Particle Graphics on Reconfigurable Hardware

JOHN S. BEECKLER and WARREN J. GROSS
McGill University

Particle graphics simulations are well suited for modeling complex phenomena such as water, cloth, explosions, fire, smoke, and clouds. They are normally realized in software as part of an interactive graphics application. The computational complexity of particle graphics simulations restricts the number of particles that can be updated in software at interactive frame rates. This article presents the design and implementation of a hardware particle graphics engine for accelerating real-time particle graphics simulations. We explore the design process, implementation issues, and limitations of using field-programmable gate arrays (FPGAs) for the acceleration of particle graphics. The FPGA particle engine processes million-particle systems at a rate from 47 to 112 million particles per second, which represents one to two orders of magnitude speedup over a 2.8 GHz CPU. Using three FPGAs, a maximum sustained performance of 112 million particles per second was achieved.

Categories and Subject Descriptors: B.6.1 [**Logic Design**]: Design Styles—*Parallel circuits*; B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Algorithms implemented in hardware; gate arrays*

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Reconfigurable computing, FPGAs, particle systems, special-purpose architectures

ACM Reference Format:

Beeckler, J. S., and Gross, W. J. 2008. Particle graphics on reconfigurable hardware. ACM Trans. Reconfig. Technol. Syst. 1, 3, Article 15 (September 2008), 27 pages, DOI = 10.1145/1391732.1391735 http://doi.acm.org/10.1145/1391732.1391735

1. INTRODUCTION

Particle graphics simulations are well suited for modeling complex phenomena such as water, cloth, explosions, fire, smoke, and clouds [Reeves 1983]. Dynamic simulations of the physics of large groups of individually simple particles can create graphical models of objects and phenomena that are otherwise difficult to

Authors' addresses: J. S. Beeckler, W. J. Gross (corresponding author), Department of Electrical and Computer Engineering, McGill University, 3480 University Street, Montreal, Quebec, H3A 2A7, Canada; email: warren.gross@mcgill.ca.

model and render realistically. In these simulations, systems of simple elements (e.g., point masses) interact with their environment and evolve together, subject to a set of rules. The properties and evolution of the system are also determined by randomly varying initial conditions. The visual ensemble of such a group of particles exhibits a great degree of complexity and detail, closely resembling the detail and randomness seen in nature. Particle graphics have many applications, including scientific visualization, movie rendering, and video games. In the case of video games, the demands on the computation and rendering of the particle system are relatively tight, due to the “real-time” interactive nature of the application.

Graphical particle simulations are traditionally implemented in software on a general-purpose CPU. The computational complexity of particle graphics limits the number of particles that can be computed in a single frame at interactive rates. In the literature it is reported that CPU implementations are able to handle tens of thousands of particles per frame [Kolb et al. 2004]. Our own software implementation can calculate approximately 36 thousand particles per frame at 60 Hz (see Table IV). The number of particles that can be handled will surely increase with the steady improvement in CPU performance; however, there is a long way to go before reaching the goal of simulating a million-particle system in real time on a CPU [Kolb et al. 2004; Latta 2004]. This is significant because a million-particle system may be used to model a complex effect, or multiple effects of lower complexity.

Fortunately, particle graphics algorithms exhibit a very high degree of parallelism that is easily uncovered and exploited. Many useful particle graphics effects can safely ignore interparticle interactions, exposing data parallelism at particle level. Data-level parallelism can be exploited on multiprocessor systems where each processor is responsible for updating a subset of the particles [Sims 1990]. In addition to splitting particles among processing elements in a multiprocessor, hardware accelerators can be applied to speed-up computations within a given processing element. Accelerators exploit operand-level parallelism with structures such as pipelines and SIMD (single-instruction multiple-data) arithmetic units.

Recently, there have been a number of successful attempts at implementing particle graphics engines using the programmable stream processors in graphics processing units (GPUs) on graphics cards [Kipfer et al. 2004; Latta 2004; Krüger et al. 2005]. In this article, we consider an alternative approach to hardware particle graphics acceleration: field-programmable gate arrays (FPGAs). We show that custom particle graphics engines on FPGAs are viable and can achieve very high performance, on the order of millions of particles per frame. Section 2 gives a brief overview of particle graphics. In Section 3 we describe the design and FPGA implementation of a hardware particle graphics accelerator. Section 4 describes a next generation implementation of a particle graphics accelerator that was coded in a high-level programming language, avoiding the need for detailed hardware design. Conclusions are offered in Section 5.

In Beeckler and Gross [2005] we introduced the concept of FPGA particle graphics engines and gave initial results on FPGA synthesis of such an engine.

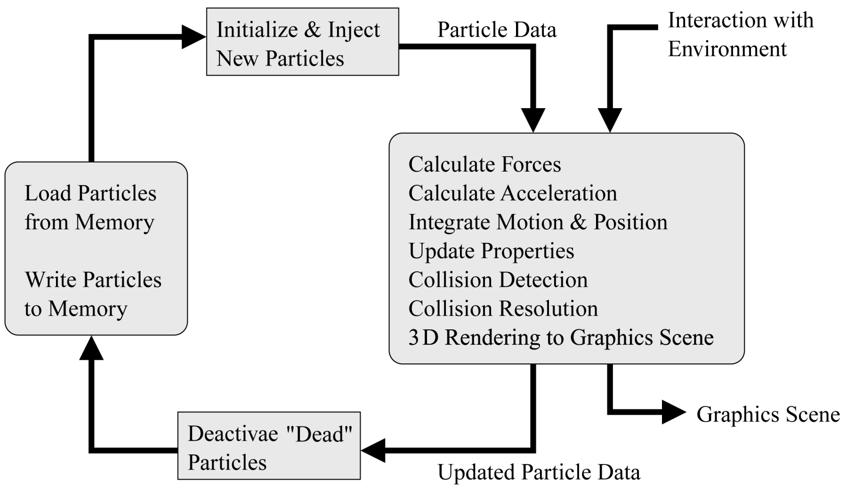
Initialize & Inject New Particles
Particle Data
Interaction with Environment
Load Particles from Memory
Write Particles to Memory
Calculate Forces
Calculate Acceleration
Integrate Motion & Position
Update Properties
Collision Detection
Collision Resolution
3 D Rendering to Graphics Scene
Graphics Scene
Deactivae "Dead" Particles
Updated Particle Data

Fig. 1. General flow of particle graphics simulations.

This article expands on details of the design and addresses system performance issues. In this work we also describe the next generation of our particle graphics engine, designed in a high-level “C”-like language and implemented on higher-performance FPGA hardware.

2. PARTICLE GRAPHICS

The general flow of a particle graphics simulation is shown in Figure 1. All particles in the system are loaded sequentially from memory, updated, rendered, and then written back to memory. At each frame, a new particle can be created with randomly varying initial conditions. Dead particles which have been alive for their useful lifetime or have reached their final state can be deactivated and replaced by newly initiated particles. For each particle processed, a set of forces acting on it is calculated, using its current state as well as some environmental data. These forces might include gravitational, electrical, vortex, wind, viscosity, friction, explosive, and spring forces. From these forces a particle acceleration is calculated. Next, the particle’s motion is numerically integrated, giving the new velocity and position of the particle. Subsequently, collision detection and collision resolution are performed, allowing particles to collide and interact with their environment. Finally, the particle is rendered. A particle could be rendered very simply as a single colored pixel, or as a more complex object such as a cloud or streak.

The type of large particle systems used for real-time graphics applications are usually first order, meaning that the particles only interact with their environment, and not with other particles in the system. Second-order extensions create special force-carrying particles [Ilmonen and Kontkanen 2003], but the overall number of interparticle interactions is not proportional to n^2. The benefit of a first-order particle system is that all particles in the system can be updated in a single pass through the particle data, with the processing of each particle being done independently of all other particles. This means that the processing is embarrassingly parallel and can be completed in one pass through a pipeline.

2.1 Recent Progress in Hardware Particle Engines

There have been a number of promising results using special-purpose hardware as accelerators for particle systems. One approach is to use the programmable floating-point stream processors in graphics processing units (GPUs) [Kipfer et al. 2004; Latta 2004; Krüger et al. 2005]. This technique creates a set of double-buffered streams of data containing particle position data, particle velocity data, and a depth map for collision detection [Kolb et al. 2004]. The output data stream is created from an input stream by executing a pixel shader program on a GPU. GPUs are reported to enable the implementation of systems as large as 1024×1024 particles. Current GPUs can process systems of up to 8192×8192 particles.

Another approach is to build a special-purpose computer for simulating particle systems. There have been a number of custom hardware implementations of particle systems reported in the literature, mainly for astrophysical N-body simulations and molecular dynamics (see e.g., Makino [2006], Azizi et al. [2004]), but not specifically for particle graphics. The main difference is that the number of interactions that need to be calculated is $O(n^2)$ in N-body simulations, while in particle graphics it is $O(n)$, where n is the number of particles in the simulation. The GRAPE (gravity pipe) series of machines were developed around a custom hardware chip specialized for N-body calculations. An alternative to developing a custom chip is to use a reconfigurable hardware device such as a field-programmable gate array (FPGA) [Hamada et al. 1998; Azizi et al. 2004; Beeckler and Gross 2005]. FPGAs are particularly suitable for accelerating problems that exhibit data-level parallelism. In this article we describe the design of a particle graphics engine implemented on an FPGA. Previous work by other authors in Zemcik et al. [2003, 2004], and Herout and Zemcik [2005] implemented the rendering pipeline for particles on FPGAs (e.g., drawing clouds instead of points), but did not accelerate the physics engine that performs the computationally expensive determination of the locations of these particles.

3. DESIGN AND IMPLEMENTATION OF A HARDWARE PARTICLE GRAPHICS ENGINE

Inspired by the hardware implementations of N-body simulators, we propose using FPGAs as programmable accelerators for implementing high-performance particle graphics engines. However, the design of a particle graphics engine is subject to a different set of criteria than for a general N-body simulation. The differentiating characteristics are as follows.

(1) *Real-Time Constraints*. The simulation must update the set of particles at the frame rate.
(2) *Interparticle Interactions*. We consider classical *first-order* particle systems requiring only one pass through the particle data. This assumption

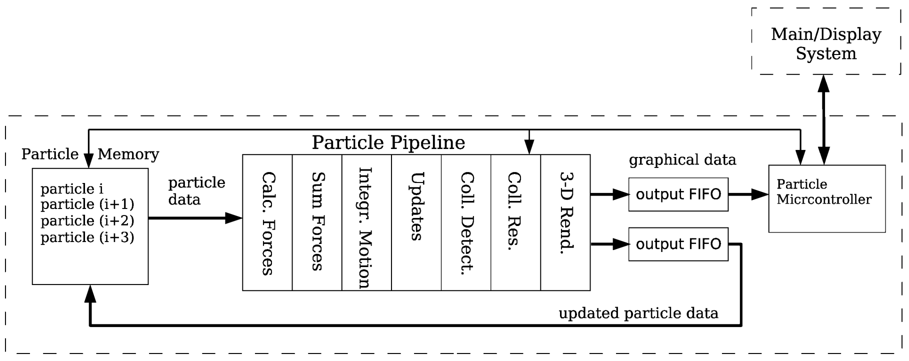

Fig. 2. Architecture of the particle graphics accelerator.

significantly reduces the computational complexity, making it easier to meet the real-time constraint.

(3) *Programmability*. It must be easy to reconfigure the accelerator to change the effect. This is one reason why a *programmable* hardware accelerator like a GPU or FPGA is required and a custom chip is not suitable. The ease and method of programmability are also important considerations since the developers are likely to be software designers and not hardware engineers.

(4) *Precision*. The precision of the variables can be reduced, so long as the visual quality of the effect is preserved.

(5) *Cost*. Cost plays a larger role in video game platforms than in general scientific computing.

In the remainder of this section, the design of a custom particle graphics acceleration system for runtime implementation in an FPGA will be described in detail.

3.1 System Overview

The particle pipeline system shown in Figure 2 is a self-contained hardware coprocessing system that completely contains, manages, executes, and renders particle graphics simulations for an application running on the host CPU. The system is intended for implementation in an FPGA that has access to its own dedicated RAM, and is able to communicate with the application CPU. The system is comprised of the following major components: the particle microcontroller, particle memory, and particle pipeline. These three major components will be described in the next three subsections.

3.2 The Particle Microcontroller

The particle microcontroller is the interface between the particle pipeline and the host (application) CPU. The host CPU sends the particle system parameters to the particle microcontroller. The latter then continuously creates and initializes new particles in particle memory between frames, introducing them into running simulations with the initial conditions created according to specified simulation properties. The microcontroller also sets and continuously controls the contents of parameter registers in the particle pipeline, giving it the ability to dynamically modify the functionality and properties of the simulation.

3.3 Particle Memory

The particle pipeline system needs high-bandwidth and exclusive access to a RAM device dedicated to containing a large pool of all available particles for simulations. This memory, the particle memory shown in Figure 2, should be part of the FPGA system used to implement the particle pipeline system and needs to be separate or isolated from main system memory. The pipeline design is capable of beginning one new particle computation and completing one in-flight particle computation every FPGA clock cycle. Therefore, to keep the pipeline full, a particle memory needs to provide sustained read and write access rates, given by

$$\begin{aligned} R_{pmem,read} &= b_p \times f_{pipe}, \\ R_{pmem,write} &= b_p \times f_{pipe}, \end{aligned} \tag{1}$$

where b_p is the width of one particle's dataset in bits, f_{pipe} is the frequency of the particle pipeline clock in Hz, and $R_{pmem,read}$ and $R_{pmem,write}$ are the required sustained read and write access rate, respectively, in units of bits per second. Fortunately, all accesses to particle memory, with the exception of the initialization of new particles, are made in a regular, sequential order. This means that the burst modes of RAM devices can be fully exploited to help achieve the required access rate.

Particle data is stored in particle memory as one large packed array. A particle's dataset, namely its entry in particle memory, must include all the data fields needed to create any of the simulations. These fields include a position vector, velocity vector, color, life count, and a type field. The position and velocity vectors are three-dimensional vectors with each component represented in the 18-bit format described to follow in Section 3.3.1. The life count is an integer set when a new particle is initialized and is decremented on every pass through the pipeline. When a particle's life count reaches zero, it can be considered "dead" or inactive, and will no longer contribute to the simulation. The particle memory entries occupied by inactive particles then become available for the initialization and creation of new particles by the microcontroller. Inactive particles output from the pipeline are counted by hardware counters, but do not enter the particle data output buffer to be stored back to particle memory. This means that an entry in particle memory for each inactivated particle will float to the top as all active particle data is shifted down in memory, replacing inactive entries. The inactive space at the top of particle memory is used for new particles and space remaining after the injection and initialization of new particles must be cleared to the inactive state, as it still contains data from other shifted particles. New particles are copied from the microcontroller's "nursery" of new particles in the microcontroller data RAM and written to the empty region of particle memory.

A number of independent particle effects can coexist in the particle memory, each identified by a *type* identifier. The type field of a particle identifies which group this particle belongs to and controls the functionality of the effect on each particle as it flows through the pipeline.

3.3.1 *Fixed-Point Data Format.* A fixed-point representation is chosen for the particle data format. There are several factors influencing the choice of fixed-point representation. Primarily, a fixed-point format used in the pipeline will limit precision and range, which can be tolerated so long as the visual quality of the effect is preserved. Another factor influencing the decision is the width of particle memory. It is best for the dataset of each particle to fit exactly in an integer number of memory words. If the pipeline clock frequency is relatively slow when compared to the particle memory access time, then particle datasets can be stored in multiple words of memory. However, if the pipeline frequency and memory access time are comparable, then each particle data entry will need to fit in as few words as possible.

The particle pipeline contains numerous fixed-point additions, subtractions, multiplications, and divisions, all of which are pipelined. These circuits become more and more complex with larger bit-widths. FPGAs contain dedicated hardware multiplier circuits which the particle pipeline design uses to implement many high-speed parallel multiplications without using programmable logic resources. Altera Stratix FPGAs and Xilinx Virtex FPGAs both have hardwired resources for implementing hundreds of 18×18-bit multiplications. For these reasons, an 18-bit (4.14) fixed-point format is used in the pipeline. This format makes the best use of dedicated FPGA multiplier circuits while still allowing particle datasets to fit perfectly into a low multiple of 32 bits, making efficient use of particle memory.

3.4 Particle Pipeline Architecture

The particle pipeline, shown in the center of Figure 2, is a fully pipelined particle update processor capable of simultaneously updating several particles. The particle pipeline includes the following four major systems:

(1) force calculation and summation,
(2) integration and updates,
(3) collision detection and response, and
(4) 3D rendering.

Each functional unit or block in the pipeline is enabled, disabled, and customized by its own set of parameter registers. Before a particle enters a function unit or block, a set of values for the unit's parameter registers, specific to the type of particle [Latta 2004; Kolb et al. 2004; Kipfer et al. 2004] entering the function unit, is selected from a table using the particle's *type field* as a table index. This is depicted in Figure 3. These selected parameters then synchronously pass through the function unit, from register stage to register stage, together with the particle data.

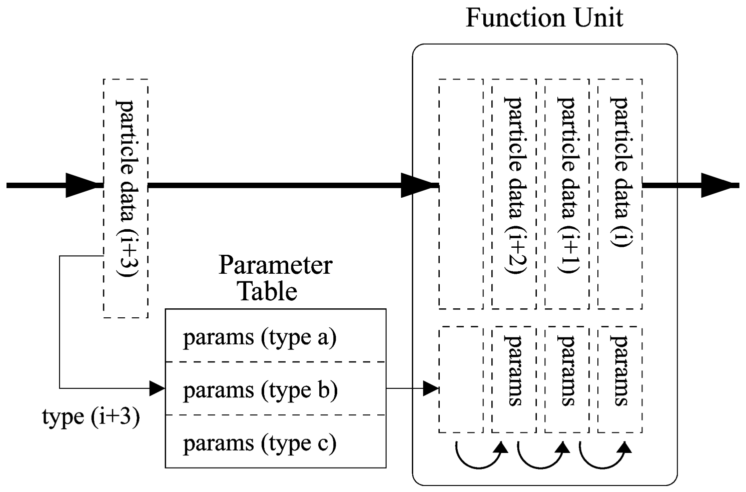

Fig. 3. Pipeline parameter selection.

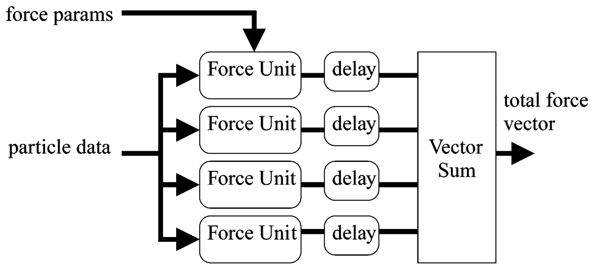

Fig. 4. The force system.

On the application side, to create simulations, application software defines a number of groups (or types) of particles. By sending commands to the pipeline microcontroller, application software fills in the values of the parameter tables for each function unit of the pipe, specifying its configuration for each particle type. For example, if there is a uniform-force block in the pipeline, its job is to add a vector to each particle's sum of forces. Such a block might have two configuration fields. One would be an enable bit, used to determine if this block should be enabled for a particle, and another would contain the force vector to be added. Consider a simulation in which there are three types of particles. Particles of type a will not experience this force at all. Particles of type b will experience the $(1.0, 0.0, 0.0)$ force vector, and particles of type c will experience the $(0.0, -2.2, 0.0)$ force vector. To accomplish this, the parameter tables for this function unit should be initialized such that the enable bits for particle types a, b, and c are 0, 1, and 1, respectively.

3.4.1 *Forces.* The force system, shown in Figure 4, contains a set of parallel force units. Each force unit receives as input the current particle data, together

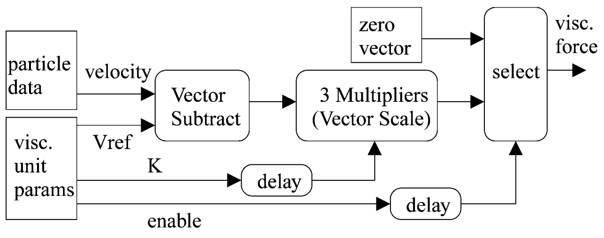

Fig. 5. The viscosity-force unit.

with a set of type-selected force parameters, and outputs a resulting 3D force vector in the pipe's fixed-point format. Each force unit's output vector is delayed for synchronization and summed to one total force vector.

Examples of easily implementable force units include uniform forces, viscosity forces, vortex forces, attractive and repulsive forces, spring forces, "random nudge" forces, and many more. Figure 5 gives a detailed look at the implementation of a viscosity-force unit. The unit calculates a general viscous force using

$$\vec{f}_{visc} = k_{visc}(\vec{v}_{ref} - \vec{v}_{particle}), \tag{2}$$

where $\vec{f}_{visc}$ is the viscous-force result, k_{visc} is the scalar viscosity factor, $\vec{v}_{ref}$ is the reference velocity (analogous to the velocity vector of the fluid in which the particle is immersed), and $\vec{v}_{particle}$ is the velocity vector of the particle.

3.4.2 *Acceleration.* An acceleration vector must be obtained from the total force vector. This is done in the force-to-acceleration stage using the well-known relationship

$$\vec{a} = \frac{1}{m}\vec{f}_{total}. \tag{3}$$

First, one hardware division obtains the $\frac{1}{m}$ term from the particle's mass value. This term is then multiplied with each component of the total force vector to obtain the particle's acceleration. A less flexible particle system may be implemented entirely without this stage, thus eliminating one inversion and three fixed-point multiplications from the pipeline design. This can be done by forcing all particles to have the same mass and making the proper choice of units. If it is not practical to include a mass term in the particle datasets, particle mass may be treated as a type-selected parameter in the acceleration stage. Each particle type may be assigned a type mass, and this value is stored in the pipeline parameter tables. In the next generation of the particle pipeline, described in Section 4, the particle dataset contains the inverse of the particle mass, eliminating the divider and replacing it with a multiplier.

3.4.3 *Integration.* Each particle's path of motion, position, and velocity over time needs to be integrated and updated according to Eq. (3). The Euler method is chosen for the integrator. Figure 6 shows the particle pipeline's motion integration circuit. The pipeline described in Section 4 implements

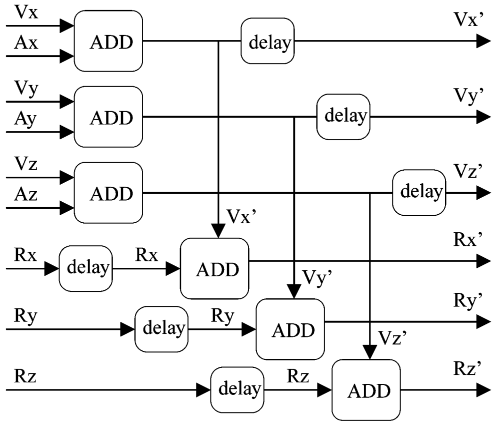

Fig. 6. Euler integration stage.

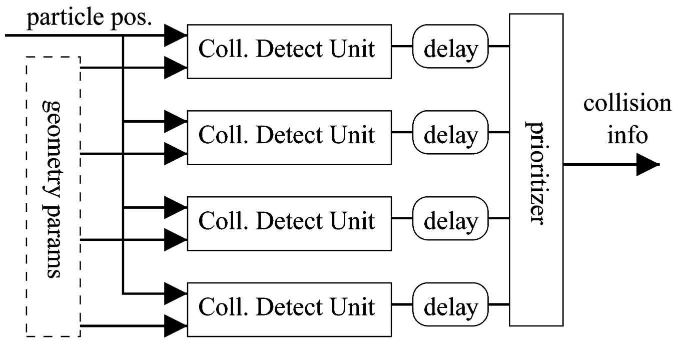

Fig. 7. Collision detection system.

higher-order Runge-Kutta methods as well. With the Euler method, the position and velocity vectors are updated as follows.

$$\begin{aligned}\vec{v}_{new} &= \vec{v} + h\vec{a} \\ \vec{r}_{new} &= \vec{r} + h\vec{v}\end{aligned} \tag{4}$$

The problem is scaled such that h is unit time.

3.4.4 *Update.* The update stage implements various particle update rules, such as:

—decrementing particle *life count*;

—fading particle colours by a *color step*;

—*killing* particles which satisfy some condition, such as energy or position beyond a given value; or

—*interpolating* between colors based on another value, such as time, energy, or life.

3.4.5 *Collision Detection.* The collision detection system shown in Figure 7 is comprised of a parallel collection of collision detection units. Each such unit detects and reports collision information with one type of geometry. For example, the plane collision detection unit detects collisions of particles with a plane, and reports information about that collision if detected. The collision detection system shares the same modular approach as for the force system, facilitating the inclusion of new units for custom geometries. Inputs to the collision detection units include the particle position and a set of type-selected parameters defining the collision geometry. Each collision detection unit output contains a collision flag indicating whether a collision was detected, an estimate for the point of intersection, a surface normal vector at the intersection point, and surface friction and bounce factors. In our example of the plane detection unit, parameter registers would define the exact orientation and location of the plane, and on what side of this plane the particles should collide. They also provide the surface properties, bounce-, and friction factors which will be used to respond to a detected collision. Each collision detection unit detects collisions for a basic geometry. Collision detection for slightly more complex shapes can be achieved by approximating the desired shape with a hierarchy of several basic detection units.

As shown in Figure 7, the collision detection units in parallel resemble the force system. The collision information sets are each delayed and synchronized. Finally, one set of collision information is selected from the detection unit outputs and passed forward to the collision response stage.

3.4.6 *Collision Response.* Figure 8 shows a simplified version of the collision response system. If there was a collision detected and the collision flag received as part of the input collision information was set, the collision response system will need to perform the following tasks.

(1) Replace the particle position by the intersection point estimate.
(2) Calculate the component of particle velocity tangent to the collision surface.
(3) Calculate the component of particle velocity normal to the collision surface.
(4) Scale the tangential particle velocity by the surface friction factor.
(5) If the projection of the particle velocity on the surface normal is negative, the particle must be “bounced” off the surface by multiplying normal particle velocity by bounce factor.
(6) Combine the updated normal and tangential particle velocity components to form a new total velocity vector for the particle.

The first operation in the collision response system, shown in Figure 8, breaks the particle velocity into a normal velocity vector and tangential velocity vector (relative to the surface), using the relationships

$$\begin{aligned}\vec{v}_{norm} &= (\vec{v} \cdot \hat{n}_{surface})\hat{n}_{surface} \\ \vec{v}_{tang} &= \vec{v} - \vec{v}_{norm}.\end{aligned} \tag{5}$$

First, the dot-product of the particle velocity $\vec{v}$ and surface normal $\hat{n}_{surface}$ is computed. The result of the dot-product is then used to scale a surface normal vector, which produces the normal velocity component in vector form. Then, the result of the scaling, that is, the normal velocity vector, is subtracted from

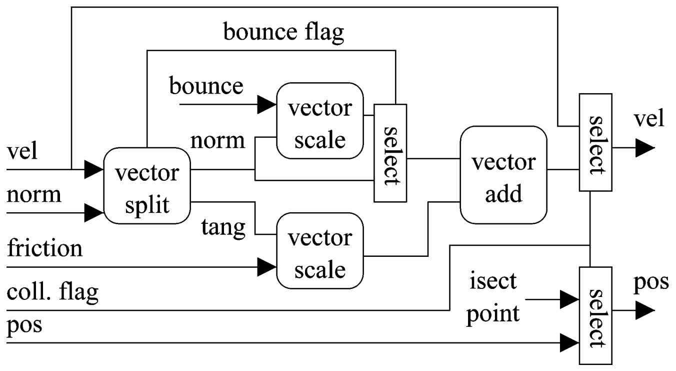

Fig. 8. Simplified collision response system.

a delayed version of the original particle velocity, producing the tangential velocity vector.

During the second section of collision response, the tangential velocity vector is scaled by the friction factor and, in parallel, the normal velocity vector is scaled by the bounce factor.

$$\begin{aligned}\vec{v}_{tang} &\leftarrow k_{friction}\vec{v}_{tang}\\ \vec{v}_{norm} &\leftarrow k_{bounce}\vec{v}_{norm}\end{aligned} \tag{6}$$

The surface friction factor would usually be a fixed-point number between zero and one, while the surface bounce factor should be a negative number to create the bounce, a reversing of the normal velocity component.

Recall that the dot-product of the original particle velocity vector and the surface normal has already been computed during the first section of collision response. The sign-bit of this result (the bounce flag) is delayed so that it can be used to determine whether the particle should be bounced. If the sign-bit is set, the bounced normal velocity just calculated is selected; otherwise, a delayed version of the original normal velocity is selected. Some examples of collisions without bounce are: a particle that is sliding along a surface, at rest on a surface, or that has already bounced but is still in collision with a surface.

Next, the selected normal velocity, either bounced or not bounced, is combined with the scaled tangential velocity to create a new total velocity vector which is the proper response to a potential collision. Finally, shown at the end of the collision response stage in Figure 8, if the collision flag was set, indicating that there was in fact a collision, the particle velocity is replaced with this new collided velocity, and the particle position is replaced with the intersection position estimate; otherwise, the original position and velocity are used.

3.4.7 *Rendering.* A simplification of the pipeline's rendering system is shown in Figure 9. Rendering is the last stage of the pipeline. At this stage, particle data has been completely updated. Graphical information is calculated using the newly updated particle data together with rendering parameters, initialized and updated continuously by the particle microcontroller. The graphical information calculated and output for each particle includes:

—a visibility flag,
—screen-pixel coordinates,
—a frame buffer address or index,
—a color value, and
—a *z-buffer* depth value.

As shown in Figure 2, after the rendering stage, the rendered graphical information and particle data are output together from the pipe. Particle data will be sent back to particle memory, and rendering results will be sent to the graphics system for display.

Rendering is described in detail in Shirley et al. [2005]. The first task in rendering is to find a set of *view coordinates* for the particles. Simulations exist in *world space*, and the particle position vectors are in world space coordinates. In Figure 9(a), to convert particle position coordinates from world space to view space, the transform

$$\begin{aligned} x_{view} &= \vec{r}_{world} \cdot \vec{right} + d_x \\ y_{view} &= \vec{r}_{world} \cdot \vec{up} + d_y \\ z_{view} &= \vec{r}_{world} \cdot \vec{dir} + d_z \end{aligned} \tag{7}$$

is applied, where $\vec{right}$, $\vec{up}$, and $\vec{dir}$ are three vectors defining the camera orientation in world-space coordinates. The vector $\vec{d}$ represents the location of the world origin in view-space coordinates. These values are held in parameter registers which software uses to control the view throughout a simulation.

After having obtained view space coordinates for the particle position, the view-space-position vector needs to be projected onto a 2D surface, the viewing screen. This is accomplished, as shown in Figure 9(b), using the following relationships.

$$\begin{aligned} x_{screen} &= k_{xscale}\, x_{view} \frac{1}{z_{view}} \\ y_{screen} &= k_{yscale}\, y_{view} \frac{1}{z_{view}} \end{aligned} \tag{8}$$

The previous two equations provide coordinates relative to the center of the screen. Generally, pixel coordinates are most conveniently specified relative to the top-left corner of the screen with positive values going down and to the right. To correct this, as shown in Figure 9(c), there is a transformation of the screen coordinates into more useful pixel coordinates.

$$\begin{aligned} x_{pixel} &= x_{center} + x_{screen} \\ y_{pixel} &= y_{center} - y_{screen} \end{aligned} \tag{9}$$

Finally, the visibility of the particle is determined by comparing the view space z value to a minimum value, and checking that x_{pixel} and y_{pixel} are within the valid range.

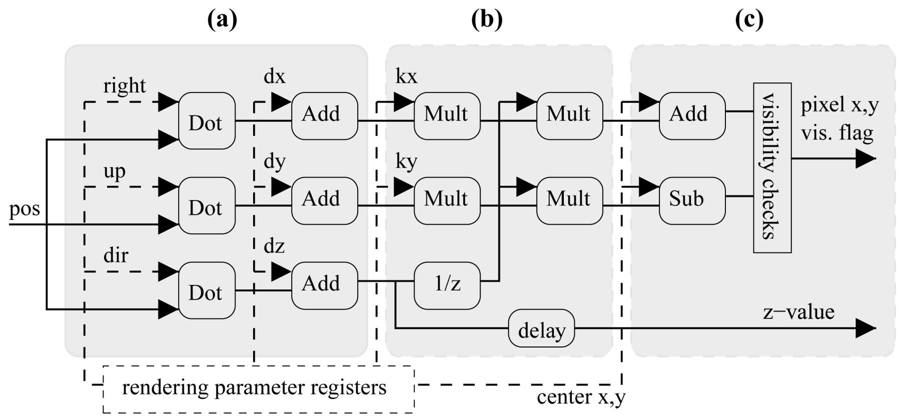

Fig. 9. Simplified rendering system.

3.5 System Implementation

A proof-of-concept test system implemented on an Altera Stratix FPGA (Particle Pipeline 1, or PP1) is shown in Figure 10. In this section we discuss in detail the design and implementation of PP1 and present measured performance results.

3.5.1 *Hardware Library, Configuration, and Simulation.* The particle pipeline, together with its supporting hardware logic, was designed as a modular, parameterized, and configurable VHDL hardware library. Scripts were developed for automatic generation and configuration of particular particle pipeline design instances. These scripts configure data formats, particle dataset members, and functional unit parameters. These scripts also generate the necessary software header files. Of particular importance is the microcontroller's interface to the pipeline parameter bus system. These tables and the parameter bus interface are specific to the particular pipeline configuration created, and therefore need to be automatically generated by the configuration system.

A functional-C software model was created in parallel with the development of the particle pipeline VHDL hardware design. It is a bit-accurate, functionally equivalent software model of the particle pipeline that can be used as a numerical and graphical testbench for testing and verifying particle pipeline configurations, designing and verifying new hardware units, and developing and experimenting with new effects, graphically displaying interactive particle simulations such as the one shown in Figure 11.

This structured design approach allows flexible and efficient customization and extension of particle pipeline design instances, without extensive knowledge or exhaustive understanding of the particle pipeline. In this way, particle pipeline systems can be designed and modified in much the same way that traditional software design can be done, using the framework and functionality of existing works and libraries.

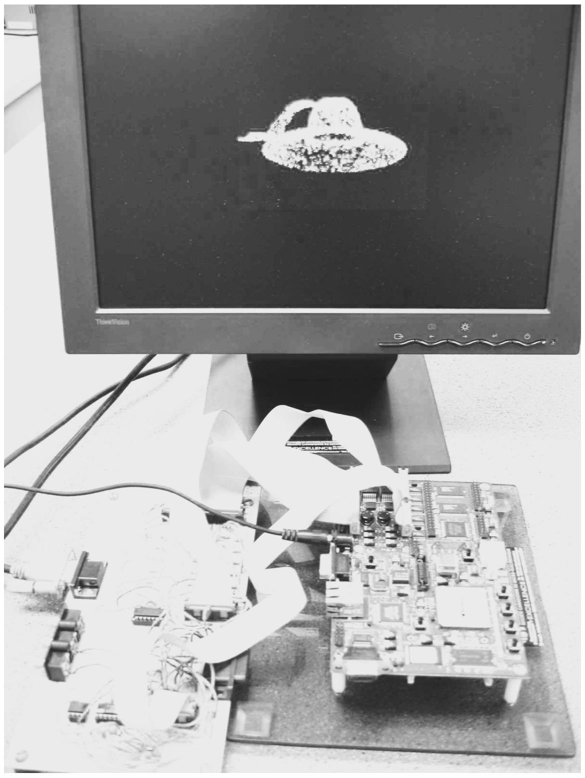

Fig. 10. PP1 test system.

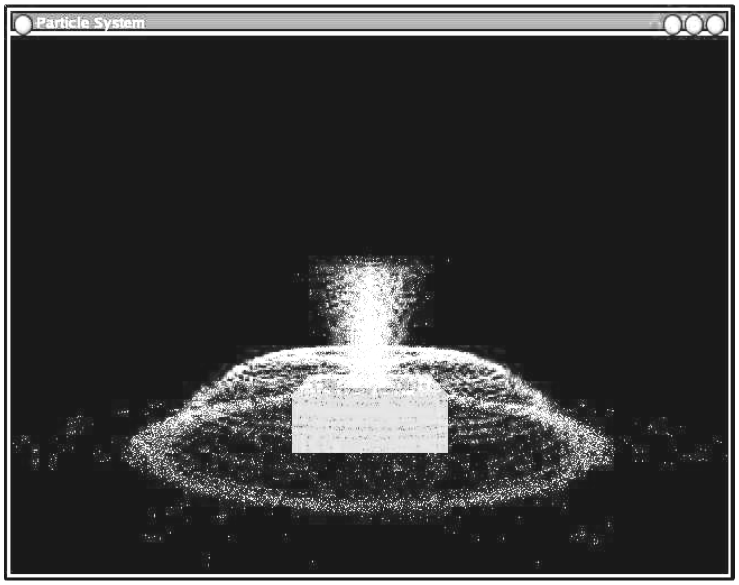

Fig. 11. Particle graphics simulation.

3.5.2 *System Configuration.* The first step in implementing the particle pipeline system was to generate a particle pipeline configuration. The test effect is a water-fountain simulation, as shown in Figure 11 generated by the simulator, and as shown in Figure 10 generated by the PP1 system. The software pipeline simulator was then used to test and experiment with this configuration, verifying that it was capable of generating the desired effects. Note that the ribbon cables in Figure 10 connect the FPGA board to a custom-built VGA interface. An 18-bit fixed-point format was chosen for the particle pipeline's internal fixed-point number representation. This was done to best utilize the Stratix EP1S40 FPGA's hardwired DSP units, which can be used for either fifty-six 18×18 or fourteen 36×36 multiplier circuits.

The particle datasets were configured to include the following fields:

—velocity vector: three 18-bit fixed-point values;
—position vector: three 18-bit fixed-point values;
—color: 8 bits (4 bits for blue and 4 bits for green);
—life: 11 bits; and
—type: 1 bit.

With these chosen formats, a packed particle dataset is 128 bits, which can be read from or written to particle memory in four accesses to a 32-bit-wide RAM device.

All fixed-point numbers in the particle pipeline, as well as in the parameter table system, will then use the same 18-bit format. Generation and configuration scripts create a particle pipeline hardware design configured to the specifications, also generating a pipeline parameter table and bus system specific to the chosen data formats and pipeline configuration. Finally, source-code is generated or configured, defining the software interfaces used by the Nios microcontroller to interface to the pipeline and its parameter table system, particle-loading and -storing hardware, the graphics data output buffer, and particle memory for the initialization of new particles.

3.5.3 *Nios Microcontroller System.* The Nios soft-core microcontroller was used as the *particle microcontroller*. The Nios system has flexible tools for designing reconfigurable 32-bit microcontroller systems in Altera FPGAs, with pipelined, multimaster bus architectures. The development board used provides the FPGA with access to two separate RAM devices. One of these devices, the smaller SRAM, is used as Nios system memory, holding microprocessor code and program data, while the other is used as particle memory. A section of the Nios memory is reserved for use as a video-frame buffer, accessed by both the Nios microcontroller and the video controller. With the use of two separate on-chip bus systems and two independent RAM devices, shown in Figures 12 and 13, transfers on the microcontroller bus system will not conflict or compete with streaming particle data on the particle data bus between particle memory and the particle pipeline. The Nios microcontroller does, however, also have access to the particle data bus. Between simulation passes when there is no streaming

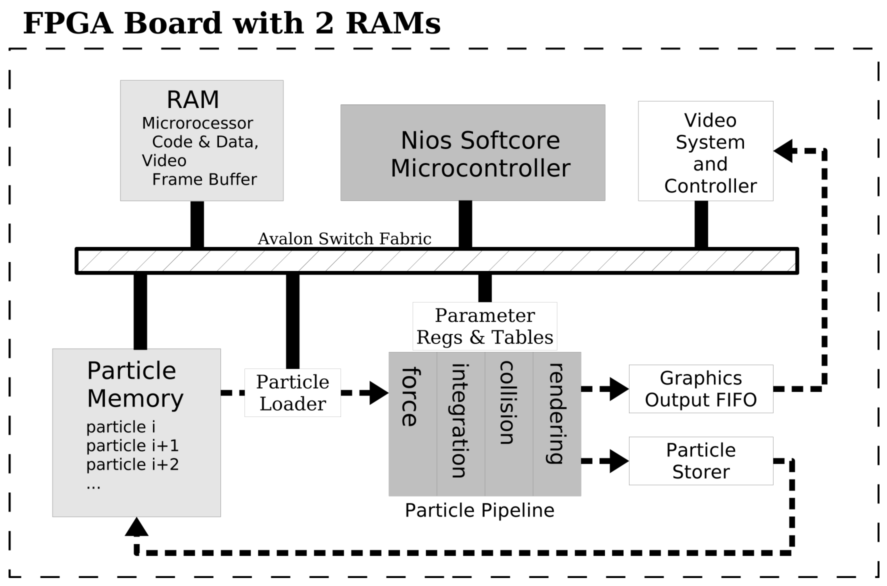

Fig. 12. PP1 system architecture overview.

particle data, the Nios microcontroller will access particle memory to initialize new particles.

Figures 12 and 13 show that the Nios has access to its own microcontroller bus system, the particle data bus, and pipeline parameter bus. The Nios microcontroller bus allows it to access the RAM used for microprocessor code, program data, and a video-frame buffer. It also allows the microprocessor to configure and control the various hardware units supporting the pipeline through each unit's configuration and control slave interface. These units include the particle-loading and -storing hardware, the pipe's output buffers, and the video system's configuration port. The Nios has access to particle memory for initialization of new particles and also for testing purposes. The pipeline parameter bus allows it to access the pipe's parameter tables during simulations and initialization. Finally, the video controller is also a master of the microcontroller bus, giving it access to the video-frame buffer located in microprocessor data memory.

3.5.4 *Particle Data Flow.* The flow of particle data represents the biggest challenge faced when implementing a particle pipeline system. The particle pipeline is designed to be capable of receiving one input particle dataset and outputting one updated particle dataset, together with its graphical data, in every clock cycle. In the case of our test system implementation, data formats were chosen such that each particle dataset fits into exactly 128 bits, or four 32-bit words. For each particle that goes through the pipeline, four 32-bit words of particle data must be read from and written to particle memory. To help achieve the highest possible throughput of particle data, it is necessary to:

—use special hardware for the loading, storing, and transfer of particle data;

—store particle data in a dedicated RAM device, isolated from other bus transactions;

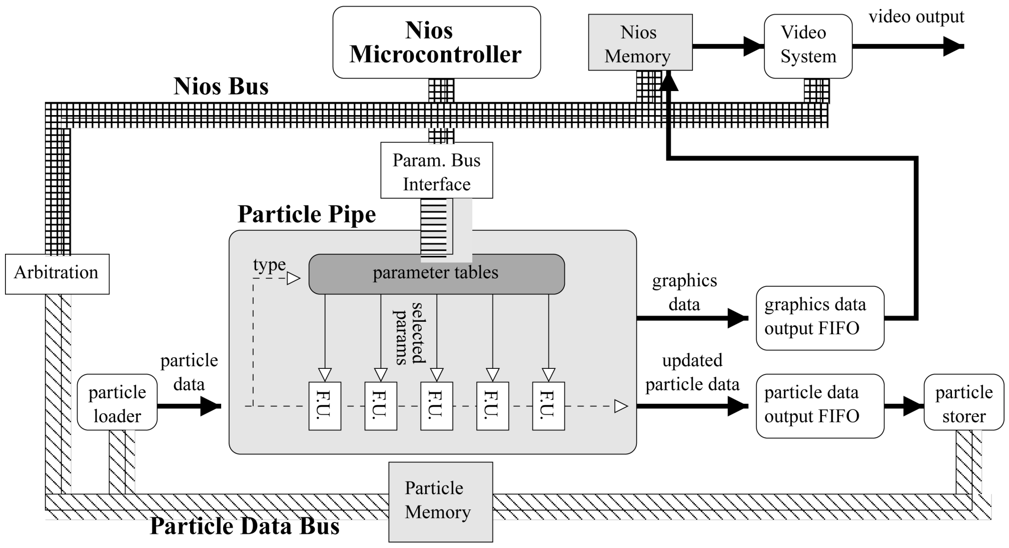

Fig. 13. PP1 system buses.

—transfer particle data using an isolated and dedicated particle data bus; and
—attempt to fully exploit bus- and RAM-device high-throughput burst modes by designing buffered hardware which accesses particle memory in bursts of sequential accesses.

Figure 12 shows how the particle data flows through the particle test system in its own data path. It is streamed from particle memory into the pipeline by the particle-loading hardware. From the pipe's output it enters the pipeline output FIFO buffers, wherein it will be written back to particle memory by the particle-storing hardware. This whole process of streaming data through the particle pipeline occurs independently and in isolation, without direct participation by the particle microcontroller and its bus. Therefore, it occurs without contending or competing with the bus traffic of the microcontroller's video system, particle initialization system, or parameter bus system. A hardware-loading unit streams data out of particle memory from a specific address range specified during initialization, and feeds the data to the input end of the particle pipeline at a specified rate. Since the width of the particle datasets, and therefore the width of the pipe's input port, is 128 bits, or four 32-bit words, the particle loader will fetch all four words of one particle then apply them together to the pipe's input port, forming a valid input cycle. When the loading hardware is not capable of providing the pipeline with particle data at its throughput rate of one dataset per pipeline clock cycle, or when it is necessary to decrease the rate of input data flow to avoid overflowing the pipe's output buffers, blank entries or pipeline bubbles are inserted to allow the pipeline to progress when no new data is available.

As the particle data moves through the pipe, updated particle data (together with graphics data for visible particles) will stream out of the output end of the pipe. This output data must be buffered before it can be handled properly. The particle data and graphics data need to be buffered separately because they are processed at different rates and by different mechanisms. Therefore, as shown in Figure 12, the output end of the particle pipeline feeds into two FIFO buffers, one for the particle data and one for the graphics data. These pipeline output buffers contain logic and signals for tracking information about the number of “valid” and “active” particles output from the pipeline, and for making this information available to the Nios microcontroller over its control bus. The state of these FIFOs can be accessed by both the particle-storing hardware (to control the rate of streaming data though the pipe) and by the Nios microcontroller.

For inactive particles, as well as for those that were determined to be invisible by the rendering hardware, the graphics data output is flagged as invisible and will not enter the graphics data output buffer. Similarly, during “invalid” or stalled cycles, or when particles have an inactive state, they will not enter the particle data output buffer. Each buffer tracks the number of valid cycles or datasets that were presented at its input, the number of datasets that have been read out of the FIFO, the number of datasets currently in the FIFO, and its empty or full status.

The particle-storing hardware continuously writes data from the pipe’s particle data output buffer back to particle memory, in the address range specified by the Nios microcontroller during initialization. Since inactive particles are not stored back to particle memory, as all valid and active particles are stored back sequentially to the same block of particle memory from which they were read, the spaces or entries in particle memory occupied by inactive particles will be shifted to the top of particle memory. At the end of each simulation pass, by knowing how many “active” particle datasets were stored back to particle memory, the microcontroller knows at what address the “inactive” and available space begins and how large it is. This block of available, inactive space at the top of particle memory can be used between frames for the initialization and injection of new particles. Any remaining space in this range which is not written over with newly initialized particle data must be cleared to an inactive state, as it still contains data from particles left over from the previous simulation frame.

3.6 Results

The PP1 system provides a functioning and interactive *proof-of-concept* for use in pipelined FPGA hardware designs to accelerate and enhance particle graphics techniques in real-time applications.

The PP1 system was implemented using an Altera EP1S40 Stratix FPGA. Table I shows the FPGA resource utilization. The particle pipeline alone was synthesized for the aforementioned device to a maximum operating frequency of 130 MHz. The complete test system, including the Nios microcontroller, buses, controllers, and other components, operates at a system frequency of 80 MHz. Since the particle pipeline design is capable of updating and rendering one particle dataset at each clock cycle, the particle pipeline itself has a potential

Table I. PP1 Stratix EP1S40 FPGA Resource Utilization

	Logic Cells	Registers	18x18 Mult.
Force System	5,157 (13%)	4,962 (11%)	3 (5%)
Coll. Detection	2,472 (6%)	2,301 (5%)	0 (0%)
Coll. Response	12,699 (31%)	12,031 (27%)	12 (21%)
Integrate Motion	624 (2%)	582 (1%)	0 (0%)
Rendering	8,382 (20%)	7,034 (16%)	13 (23%)
Nios Microcontroller	6,397 (15.5%)	2,424 (5%)	1 (2%)
Pipeline interfaces	2679 (6%)	1303 (3%)	0 (0%)
Total System Design	40,764 (99%)	32,215 (72%)	29 (52%)
Avail. in FPGA	41,250 (100%)	44,860 (100%)	56 (100%)

Table II. PP1 Performance at 75 MHz.

Frame Rate	Video System	Num. of Particles	Total Frame Rate
30 Hz	off	45,000	29.83 Hz
30 Hz	on	41,000	29.71 Hz
60 Hz	off	22,000	59.86 Hz
60 Hz	on	18,000	59.45 Hz

throughput of 130 million particles per second. This corresponds to simulations and effects with 2.1 million particles in each frame at a frame rate of 60 Hz, or 4.3 million particles per frame at 30 Hz.

These throughputs depend on a particle memory capable of providing the required access rates, and a system integration capable of processing and displaying the generated graphical data. Given a particle-dataset width of 128 bits, to achieve the pipeline's maximum throughput, the read rate required of particle memory is

$$\left(128\ \frac{bits}{part.}\right)(130\ MHz)\left(1\ \frac{part.}{cycle}\right) = 2.08\ \frac{gigabytes}{sec}\ . \tag{10}$$

This can be realized with standard DDR-SDRAM memory modules. The writes to memory require the same access rate. A rate of 1.28GB per second is required to fully utilize the particle pipeline in an 80 MHz test system. This would correspond to a peak performance of 1.3 million particles per frame at a frame rate of 60 Hz, or 2.6 million particles per frame at 30 Hz.

Table II shows the actual performance observed from the PP1 system running demonstration code at 75 MHz, generating the effect shown in Figure 10 while targeting frame rates of 30 and 60 Hz. The system was tested at 75 MHz to match the 75 MHz clock required for the video-display interface. The largest particle simulation implementable on the test system contains 45,000 particles at a frame rate of 30 Hz. The video system, when enabled, reduced the simulation size to 41,000 particles.

The video-processing and -display capabilities of the test system are low in performance compared to standard computer hardware. The inability of the test system as a whole to efficiently process and remove graphical data from the particle pipeline's output buffer severely limits the speed and size of simulations.

Combined with low particle memory access rates, the test system's performance is limited to well below the particle pipeline peak performance. The particle memory copy rate achievable by the PP1 system's loading and storing hardware has been measured as limited to 0.1365 bytes per clock cycle. With this in mind, considering that the particle pipeline at maximum throughput would require a copy rate of 128 bits (16 bytes) per cycle, particle memory starvation is responsible for limiting particle pipeline utilization to $0.1365 \times 8/128 = 0.85\%$ of its peak performance. During simulations on the test system, the particle pipeline is actually stalled and not in use for 99.15% of all cycles. As a first implementation of the particle pipeline, and primarily intended for the testing and verification of the concept and operation of the particle pipeline, the PP1 system was designed and organized in the most direct way possible. The particle-data-loading and -storing units were designed as two separate controllers, each a master on the particle data bus. The storing unit blindly attempts to empty the particle data output buffer, while the loading unit loads particle data at a specified rate. Coordination and arbitration between these two bus masters attempting to simultaneously access particle memory are resolved by the bus arbitration logic. As the loading and storing units attempt to simultaneously read and write from the single particle memory device, access is shared between the two with alternating permission. The result of this simple solution is that what should be two sequential, pipelined, high-speed bursts (one read stream and one write stream) becomes alternating random accesses by two different masters. The bus transfers are not pipelined, the SDRAM controller cannot reach a burst mode of operation, and the access rates seen by the particle pipeline are extremely slow. Double-buffering to separate memories is one solution for this issue.

A better particle data system should be designed to maximize the performance of the particle memory, since, as we have seen, access to particle memory is the major factor limiting particle pipeline performance. For the design of a high-performance particle pipeline system, the particle loading and storing units should be merged into a single pipelined design capable of exclusively bursting sequential data from particle memory, while all pipeline output is buffered for some duration. Then, the pipeline output should be exclusively bursted back to particle memory, while the pipeline is either not fed with input data or is fed from a read buffer. The transfer of data should be separated from the transfer-of-address and control information to allow multiple, simultaneous pending read operations. A particle pipeline system designed in such a way could be expected to achieve particle memory access rates close to the performance limits of the memory device itself, or of the clock frequency and width of the system bus.

4. HIGH-LEVEL LANGUAGE DESIGN OF A HIGH-PERFORMANCE HARDWARE PARTICLE ENGINE

In Section 3 we demonstrated that a particle graphics engine capable of processing approximately 2 million particles per frame at 60 Hz could be built using relatively modest resources in an FPGA. The PP1 system built to exercise the pipeline, however, could not support this level of performance. It was calculated that to achieve maximum performance, about 2GB per second of bandwidth was required to fill and drain the pipeline, and that this was achievable with commodity memory. The FPGA development board we had available at the time was not adequate to support the required memory bandwidth. In the next phase of this work, an FPGA board with improved memory bandwidth became available. In this section, we describe a new version of the particle engine, PP2, that takes advantage of the memory bandwidth available in the new FPGA board and extends the previous work. The main features of the new pipeline are listed next.

(1) The pipeline is interfaced to memory over a high-speed 3.2GB per-second link.
(2) The pipeline is programmed in a high-level C-like language, which is accessible to software designers and does not require any knowledge of hardware design.
(3) Floating-point data types are used, instead of fixed-point.
(4) Higher-order integration schemes are used.
(5) The system is scalable to use multiple particle pipelines.

4.1 Hardware Configuration

The test system is an SGI Altix 350 with 16GB of main memory. The CPU is a 1.5 GHz Itanium 2. The system contains four Xilinx Virtex-4 LX200 FPGAs, each connected to the shared main memory by the NUMALink bus with 3.2GBper-second unidirectional bandwidth per link. Each FPGA is connected to 32MB of local high-speed memory, arranged as $2 \times 2^{10} \times 128$-bit words, that serves as a software-controlled cache.

4.2 Programming

The main barrier to the widespread adoption of FPGAs as processors for graphics acceleration has been the significant barrier-to-entry for programmers. Generally, programming FPGAs has been the domain of hardware designers working with hardware description languages such as VHDL or Verilog and taking into account low-level hardware structures and detailed clock-cycle timing. Recently, a number of commercial C-like languages for programming FPGAs have been introduced to make FPGAs more accessible to software programmers [Styles and Luk 2000; El-Araby et al. 2007; Koo et al. 2007].

The particle pipeline was reprogrammed in Mitrion-C. Mitrion-C is a functional programming language where the programmer describes the data flow of the algorithm, independent of the underlying hardware implementation. The compiler then translates the data flow into a hardware configuration for an FPGA. The language contains several constructs that make expressing parallelism very easy. There are two array types: *vectors* and *lists*. Elements in a vector can be accessed in parallel, while the elements in a list are accessed in sequence. A parallel for-loop, called *foreach*, indicates that the iterations of the loop are independent and can be executed in parallel. Applying a foreach loop

Table III. PP2 FPGA Resource Utilization

	Euler	RK2	RK3
Registers (178,176 total)	24283 (13%)	27,821 (15%)	37,999 (21%)
4-LUTs (178,176 total)	21749 (12%)	26,926 (15%)	41,547 (23%)
18x18 Multipliers (96 total)	16 (17%)	25 (26%)	34 (35%)
Block RAMs (336 total)	25 (7%)	28 (8%)	41 (12%)

to a vector creates explicit parallel hardware structures, while a foreach loop on a list creates a pipeline.

It is straightforward to implement the particle pipeline in Mitrion-C. The particles are arranged as a list and processed in a foreach loop. This results in the pipeline structure. Within the processing units of the pipeline (integration, collision detection, etc.), parallelism is expressed by foreach loops operating on vectors of data. The description and simulation of the particle pipeline in Mitrion-C was completed in about 2 weeks, which is significantly faster than the design time for PP1.

4.3 Implementation

The particle pipeline was described in Mitrion-C and synthesized to a Xilinx Virtex 4 LX200 device running at 100 MHz. The Mitrion tools produce a circuit that is always clocked at a fixed frequency of 100 MHz. The configuration of the pipeline described in Section 3 was upgraded in several areas.

4.3.1 *Data Type.* The data type used for all variables is the 16-bit (6-bit exponent and 9-bit mantissa) floating-point type from the OpenEXR libraries that has also been used in recently reported particle engine implementations [Krüger et al. 2005]. The 16-bit data size enables one particle's dataset to fit in a single 128-bit word of the FPGA local memory.

4.3.2 *Integration.* To investigate the hardware cost of higher-order integration schemes, the pipeline can be configured to use Euler, second-order Runge-Kutta (RK2), or third-order Runge-Kutta (RK3) integrators.

4.3.3 *Double-Buffering and Multi-FPGA Scaling.* The FPGA reads and writes its data from and to local SRAM memories. In order to maintain the pipeline data flow, the memory is split into two independent 128-bit wide banks of 16MB each. Each of these banks can hold 1024×1024 particles, assuming a 128-bit particle representation. To accommodate a larger number of particles, double-buffering is used. Each of the two 16MB memory banks are split further into two 8MB regions, one of which is used by the pipeline for data access and one for supplying (and draining) data from (and to) the NUMALink bus. This double-buffering technique hides the data transfer times to the FPGA local memory, with the exception of the first and last blocks to and from the FPGA.

Even with double-buffering, we found that the available data transfer bandwidth, and not the computational bandwidth, was the bottleneck in our implementation (see Section 4.4). To alleviate this, we scaled the design to use up to four parallel pipelines, each on a separate FPGA with its own connection to

Table IV. Particle Engine Performance Comparison (Measured in millions of particles per second (Mpps))

Implementation	Number of Particles	Euler	RK2	RK3
2.8 GHz Pentium 4 CPU	1,048,576 – 52,428,800	2.14	1.27	0.81
GPU [Krüger et al. 2005]	1,048,576	59	51	41
1 FPGA (PP2)	1,048,576	46.78	47.27	46.84
1 FPGA (PP2)	52,428,800	55.18	55.41	54.79

Table V. Speedup of PP2 FPGA System over a 2.8 GHz Pentium 4 CPU (52,428,800 particles)

# FPGAs	Euler	RK2	RK3
1	26	44	68
2	48	81	127
3	52	89	139
4	53	89	139

NUMALink. The double-buffering and multi-FPGA scaling were automatically handled by SGI libraries.

4.4 Results

The FPGA resource utilization for PP2 is reported in Table III. The PP2 system performance was measured with hardware timers and includes all overhead incurred by data transfers between memory and the FPGA and control of the FPGA. The results are shown in Table IV, expressed in millions of particles per second. The results are compared with a software implementation running on a 2.8 GHz Pentium 4 CPU and the GPU implementation in Krüger et al. [2005]. Note that the GPU performance results in Table IV are for particle updates only and do not include the rendering of the particle sprites. Similarly, the PP2 implementation does not perform the rendering step. With one FPGA processing frames with 1,048,576 particles, the PP2 system achieves an average performance of 47 million particles per second, representing a speedup over the CPU of 22, 37, and 58 times for Euler, RK2, and RK3 integration, respectively. It is interesting to note that the clock frequency of the CPU is 28 times higher than the FPGA and the clock frequency of the GPU (525 MHz ATI X800 XT) in Krüger et al. [2005] is 5.25 times higher than the FPGA. The CPU achieves at best approximately 0.0008 particles per cycle, the GPU achieves 0.11 particles per cycle, and the FPGA achieves 0.47 particles per cycle. The efficiency of the FPGA improves as the size of dataset increases, due to the diminished impact of the communications overhead. With a data buffer size of 1024^2 particles (16MB), the computation time on the FPGA is not long enough to completely overlap the communications overhead. To test the limits of PP2, a very large dataset of 52,428,800 particles was tested, achieving 0.55 particles per cycle and a speedup of 26 to 68 times over the CPU implementation.

A single PP2 pipeline is therefore able to deliver an order-of-magnitude improvement in speed over a software implementation. This is a comparable result to the GPU implementation from Krüger et al. [2005], which has speedups

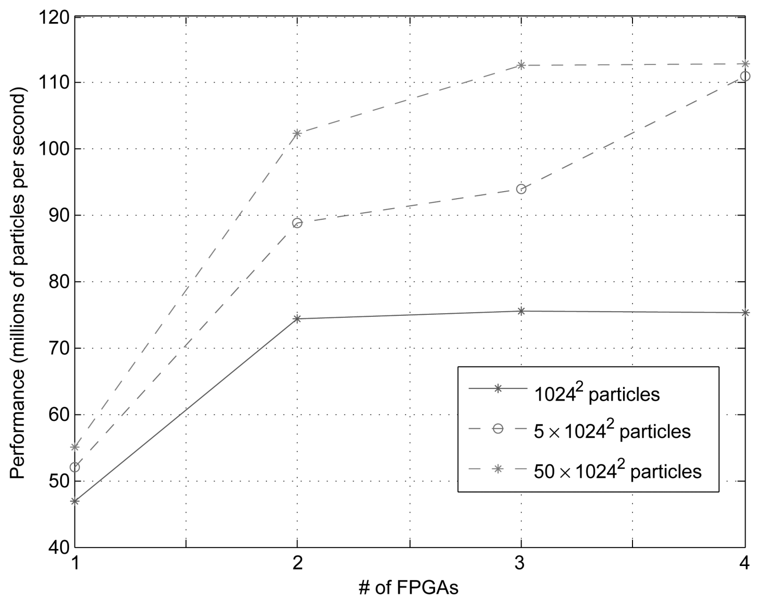

Fig. 14. Performance of PP2 with multiple FPGAs.

ranging from 25 to 56 for floating-point data. However, it is interesting to note that the GPU performance decreases as the complexity of the particle pipeline increases, for example, when implementing higher-order integration methods. This is because the higher-order integrators require multiple passes through the data. On the other hand, the performance of the FPGA implementation is the same for all integration methods because the FPGA is able to trade-off increased area for speed. The FPGA design fully unrolls and pipelines all loops, requiring a single pass through the pipeline for each particle. The space increase is modest, however, leaving abundant free space on the device for further improvements.

From Table IV the performance of a single FPGA in PP2 is limited to 55 Mpps. The performance bottleneck is the bandwidth limit of a single NUMA-Link connection. To increase the system performance, we scaled the PP2 system to up to four FPGAs, each running an identical copy of the PP2 pipeline. Each FPGA has its own connection to NUMALink. The performance results are presented in Table V in terms of speedup over the Pentium 4 CPU. The maximum performance obtained was 139 times speedup, using three FPGAs and RK3 integration. The characteristics of scaling as a function of the number of FPGAs is plotted in Figure 14. No discernible difference in performance is measured for the different integration methods, therefore the value reported is an average over all methods. The peak performance observed was 112 Mpps with three or four FPGAs. A million-particle system can achieve 75 Mpps, corresponding to frame rates of up to 71 Hz with two FPGAs. Note that the resources for two particle pipelines can fit in a single FPGA, and the purpose of multiple FPGAs here is to deliver increased communications bandwidth. We expect that upcoming improvements in system software from the vendor will further increase the PP2 performance towards its maximum value of 100 Mpps per FPGA.

5. CONCLUSIONS

In this article we explored the design and implementation of an FPGA-based hardware particle graphics engine. Results show that the FPGA-based engine achieves up to 139 times speedup over CPUs. Custom hardware implementations are competitive with GPUs for simple pipeline processing, but gain an edge when the order of the integration is increased because the FPGA can process the particles in a single pass, in contrast to the multipass GPU algorithm. A single FPGA could process a million-particle system at a rate of 47 million particles per second, while a system of three FPGAs achieved 112 million particles per second. We showed that the design of a complex physics-based algorithm could be implemented in hardware using a high-level software language.

We expect that recent trends in FPGA technology, both in hardware and software, will make FPGAs an attractive programmable processing element to augment CPUs and GPUs. Combined with high-performance CPU-FPGA interconnections such as those recently announced as direct connections to CPU front-side busses [Maxwell 2007], FPGAs offer a potential for low-cost, high-performance custom hardware accelerators for a host of graphics and physics algorithms. Future graphics system architectures could integrate FPGAs and GPUs over a high-bandwidth and low-latency interface. Future work could include implementing parallel particle pipelines on a single FPGA to take advantage of increased communications and memory bandwidth. This would serve as a good point of comparison with modern GPUs based on the unified shader architecture.

REFERENCES

AZIZI, N., KUON, I., EGIER, A., ARABIHA, A., AND CHOW, P. 2004. Reconfigurable molecular dynamics simulator. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. 197–206.

BEECKLER, J. S. AND GROSS, W. J. 2005. FPGA particle graphics hardware. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. Napa, CA, 85–94.

EL-ARABY, E., TAHER, M., ABOUELLAIL, M., EL-GHAZAWI, T., AND NEWBY, G. B. 2007. Comparative analysis of high level programming for reconfigurable computers: Methodology and empirical study. In *Proceedings of the 3rd Southern Conference on Programmable Logic (SPL)*, 99–106.

HAMADA, T., FUKUSHIGE, T., KAWAI, A., AND MAKINO, J. 1998. PROGRAPE-1: A programmable special-purpose computer for many-body simulations. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 256–257.

HEROUT, A. AND ZEMCIK, P. 2005. Hardware pipeline for rendering clouds of circular points. In *Proceedings of the 13th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision WSCG*, 17–22.

ILMONEN, T. AND KONTKANEN, J. 2003. The second order particle system. In *Proceedings of the 13th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision WSCG*. 11, 240–247.

KIPFER, P., SEGAL, M., AND WESTERMANN, R. 2004. UberFlow: A GPU-based particle engine. In *Proceedings of the ACM SIGGRAPH / EUROGRAPHICS Conference on Graphics Hardware*, 115–122.

KOLB, A., LATTA, L., AND REZK-SALAMA, C. 2004. Hardware-Based simulation and collision detection for large particle systems. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 123–131.

KOO, J. J., FERNÁNDEZ, D., HADDAD, A., AND GROSS, W. J. 2007. Evaluation of a high-level-language methodology for high-performance reconfigurable computers. In *Proceedings of the IEEE 18th International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, 30–35.

KRÜGER, J., KIPFER, P., KONDRATIEVA, P., AND WESTERMANN, R. 2005. A particle system for interactive visulalization of 3D flows. *IEEE Trans. Visual. Comput. Graph. 11*, 6 (Nov./Dec.), 744–756.

LATTA, L. 2004. Building a million particle system. In *Proceedings of the Game Developers Conference*, 54–60.

MAKINO, J. 2006. The GRAPE project. *IEEE Comput. Sci. Eng. 8*, 1 (Jan./Feb.), 30–40.

MAXWELL, C. 2007. FPGA-Based solution interfaces to Intel bus. *EETimes Online*. http://www.eetimes.com/showArticle.jhtml?articleID=203100031.

REEVES, W. T. 1983. Particle systems—A technique for modeling a class of fuzzy objects. *ACM Trans. Graph. 2*, 2 (Apr.), 91–108.

SHIRLEY, P., ASHIKHMIN, M., GLEICHER, M., MARSCHNER, S., REINHARD, E., SUNG, K., THOMPSON, W., AND WILLEMSEN, P. 2005. *Fundamentals of Computer Graphics*, 2nd ed. A. K. Peters, Ltd., Natick, MA.

SIMS, K. 1990. Particle animation and rendering using data parallel computation. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, 405–413.

STYLES, H. AND LUK, W. 2000. Customising graphics applications: Techniques and programming interface. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 77–87.

ZEMCIK, P., HEROUT, A., CRHA, L., FUCIK, O., AND TUPEC, P. 2004. Particle rendering engine in DSP and FPGA. In *Proceedings of the 11th IEEE International Conference and Workshop on Engineering of Computer-Based Systems*, 361–368.

ZEMCIK, P., TISNOVSKY, P., AND HEROUT, A. 2003. Particle rendering pipeline. In *Proceedings of the 19th Spring Conference on Computer Graphics*, 165–170.

Received March 2008; revised May 2008; accepted June 2008