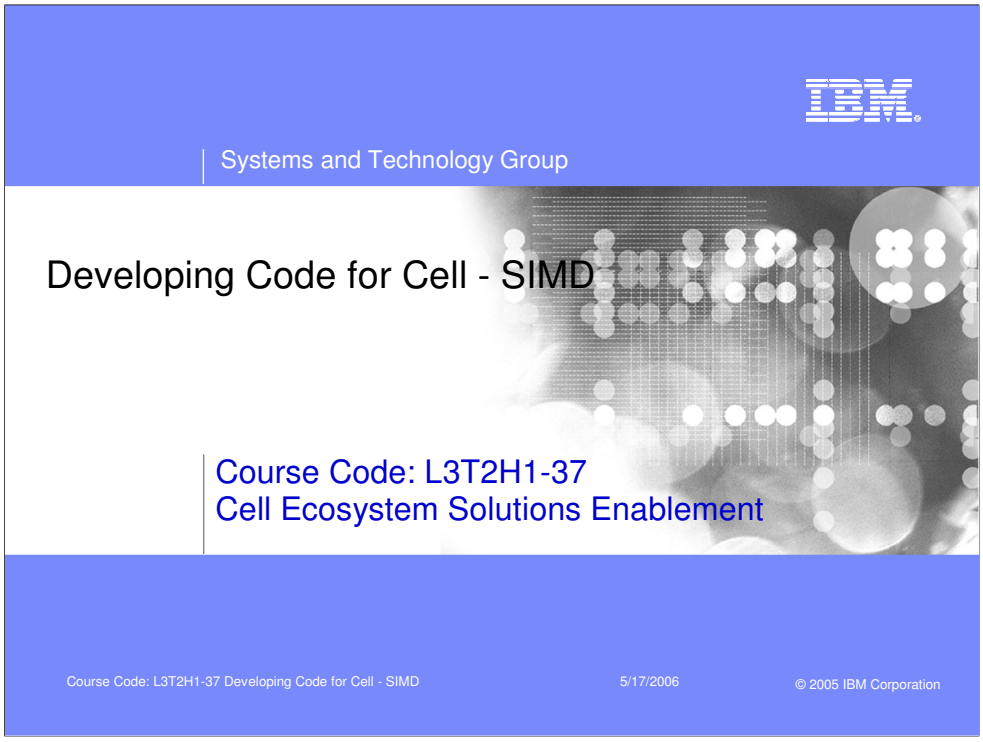

IBM
Systems and Technology Group
Developing Code for Cell - SIMD
Course Code: L3T2H1-37
Cell Ecosystem Solutions Enablement
Course Code: L3T2H1-37 Developing Code for Cell - SIMD
5/17/2006
© 2005 IBM Corporation

Class Objectives – Things you will learn

- SIMD Support In PPE and SPE
- The differences between PPE and SPE in SIMD support in terms of architectural and language extensions
- SIMD instructions and vector data types
- Examples of SIMD instructions
- SIMD programming: native vs. traditional
- Language extensions to support SIMD
- Vectorization and examples of vector operations
- Techniques used to port code from PPE to SPE
- Walk through the Euler particle-system simulation program

Class Agenda

- **SIMD Support In PPE and SPE**
  - SIMD Architecture
  - Differences Between PPE and SPE SIMD Support
  - PPE and SPE SIMD-Support Comparison
- **SIMD Instructions**
  - Language-Extension Differences - PPE versus SPU Vector Data Types
  - A SIMD Instruction Example
  - SIMD “Cross-Element” Instructions
  - Shuffle / Permute – A Simple Example
- **SIMD Programming**
  - C/C++ Extensions to Support SIMD
  - Vectorizing a Loop – A Simple Example
  - Vectorization
- **Porting SIMD Code from the PPE to the SPEs**
  - Simple Macro Translation
  - Code-Mapping Considerations
  - Use Intrinsics that Map One-to-One
- **Euler Particle-System Simulation**

SIMD Support In PPE and SPE

SIMD Architecture

- **SIMD = "single-instruction multiple-data"**
- **SIMD exploits data-level parallelism**
  - a single instruction can apply the same operation to multiple data elements in parallel
- **SIMD units employ "vector registers"**
  - each register holds multiple data elements
- **SIMD is pervasive in the BE**
  - PPE includes VMX (SIMD extensions to PPC architecture)
  - SPE is a native SIMD architecture (VMX-like)
- **SIMD in VMX and SPE**
  - 128bit-wide datapath
  - 128bit-wide registers
  - 4-wide fullwords, 8-wide halfwords, 16-wide bytes
  - SPE includes support for 2-wide doublewords

Differences Between PPE and SPE SIMD Support

- ***Architectural Differences***
  - The PPE processes SIMD operations in the VXU within its PPU. The operations are those of the Vector/SIMD Multimedia Extension instruction set.
  - The SPEs process SIMD operations in their SPU. The operations are those of the SPU instruction set.
- ***Language-Extension Differences***
  - The SPE's SPU instruction set is similar to that of the PPE's Vector/SIMD Multimedia Extension instruction set, in that both operate on 128-bit SIMD vectors. However, from a programmer's perspective, these instruction sets are quite different, and their respective language extensions have different intrinsics and data types.

PPE and SPE SIMD-Support Comparison

Feature	PPE	SPE
Number of processing elements	1	8
Modes supported	user and supervisor	user only
Number of SIMD registers	32 (128-bit)	128 (128-bit)
Organization of register files	separate fixed-point, floating-point, and SIMD registers	unified SIMD registers
Load latency	variable (cached)	fixed
Addressability	2^{64}-byte main storage	256-KB LS 2^{64}-byte main storage via DMA
Memory architecture	2-level caching	software-controlled LS
SIMD instruction set	general SIMD, supported by PowerPC scalar and control instructions	SIMD only, optimized for single-precision floating-point, 16-bit fixed-point, and 32-bit fixed-point
Single-precision floating-point SIMD	IEEE 754-1985 and SPE-compatible graphics-rounding mode supported	extended range[1]
Double-precision floating-point SIMD	not supported	IEEE 754-1985 supported
Doubleword fixed-point SIMD	not supported	supported
Inter-element communication facility	MMIO	channel

SIMD Instructions

Language-Extension Differences - PPE versus SPU Vector Data Types

Vector Data Type	PPE	SPU
vector unsigned char	x	x
vector signed char	x	x
vector bool char	x	—
vector unsigned short	x	x
vector signed short	x	x
vector bool short	x	—
vector pixel	x	—
vector unsigned int	x	x
vector signed int	x	x
vector bool int	x	—
vector float	x	x
vector unsigned long long	—	x
vector signed long long	—	x
vector double	—	x

The key differences are:

- Only the Vector/SIMD Multimedia Extension instruction set supports pixel vectors.
- Only the SPU instruction set supports doubleword vectors.

Language-Extension Differences - PPE versus SPU Vector Data Types (Cont'd)

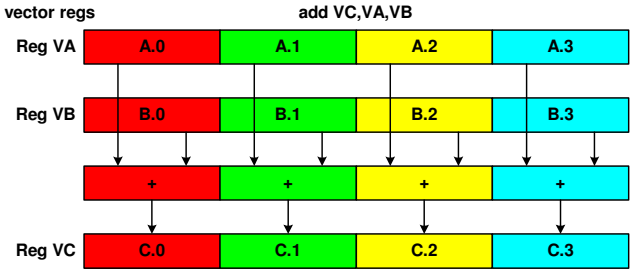
Vector/SIMD Multimedia Extension instruction not directly supported by a single instruction in the SPU instruction set:

- Saturating math
- Sum-across
- Log2 and 2x
- Ceiling and floor
- Complete byte instructions

SPU instruction set not directly supported by a single instruction in the Vector/SIMD Multimedia Extension instruction set:

- Immediate operands
- Double-precision floating-point
- Sum of absolute difference
- Count ones in bytes
- Count leading zeros
- Equivalence

A SIMD Instruction Example

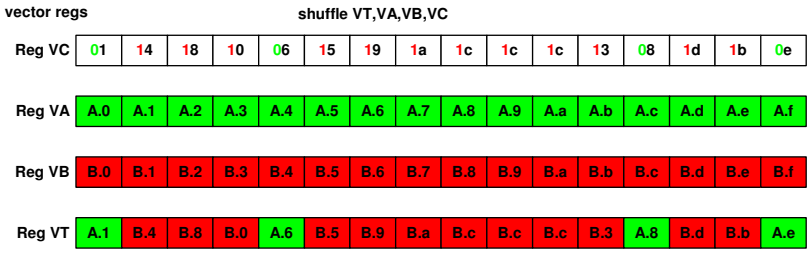


- **Example is a 4-wide add**
  - each of the 4 elements in reg VA is added to the corresponding element in reg VB
  - the 4 results are placed in the appropriate slots in reg VC

SIMD “Cross-Element” Instructions

- **VMX and SPE architectures include “cross-element” instructions**
  - shifts and rotates
  - permutes / shuffles
- **Permute / Shuffle**
  - selects bytes from two source registers and places selected bytes in a target register
  - byte selection and placement controlled by a “control vector” in a third source register
    - extremely useful for reorganizing data in the vector register file

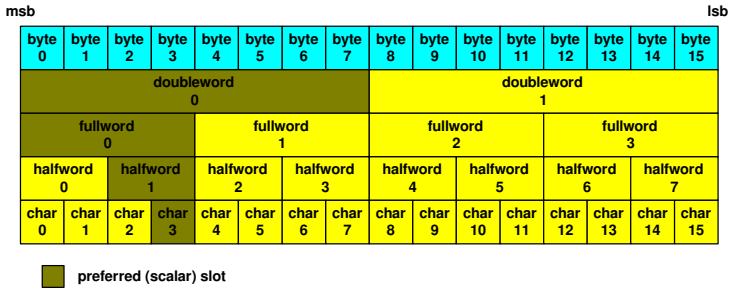
Shuffle / Permute – A Simple Example



- Bytes selected from regs VA and VB based on byte entries in control vector VC
- Control vector entries are indices of bytes in the 32-byte concatenation of VA and VB
- Operation is purely byte oriented
- SPE has extended forms of the shuffle / permute operation

Scalar Overlay on SIMD in SPE

- **Scalar operations use "preferred slot" in the 128-bit vector registers**
- **Rotate instructions used to move data into the appropriate locations**
  - generally handled by the compiler



SIMD Programming

SIMD Programming

- **"Native SIMD" programming**
  - algorithm vectorized by the programmer
  - coding in high-level language (e.g. C, C++) using intrinsics
  - intrinsics provide access to SIMD assembler instructions
    - e.g. c = spu_add(a,b) → add vc,va,vb
- **"Traditional" programming**
  - algorithm coded "normally" in scalar form
  - compiler does auto-vectorization
  - ➢ but auto-vectorization capabilities remain limited

C/C++ Extensions to Support SIMD

- **Vector datatypes**
  - e.g. “vector float”, “vector signed short”, “vector unsigned int”, …
  - SIMD width per datatype is implicit in vector datatype definition
  - vectors aligned on quadword (16B) boundaries
  - casts from one vector type to another in the usual way
  - casts between vector and scalar datatypes not permitted
- **Vector pointers**
  - e.g. “vector float *p”
  - p+1 points to the next vector (16B) after that pointed to by p
  - casts between scalar and vector pointer types
- **Access to SIMD instructions is via intrinsic functions**
  - similar intrinsics for both SPU and VMX
  - translation from function to instruction dependent on datatype of arguments
  - e.g. spu_add(a,b) can translate to a floating add, a signed or unsigned int add, a signed or unsigned short add, etc.

C/C++ Extensions to Support SIMD (Cont'd)

- **Vector declaration**
  - Vector signed char a={1, 2, 3, 4, 5, 6, 7, 8}
- **Operating on vectors**
  - c = spu_add(a,b);
  - c = spu_mul(a,b);
  - c = spu_sub(a,b);
  - c = spu_avg(a,b);
  - . . .

Vectorizing a Loop – A Simple Example

```
/*  scalar version  */

int mult1(float *in1, float *in2, float *out, int N)
{
  int i;

  for (i = 0, i < N, i++)
  {
        out[i] = in1[i] * in2[i];
  }

  return 0;
```

```
/*  vectorized version   */

int vmult1(float *in1, float *in2, float *out, int N)
{
  /*  assumes that arrays are quadword aligned  */
  /*  assumes that N is divisible by 4  */

  int i, Nv;
  Nv = N >> 2;                  /* N/4 vectors  */
  vector float *vin1 = (vector float *) (in1);
  vector float *vin2 = (vector float *) (in2);
  vector float *vout = (vector float *) (out);

  for (i = 0, i < Nv, i++)
  {
        vout[i] = spu_mul(vin1[i],vin2[i]);
  }

  return 0;
```

- **Loop does term-by-term multiply of two arrays**
  - arrays assumed here to remain scalar outside of subroutine
- **If arrays not quadword-aligned, extra work is necessary**
- **If array size is not a multiple of 4, extra work is necessary**

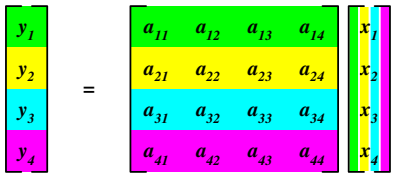
Vectorization

- **For any given algorithm, vectorization can usually be applied in several different ways**
- **Example: 4-dim. linear transformation (4x4 matrix times a 4-vector) in a 4-wide SIMD**

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

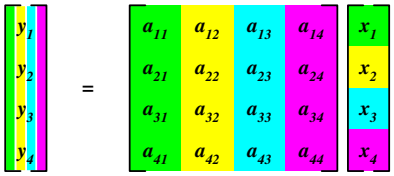
- **Consider two possible approaches:**
  - dot product: each row times the vector
  - sum of vectors: each column times a vector element
- **Performance of different approaches can be VERY different**

Vectorization Example – Dot-Product Approach



- **Assume:**
  - each row of the matrix is in a vector register
  - the x-vector is in a vector register
  - the y-vector is placed in a vector register
- **Process – for each element in the result vector:**
  - multiply the row register by the x-vector register
  - perform vector reduction on the product (sum the 4 terms in the product register)
  - place the result of the reduction in the appropriate slot in the result vector register

Vectorization Example – Sum-of-Vectors Approach



- **Assume:**
  - each column of the matrix is in a vector register
  - the x-vector is in a vector register
  - the y-vector is placed in a vector register (initialized to zero)
- **Process – for each element in the input vector:**
  - copy the element into all four slots of a register (“splat”)
  - multiply the column register by the register with the “splatted” element and add to the result register

Vectorization Trade-offs

- **Choice of vectorization technique will depend on many factors, including:**
  - organization of data arrays
  - what is available in the instruction-set architecture
  - opportunities for instruction-level parallelism
  - opportunities for loop unrolling and software pipelining
  - nature of dependencies between operations
  - pipeline latencies

Porting SIMD Code from the PPE to the SPEs

Porting SIMD Code from the PPE to the SPEs

General approach

- Write the SIMD programs first for the PPE, then porting them to the SPEs.

➔ This approach postpones some SPE-related considerations of dealing with the local store (LS) size, data movements, and debug until after the port.

➔ The approach can also allow partitioning of the work into simpler (perhaps more digestible) steps on the SPEs.

- After the Vector/SIMD Multimedia Extension code is working properly on the PPE, a strategy for parallelizing the algorithm across multiple SPEs can be developed.
- Convert from Vector/SIMD Multimedia Extension intrinsics to SPU intrinsics, adding data-transfer and synchronization constructs, and tuning for performance.
- Test the impact of various techniques, such as DMA double buffering, loop unrolling, branch elimination, alternative intrinsics, number of SPEs, and so forth.
- Use debugging tools such as the static timing-analysis tool and the IBM Full System Simulator for the Cell Broadband Engine to assist this effort,
- Alternatively, experienced Cell Broadband Engine programmers may prefer to skip the Vector/SIMD Multimedia Extension coding phase and go directly to SPU programming.

➔ In some cases, SIMD programming can be easier on an SPE than the PPE because of the SPE's unified register file.

Code-Mapping Considerations

Challenges associated with mapping code designed for one instruction set and compiled for another instruction set

- Performance
- Unmappable constructs
- Limited size of LS
- Equivalent precision

Performance

Simple remapping of low-level intrinsics can result in less-than-optimal performance, depending upon the intrinsics used.

Understanding the dynamic range of the remapping's operands can reduce the performance impact of simple remapping.

Unmappable Constructs

Differences in the processing of intrinsics make simple translation of certain intrinsics unmappable The unmappable SPU intrinsics include:

- stop and stopd
- conditional halt
- interrupt enable and disable
- move to and from status control and special-purpose registers
- channel instructions
- branch on external data

Code-Mapping Considerations (Cont'd)

Limited Size of LS

Vector/SIMD Multimedia Extension programs mapped to SPU programs might not fit within the LS of the SPE, either because

- The program is initially too big
- Mapping expands the code.

Equivalent Precision

The SPU instruction set does not fully implement the IEEE 754 single-precision floating-point standard (default rounding mode is r*ound to zero*, denormals are treated as zero, and there are no infinities or NaNs).

Floating-point results on an SPE may differ slightly from floatingpoint results using the PPE's PowerPC instruction set. In addition, all estimation intrinsics (for example, ceiling, floor, reciprocal estimate, reciprocal square root estimate, exponent estimate, and log estimate) do not have equivalent accuracy on the SPU and PPE PowerPC instruction sets.

The instructions in the PPE's Vector/SIMD Multimedia Extension have a *graphics rounding mode* (enabled by default) that allows programs written with Vector/SIMD Multimedia Extension instructions to produce floating-point results that are equivalent in precision to those written in the SPU instruction set. In this Vector/SIMD Multimedia Extension mode, as in the SPU environment, the default rounding mode is r*ound to zero*, denormals are treated as zero, and there are no infinities or NaNs.

Simple Macro Translation

Use a simple macro translation strategy for developing code that is portable between the Vector/SIMD Multimedia Extension and SPU instruction sets.

• *Use a Compatible Vector-Literal Construction Format*—The PPE Vector/SIMD Multimedia Extension and the SPE's SPU instruction set support two styles of constructing literal vectors: curly brace and parenthesis. Most compilers support both styles. A set of construction macros can be used to insulate programs from any differences in the tools.

• *Use Single-Token Vector Data Types*—The *SPU C/C++ Language Extensions* document specifies a set of single-token vector data types. Because these are single-token, the data types can be easily redefined by a preprocessor to the desired target processor. Additional single-token data types must be standardized for the unique Vector/SIMD Multimedia Extension data types.

Vector Data Type	Single-Token Data Type
vector bool char	vec_bchar16
vector bool short	vec_bshort8
vector bool int	vec_bint4
vector pixel	vec_pixel8

Use Intrinsics that Map One-to-One

Use Intrinsics that Map One-to-One

Regardless of the technique used to provide portability, performance will be optimized if the operations map one-to-one between Vector/SIMD Multimedia Extension intrinsics and SPU intrinsics.

- The SPU intrinsics that map one-to-one with Vector/SIMD Multimedia Extension
- The Vector/SIMD Multimedia Extension intrinsics that map one-to-one with SPU

SPU Intrinsics with One-to-One Vector/SIMD Multimedia Extension Mapping

SPU Intrinsic	Vector/SIMD Multimedia Extension Intrinsic	For Data Types
spu_add	vec_add	vector operands only, no scalar operands
spu_genc	vec_addc	all
spu_and	vec_and	vector operands only, no scalar operands
spu_andc	vec_andc	all

SPU Intrinsics with One-to-One Vector/SIMD Multimedia Extension Mapping

SPU Intrinsic	Vector/SIMD Multimedia Extension Intrinsic	For Data Types
spu_avg	vec_avg	all
spu_cmpeq	vec_cmpeq	vector operands only, no scalar operands
spu_cmpgt	vec_cmpgt	vector operands only, no scalar operands
spu_convtf	vec_ctf	limited scale range (5 bits)
spu_convts	vec_cts	limited scale range (5 bits)
spu_convtu	vec_ctu	limited scale range (5 bits)
spu_madd	vec_madd	float only
spu_mulhh	vec_mule	all
spu_mulo	vec_mulo	halfword vector operands only, no scalar operands
spu_nmsub	vec_nmsub	float only
spu_nor	vec_nor	all
spu_or	vec_or	vector operands only, no scalar operands
spu_re	vec_re	all
spu_rl	vec_rl	vector operands only, no scalar operands
spu_rsqrte	vec_rsqrte	all
spu_sel	vec_sel	all
spu_sub	vec_sub	vector operands only, no scalar operands
spu_genb	vec_subc	vector operands only, no scalar operands
spu_xor	vec_xor	vector operands only, no scalar operands

Vector/SIMD Multimedia Extension Intrinsics with One-to-One SPU Mapping

Vector/SIMD Multimedia Extension Intrinsic	SPU Intrinsic	For Data Types
vec_add	spu_add	halfwords, words, and floats only (not bytes)
vec_addc	spu_genc	all
vec_and	spu_and	all
vec_andc	spu_andc	all
vec_avg	spu_avg	unsigned chars only
vec_cmpeq	spu_cmpeq	all
vec_cmpgt	spu_cmpgt	all
vec_ctf	spu_convtf	all
vec_cts	spu_convts	all
vec_ctu	spu_convtu	all
vec_madd	spu_madd	all
vec_mulo	spu_mulo	halfwords only (not bytes)

Vector/SIMD Multimedia Extension Intrinsics with One-to-One SPU Mapping (Cont'd)

Vector/SIMD Multimedia Extension Intrinsic	SPU Intrinsic	For Data Types
vec_nmsub	spu_nmsub	all
vec_nor	spu_nor	all
vec_or	spu_or	all
vec_re	spu_re	all
vec_rl	spu_rl	halfwords and words only (not bytes)
vec_rsqrte	spu_rsqrte	all
vec_sel	spu_sel	all
vec_sub	spu_sub	halfwords, words, and floats only
vec_subc	spu_genb	all
vec_xor	spu_xor	all

Euler Particle-System Simulation

Euler Particle-System Simulation

The Euler-based particle-system simulation program illustrates the following steps involved in coding for the Cell Broadband Engine:

1. Transform scalar code to vector code (SIMDize) for execution on the PPE's VXU.
2. Port the code for execution on the SPE's SPU unit.
3. Parallelize the code for execution across multiple SPEs.

The Euler algorithm

- Use Euler's method of integration techniques to animate a large set of particles.
- 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 simple 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)

C implementation of the Euler algorithm

The following code shows a C implementation of the Euler algorithm, implemented for a uniprocessor using scalar data. There are no intrinsics calls in this listing.

```
#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);
}
```

Step 1: SIMDize the Code for Execution on the PPE

Strategies for SIMDizing code for execution either on the PPE's VXU or on an SPE's SPU unit depend upon

- The type of data being operated on
- The interdependencies of the data computations

• *Let the Compiler Do It*—Work effectively for some code samples (like this simple example), but it tends to be unsuccessful for more complicated code. Results will vary depending upon the algorithm, the language the code is expressed in, coding style, and capabilities of the compiler.

• *Array-of-Structures (AOS) Form*—Most common technique when the input data is naturally expressed as a vector (also call vector-across form). 3-D graphic applications express geometry as 3-component or 4-component vectors. These components naturally fit within a 4-component, single-precision floating-point vector.

• *Structure-of-Arrays (SOA) Form*—In this form, you collect the individual elements of the natural vectors into separate arrays (also called parallel-array form). The code is then written as if it were to execute scalar instructions, but it will be executing SIMD instructions. This results in code that computes four single-precision floats results simultaneously.

• *Hybrid Forms*—Often it is important that the input vector format remain unchanged. But SOA solutions are easier to code and more efficient than the AOS solutions. In this case, one can:

– Input the data in its natural, AOS form.

– Transform each data element on the fly into SOA form, using either the vec_perm (Vector/ SIMD Multimedia Extension) or the spu_shuffle (SPU) intrinsic.

– Perform computation using the SOA technique.

– Translate each output back into its natural, AOS form.

➔ Assuming the compiler auto-SIMDization is either unavailable or ineffective, you must adjust the data structures for efficient SIMD access. This decision cannot be made without also considering the SPE data-accessing method and the data-parallelization method. In addition, data should be aligned or padded for efficient quadword accesses, using the aligned attribute.

Step 1a: SIMDize in Array-of-Structures Form for Vector/SIMD Multimedia Extension

```
#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);
}
```

Step 1b: SIMDize in Structure-of-Arrays Form for Vector/SIMD Multimedia Extension

How to SIMDize in the SOA form, assuming that the number of particles is a multiple of 4.

```
#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

This step includes

1. Creating an SPE thread of execution on the PPE
2. Migrating the computation loops from Vector/SIMD Multimedia Extension intrinsics to SPU intrinsics
3. Adding DMA transfers to move data in and out of the SPE's local store (LS)

➔ Assuming the particle data structures cannot be restructured into SOA form, therefore, we use Step 1a. SPU intrinsics are used, and identified by their prefix, “spu_”.

Moving the code from the PPE to the SPE requires

- Creating a control-structure, called *context*, that defines the parameters to be computed on the SPE. This includes pointers to the particle array data, current force information, and so forth. The pointer to the context control-structure defined in the PPE is passed to the SPE thread by using the parameter passing mechanism in spe_create_thread. Alternatively, this information could have been passed via the mailbox.
- Porting the computation for execution on the SPE. The complexity of this operation depends upon the types of data and types of intrinsics used. For this case, some of the intrinsics only require a simple name translation (for example, vec_madd to spu_madd). The translation of the scalar values is a little more extensive.
- Adding an additional looping construct to partition the data arrays into smaller blocks. This is required because all the data does not fit within the SPE's local store.
- Adding DMA transfers to move data in and out of the SPE's local store.

Particle.h

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;
```

PPE Makefile

PPE Makefile:

```
############################################################
##########
# Subdirectories
############################################################
##########
DIRS := spu
############################################################
##########
# Target
############################################################
##########
PROGRAM_ppu := euler_spe
############################################################
##########
# Local Defines
############################################################
##########
IMPORTS := spu/lib_particle_spu.a -lspe
############################################################
##########
# make.footer
############################################################
##########
include ../../../../../make.footer
```

PPE 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);
}
}
// Wait for final DMAs to complete before terminating SPE thread.
(void)spu_mfcstat(2);
return (0);
}
}
}
// Wait for all the SPEs to complete.
for (i=0; i<SPE_THREADS; i++) {
(void)spe_wait(spe_ids[i], &status, 0);
}
return (0);
}
```

Step 3: Parallelize Code For Execution Across Multiple SPEs

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);
```

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