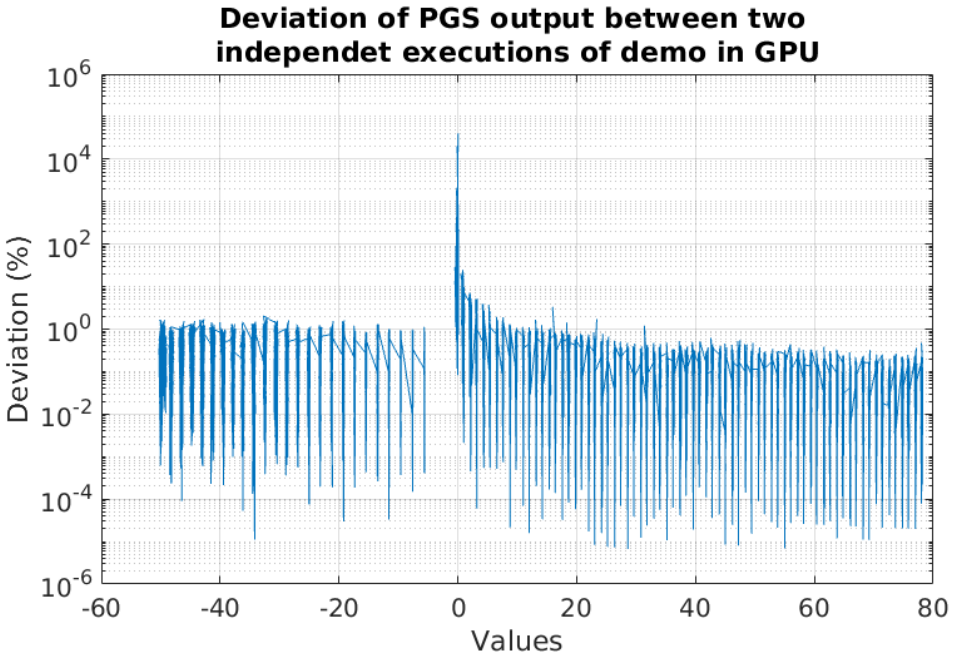


FIGURE 4.2: The deviation in positions between two executions of demo. The Y-Axis has logarithmic scale

The deviation of PGS output between independent executions of demo is varies between 1% and 3%. This is an acceptable error in physics engines because it does not ignore the laws of physics in the scene and because the real change in the position of the bodies is negligible within the overall scene. This error is derived from the lack of determinism that exists in Bullet physics engine implementation on GPU.

4.6 Comparison of Float and Half-Float

We have implemented the PGS and Integration algorithms with two different approaches. Firstly we used the floating point precision to represent the values in our algorithm. We noticed that the values do not have a big range on the integer part. We also noticed that even if we lose some resolution, the final results will be really close to the desired ones. This led us to think of using

half-precision floating point to represent our values. We have implemented the floating point to half-precision floating point conversion in MATLAB using a custom toolbox we found [39]. We have also used the Vivado HLS library for half-precision floating point arithmetic [42]. We have compared those two methods and we conclude that they give exactly the same results.

4.6.1 Float to Half-Float Conversion and Vice-Versa

The IEEE 745 floating point representation uses a sign bit, an 8-bit exponent with a bias of 128, and a 23-bit mantissa as is shown in Figure 4.3. Biasing is used because exponents have to be signed values in order to be able to represent both tiny and huge values, but two's complement, the usual representation for signed values, would make comparison harder. A biased exponent is the result of adding some constant (called the bias) to the exponent chosen to make the range of the exponent non-negative [38]. In order to interpret, it is converted into an exponent within a signed range by subtracting the bias. To convert a

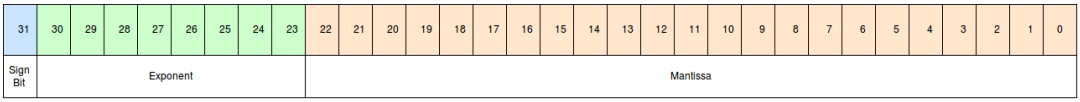


FIGURE 4.3: The format of a floating point number

floating point number into a half-precision floating point number the first step is to shift and mask the sign bit, then to mask off the exponent, and to subtract the bias-correction. The result is shifted, and finally the mantissa is shifted and masked off. All the pieces are then assembled together to compose the half-precision floating point representation of the initial floating point number. This process is working for the general cases but it doesn't handle zero, Infinity, NaN, or small float numbers which are only presentable as subnormal half-floats [48]. These special cases are being solved by using some arrays of constants, but we will not further analyze these cases.

The half-precision floating point data type in IEEE 754 standard, sacrifices range and accuracy in favor of representation size. A half-precision floating point is composed of a sign bit, a 5-bit exponent with a bias of 15, and a 10-bit mantissa as is shown in Figure 4.4. Conversion of half-precision floating point number to floating point number is implemented by executing the following steps: copy the sign bit, subtract the half-precision floating point bias (15) from the exponent and add the floating point bias (127), finally append 13 zero-bits to the mantissa. The above conversion it does not work for special

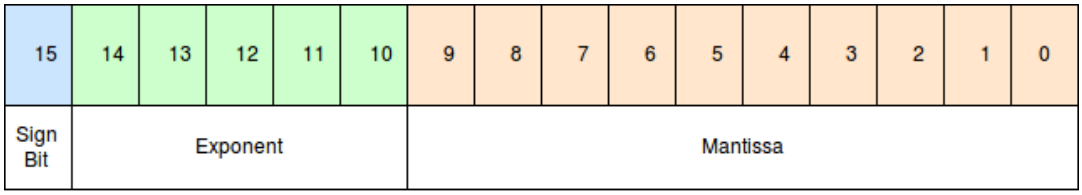


FIGURE 4.4: The format of a half-precision floating point number

cases such as zero or subnormal (denormal) numbers [48]. In order to be able to convert half-precision floating point numbers that belong to these special cases some arrays containing constants are needed, but we are not going to analyze the procedure they execute.

4.6.2 Half-Float Representation of Input Data

We have converted the data that our algorithm takes as input, from floating point into half-precision floating point using MATLAB. We are going to juxtapose them and to inspect whether the error that occurs is acceptable for our application or not. The distribution of the input data for the floating point is shown in Figure 4.5, and for the half-precision floating point in Figure 4.6.

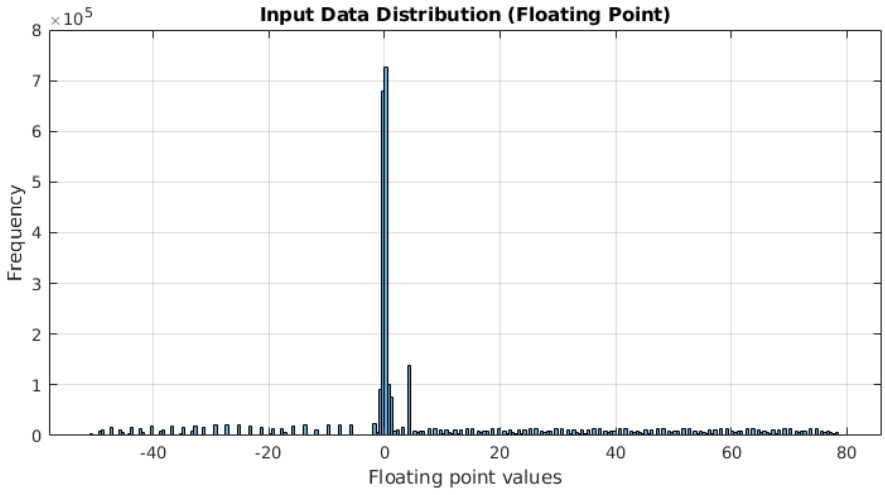


FIGURE 4.5: The distribution of the input data of algorithm expressed in floating point

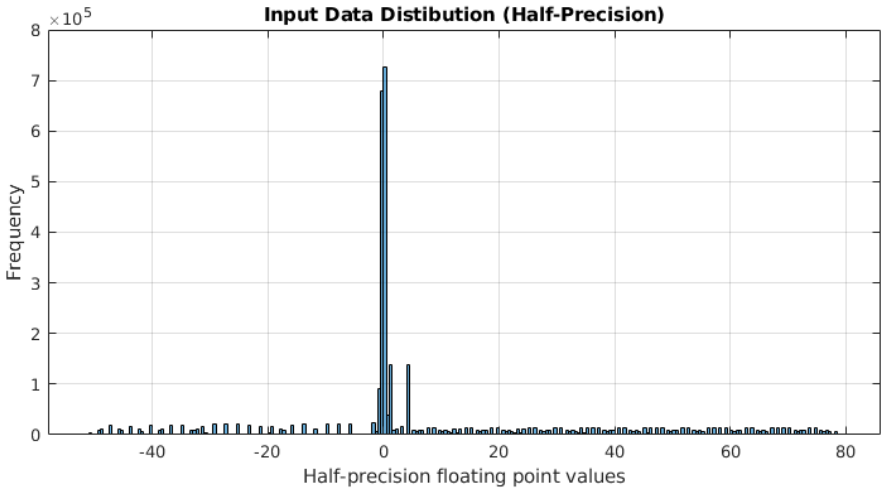


FIGURE 4.6: The distribution of the input data of algorithm expressed in half-precision floating point

As we can see from the two histograms, the half-precision floating point values are very close to the original floating point values. In order to be able to better observe their difference, we have calculated the error between the values of half-precision floating point compared with the initial floating point values. This error is a measure that indicates how close the values are in the two representations. The expression we used to calculate the error is shown in 4.1.

In Figure 4.7 below we present a plot showing the percentage error of the converted half-precision floating point values relative to the initial floating point values. As we can see the error in values with zero integer part is high enough. However, this figure is not showing the whole truth. As we mentioned above when we have values with zero integer part just a small change in the value can lead to a really high error compared to the initial value.

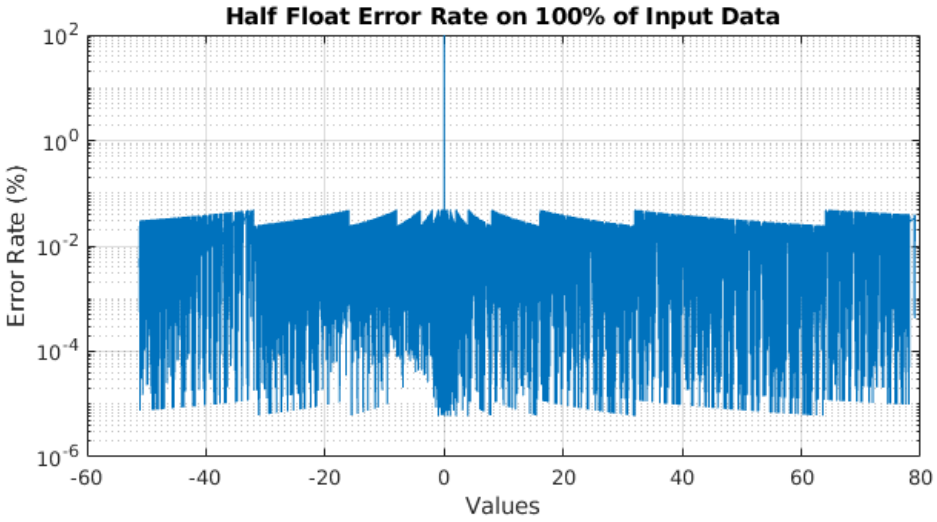


FIGURE 4.7: The error rate of half float compared to float on 100% of input data. The Y-Axis has logarithmic scale

In our case the 99.93% of the values have error rate less than 1% as presented in 4.8 . The rest of the values (0.07%) with the high error rate are of the order of 10^{-4} and even smaller, but this is an accepted error since the real change in the position of the bodies is negligible within the overall scene. The average error of the half-precision floating point values compared with the initial floating point values is equal to 0.02%.

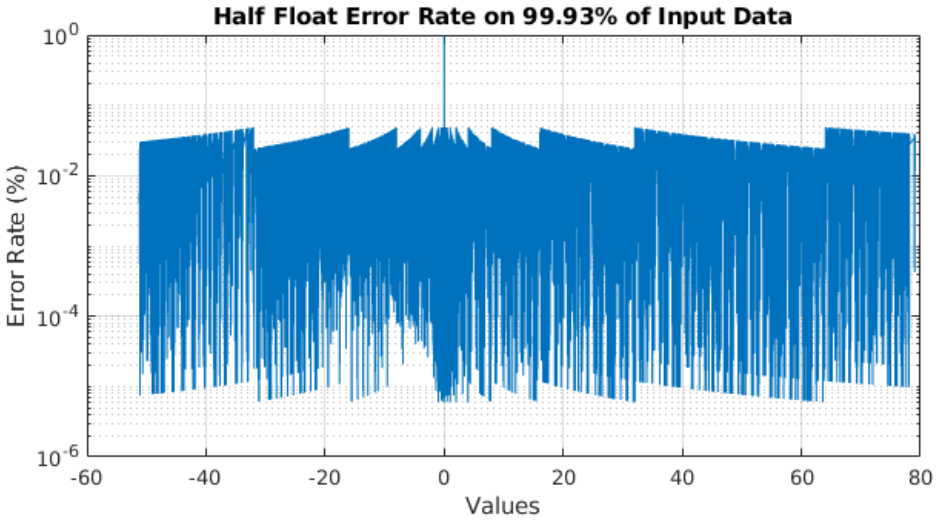


FIGURE 4.8: The error rate of half float compared to float on 99.93% of input data. The Y-Axis has logarithmic scale

4.6.3 Evaluation of Results Using Half-Float Representation

Output of PGS using Float

Thereafter we have executed the algorithm in FPGA using float values as input, and we compared its output with the GPU's. The deviation on output values of then algorithm in FPGA using float values as input is 2.97% compared to the GPU's output. This is presented in Figure 4.9. As we mentioned above in the analysis about GPU's lack of determinism, GPU is executing the algorithms based on some batches it creates. The deviation we observe between FPGA's and GPU's implementations derives from the fact that FPGA's implementation is deterministic while GPU's is not. This is within the permissible limits for the deviation of the final positions of the bodies. However, it was calculated that 84.29% of the position's values from FPGA implementation has deviation less than 1% compared with the position's values from GPU implementation.

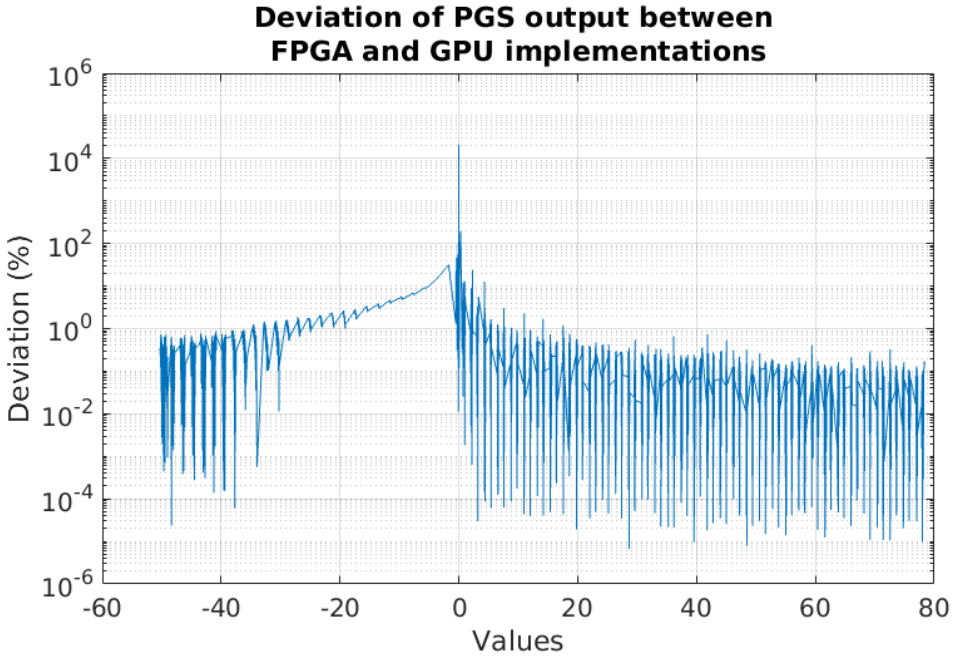


FIGURE 4.9: The deviation of PGS output between FPGA's and GPU's implementation using floating point. The Y-Axis has logarithmic scale

Output of PGS using Half-Float

We have executed the algorithm in FPGA using half values as input, and we compared its output with the GPU's. We did not notice a big difference in

results of FPGA float comparison with GPU. In order to be sure about the validation of results in FPGA using half-float values we have also compared this implementation with the FPGA implementation using float values.

The error rate between those two implementations is shown in Figure 4.10. We saw that the 99.64% of the position's values from FPGA implementation with float has error less than 1% compared with the position's values from FPGA implementation with half-float and that the average error on output values is 0.06%. Knowing that our FPGA's implementation with float produces valid results and knowing that we have such a small error between half-float and float implementations we can tell for sure that our implementation with half-float leads to valid results.

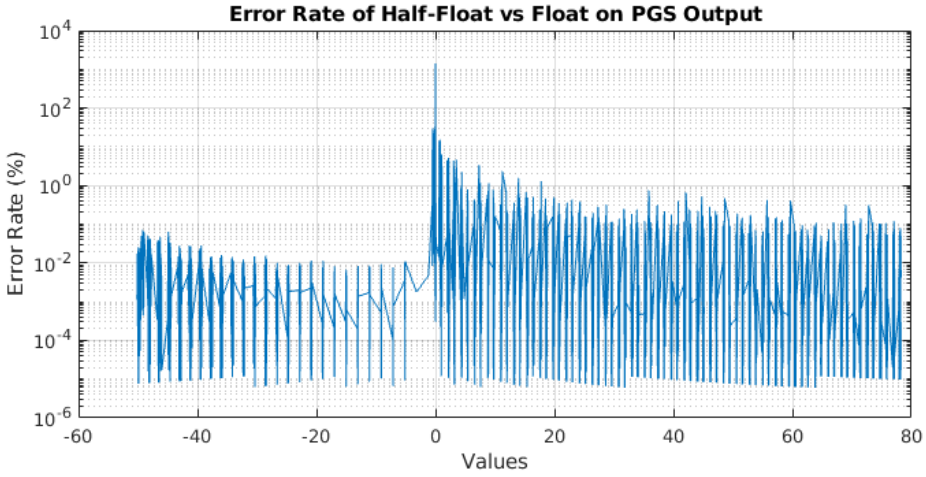


FIGURE 4.10: The error rate of PGS output between FPGA's implementation with float and half-float. The Y-Axis has logarithmic scale

4.6.4 Half-Float to Float Conversion

In our quest to create the implementation of PGS and Integration with half-precision floating point values, we have encountered some problems. Converting a floating point number to a half-precision floating point gives us a 16-bit unsigned integer value. The parts of the half-precision floating point number, i.e. sign bit, exponent, and mantissa, are generated by the conversion process, and they produce a binary value that is represented by the 16-bit unsigned integer mentioned above. Vivado HLS does not allow us to pass directly half-precision floating point numbers to the FPGA. For this reason we had to pass its integer representation and convert it to a half-precision floating point within the function.

The first approach was to use the library provided by Vivado HLS in order to manipulate half-precision floating point numbers. On the test bench, all conversions between 16-bit integer and half-precision floating point worked fine. But they could not pass from synthesis. So we had to find a different way to make these conversions.

We first thought of using the union just as we use it to represent the floating point in a 32-bit unsigned integer and vice versa. But this conversion was not possible on half-precision floating point. We then implemented the logic that the union uses internally, parameterizing it to the demands of our own problem. In particular, we specified a memory address of type "void" to store the integer representation of the half-precision floating point and then we were reading from the same address by casting to the type that we wanted i.e. half. This approach also worked on the test bench but did not go through the synthesis.

Since we did not have any other options, we proceeded to a solution trying to create a 16-bit representation of a floating point number. We observed the representation of floating point in 32-bit integers in contrast to the 16-bit representation of the half-precision floating point. After many tests with different numbers, we found a pattern in the process of conversion. The first bit is the sign bit, so we keep it as it is. We also noticed that the second bit i.e the first bit of the exponent indicates if the exponent is negative or positive or equivalent if we need to shift left or right, so we had to keep this bit also untouched. The range of the values in our application lies between $3.0 \cdot 10^{-5}$ and 450.0. Knowing that, we can be sure for the values of the 2_{nd}, 3_{rd} and 4_{th} bit of the exponent for every value in our application. If the 1_{st} bit of the exponent is equal to 0 then the next three bits will be equal to 1. If its equal to 1 then the next three bits will be equal to 0. We only need to keep the 4 LSB of the exponent. Finally we need the 10 MSB of the mantissa. The combination of those three parts gives us the 16-bit integer representation of a floating point number as shown in Figure 4.11. This way we can represent values in a range of $6.10352 \cdot 10^{-5}$ to 512.0 so we are able to represent all the values in our application.

In order to covert back to 32-bit representation of floating point value we apply the inverse process by adding the 3 bits we have not include in the exponent and zero padding the last 13 bits of mantissa as is shown in Figure 4.12.

Depending on the range of values in an application we can cut of bits from the exponent and add more bits to mantissa to achieve better precision. We can also represent a floating point value with less than 16 bits cutting of bits from mantissa. This way we sacrifice precision in order to lower the number of bits we are going to use.

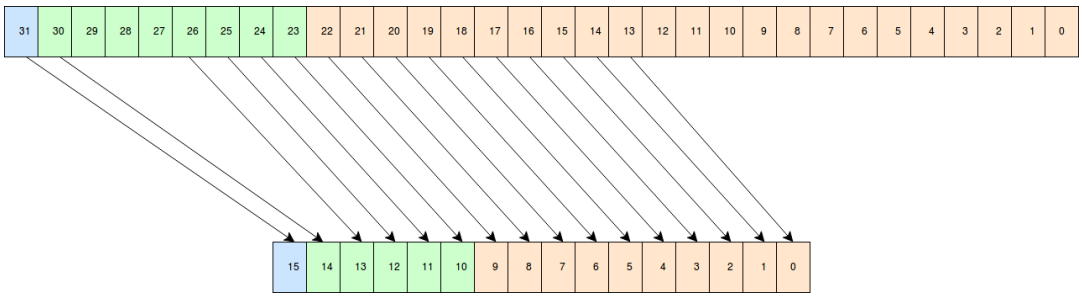


FIGURE 4.11: Conversion of a floating point value to half-precision floating point

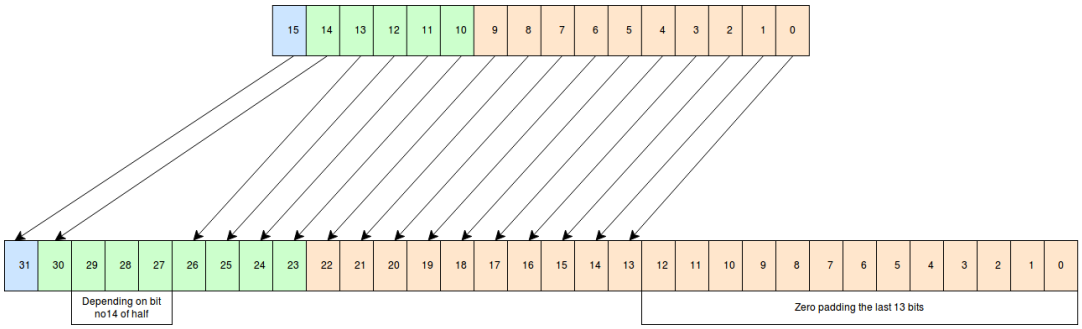


FIGURE 4.12: Conversion of a half-precision floating point value to floating point

This custom conversion does not work in all cases. There are special cases like subnormal numbers etc. which need a different approach. If we want to include these special cases in our conversion we need some arrays of constants and a more complex conversion, something we have not implemented since we did not need it. The way this conversion is implemented can be found in [48].

Besides all the previous work we have also found a workaround with a different library of Vivado HLS to implement the conversion easier. The library mentioned above is "x_hls_utils.h" [42].

4.7 Sorting Input Data on PGS

Bullet's implementation of the PGS algorithm in GPU uses a pre-sorting on input data before moving to the main processing part of the algorithm. The process of solving pair-wise constraints means that the data for each of the bodies involved must be updated. That's why solving multiple constraints is not embarrassingly parallel because there might be needed access to the same bodies. This is why GPU is sorting the constraints in independent batches, where the constraints in each batch don't have read-write access to the same

bodies. This sorting on input data also helps GPU to execute the processing of the constraints in parallel and so achieve better execution times.

4.7.1 Possible Benefits using Sorting

Based on this idea we implemented in MATLAB a sorting wise algorithm so that we can exploit this parallelism in the FPGA. We started with the idea of resolving all the collisions concerning a particular body. Then we will resolve the collision of the active (body that has already been processed in a previous collision) body with the fewest remaining collisions. We have followed this deapth first search approach so that we can start the second iteration of the external PGS loop without having to complete the first one. That way we create a pipeline between the iterations of the PGS outer loop and we can achieve better latency speedup and reduce the BRAM footprint significantly. Implementing the PGS without sorting the input we need to keep data (i.e linear and angular velocities) for all 38880 rigid bodies in our memory. In case of PGS with sorted input we found that in the worst case we need to store data for 1528 bodies in memory. That's the $3,85\%$ of the memory we needed in case of unsorted input. In figure 4.13 we can see a plot showing the number of bodies we need to store during the execution of the algorithm. Also in 3 we present the sorting algorithm we used.

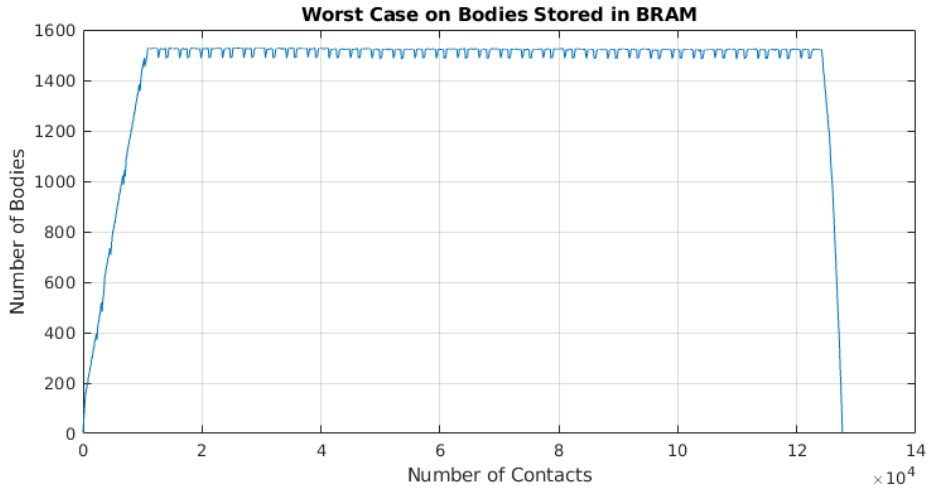


FIGURE 4.13: Worst case scenario of bodies need to be sorted in BRAM

The PGS algorithm produces different results, if the constraints of contacting bodies are solved in a different order. However, the application will behave

Algorithm 3 Sorting algorithm

1: $pairs\_array$ ▷ The unsorted array of pairs
2: $cur\_body \leftarrow body\_1$ ▷ Initializing with body no 1
3: $active\_bodies\_ins(cur\_body)$ ▷ Inserts the cur_body in active_bodies array
4: **while** cont_left(cur_body)>0 **do** ▷ Shows the number of collisions left in cur_body
5: $k \leftarrow get\_pair(pairs\_array, cur\_body)$ ▷ Gets the 1_{st} pair of cur_body in pairs_array array
6: $sorted\_array \leftarrow add\_pair(cur\_body, k)$ ▷ Add this pair to the final array
7: $swap\_pairs(pairs\_array, cur\_body)$ ▷ Brings the next pair of cur_body in 1_{st} place of pairs_array array
8: $active\_bodies\_ins(k)$
9: **end while**
10: $active\_bodies\_del(cur\_body)$ ▷ Deletes the cur_body from active_bodies array
11: **for** all bodies **do**
12: $cur\_body \leftarrow schedule()$ ▷ Gets body with fewer collisions in active_bodies array
13: **while** cont_left(cur_body)>0 **do**
14: $k \leftarrow get\_pair(cur\_body)$
15: $add\_pair(cur\_body, k)$
16: $active\_bodies\_ins(k)$
17: **end while**
18: $active\_bodies\_del(cur\_body)$
19: **end for**
20: **return** $sorted\_array$

similarly in both cases and the deviation of the final results after many simulation steps will be small enough to be acceptable in real-time applications. In our implementation on FPGA, we have not applied the sorting on input data. The whole analysis was implemented in MATLAB.

Chapter 5

System Implementation

In this chapter, we are going to present our implementation of the PGS and Integration algorithms targeting ZCU102. We will start by giving some general information about the tools we used and their capabilities, as also the capabilities of the FPGA (ZCU102). Following, we will present our first approach in the algorithm implementation combined with the problems we have faced. Last but not least, we are going to fully analyze our two final implementations of the algorithm. The first one uses values of floating point precision while the second uses half-precision floating point values.

5.1 Tools Used

In this thesis, we worked on Xilinx's design tools for FPGA and more specifically in Vivado HLS 2017.1. The targeted FPGA in our architectures was Zynq UltraScale+ ZCU102. Below we are going to provide some general information about the tools we have used and the FPGA we have targeted.

5.1.1 Vivado HLS

Vivado HLS[43] is a tool created by Xilinx[41] in order to transform a C specification (i.e code written in C, C++, SystemC, or as an OpenCL API C kernel) into a register transfer level (RTL) implementation that is synthesizable into a Xilinx FPGA. This way, HLS can take advantage of the capabilities of a programming language with a higher level of abstraction to produce IP blocks, generating the appropriate VHDL and Verilog code. Those IP blocks can be integrated into a hardware design.

Firstly HLS executes the scheduling phase by determining which operations occur during each clock cycle. In order to schedule these operations takes into account the clock frequency, the time it takes for the operation to complete and any possible optimization directives that have been applied. The next phase

is binding, where HLS determines which hardware resource implements each scheduled operation. Finally it extracts the control logic to create a finite state machine (FSM) that sequences the operations in the RTL design.

When an IP block is created HLS exports a synthesis report showing the performance metrics of the generated design. The report's information on performance metrics are presented below:

- **Area:** Amount of hardware resources required to implement the design based on the resources available in the target FPGA. The types of resources are, Look-Up Tables (LUT), Flip Flops (FF) , Block RAMs (BRAMs), and DSP48s.
- **Latency:** Number of clock cycles required for the function to compute all output values.
- **Iteration Interval (II):** Number of clock cycles before the function can accept new input data.
- **Loop Iteration Latency:** Number of clock cycles it takes to complete one iteration of the loop.
- **Loop Initiation Interval:** Number of clock cycle before the next iteration of the loop starts to process data.
- **Loop Latency:** Number of cycles to execute all iterations of the loop.

In our implementation we used C++ to write our code as we have already mentioned. In order to debug our code we used a C test bench to simulate the C function prior to synthesis. Finally, the tool allows for directives to be added to the code to direct the synthesis process to implement a specific behavior or optimization. Directives are optional and do not change the behavior of the C code in the simulations, only the synthesized IP block.

HLS Optimization Directives

To optimize a design, we need to reduce the latency, the used area, the iteration interval, etc. To be able to do that we make use of the directives provided by HLS. There are many directives available but we will analyze only the directives we used in our architectures[44].

- **PIPELINE**
  Reduces the initiation interval for a function or loop by allowing the concurrent execution of operations. A pipelined function or loop can process

new inputs every N clock cycles, where N is the initiation interval (II) of the loop or function. In case where $II = 1$ we can process a new input data every clock cycle. In Figure 5.1 we present a loop without using pipeplining and in Figure 5.2 we present a loop with the use of pipeplining.

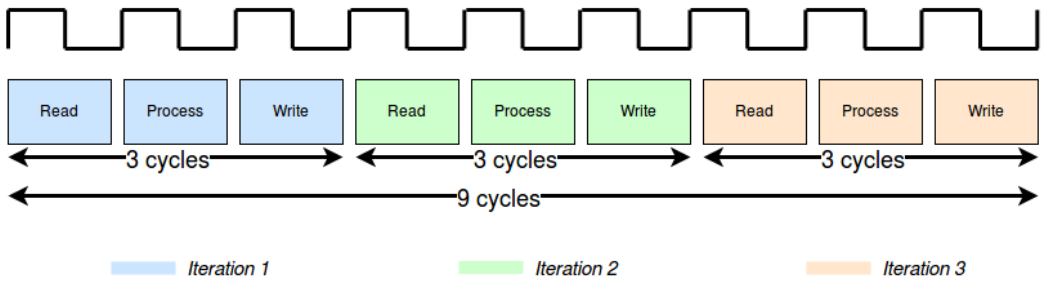


FIGURE 5.1: Presentation of a loop without using pipelining

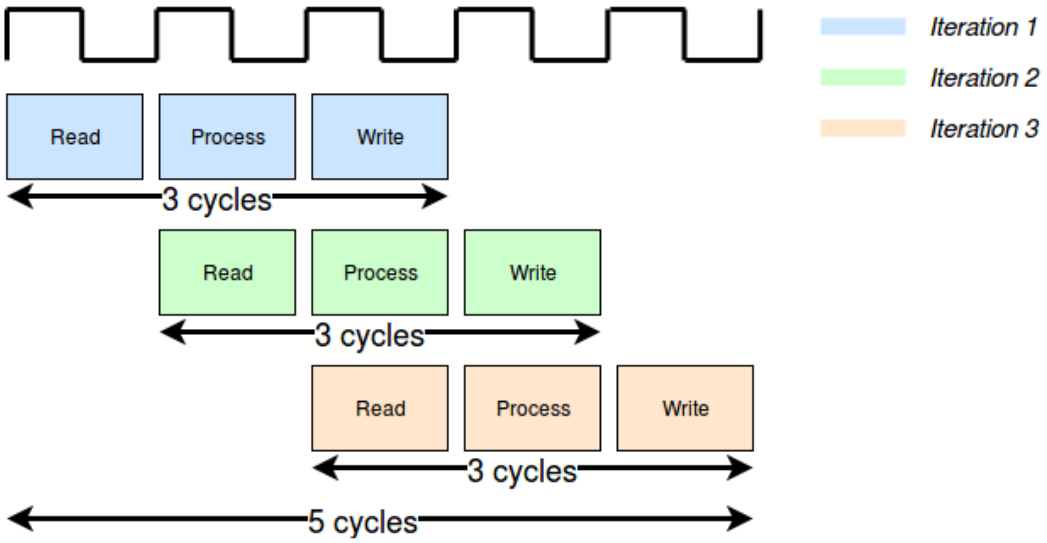


FIGURE 5.2: Presentation of a loop with the use of pipelining

- **ARRAY_PARTITION**
  Partitions an array into smaller arrays or individual elements. The partitioning:
  - Results in RTL with multiple small memories or multiple registers instead of one large memory.
  - Effectively increases the amount of read and write ports for the storage.

  - Potentially improves the throughput of the design.
  - Requires more memory instances or registers

- **DEPENDENCE**
  Provides additional information that can overcome loop-carry dependencies and allow loops to be pipelined (or pipelined with lower intervals). There are two kind of dependencies.

  - Dependencies within loops (loop-independent dependence)
  - Dependencies between different iterations of a loop (loop-carry dependence)

  These dependencies impact when operations can be scheduled, especially during function and loop pipelining.

- **UNROLL**
  Unroll loops to create multiple independent operations rather than a single collection of operations. The directive transforms loops by creating multiples copies of the loop body in the RTL design, which allows some or all loop iterations to occur in parallel. Loops in the C/C++ functions are kept rolled by default. When loops are rolled, synthesis creates the logic for one iteration of the loop, and the RTL design executes this logic for each iteration of the loop in sequence. This directive can increase data access and throughput by unrolling the loops.

- **ARRAY_MAP**
  Combines multiple smaller arrays into a single large array to help reduce BRAM resources. Each array is mapped into a block RAM or UltraRAM. The basic BRAM unit provided in an FPGA is 18K. If many small arrays do not use the full 18K, a better use of the block RAM resources is to map many small arrays into a single larger array. There are two ways of mapping small arrays into a larger one:

  - **Horizontal Mapping:** This corresponds to creating a new array by concatenating the original arrays. Physically, this gets implemented as a single array with more elements. The arrays are concatenated in the order that the pragmas are specified, starting at target element zero.
  - **Vertical mapping:** : This corresponds to creating a new array by concatenating the original words in the array. Physically, this gets

implemented as a single array with a larger bit-width. The arrays are concatenated in the order that the pragmas are specified, starting at bit zero.

- **DATA_PACK**
  Packs all the elements of a struct into a single wide vector to reduce the memory required for the variable, while allowing all members of the struct to be read and written to simultaneously. The bit alignment of the resulting new wide-word can be inferred from the declaration order of the struct fields. The first field takes the LSB of the vector, and the final element of the struct is aligned with the MSB of the vector.

5.2 Memory I/O Interfaces

In order to be able to exploit the bandwidth of the DDR as much as possible, we did some research for the I/O interface between processor's DDR and FPGA. There are two methods of performing I/O between the CPU (in our case the DDR of CPU) and peripheral devices (in our case the FPGA):

1. **Memory Mapped I/O:** Memory-mapped I/O uses the same address space to address both memory (e.g DDR) and I/O devices (e.g FPGA). The memory and registers of the I/O devices are mapped to address values. This method is appropriate in applications where there is no need for low I/O bandwidth. Its random access nature can not allow it to efficiently drive multiple requests because for each request there is a 30 - 50 clock cycles penalty for the initial interval.

2. **Direct Memory Access (DMA) I/O:** DMA is a method that allows an I/O device (e.g FPGA) to send or receive data directly to or from the main memory (e.g DDR), bypassing the CPU to speed up memory operations. The DMA method is perfectly suited for applications that needs high I/O bandwidth. Its like creating a FIFO between the DDR and FPGA, in which we send data without the need of sending requests, so we reduce the penalty of the initial interval we have to pay.

Since we have an application with a high bandwidth I/O we have figured out that the best choice was to use the DMA I/O method. This method allows us to use streaming interface to pass our data in FPGA which is something that allows us to exploit the available bandwidth of the DDR in the best possible way.

5.3 A First,Naive Approach

As a first step, we transferred the C ++ code of the individual parts of the algorithm to HLS to confirm their synthesizability.

5.3.1 Bottom-Up Strategy

In first approach we followed the bottom-up strategy. This strategy is merging many small individual problems/systems, in our case the 4 different parts of the PGS and Integration algorithms, in a single more complex final system. The three individual parts (kernels in GPU implementation) of PGS algorithm and the Integration are presented below:

- **Contact to Constraint Conversion (Cont2Constr):** Gets all the pairwise contact points for all rigid bodies and creates the corresponding contact constraints for every single collision.
- **Contact Constraint Solver (Constr_Solver):** Solves the collisions based on the constraints between the two colliding bodies.
- **Contact Friction Solver (Frict_Solver):** Solves the collisions based on the friction force between the two colliding bodies.
- **Integration (Integrate):** Updates the position of the bodies based on the new values of angular and linear velocities.

We have tested the functionality of each part separately to be able to move on the next step i.e. the combination of those parts.

5.3.2 Architecture Design

In order for the algorithm to be functional, we had to link all the individual parts that we had implemented. So we created a first naive design where we have executed the four pieces of the algorithm sequentially. The input of the system was led to Cont2Constr whose output went as input to Constr_Solver. Then its output came as an entry to Frict_Solver, where again the output was the input for the last piece, Integrate. The final output of the system was Integrate's output. We present this naive design in Figure 5.3.

Here it should be stressed that input data from the outside world for each of the functions is different and not used in any other function. They are used to produce the output of each function in combination with the input from the previous function.

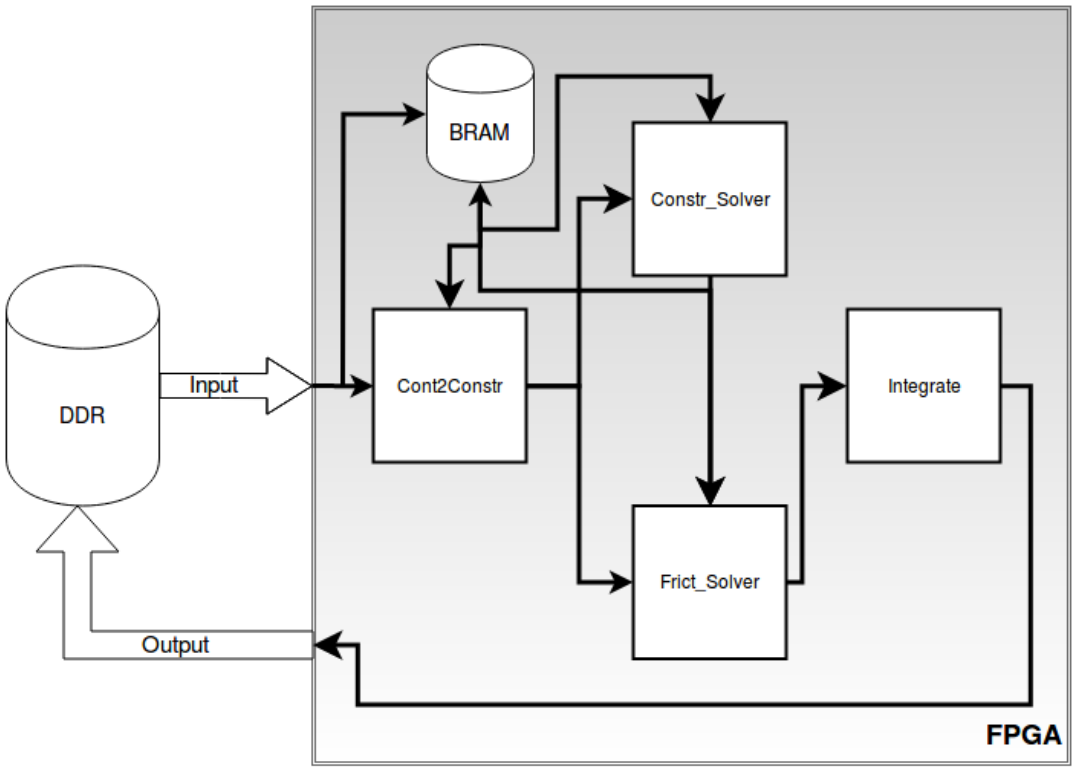


FIGURE 5.3: The first naive design of the algorithm

This design was implemented in order to check the full functionality of PGS and Integration algorithms and to get a first designing experience with HLS, to understand the potential it could offer us. The data transfer between the parts of the algorithm was performed through DDR since the goal of this implementation was to test the tools we had at our disposal.

5.3.3 Optimizations on the Frist Design

Since we had a fully functional algorithm, we started to think about some optimizations we could possibly implement.

Our data were stored in arrays of structs. We used the DATA_PACK directive in order to pack all the elements together and having simultaneously access on those elements. But the size of each structs was big enough and we couldnt achieve the desired result. So we divided all of the arrays of structs into many individual arrays, which were stored in different memory locations and we could have access on each one of them concurrently.

As we mentioned above we were passing the data to algorithm directly through DDR (i.e Memory Mapped I/O). Since our application needs a lot of data in every single iteration this was not the best possible option. We have

applied the streaming interface in order to exploit the bandwidth in a better way. The buses we used while streaming the data in and out of algorithm were 32-bits. Using the streaming interface the overall latency of the algorithm reduced significantly but still, it was not even close to the latency of GPU's implementation.

5.4 Floating Point Architecture

This is the first complete architecture of the PGS and Integraton algorithms. Below, we will analyze the process through which the final design emerged. Several optimization techniques have been tested. We will present the most important and those that gave us the best results.

5.4.1 Algorithmic Level Optimization

We applied a different approach to the order and manner in which the separate parts of the algorithm are executed. As shown in Figure 5.3, Cont2Constr is initially executed, and then the two basic collision-resolving functions are executed. The Cont2Constr function creates the input data to the Constr_Solver and Frict_Solver in combination with the input data from the outside world.

We divided Cont2Constr into two parts, one for creating input data for Constr_Solver and one for Frict_Solver. Then we inserted the first part of Cont2Constr into the Constr_Solver and the second part of Cont2Constr into the Frict_Solver. This way we added some complexity on both of the solver functions but we reduced the input data from the outside world significantly and we eliminated the piece of communication between the Cont2Constr and both of the solvers.

I/O Data Reduction

Based on the analysis in Chapter 4, we implemented all the optimizations for reducing the size of the input data. This reduction gives us many possibilities to reduce the latency of the algorithm as we will need fewer read/writes through the streams. An extensive analysis of how we can achieve this is presented in Chapter 4.

Memory Footprint

As we have already analyzed in Chapter 4, because of the nature of the algorithm we need to store some essential data for all the rigid bodies in the BRAM

of FPGA. The rest of the data the algorithm needs will be acquired through incoming streams from the outside world.

In particular, the data on rigid bodies to be stored at BRAM will be sent at the beginning. Then, while executing the main loop of the algorithm, we will be passing the data of the collisions through streams. Whenever necessary, BRAM data for rigid bodies will be updated. The exact same procedure is followed for the 2 solvers of our algorithm. Lastly, for the implementation of the Integration, the data it needs will come through the BRAM and stream. Rather, data from the outside world will not concern collisions but the rigid bodies. There are additional data that was not stored in BRAM due to lack of space that must come within the algorithm with the usage of streams.

5.4.2 Exploitation of the Available Bandwidth

Larger Streaming Buses

Having already reduced data to our system's input, the next optimization is to fully exploit the available bandwidth to bring the data to the FPGA in the best possible way.

The first step was to increase the size of the streaming buses in order to pass more data with a single stream read or write. The maximum memory bus that DMA can support is 1024 Bits. Although this is limited by the High-Performance (HP) ports of the ZCU102 where the DMA has to go through. The maximum memory bus the HP ports allow from CPU memory to FPGA is 128-bit, so this the the maximum size of the bus we can have when streaming data in and out of FPGA.

Using 128-bit streaming buses instead of 32-bit we can have the $4x$ bandwidth than before and ideally we can achieve a $4x$ speedup.

Multiple DMA's

The next thought was to use multiple DMA's in order to exploit even better the available bandwidth. Using the information we obtained from some of our fellow students analysis for the maximum possible bandwidth from DDR (16 GB/s) we concluded that we can use multiple DMA's. Each one of them would pass through an HP port (ZCU102 has 6 of them). Theoretically, we could use 6 DMA's, but the analysis of our fellow students showed that in the best case we can use 4 of them efficiently.

Using 4 DMAs with 128-bit streaming buses can have the $16x$ bandwidth than the first approach and ideally we can achieve a $16x$ speedup. In our case

we used 3 DMAs with 128-bit buses and 1 DMA with 64-bit bus because that was enough to pass all the data we needed. Further analysis will be performed later on this chapter.

In Figure 5.4 below we present the new datapath that came up after implementing all the optimizations above. Where "Part 1" is the first part of the Cont2Constr function inserted inside the functionality of Constr_Solver and the "Part 2" is the is the second part of the Cont2Constr function inserted inside the functionality of Frict_Solver.

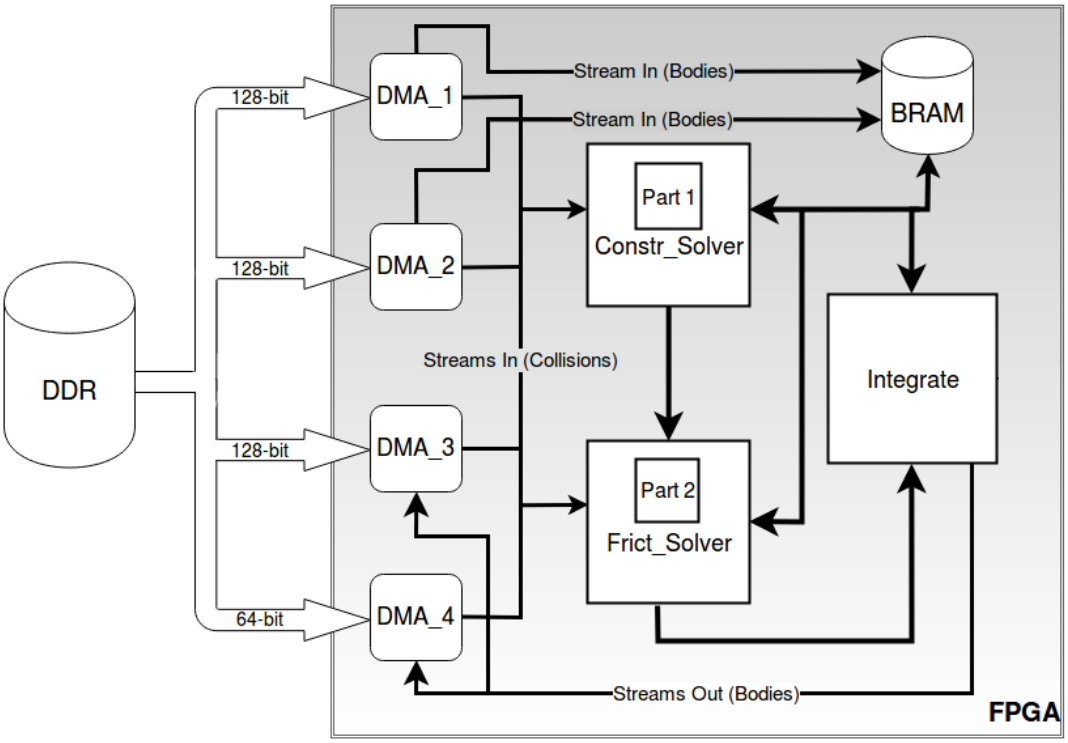


FIGURE 5.4: The floating point architecture datapath

5.4.3 Array Partition

During the execution of the main loop of PGS there are many accesses to BRAM. In particular in a single iteration of the loop we will need to access the same array in BRAM many times. Since BRAM is dual port we can only have two accesses in a single clock cycle. So, we can either read/write, read/read (from different memory addresses) or write/write. Our goal is to reduce as much as possible the cycles we need to read data from BRAM so that it does not restrict us to achieve low value for iteration interval in the basic loop of our algorithm. That's why we used the ARRAY_PARTITION directive that HLS provides us. AS we mentioned above ARRAY_PARTITION effectively

increases the amount of read and write ports for the storage by creating multiple instances of arrays. This also leads to increased size of data is being stored in BRAM. Using this directive we manage to achieve the desirable read and write accesses to BRAM in a single clock cycle.

5.4.4 Pipeline

As shown in Figure 5.2 it is obvious that using pipeline can reduce significant the latency of a loop. The two solvers of the algorithm are based on a loop with many iterations so the PIPELINE directive is perfectly suited for our case. The most important in pipeline is to be able to achieve low value for II, if possible equal to 1. This is not always easy since there are many restrictions related to I/O, the accesses in BRAM and many more.

We have reduced the restrictions from accesses in BRAM with the use of ARRAY_PARTITION directive. The data we need from the outside world for a single iteration of the loop, can be passed into the FPGA with 7 streams of 128-bit buse each. This mean we have to pass into the FPGA 896 bits for every iteration of the main loop of PGS. As we analized above we can efficiently use only 4 DMA's so we can not "feed" the FPGA with all the data in a single clock cycle. We can conclude that the best possible II for this architecture is equal to 2, because we cannot bypass the restrictions derive from I/O.

Dependencies

Knowing that our bottom line was $II = 2$, we started by setting it as a target. At first, we could not achieve this because there were additional restrictions that we had to overcome. One of these was some inter-dependencies that arose from the multiple accesses to BRAM. We have managed to overcome this obstacle by using the DEPENDENCE directive. The tool gave us the information that there were dependencies between the iteration of the loop which actually was not problem for us. Like GPU's implementation we could afford to calculate some values for the bodies involved in a collision without always have the last updated values of the bodies. That's why we have declared these dependencies as false and we bypassed the specific problem.

Custom Loop Unroll

By default using the PIPELINE directive all the loops inside the loop being pipelined are being unrolled. Of course, loops with many iterations can not unrolled since it would demand really high usage of FPGA resources. In the PGS

main loop, there was a loop that could be unrolled and the use of the PIPELINE directive did just that. Because of this action, some conflicts emerged which, in theory, should not exist. Because of this, we implemented the unroll custom so we can overcome the problems that arose from the default unroll from HLS. After several changes to the code and using a "datacarrier" as we called it (discussed in the following paragraph), we were finally able to overcome the problems that had arisen.

Remove If-Statements

It is well known that branches must be avoided in hardware design. Their implementation is many times costly and in the case where there are many branches, the system can be significantly delayed. In PGS algorithm there is a statement needs to be checked before updating a rigid body's velocities. This statement depends on a value coming from the outside world so it is not known before run time. This if-statement needs to be checked 4 times in a single iteration loop, one time for each axis (X, Y, Z) and one time for some special cases. All these if-statements increase the use of FPGA resources and also make it difficult to achieve the desired $II = 2$.

In order to get rid of these if-statements we have created a code where all the 4 cases are executed. In order to choose which one of the results to keep we also multiply the intermediate results with the value on original if-statement, which has been converted in boolean value. If the value is equal to 1 then we need to update the velocities else we do not. We have also added an attribute of 384 bit width, which we have named "datacarrier", to reduce the accesses in BRAM. In "datacarrier" we store all the velocities (12 floating point values) of the two bodies involved in the specific collision. In order to take advantage of the huge bandwidth of BRAM the "datacarrier" stores 12 floats in a single attribute so we can read or write to this attribute in a single clock cycle. Below we present the pseudocode of the original and converted part of the algorithm containing the if statements.

Algorithm 4 Original "Branchy" Code

```
1: coeffInv                ▷ Stream values indicating the update or not velocities
2: inter_values                                   ▷ Set of intermediate values
3: update                                  ▷ Calculates the intermediate values
4: velocities                              ▷ Velocities of bodies stored in BRAM
5: if coeffInv1 then
6:     inter_values ← update(velocities)
7:     velocities ← velocities + inter_values
8: end if
9: if coeffInv2 then
10:    inter_values ← update(velocities)
11:    velocities ← velocities + inter_values
12: end if
13: if coeffInv3 then
14:    inter_values ← update(velocities)
15:    velocities ← velocities + inter_values
16: end if
17: if coeffInv4 then
18:    inter_values ← update(velocities)
19:    velocities ← velocities + inter_values
20: end if
21: new_velocities ← velocities
```

Algorithm 5 Updated "Branchless" Code

```
1: coeffInv                ▷ Stream values indicating the update or not velocities
2: coeffInv_bol                  ▷ 1 if we need to update velocities else 0
3: inter_values                                   ▷ Set of intermediate values
4: update                                  ▷ Calculates the intermediate values
5: velocities                              ▷ Velocities of bodies stored in BRAM
6: coeffInv_bol ← (bool)coeffInv
7: datacarrier ← velocities
8: inter_values1 ← update(datacarrier) ∗ coeffInv1_bol
9: datacarrier ← datacarrier + inter_values1
10: inter_values2 ← update(datacarrier) ∗ coeffInv2_bol
11: datacarrier ← datacarrier + inter_values2
12: inter_values3 ← update(datacarrier) ∗ coeffInv3_bol
13: datacarrier ← datacarrier + inter_values3
14: inter_values4 ← update(datacarrier) ∗ coeffInv4_bol
15: datacarrier ← datacarrier + inter_values4
16: new_velocities ← velocities
```

5.4.5 Array Map

to reduce resources in the FPGA used by our algorithm and especially in BRAM, we used the ARRA_MAP directive. This directive combines multiple smaller arrays a single larger array, which can then be targeted to a single larger resource. We used vertical mapping since it is creating a new array by concatenating the original words in the array. Physically, this gets implemented as a single array with a larger bit-width. Using this directive we reduced the BRAM by 3%.

5.5 Half-Precision Floating Point Architecture

The second architecture we propose for the PGS algorithm was implemented using half-precision floating point. Based on the floating point architecture and taking advantage of the fact that all of our data require half the memory footprint relative to previous architecture we came up with some optimizations and transformations of the previous architecture to end up with better speedup and lower memory footprint.

5.5.1 Conversion to Half

The first step was to convert everything in our implementation in half-precision floating point. The arrays stored in BRAM, the way our streams work and all of the variables in our code had to change in order to be able to handle half-precision floating point values. This way we now need half-sized arrays in BRAM than before and we can also send the double information using the same exact streams we used. So there will be no more I/O restriction that did not allow us to achieve $II = 1$.

However, there has been another restriction on accesses to BRAM. During the iteration of the loop, we need to have 4 accesses to the same memory array. Because of the BRAM (dual port) design, we can only have 2 accesses on the same array within a clock cycle, so we will have $II = 2$. Using the ARRAY_PARTITION directive could not solve this problem. A solution to this problem will be presented later in this chapter.

5.5.2 Multiple Instances of Algorithm

The information we must send in FPGA for a single iteration of the loop is exactly half the size of the previous one. So, we want to send 448-bits using

streams with 128-bit buses. So we need "3.5" stream accesses (actually 4) to send this information to the FPGA. To be able to exploit the remaining 0.5 from a access we created 2 instances of our algorithm in the base loop. So in a single loop iteration, we send data and we calculate results for two collisions instead of one. Of course, the total iterations of the main loop were halved.

So now the main loop of our algorithm will be half the size and in each iteration will perform twice as much work. It is easy to see that the best II we could achieve with the pipeline would be $II = 2$ (as if we had $II = 1$ in the original loop). Because of the limitations of BRAM at this point of design, the II we could achieve was equal to 4 (as if we had $II = 2$ in the original loop).

5.5.3 Multiple Instances of Arrays in BRAM

In order to overcome the restrictions due to BRAM we have thought of creating two instance of our arrays stored in BRAM. In this way, we can have double accesses in the data we need as we can get them from different memory locations. Of course, we should always update both of the instances of the array, which will eventually restrict us. We recall that we need 4 accesses on the same array during a clock cycle, 2 reads and 2 writes. Exploiting the second instance of the array we can perform the first read in the first instance of the array and the second read in the second instance of the array. However, because writing should be done on both arrays, we need 2 more accesses in each. So a total of 3 accesses than the 4 needed before. This way, we ended up in $II = 3$ (as if we had "$II = 1.5$" in the original loop) in relation to $II = 4$ (as if we had $II = 2$ in the original loop) of the array. This is the best II we can achieve in the way this design has been implemented. Even if we extend this logic of the many instances of the original array, the fact that we have 2 write operations in each instance, limits us to being unable to descend below $II = 3$.

Because of this restriction and because of final achieved $II = 3$ for the inner loop pipeline we only need 3 DMAs instead of 4. More precisely we have 2 DMAs with 128-bit buses and 1 DMA with 64-bit bus.

Chapter 6

Results

This chapter summarizes the results obtained from the two architectures that we proposed for the implementation of PGS and Integration algorithms in FPGA, as well as their comparison with the corresponding implementations in GPU and CPU. The comparisons are based on the latency of the two algorithms as well as on their energy consumption.

6.1 Specifications of Compared Platforms

6.1.1 Zynq UltraScale+ ZCU102

The ZCU102 features a Zynq UltraScale+ MPSoC device with a quad-core ARM Cortex-A53, dual-core Cortex-R5 real-time processors, and a Mali-400 MP2 graphics processing unit based on Xilinx's 16nm FinFET+ programmable logic fabric[45]. The features of ZCU102 are presented below:

- Optimized for quick application prototyping with Zynq Ultrascale+ MPSoC
- DDR4 SODIMM – 4GB 64-bit w/ ECC attached to Processor Subsystem (PS)
- DDR4 Component – 512MB 16-bit attached to Programmable Logic (PL)
- PCIe Root Port Gen2x4, USB3, Display Port and SATA
- 4x SFP+ cages for Ethernet
- 2x FPGA Mezzanine Card (FMC) interfaces for I/O expansion including 16 x 16.3 Gb/s GTH transceivers and 64 user defined differential I/O signals

We also present the specifications of the *Zynq UltraScale XCZU9EG-2FFVB1156 FPGA* in Table 6.1:

Table 6.1: The specifications of Zynq UltraScale XCZU9EG-2FFVB1156 FPGA

System Logic Cells	Memory Block RAM	DSP Slices	Maximum I/0 Pins
600 K	4 MB	2,520	328

6.1.2 NVIDIA GeForce GTX 980

In Table 6.2 below we present the specifications of NVIDIA GeForce GTX 980 GPU:

Table 6.2: The specifications of NVIDIA GeForce GTX 980 GPU

	NVIDIA GeForce GTX 980
CUDA Cores	2048
Clock Frequency	1126 MHz
Memory Clock Frequency	1750 MHz
Memory	4GB GDDR5
Memory Bus	256-bit
Bandwidth	224 GB/s
Thermal Design Power (TDP)	165W

6.1.3 Intel Core i7 3770

In the following 6.3 we present the specifications of Intel Core i7 3770 CPU:

Table 6.3: The specifications of Intel Core i7 3770 CPU

Cores	Clock Frequency	Threads	Cache	Thermal Design Power (TDP)
4	3.4 GHz	8	8 MB	77 W

6.2 Speedup

We mention the concept of speedup in many chapters of this diplomatic work, so it's time to give a definition of what speedup means. Having two different systems processing the same problem we can measure their relative performance

by a number called speedup. Technically speaking, it is the improvement in the speed of execution of a task executed on two similar architectures with different resources. The inspirator of the concept of speedup is Amdahl with the much-known Amdahl's law, which was especially focused on parallel processing.

Speedup can be defined for two different types of quantities: latency and throughput.

- **Latency:** It is the reciprocal of the execution speed of a task of an architecture.

$$L = \frac{1}{v} = \frac{T}{W} \tag{6.1}$$

  where v is the execution speed of the task. T is the execution time of the task and W is the execution workload of the task.

- **Throughput:** It is the the execution rate of a task of an architecture.

$$Q = \rho \cdot v \cdot A = \frac{\rho \cdot A \cdot W}{T} = \frac{\rho \cdot A}{L} \tag{6.2}$$

  where ρ is the execution density (e.g. the number of stages in an instruction pipeline for a pipelined architecture) and A is the execution capacity (e.g. the number of processors for a parallel architecture).

6.2.1 Latency Speedup

Let us consider two different two different systems processing the same problem. Setting L_1 to be the the latency of the architecture 1 and L_2 to be the the latency of the architecture 2 we get the following:

$$S_{latency} = \frac{L_1}{L_2} \tag{6.3}$$

where $S_{latency}$ is the speedup in latency of the architecture 2 with respect to the architecture 1.

6.2.2 Throughput Speedup

Following the same pattern we set Q_1 to be the the throughput of the architecture 1 and Q_2 to be the the throughput of the architecture 2 and we get the following:

$$S_{throughput} = \frac{Q_2}{Q_1} \tag{6.4}$$

where $S_{throughput}$ is the speedup in throughput of the architecture 2 with respect to the architecture 1.

6.3 Power and Energy Consumption

Power consumption of a system is the electrical energy per unit time that this system needs in order to execute a task. Power consumption is usually measured in units of watts (W). In order to measure the energy consumption of a system we have to multiply the power consumption by the total time our system needs to complete the execution of an application. Energy consumption is usually measured in units of joule (J). Our measurements in FPGA is in theoretical level since we did not went through Vivado IDE tool which indicates the values for power consumption in our implementations. We might actually have better power consumption than the maximum but since we can not measure it we get the worst case scenario. So we set the power consumption for our architectures as the maximum power consumption of ZCU102.

6.4 Floating Point Architecture

Below we present the comparison of our floating point architecture implementation of PGS and Integration algorithms in FPGA with the corresponding implementations of GPU and CPU. Starting with Table 6.4 we present the results of FPGA implementation.

Table 6.4: The results of Floating Point Implementation

Clock Cycles	Clock Frequency	BRAM_18K Usage	DSP48E Usage	FF Usage	LUT Usage
2062697	275 MHz	70%	42%	32%	47%

In Table 6.5 we present the results of the PGS and Integration algorithm implemented on GPU and CPU compared to our implementation in ZCU102.

TABLE 6.5: Comparison between FPGA, GPU, CPU and FPGA for the floating point architecture

	ZCU102	GTX 980	i7 3770
Clock Frequency	275 (MHz)	1126 (MHz)	3.4 (GHz)
Latency (ms)	7.49	13.64	53.01
Total On Chip Power (Watt)	15	288	133
Energy Consumption (Joule)	0.11	3.93	7.05

The next Table 6.6 is showing the speedup we get in FPGA over GPU and CPU and in the final Table 6.7 we present the energy efficiency over GPU and CPU.

TABLE 6.6: The latency speedup we achieve in FPGA over GPU and CPU for the floating point architecture

	GTX 980	i7 3770
Latency Speedup	1.82×	7.08×

Based on Amdahl's law

$$S_{latency}(s) = \frac{1}{(1-p) + \frac{p}{s}} \tag{6.5}$$

where $S_{latency}$ is the speedup for the whole task, s is the speedup of the part of the task being accelerated, and p is the proportion of execution time of this part in relation to the total execution time of the task.

In our case we can achieve 1.37× speedup in the whole simulation step for the floating point architecture.

TABLE 6.7: The power and energy efficiency of FPGA over GPU and CPU for the floating point architecture

	GTX 980	i7 3770
Power Efficiency	19.2×	8.67×
Energy Efficiency	35.72×	64.09×

6.5 Half-Precision Floating Point Architecture

Below we present the comparison of our half-precision floating point architecture implementation of PGS and Integration algorithms in FPGA with the

corresponding implementations of GPU and CPU. Starting with Table 6.8 we present the results of FPGA implementation.

TABLE 6.8: The results of half-precision floating point implementation

Clock Cycles	Clock Frequency	BRAM_18K Usage	DSP48E Usage	FF Usage	LUT Usage
1557317	250 MHz	48%	42%	29%	49%

In Table 6.9 we present the results of the PGS and Integration algorithm implemented on GPU and CPU compared to our implementation in ZCU102.

TABLE 6.9: Comparison between FPGA, GPU, CPU and FPGA for the half-precision floating point architecture

	ZCU102	GTX 980	i7 3770
Clock Frequency	250 (MHz)	1126 (MHz)	3.4(GHz)
Latency (ms)	6.23	13.64	53.01
Total On Chip Power (Watt)	15	288	133
Energy Consumption (Joule)	0.09	3.93	7.05

The next Table 6.10 is showing the speedup of FPGA over GPU and CPU and in the final Table 6.11 we present the energy efficiency over GPU and CPU.

TABLE 6.10: The latency speedup we achieve in FPGA over GPU and CPU for the half-precision floating point architecture

	GTX 980	i7 3770
Latency Speedup	2.19×	8.5×

We can achieve 1.48× speedup in the whole simulation step for the half-precision floating point architecture based on 6.5.

Below 6.1, we present a chart of FPGA speedup over GPU and CPU for three different architectures. The floating point architecture, the half-precision floating point architecture and the first naive design.We have not analyzed the naive design but we include it in the chart for completion reasons.

TABLE 6.11: The power and energy efficiency of FPGA over GPU and CPU for the half-precision floating point architecture

	GTX 980	i7 3770
Power Efficiency	19.2×	8.67×
Energy Efficiency	43.67×	78.34×

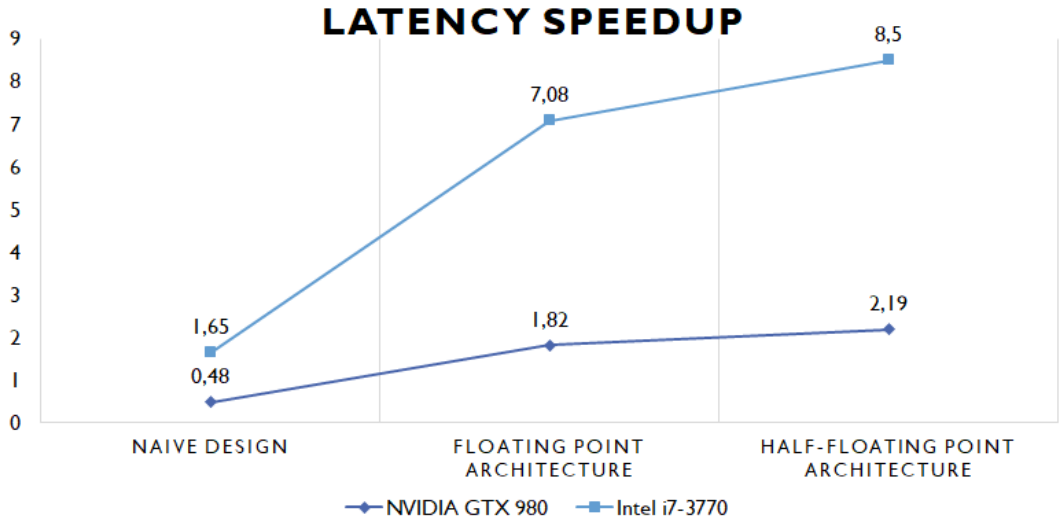


FIGURE 6.1: The speedup of FPGA over GPU and CPU for our three different architectures

Finally in 6.2, we present a chart of FPGA energy efficiency over GPU and CPU for the same three architectures.

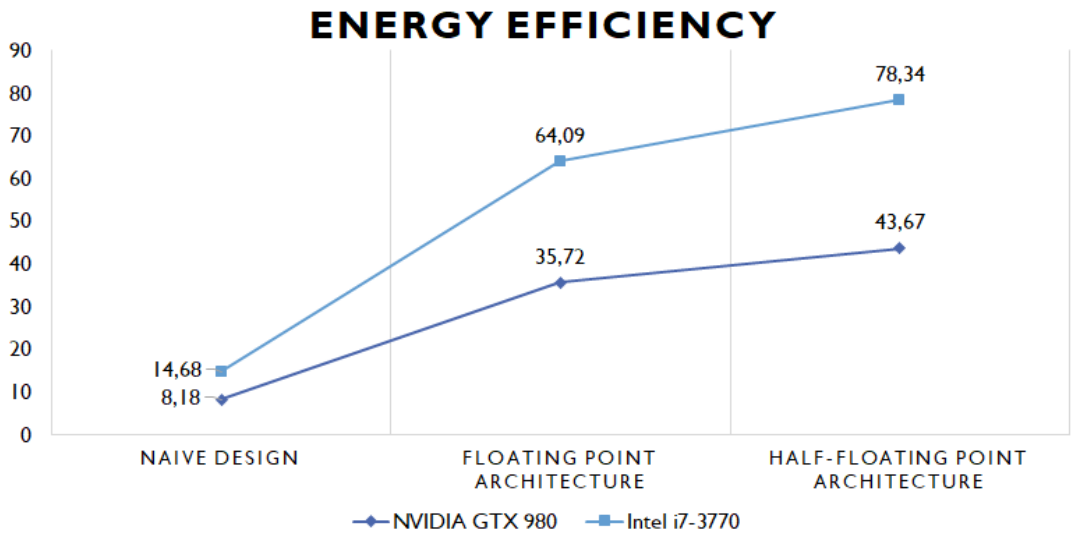


FIGURE 6.2: The energy efficiency of FPGA over GPU and CPU for our three different architectures

Chapter 7

Conclusions and Future Work

In this last chapter we will refer to the conclusions that have emerged during the course of this thesis as well as the lessons we have learned. We will also mention a number of proposals as a future work, extending the work of this thesis.

7.1 Conclusions

During this work, we have been able to conclude many interesting things about how physics engines work, their particularities, their bottlenecks, and so on. We have succeeded in accelerating in FPGA a basic algorithm (PGS) that physics engines use and implement in the GPU. This could possible lead to the creation of heterogeneous systems consisting of FPGAs and GPUs, where the FPGA will deal with processing and the GPU with rendering of the whole physics engine simulation loop. We have also concluded that in certain applications based on physics engines the half-precision floating point values are acceptable and can lead to a better speedup and resource utilization overall.

7.2 Lessons Learned

During the preparation of this work there were several problems and obstacles that we had to overcome. One of the most challenging steps was to understand in depth the nature of the problem and to make a very good analysis so as to reveal opportunities for exploiting possible weaknesses in the already existing implementation. A very time consuming process involved the understanding of the library code we worked on, as we had to fully understand the mentality of the programmer who had composed it, which is often not easy at all.

We learned and understood in depth the Xilinx tools and more specifically the vivado HLS. We also expanded our knowledge of programming languages C ++, C and we learned some things about how OpenCL and OpenGL work.

Last but not least, we have learned some very important lessons about the way we should approach a problem, how important it is to be able to properly manage the available time we have, and the absolute need for a proper planning of the work and to stay faithful to this plan.

7.3 Future Work

Starting with the analysis described in Chapter 4 for the case we are sorting the data that comes as an input to the FPGA, we suggest as a future work to implement this kind of sorting within the FPGA i.e the transference of software implementation in MATLAB to HLS. As we analyzed, such an approach could significantly reduce BRAM and allow us to introduce pipeline between the iterations of the outer loop of the two solvers as also a pipeline between the two solvers.

Another idea for future work is to implement the part of the simulation loop which concerns the collision detection, i.e broad-phase, and narrow-phase collision detection. This will lead to the implementation of the entire simulation loop in the FPGA. It is possible that it can not fit into a small FPGA, which may require two FPGAs to communicate, or a larger FPGA that can fit the entire design.

Finally, we suggest an idea of combining an FPGA together with a GPU into one system. The idea is to use the FPGA to run the algorithm's entire simulation loop and to communicate with the GPU to send out the results for rendering.

References

[1] A. Choudhary P. Banerjee A. Nayak M. Haldar. "Precision and error analysis of MATLAB applications during automated hardware synthesis for FPGAs". In: *Proceedings Design, Automation and Test in Europe.* (2001). URL: https://ieeexplore.ieee.org/document/915108.

[2] Martin Rölin Axel Seugling. "Evaluation of Physics Engines and Implementation of a Physics Module in a 3d-Authoring Tool". MA thesis. Umeå University Department of Computing Science, Sweden, 2006.

[3] Ian Robert Ballantyne. "Collision Overload: Reducing the Impact in Real-time Physics". MA thesis. 2009.

[4] Somnath P Mukherjee S Pal Subhradeep Biswas B N Chatterjee. "A DISCUSSION ON EULER METHOD: A REVIEW". In: *Electronic Journal of Mathematical Analysis and Applications* 1 (2013). URL: https://www.researchgate.net/publication/239525844_A_DISCUSSION_ON_EULER_METHOD_A_REVIEW.

[5] Adrian Boeing. "Evaluation of real-time physics simulation systems". In: *GRAPHITE. Proceedings of the 5th international conference on Computer graphics and interactive techniques, Australia and Southeast Asia* (2007). URL: https://dl.acm.org/citation.cfm?id=1321312.

[6] Erin Catto. "Iterative Dynamics with Temporal Coherence". In: (2005). URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.574.6641&rep=rep1&type=pdf.

[7] Xiao jian Liu Cheng Liang. "The Research of Collision Detection Algorithm Based on Separating axis Theorem". In: *International Journal of Science* 2 (2015). URL: https://pdfs.semanticscholar.org/65c6/ce78829efaeacbb29e753a13c3a1838e53db.pdf.

[8] David Chui. "An FPGA Implementation of the Ewald Direct Space and Lennard-Jones Compute Engines". In: *Master of Applied Science* (2005). URL: https://www.semanticscholar.org/paper/An-FPGA-Implementation-of-the-Ewald-Direct-Space-Chui/40347f255de5769232cea8e333e32d33744f6e20.

[12] Tomas Berglund Da Wang Martin Servin. "Warm starting the projected Gauss-Seidel algorithm for granular matter simulation". In: *Computational Particle Mechanics* 3 (2016). URL: https://link.springer.com/article/10.1007/s40571-015-0088-x.

[13] Oliver G. Staadt Daniel S. Coming. "Kinetic Sweep and Prune for Collision Detection". In: *Workshop On Virtual Reality Interaction and Physical Simulation* (2005). URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.115.4443&rep=rep1&type=pdf.

[15] Sukamto Elfizar. "Analysis of Axis Aligned Bounding Box in Distributed Virtual Environment". In: *International Journal of Computer Applications (0975 – 8887)* 105 (2014). URL: https://www.semanticscholar.org/paper/Analysis-of-Axis-Aligned-Bounding-Box-in-Virtual/c3b7cb8acc91dc9e1ac0912506d155748e844f81.

[16] Tom Erez, Yuval Tassa, and Emanuel Todorov. "Simulation tools for model-based robotics: Comparison of Bullet, Havok, MuJoCo, ODE and PhysX". In: *2015 IEEE International Conference on Robotics and Automation (ICRA)* (2015). URL: https://ieeexplore.ieee.org/document/7139807/media.

[17] Christer Ericson. *Real-Time Collision Detection*. 2005. URL: http://realtimecollisiondetection.net/books/rtcd/.

[18] Earl Wells Falco Girgis. "GPGPU Acceleration for Video Game Physics Engines". In: (). URL: http://elysianshadows.com/updates/wp-content/uploads/2011/05/cpe790FINAL.pdf.

[21] Fredrik Fossum. "Real-Time Rigid Body Interactions". MA thesis. Norwegian University of Science, Technology Department of Computer, and Information Science, 2011.

[22] Stefan Gottschalk. "Collision Queries using Oriented Bounding Boxes". PhD thesis. Univerisy of North Carolina at Chapel Hill, 2000.

[24] Tamer Abd Elmouty Elawady Hussein A. Aly. "A new narrow phase collision detection algorithm using height projection". In: *4th European Education and Research Conference (EDERC 2010)* (2010). URL: https://ieeexplore.ieee.org/document/6151418/?part=1.

[25] Jeff Trinkle Jan Bender Kenny Erleben and Erwin Coumans. "Interactive Simulation of Rigid Body Dynamics in Computer Graphics". In: *STAR Proceedings of Eurographics* (2014). URL: https://doi.org/10.1111/cgf.12272.

[26] Hassan Kianinejad Peng Wei Jason Cong Zhenman Fang. "Revisiting FPGA Acceleration of Molecular Dynamics Simulation with Dynamic Data Flow Behavior in High-Level Synthesis". In: (2016). URL: https://arxiv.org/abs/1611.04474v1.

[27] Walid A. Najjar Jason Villarreal. "Compiled hardware acceleration of Molecular Dynamics code". In: *International Conference on Field Programmable Logic and Applications* (2008). URL: https://ieeexplore.ieee.org/document/4630035.

[28] Dinesh Manocha Madhav K. Ponamgi Jonathan D. Cohen Ming C. Lin. "I-COLLIDE: An Interactive and Exact Collision Detection System for Large-Scale Environments". In: (2005). URL: http://www.cs.jhu.edu/~cohen/Publications/icollide.pdf.

[29] Daniel Wagner Wolfgang J. Paul Philipp Slusallek Jörg Schmittler Sven Woop. "Towards a Field-Programmable Physics Processor (FP)". In: *7th Irish Workshop on Computer Graphics (Eurographics Ireland Chapter 2006), At Dún Laoghaire, Dublin, Ireland* (2006). URL: https://www.researchgate.net/publication/242482020_Towards_a_Field-Programmable_Physics_Processor_FP.

[30] Evangelos Kokkevis. "Practical Physics for Articulated Characters". In: *Game Developers Conference* (2004). URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.445.9996&rep=rep1&type=pdf.

[31] Patrick Lindemann. "The Gilbert-Johnson-Keerthi Distance Algorithm". In: 2 (). URL: https://www.medien.ifi.lmu.de/lehre/ss10/ps/Ausarbeitung_Beispiel.pdf.

[32] Sarah Niebe Morten Silcowitz and Kenny Erleben. "Projected Gauss-Seidel Subspace Minimization Method for Interactive Rigid Body Dynamics - Improving Animation Quality using a Projected Gauss-Seidel Subspace Minimization Method." In: (). URL: https://www.researchgate.net/publication/220868749_Projected_Gauss-Seidel_Subspace_Minimization_Method_for_Interactive_Rigid_Body_Dynamics_-_Improving_Animation_Quality_using_a_Projected_Gauss-Seidel_Subspace_Minimization_Method.

[33] Panos M. Pardalos. "The Linear Complementarity Problem". In: (). URL: https://page-one.springer.com/pdf/preview/10.1007/978-94-015-8330-5_3.

[36] Emil Rönnbäck. "Parallel implementation of the projected Gauss-Seidel method on the Intel Xeon Phi processor – Application to granular matter simulation". MA thesis. Umeå University Department of Computing Science, Sweden, 2014. URL: http://www8.cs.umu.se/education/examina/Rapporter/EmilRonnback.pdf.

[38] Christopher Stover. *Biased Exponent*. URL: http://mathworld.wolfram.com/BiasedExponent.html.

[40] Michael Woulfe Muiris Manzke. "Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip". In: *Graphics Hardware* (2004). URL: http://www.sven-woop.de/papers/2004-GH-SaarCOR.pdf.

[46] Y. J. Uncertain. Anal. Appl Yang X. Shen. "Runge-Kutta Method for Solving Uncertain Differential Equations". In: *Electronic Journal of Mathematical Analysis and Applications* 3 (2015). URL: https://doi.org/10.1186/s40467-015-0038-4.

[47] Dou Y. Yang X. Mou S. "FPGA-Accelerated Molecular Dynamics Simulations: An Overview". In: *Reconfigurable Computing: Architectures, Tools and Applications* (2007). URL: https://page-one.springer.com/pdf/preview/10.1007/978-94-015-8330-5_3.

[48] Jeroen van der Zijp. *Fast Half Float Conversions*. Tech. rep. 2008. URL: http://www.fox-toolkit.org/ftp/fasthalffloatconversion.pdf.

External Links

[9] Erwin Coumans. *Bullet Physics SDK Manual*. 2015. URL: https://github.com/bulletphysics/bullet3/blob/master/docs/Bullet_User_Manual.pdf.

[10] Erwin Coumans. *Bullet Real-Time Physics Simulation*. URL: https://pybullet.org/wordpress/.

[11] Erwin Coumans. *GPU Rigid Body Simulation using OpenCL*. URL: https://github.com/bulletphysics/bullet3/blob/master/docs/GPU_rigidbody_using_OpenCL.pdf.

[14] Jonathan "lonesock" Dummer. *A Simple Time-Corrected Verlet Integration Method*. 2016. URL: http://lonesock.net/article/verlet.html.

[19] Glenn Fiedler. *Fix Your Timestep!* URL: https://gafferongames.com/post/fix_your_timestep/.

[20] *Forward and Backward Euler Methods*. URL: http://web.mit.edu/10.001/Web/Course_Notes/Differential_Equations_Notes/node3.html.

[23] Khronos Group. *OpenCL Overview*. URL: https://www.khronos.org/opencl/.

[34] *Physics Tutorial: Collision Response*. URL: https://research.ncl.ac.uk/game/mastersdegree/gametechnologies/physics6collisionresponse/2017\%20Tutorial\%206\%20-\%20Collision\%20Response.pdf.

[35] *Runge-Kutta Methods*. 2016. URL: http://web.mit.edu/10.001/Web/Course_Notes/Differential_Equations_Notes/node3.html.

[37] Oliver Smart. *Non-bonded Interactions*. URL: http://www.cryst.bbk.ac.uk/PPS2/course/section7/os_non.html.

[39] James Tursa. *IEEE 754r Half Precision floating point converter*. 2009. URL: https://www.mathworks.com/matlabcentral/fileexchange/23173-ieee-754r-half-precision-floating-point-converter.

[41] Xilinx. *Official Xilinx Website*. URL: https://www.xilinx.com/.

[42] Xilinx. *Vivado Design Suite User Guide High-Level Synthesis*. 2018. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug902-vivado-high-level-synthesis.pdf.

[43] Xilinx. *Vivado High-Level Synthesis*. URL: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[44] Xilinx. *Vivado HLS Optimization Methodology Guide*. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1270-vivado-hls-opt-methodology-guide.pdf.

[45] Xilinx. *ZCU102 Evaluation Board User Guide*. URL: https://www.xilinx.com/support/documentation/boards_and_kits/zcu102/ug1182-zcu102-eval-bd.pdf.