FAST N-BODY SIMULATION AND DISPLAY

Kade Phillips and Scott McCuen

Overview

Our project is a demonstration of an FPGA's strengths as dedicated, parallelized, application-specific hardware. An interesting computation that showcases these properties is direct N-body simulation. On a general purpose processor, this has a time complexity of $O(n^2)$, but on an FPGA this can be done with a time complexity of $O(n)$ for reasonable n (less than 100 on a Virtex-7).
We expect that our FPGA implementation will match an equivalent program in C running on a mid-range processor at this scale, and if it were possible, would outperform the equivalent program by one to two orders of magnitude for $n > 1000$. FPGAs seem to be used frequently as accelerators for N-body simulation, but we are not aware of an FPGA being used for the computation in its entirety.

The project is divided into two logically sequential parts: calculation and display. The simulation takes place in a box one million light-years across, with 50–200 particles. There is one register+arithmetic unit per particle. Wiring each unit to every other unit would reduce the problem to constant time but this requires impossibly dense routing. Our solution is to use a bus that communicates the position of each particle every timestep. To reduce the footprint of each unit (and headache for us), we expect to use fixed-point operations instead of floating-point. With a 100 MHz clock, we expect to run at ~1000 timesteps per second.

We will display the position of particles at 30 frames per second, combining adjacent particles into groups and setting pixel brightness based on the density of particles in a group. At a minimum, we will be able to view the cube from the top, front, and side. If sufficient resources remain on the FPGA, we will add features to rotate and zoom in on the cube.

Modules

There are two primary types of modules: distributed processing units (DPUs) and graphics. Each DPU contains registers for one particle's position and previous position in three dimensions, as well as the logic to perform verlet integration. The DPUs are chained together by parallel buses carrying particle identification tags and position data. Each unit performs computations with this data and then forwards it to the next unit. In this way, every unit sees the location of every particle. The primary graphics module (PGM) sits transparently within this chain, watching what passes through. Given the particle locations, along with commands from an inertial measurement unit (IMU) or 3D mouse, the PGM draws a frame. This frame is passed on to a VGA module, which sends an image to a

display. Alternatively, the PGM can compute x-y points that are passed into a serial module, transmitted to a computer, and then drawn by the computer on its screen.

External Components

Our project will use a VGA display, IMU, and 3D mouse, and we have access to all of the external components that we will need. We will interface with the display as we did in the lab assignments, sending data to the display through a VGA connection. The IMU, which will either be on a breakout board or on a module capable of both UART and BLE communication, will send serial data packets to the FPGA that contain X, Y, and Z acceleration and rotation data from the accelerometer and gyroscope. The exact data rate is yet to be specified, but we can easily stream data at a rate in the range of tens of kilohertz. If we use the 3D mouse, we will have to determine its exact protocol on our own because we have been unable to find any documentation. However, it is a USB HID device, and we should be able to interface with it after some experimentation.

Division of Labor

We plan to divide labor primarily between the simulation and the display. Kade will focus on writing the DPU code and optimizing its performance, and Scott will focus on graphics and interfacing with the IMU and 3D mouse. However, we plan to collaborate on every aspect of the project, because we are both interested in every part of the project.

Example Implementation in C

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdint.h>

// 1 unit distance = 0.0596 ly
//           G = 1.559e-13 ly^3/(Ms*yr^2)
//           a0 = 1.261e-11 ly/yr^2
```

```
// If all particles have the same mass m, then the acceleration on a particle is
//
//     sqrt(G*m*a0) / r
//
// using modified Newtonian gravity

// Want total mass of simulation to be ~1e12 solar masses (10 000 particles => 100 million solar masses per particle)
//
// G*a0 = 1.966e-24 ly^4 / (Ms * yr^4)
//
// G*a0 * 1e8 Ms = 1.966e-16 ly^4 / yr^4
//
// sqrt(...) = 1.402e-8 ly^2 / yr^2  =  3.947e-6 u^2 / yr^2  ~  2^-18 u^2/yr^2
//
// so acceleration (in distance units / years^2) = (1/r) >> 18

// Time constant for system is ~1/sqrt(G*totalMass/length^3) = 2.5e9 years
// System can run at ~100 steps per second
// Want time constant to correspond to ~100 seconds
// => timestep ~ 250 000 years ~ 2^18
// Then 14 billion years corresponds to about 9 minutes, which sounds right
// We'll just use Δt = 2^18 years

// The numerical method we will use is Verlet
//
//     x <- 2*x - prev + (Σ(1/r) << 18)
//     prev <- x
//
// where the left shift by 18 corresponds to multiplication by 2^-18 * (2^18)^2 = sqrt(G*a0*m) * Δt^2

// Used in previous version, may come in handy
//   memset(&particles[0][idx].velocity, 0, sizeof(struct vector));

#define N 1280
#define WIDTH 1376
#define MAX_X 1365
```

```
#define MAX_Y 511
#define TRAILS 1

struct vector {long double x, y, z;};
struct particle {struct vector posn, prev;};

int main(void) {
	FILE* disp = fopen("/dev/fb0", "w");
	uint32_t (*disp_buffer) = malloc(WIDTH*(MAX_Y+1) * sizeof(uint32_t));

	fputs("\e[2J", stdout);

	struct particle (*particles)[N] = malloc(2*N*sizeof(struct particle));
	srand(42);
	for(unsigned int idx=0; idx<N; idx++){
	long double x = (1L<<31) + (rand()>>1) - (1L<<29); // 1, 29
	long double y = (1L<<31) + (rand()>>1) - (1L<<29); // 1, 29
	long double z = (1L<<31) + (rand()>>5) - (1L<<25); // 5, 25
	particles[0][idx].posn.x = x;
	particles[0][idx].posn.y = y;
	particles[0][idx].posn.z = z;
	particles[0][idx].prev.x = (y>(1L<<31) ? x+0x200000 : x-0x200000);
	particles[0][idx].prev.y = (x<(1L<<31) ? y+0x200000 : y-0x200000);
	particles[0][idx].prev.z = z + (rand()>>7) - (1L<<23);
	//particles[0][idx].prev.x = x + (rand()>>7) - (1L<<23); //(y>(1L<<31) ? x+0x200000 : x-0x200000);
	//particles[0][idx].prev.y = y + (rand()>>7) - (1L<<23); //(x<(1L<<31) ? y+0x200000 : y-0x200000);
	//particles[0][idx].prev.z = z + (rand()>>7) - (1L<<23);
	}

	struct vector (*forces) = malloc(N*(N-1)/2 * sizeof(struct vector));
	// index = (r**2 - r)/2 + c
	//   for N > r > c >= 0

	unsigned int timestep = 0;
	unsigned int s = 0;
	while (1){
```

```
for(unsigned int p1=1; p1<N; p1++){
	for(unsigned int p0=0; p0<p1; p0++){
	long double x0 = particles[s][p0].posn.x;
	long double y0 = particles[s][p0].posn.y;
	long double z0 = particles[s][p0].posn.z;

	long double x1 = particles[s][p1].posn.x;
	long double y1 = particles[s][p1].posn.y;
	long double z1 = particles[s][p1].posn.z;

	long double dx = x1-x0;
	long double dy = y1-y0;
	long double dz = z1-z0;

	// unit length vector with direction of r: (vec1 - vec2) * 1/r
	// vector with length 1/r: (vec1 - vec2) * 1/r^2
	long double r_inv_squared = 1.0/(dx*dx + dy*dy + dz*dz);

	unsigned int jdx = p1*(p1-1)/2 + p0;
	forces[jdx].x = dx*r_inv_squared;
	forces[jdx].y = dy*r_inv_squared;
	forces[jdx].z = dz*r_inv_squared;

	if(jdx == 0){
		printf("\x1B[35;1H%21s %21s %21s %21s\n", "dx", "dy", "dz", "1/r^2");
		printf("%21Le %21Le %21Le %21Le\n\n", dx, dy, dz, r_inv_squared);
	}
	}
}

unsigned int px=0;
for(unsigned int px_y=0; px_y<=MAX_Y; px_y++){
	for(unsigned int px_x=0; px_x<WIDTH; px_x++){
	if(((px_x==0 || px_x==511)&&(px_y<512)) || ((px_y==0 || px_y==511)&&(px_x<512))){
		disp_buffer[px] = 0x00FFFFFF;
	}
```

```
	else{
		disp_buffer[px] = TRAILS ? (disp_buffer[px]>>1) : 0;
	}
	px++;
	}
}

for(unsigned int p=0; p<N; p++){
	struct vector sum;
	sum.x = 0; sum.y = 0; sum.z = 0;
	for(unsigned int before=0; before<p; before++){
	unsigned int kdx = p*(p-1)/2 + before;
	sum.x -= forces[kdx].x;
	sum.y -= forces[kdx].y;
	sum.z -= forces[kdx].z;
	}
	for(unsigned int after=p+1; after<N; after++){
	unsigned int kdx = after*(after-1)/2 + p;
	sum.x += forces[kdx].x;
	sum.y += forces[kdx].y;
	sum.z += forces[kdx].z;
	}

	if(p == 0){
	printf("%21s %21s %21s\n", "sum.x", "sum.y", "sum.z");
	printf("%21Le %21Le %21Le\n\n", sum.x, sum.y, sum.z);
	}

	particles[!s][p].prev   = particles[s][p].posn;
	particles[!s][p].posn.x = particles[s][p].posn.x*2 - particles[s][p].prev.x + sum.x*262144.0*100000.0;
	particles[!s][p].posn.y = particles[s][p].posn.y*2 - particles[s][p].prev.y + sum.y*262144.0*100000.0;
	particles[!s][p].posn.z = particles[s][p].posn.z*2 - particles[s][p].prev.z + sum.z*262144.0*100000.0;

if(particles[!s][p].posn.x > 0 &&
	particles[!s][p].posn.x < (1L<<32) &&
	particles[!s][p].posn.y > 0 &&
```

```
            particles[!s][p].posn.y > 0 &&
            particles[!s][p].posn.z < (1L<<32) &&
            particles[!s][p].posn.z < (1L<<32)){
            unsigned int x_coord = (uint32_t)(particles[!s][p].posn.x) >> 23;
            unsigned int y_coord = 511 - ((uint32_t)(particles[!s][p].posn.y) >> 23);
            unsigned char intensity = ((uint32_t)(particles[!s][p].posn.z) >> 24);
            disp_buffer[y_coord*WIDTH + x_coord] = intensity + (intensity<<8) + (intensity<<16);
            }
        }
        rewind(disp);
        fwrite(disp_buffer, sizeof(uint32_t), WIDTH*(MAX_Y+1), disp);

        printf("%4u steps", timestep);
        fflush(stdout);
        s = !s;
        timestep++;
        }
        fclose(disp);
}
```