Exercise
Euler Particle System Simulation

Course Code: L3T2H1-57
Cell Ecosystem Solutions Enablement

Course Objectives

- **The student should get ideas of how to get in well-defined steps from scalar code to vectorized code for the PPU (VMX) to code for the SPU.**
- **Issues occurring during parallelization will be discussed and then applied to the example, an Euler particle system.**

Particle Simulation System

- This example shows a particle-system simulation using numerical integration techniques to animate a large set of particles. Numerical integration is implemented using Euler's method of integration. It computes the next value of a function of time, F(t), by incrementing the current value of the function by the product of the time step and the derivative of the function:
  - F(t + dt) = F(t) + dt*F'(t);
- The particle system consists of:
  - An array of 3-D positions for each particle (pos[ ])
  - An array of 3-D velocities for each particle (vel[ ])
  - An array of masses for each particle (mass[ ])
  - A force vector that varies over time (force)

Initial Scalar Code

```
#define END_OF_TIME 10
#define PARTICLES 100000
typedef struct {
    float x, y, z, w;
} vec4D;
vec4D pos[PARTICLES]; // particle positions
vec4D vel[PARTICLES]; // particle velocities
vec4D force; // current force being applied to the particles
float inv_mass[PARTICLES]; // inverse mass of the particles
float dt = 1.0f; // step in time
int main()
{
    int i;
    float time;
    float dt_inv_mass;
    // For each step in time
    for (time=0; time<END_OF_TIME; time += dt) {
        // For each particle
        for (i=0; i<PARTICLES; i++) {
        // Compute the new position and velocity as acted upon by the force f.
            pos[i].x = vel[i].x * dt + pos[i].x;
            pos[i].y = vel[i].y * dt + pos[i].y;
            pos[i].z = vel[i].z * dt + pos[i].z;
            dt_inv_mass = dt * inv_mass[i];
            vel[i].x = dt_inv_mass * force.x + vel[i].x;
            vel[i].y = dt_inv_mass * force.y + vel[i].y;
            vel[i].z = dt_inv_mass * force.z + vel[i].z;
        }
    }
    return (0);
}
```

Optimize the Code in Several Steps

1. SIMDize the Code for Execution on the PPE
   - **Where are the vectors?**
   - **Which data-structures are possible?**
   - **Which one would you chose and why?**
2. SIMDize the Code for Execution on the SPE
   - **What has to change?**
   - **How would tell the SPU what to do?**
   - **What impact has the size of the local store to the problem?**
3. Parallelize Code For Execution Across Multiple SPEs
   - **How can you extend step 2 to make full use of the complete cell ship, i.e. to use all SPEs?**

Step 1: SIMDize the Code for Execution on the PPE

- **Can the compiler do it (auto-SIMDization) ?**
  - only for simple code possible
- **Data-structures**
  - Array of structures: [x, y, z, 1]
  - Structure of arrays: [x1, x2, x3, x4], [y1, y2, y3, y4], …
- **VMX code (SIMD on PPU)**

SIMDization in Array of Structures Form for VMX

```
#define END_OF_TIME 10
#define PARTICLES 100000
typedef struct {
    float x, y, z, w;
} vec4D;
vec4D pos[PARTICLES] __attribute__ ((aligned (16)));
vec4D vel[PARTICLES] __attribute__ ((aligned (16)));
vec4D force __attribute__ ((aligned (16)));
float inv_mass[PARTICLES] __attribute__ ((aligned (16)));
float dt __attribute__ ((aligned (16))) = 1.0f;
int main()
{
   int i;
   float time;
   float dt_inv_mass __attribute__ ((aligned (16)));
   vector float dt_v, dt_inv_mass_v;
   vector float *pos_v, *vel_v, force_v;
   vector float zero = (vector float)(0.0f);
   pos_v = (vector float *)pos;
   vel_v = (vector float *)vel;
   force_v = *((vector float *)&force);
   // Replicate the variable time step across elements 0-2 of
   // a floating point vector. Force the last element (3) to
zero.
   dt_v = vec_sld(vec_splat(vec_lde(0, &dt), 0), zero, 4);
```

```
   // For each step in time
   for (time=0; time<END_OF_TIME; time += dt) {
      // For each particle
     for (i=0; i<PARTICLES; i++) {
     // Compute the new position and velocity as acted upon
by the force f.
      pos_v[i] = vec_madd(vel_v[i], dt_v, pos_v[i]);
      dt_inv_mass = dt * inv_mass[i];
      dt_inv_mass_v = vec_splat(vec_lde(0, &dt_inv_mass), 0);
      vel_v[i] = vec_madd(dt_inv_mass_v, force_v, vel_v[i]);
     }
   }
   return (0);
}
```

SIMDization in Structure of Arrays Form for VMX

```
#define END_OF_TIME 10
#define PARTICLES 100000
typedef struct {
    float x, y, z, w;
} vec4D;
// Separate arrays for each component of the vector.
vector float pos_x[PARTICLES/4],
  pos_y[PARTICLES/4],
  pos_z[PARTICLES/4];
vector float vel_x[PARTICLES/4],
  vel_y[PARTICLES/4],
  vel_z[PARTICLES/4];
vec4D force __attribute__ ((aligned (16)));
float inv_mass[PARTICLES] __attribute__ ((aligned (16)));
float dt = 1.0f;
int main()
{
    int i;
    float time;
    float dt_inv_mass __attribute__ ((aligned (16)));
    vector float force_v, force_x, force_y, force_z;
    vector float dt_v, dt_inv_mass_v;
    // Create a replicated vector for each
    // component of the force vector.
    force_v = *(vector float *)(&force);
    force_x = vec_splat(force_v, 0);
    force_y = vec_splat(force_v, 1);
    force_z = vec_splat(force_v, 2);
    // Replicate the variable time step across all
    // elements.
    dt_v = vec_splat(vec_lde(0, &dt), 0);
```

```
// For each step in time
    for (time=0; time<END_OF_TIME; time += dt) {
        // For each particle
        for (i=0; i<PARTICLES/4; i++) {
        // Compute the new position and
        //velocity as acted upon by the force f.
          pos_x[i] = vec_madd(vel_x[i], dt_v, pos_x[i]);
          pos_y[i] = vec_madd(vel_y[i], dt_v, pos_y[i]);
          pos_z[i] = vec_madd(vel_z[i], dt_v, pos_z[i]);
          dt_inv_mass = dt * inv_mass[i];
          dt_inv_mass_v = vec_splat(vec_lde(0, &dt_inv_mass), 0);
          vel_x[i] = vec_madd(dt_inv_mass_v, force_x, vel_x[i]);
          vel_y[i] = vec_madd(dt_inv_mass_v, force_y, vel_y[i]);
          vel_z[i] = vec_madd(dt_inv_mass_v, force_z, vel_z[i]);
        }
    }
    return (0);
}
```

Step 2: Port the PPE Code for Execution on the SPE

1. Creating an SPE thread of execution on the PPE
   – Initialization for the thread (context)
2. Migrating the computation loops from Vector/SIMD Multimedia Extension intrinsics to SPU
   – syntactic replacement (vec_ → spu_)
   – Mapping VMX → SPU (vmx2spu.h, vec_literal.h)
   – Partition data
   – add DMA’s to bring in data
3. Adding DMA transfers to move data in and out of the SPE's local store (LS)

PPU Code - particle.h

```
#define END_OF_TIME 10
#define PARTICLES 100000
typedef struct {
    float x, y, z, w;
} vec4D;
typedef struct {
    int particles; // number of particles to process
    vector float *pos_v; // pointer to array of position vectors
    vector float *vel_v; // pointer to array of velocity vectors
    float *inv_mass; // pointer to array of mass vectors
    vector float force_v; // force vector
    float dt; // current step in time
} context
```

Makefiles for PPU and SPU

PPU

```
PROGRAM_spu := euler_spe
DIRS := spu
IMPORTS := spu/lib_particle_spu.a -lspe
include $TOP/make.footer
```

SPU

```
PROGRAM spu := particle
LIBRARY_embed := lib_particle_spu.a
INCLUDE := -I ..
include $TOP/make.footer
```

PPU Code

```
#include <stdio.h>
#include <libspe.h>
#include "particle.h"
vec4D pos[PARTICLES] __attribute__ ((aligned (16)));
vec4D vel[PARTICLES] __attribute__ ((aligned (16)));
vec4D force __attribute__ ((aligned (16)));
float inv_mass[PARTICLES] __attribute__ ((aligned (16)));
float dt = 1.0f;
extern spe_program_handle_t particle;
int main()
{
   int status;
   speid_t spe_id;
   context ctx __attribute__ ((aligned (16)));
   ctx.particles = PARTICLES;
   ctx.pos_v = (vector float *)pos;
   ctx.vel_v = (vector float *)vel;
   ctx.force_v = *((vector float *)&force);
   ctx.inv_mass = inv_mass;
   ctx.dt = dt;
   // Create an SPE thread of execution passing the context as a parameter.
  spe_id = spe_create_thread(0, &particle, &ctx, NULL, -1, 0);
  if (spe_id) {
     // Wait for the SPE to finish
    (void)spe_wait(spe_id, &status, 0);
  } else {
    perror("Unable to create SPE thread");
    return (1);
  }
  return (0);
}
```

SPE Code

```
#include <spu_intrinsics.h>
#include <cbe_mfc.h>
#include "particle.h"
#define PARTICLES_PER_BLOCK 1024
// Local store structures and buffers.
volatile context ctx;
volatile vector float pos[PARTICLES_PER_BLOCK];
volatile vector float vel[PARTICLES_PER_BLOCK];
volatile float inv_mass[PARTICLES_PER_BLOCK];
int main(unsigned long long spe_id, unsigned long long parm)
{
   int i, j;
   int left, cnt;
   float time;
   unsigned int tag_id = 0;
   vector float dt_v, dt_inv_mass_v;
   spu_writech(MFC_WrTagMask, -1);

// Input parameter parm is a pointer to the particle context.
   // Fetch the context, waiting for it to complete.
   spu_mfcdma32((void *)(&ctx), (unsigned int)parm,
                     sizeof(context), tag_id,
     MFC_GET_CMD);
   (void)spu_mfcstat(2);
   dt_v = spu_splats(ctx.dt);
   // For each step in time
for (time=0; time<END_OF_TIME; time += ctx.dt) {
     // For each block of particles
     for (i=0; i<ctx.particles; i+=PARTICLES_PER_BLOCK) {
       // Determine the number of particles in this block.
       left = ctx.particles - i;
       cnt = (left < PARTICLES_PER_BLOCK) ? left : PARTICLES_PER_BLOCK;
       // Fetch the data - position, velocity,
       // inverse_mass. Wait for DMA to complete
       // before performing computation.
       spu_mfcdma32((void *)(pos), (unsigned int)(ctx.pos_v+i),
                  cnt * sizeof(vector float), tag_id, MFC_GET_CMD);
       spu_mfcdma32((void *)(vel), (unsigned int)(ctx.vel_v+i),
                  cnt * sizeof(vector float), tag_id, MFC_GET_CMD);
       spu_mfcdma32((void *)(inv_mass), (unsigned int)(ctx.inv_mass+i),
                  cnt * sizeof(float), tag_id, MFC_GET_CMD);
      (void)spu_mfcstat(2);
      // Compute the step in time for the block of particles
      for (j=0; j<cnt; j++) {
        pos[j] = spu_madd(vel[j], dt_v, pos[j]);
        dt_inv_mass_v = spu_mul(dt_v, spu_splats(inv_mass[j]));
        vel[j] = spu_madd(dt_inv_mass_v, ctx.force_v, vel[j]);
      }
      // Put the position and velocity data back into main storage
      spu_mfcdma32((void *)(pos), (unsigned int)(ctx.pos_v+i),
            cnt * sizeof(vector float), tag_id, MFC_PUT_CMD);
      spu_mfcdma32((void *)(vel), (unsigned int)(ctx.vel_v+i),
            cnt * sizeof(vector float), tag_id, MFC_PUT_CMD);
      }
   }
   (void)spu_mfcstat(2); // wait for DMA
   return (0);
}
```

Step 3: Parallelize Code For Execution Across Multiple SPEs

- **Most intuitive approach: Partition data**
  - problem when there are data dependencies

PPE Code

```
#include <stdio.h>
#include <libspe.h>
#include "particle.h"
#define SPE_THREADS 7
vec4D pos[PARTICLES] __attribute__ ((aligned (16)));
vec4D vel[PARTICLES] __attribute__ ((aligned (16)));
vec4D force __attribute__ ((aligned (16)));
float inv_mass[PARTICLES] __attribute__ ((aligned (16)));
float dt = 1.0f;
extern spe_program_handle_t particle;
int main()
{
   int i, offset, count;
   int status;
   speid_t spe_ids[SPE_THREADS];
   context ctxs[SPE_THREADS] __attribute__ ((aligned (16)));
   // Construct a context and thread for each
   // SPE thread. Make sure
   // that each SPE's (excluding the last)
   // particle count is a multiple
   // of 4 so that inv_mass context pointer
   // is always quadword aligned.
   for (i=0, offset=0; i<SPE_THREADS; i++, offset+=count) {
      count = (PARTICLES / SPE_THREADS + 3) & ~3;
      ctxs[i].particles =
       (i==(SPE_THREADS-1)) ? PARTICLES - offset : count;
     ctxs[i].pos_v = (vector float *)&pos[offset];
     ctxs[i].vel_v = (vector float *)&vel[offset];
     ctxs[i].force_v = *((vector float *)&force);
     ctxs[i].inv_mass = &inv_mass[offset];
     ctxs[i].dt = dt;
     // Create an SPE thread of execution passing
     // the context as a parameter.
    spe_ids[i] = spe_create_thread(0,
                 &particle, &ctxs[i], NULL, -1, 0);
    if (spe_ids[i] == -1) {
       perror("Unable to create SPE thread");
       return (1);
    }
   }
  // Wait for all the SPEs to complete.
  for (i=0; i<SPE_THREADS; i++) {
     (void)spe_wait(spe_ids[i], &status, 0);
   }
   return (0);
}
```

IBM Microelectronics Division
1580 Route 52, Bldg. 504
Hopewell Junction, NY 12533-6351

The IBM home page is http://www.ibm.com
The IBM Microelectronics Division home page is
http://www.chips.ibm.com