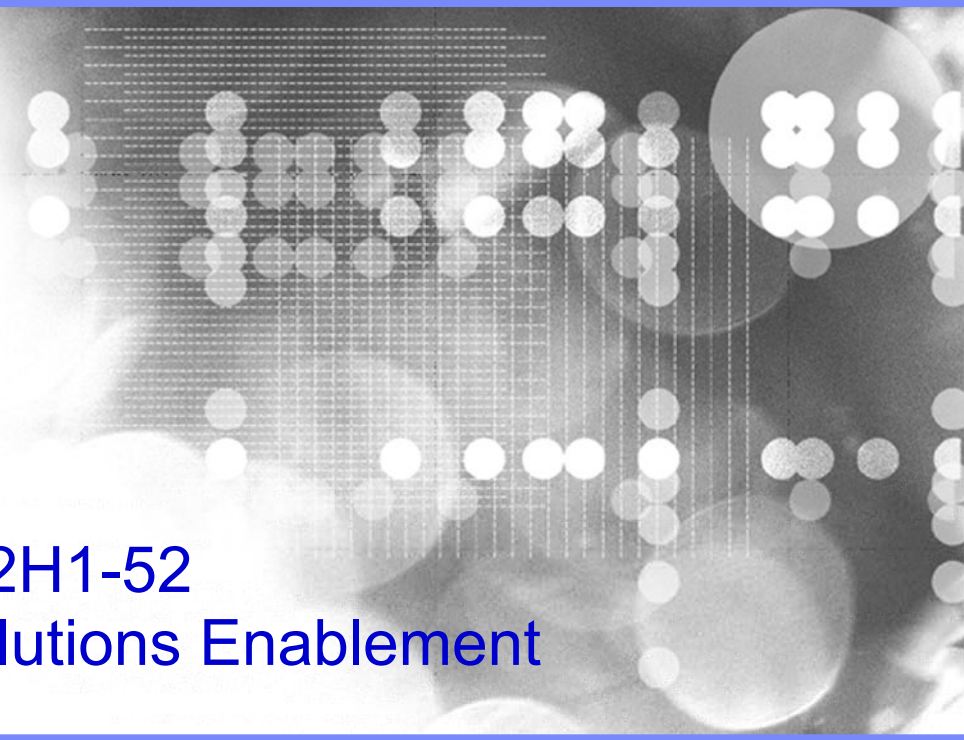


Vector Programming and Application Partitioning for the Cell BE

Course Code: L3T2H1-52
Cell Ecosystem Solutions Enablement

Course Objectives

- **Learn how to vectorize to exploit the power of Cell BE**
- **How to partition an application based on PPE or SPE resource requirements**
- **How to use intrinsics**

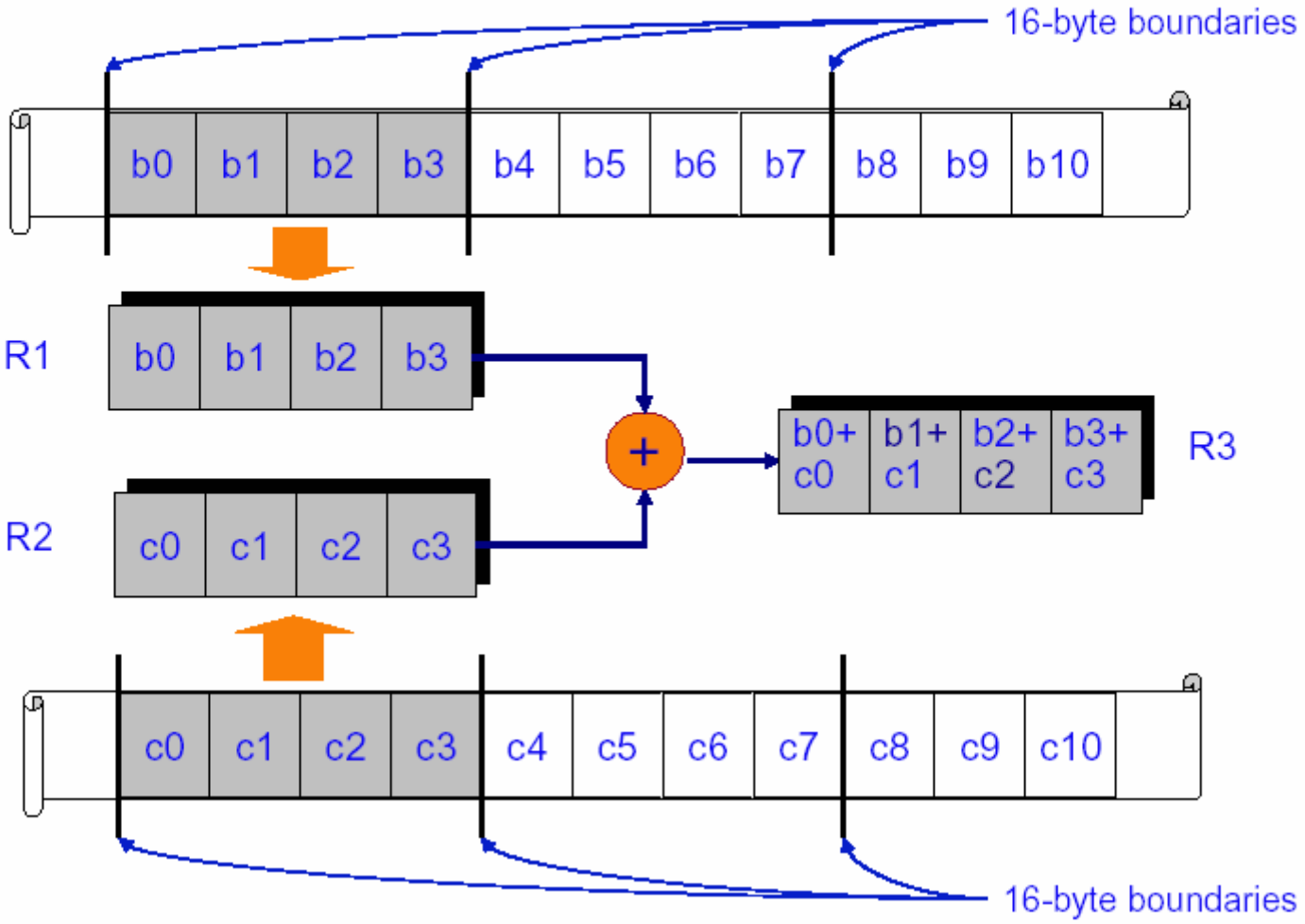
Course Agenda

- **Vector Programming (SIMD)**
  - Data types for vector programming
  - Application partitioning
- **SPU Intrinsics**

Vector Programming

Single Instruction Multiple Data (SIMD) Computation

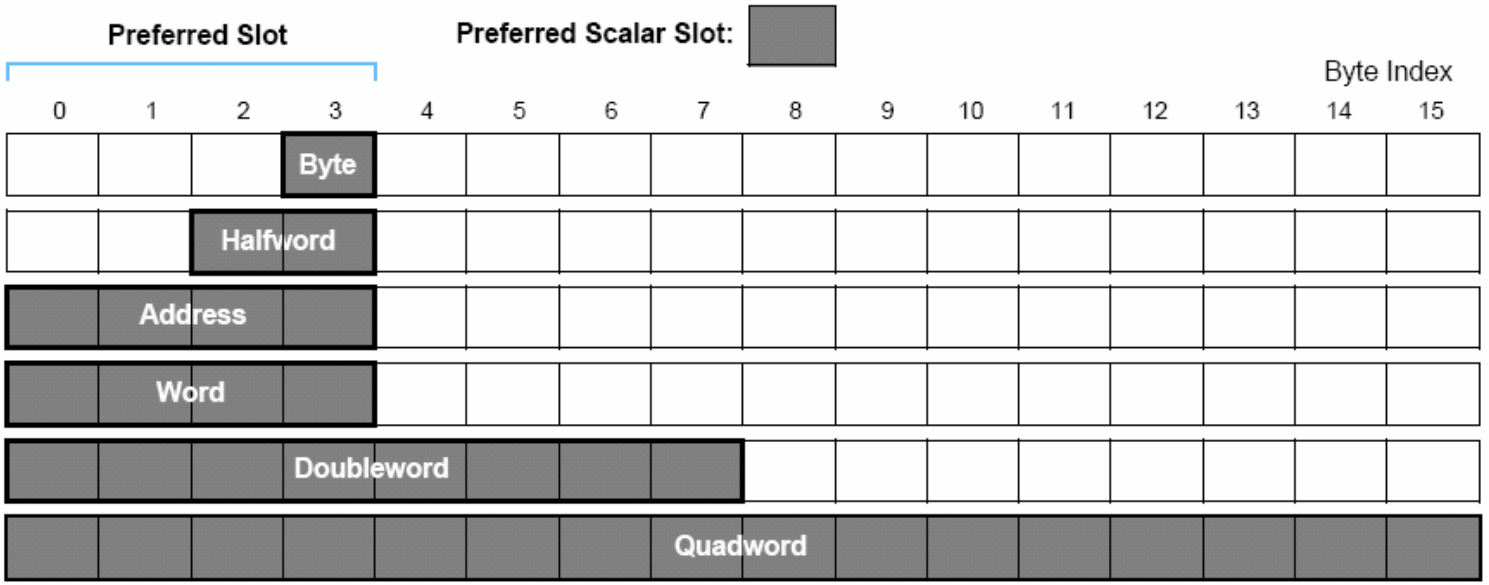
Process multiple "b[i]+c[i]" data per operations



C/C++ Extensions to Support SIMD

- **Vector datatypes**
  - vector [unsigned] {char, short, float, double}
    - e.g. “vector float”, “vector signed short”, “vector unsigned int”, …
  - SIMD width per datatype is implicit in vector datatype definition
  - casts from one vector type to another in the usual way
  - vectors aligned on quadword (16B) boundaries
- **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

Fitting SIMD Data Types into Registers



msb lsb

byte 0	byte 1	byte 2	byte 3	byte 4	byte 5	byte 6	byte 7	byte 8	byte 9	byte 10	byte 11	byte 12	byte 13	byte 14	byte 15
doubleword 0								doubleword 1							
word 0				word 1				word 2				word 3			
halfword 0		halfword 1		halfword 2		halfword 3		halfword 4		halfword 5		halfword 6		halfword 7	
char 0	char 1	char 2	char 3	char 4	char 5	char 6	char 7	char 8	char 9	char 10	char 11	char 12	char 13	char 14	char 15

Types of Intrinsics

- The C/C++ language extension intrinsics are grouped into the following three classes:
  - *Specific Intrinsics*
    - Intrinsics that have a one-to-one mapping with a single assembly-language instruction. Programmers rarely need the specific intrinsics for implementing inline assembly code because most instructions are accessible through generic intrinsics.
    - Prefixed by si_ , e.g. si_stop
  - *Generic Intrinsics and Built-Ins*
    - Intrinsics that map to one of several assembly-language instructions or instruction sequences, depending on the type of operands.
    - Prefixed by spu_ , e.g. spu_add
  - *Composite Intrinsics*
    - Convenience functions that map to assembly-language instruction sequences. A composite intrinsic can be expressed as a sequence of generic intrinsics.

Generic SPU Intrinsics

- Generic / Buit-In
  - Constant formation (spu_splats)
  - Conversion (spu_convtf, spu_convts, ..)
  - Arithmetic (spu_add, spu_madd, spu_nmadd, ...)
  - Byte operations (spu_absd, spu_avg,...)
  - Compare and branch (spu_cmpeq, spu_cmpgt,...)
  - Bits and masks (spu_shuffle, spu_sel,...)
  - Logical (spu_and, spu_or, ...)
  - Shift and rotate (spu_rlqwbyte, spu_rlqw,...)
  - Control (spu_stop, spu_ienable, spu_idisable, ...)
  - Channel Control (spu_readch, spu_writech,...)
  - Scalar (spu_insert, spu_extract, spu_promote)
- Composite
  - DMA (spu_mfcdma32, spu_mfcdma64, spu_mfcstat)

Accessing SIMD instructions

- **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.
  - Examples
    - t_v = spu_mul(t_v, (vector float)spu_splats((float)0.5));
    - t_v = spu_sub((vector float) spu_splats((float)47.11), t_v);
    - t_v = spu_mul(t_v, s_v);

Vectorizing a Loop – A Simple Example

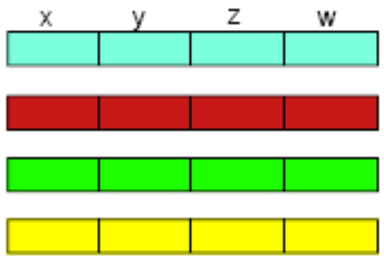
```
/*  scalar version  */

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**

SIMD Data Partitioning Strategies

- **Choose SIMD strategy appropriate for algorithm**
  - vec-across



4 independent 4-component vectors (cyan, red, green and yellow)

  - parallel-array

    - Easy to SIMD – program as if scalar, operating on 4 independent objects at a time
    - Data must be maintained in separate arrays or SPU must shuffle vec-across data into a parallel array form
- **Consider unrolling affects when picking SIMD strategy**

Complex multiplication

- **In general, the multiplication of two complex numbers is represented by**

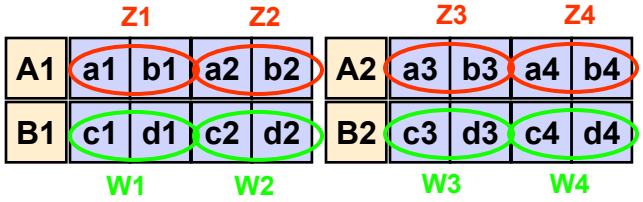
$$(a+ib)(c+id)=(ac-bd)+i(ad+bc)$$

- **Or, in code form:**

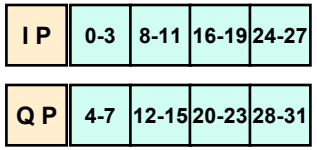
```
/* Given two input arrays with interleaved real and imaginary parts */
float input1[2N], input2[2N], output[2N];

for (int i=0;i<N;i+=2) {
   float ac = input1[i]*input2[i];
   float bd = input1[i+1]*input2[i+1];
   output[i] = (ac - bd);
 /*optimized version of (ad+bc) to get rid of a multiply*/
 /* (a+b) * (c+d) -ac - bd = ac + ad + bc + bd -ac -bd = ad + bc */
   output[i+1] = (input1[i]+input1[i+1])*(input2[i]+input2[i+1]) - ac - bd;
}
```

Complex Multiplication SPE - Shuffle Vectors

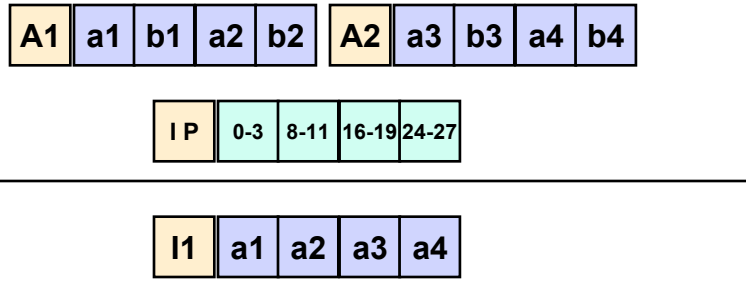
Input



Shuffle patterns



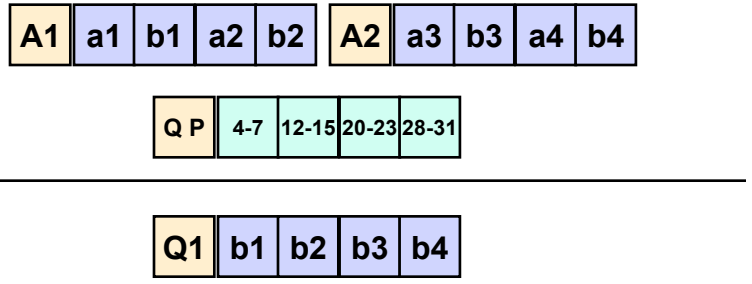

```
I1 = spu_shuffle(A1, A2, I_Perm_Vector);
```




```
I2 = spu_shuffle(B1, B2, I_Perm_Vector);
```

By analogy


```
Q1 = spu_shuffle(A1, A2, Q_Perm_Vector);
```

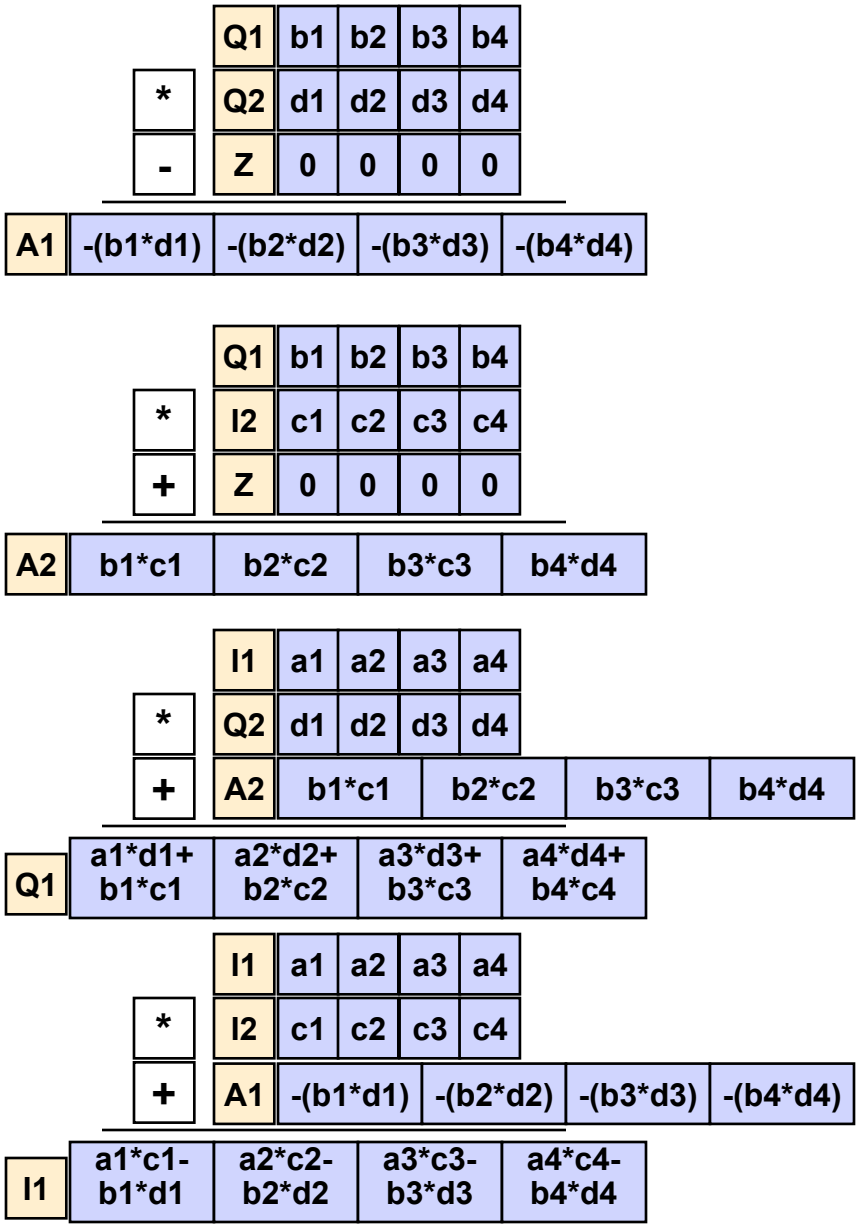



```
Q2 = spu_shuffle(B1, B2, Q_Perm_Vector);
```

By analogy

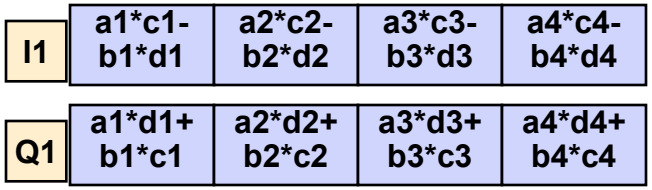
Complex Multiplication

```
A1 = spu_nmsub(Q1, Q2, v_zero);

A2 = spu_madd(Q1, I2, v_zero);

Q1 = spu_madd(I1, Q2, A2);

I1 = spu_madd(I1, I2, A1);
```

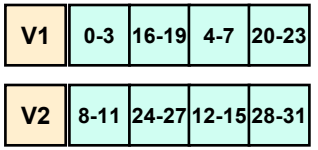


Complex Multiplication – Shuffle Back

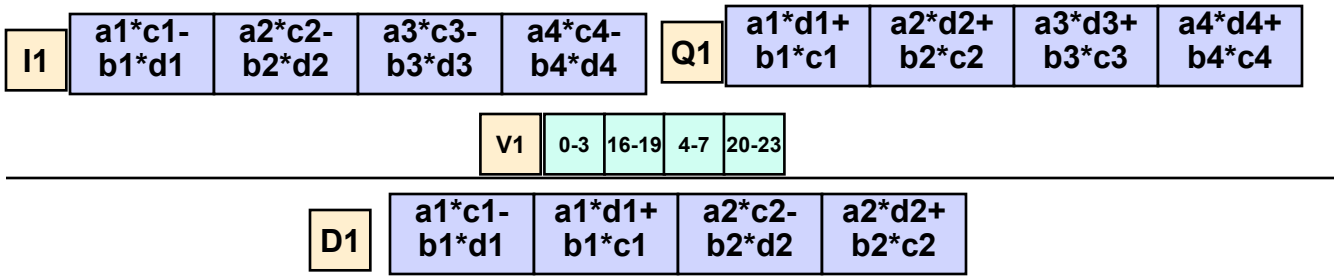
Results



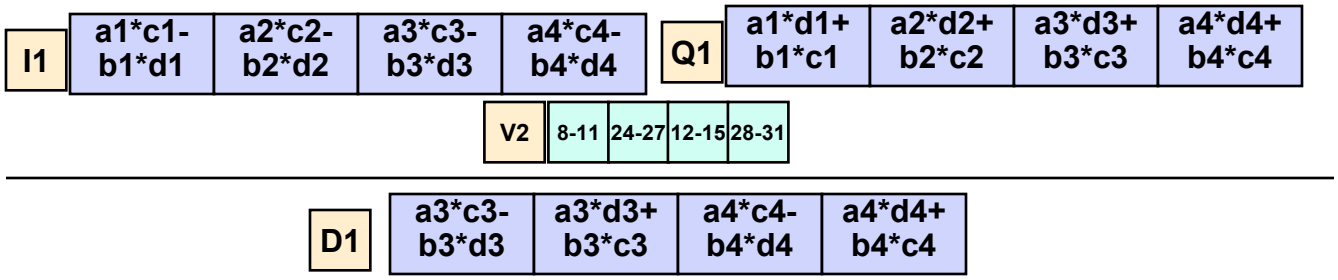
Shuffle patterns




```
D1 = spu_shuffle(I1, Q1, vcvmrgh);
```




```
D2 = spu_shuffle(I1, Q1, vcvmrgl);
```



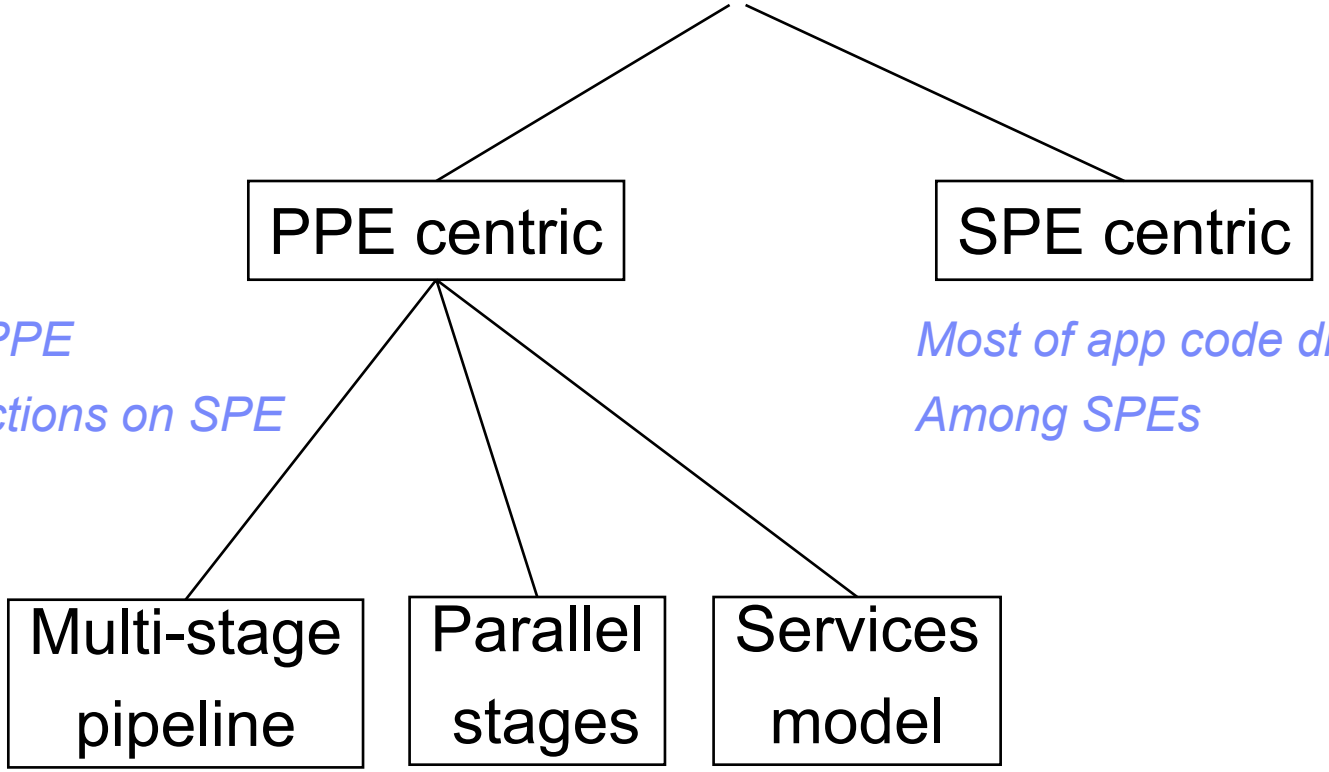
Complex multiplication – SPE - Summary

```
vector float A1, A2, B1, B2, I1, I2, Q1, Q2, D1, D2;
                                    /* in-phase (real), quadrature (imag), temp, and output vectors*/
vector float v_zero = (vector float)(0,0,0,0);

vector unsigned char I_Perm_Vector = (vector unsigned char)(0,1,2,3,8,9,10,11,16,17,18,19,24,25,26,27);
vector unsigned char Q_Perm_Vector = (vector unsigned char)(4,5,6,7,12,13,14,15,20,21,22,23,28,29,30,31);
vector unsigned char vcvmrgh = (vector unsigned char) (0,1,2,3,16,17,18,19,4,5,6,7,20,21,22,23);
vector unsigned char vcvmrgl = (vector unsigned char) (8,9,10,11,24,25,26,27,12,13,14,15,28,29,30,31);

/* input vectors are in interleaved form in A1,A2 and B1,B2 with each input vector
   representing 2 complex numbers and thus this loop would repeat for N/4 iterations
 */
I1 = spu_shuffle(A1, A2, I_Perm_Vector); /* pulls out 1st and 3rd 4-byte element from vectors A1 and A2 */
I2 = spu_shuffle(B1, B2, I_Perm_Vector); /* pulls out 1st and 3rd 4-byte element from vectors B1 and B2 */
Q1 = spu_shuffle(A1, A2, Q_Perm_Vector); /* pulls out 2nd and 4th 4-byte element from vectors A1 and A2 */
Q2 = spu_shuffle(B1, B2, Q_Perm_Vector); /* pulls out 3rd and 4th 4-byte element from vectors B1 and B2 */
A1 = spu_nmsub(Q1, Q2, v_zero);          /* calculates -(bd - 0) for all four elements */
A2 = spu_madd(Q1, I2, v_zero);           /* calculates (bc + 0) for all four elements */
Q1 = spu_madd(I1, Q2, A2);               /* calculates ad + bc for all four elements */
I1 = spu_madd(I1, I2, A1);               /* calculates ac - bd for all four elements */
D1 = spu_shuffle(I1, Q1, vcvmrgh);       /* spreads the results back into interleaved format */
D2 = spu_shuffle(I1, Q1, vcvmrgl);       /* spreads the results back into interleaved format */
```

Application Partitioning

- **Applications must be partitioned across the processing elements**
  - PPC core, SPEs
- **Partitioning involves consideration of and trade-offs among:**
  - processing load
  - program structure
  - data flow
  - data and code movement via DMA
  - loading of bus and bus attachments
  - desired performance
- **Several models:**
  - “PPC-centric” vs. “SPE-centric”
  - data-serial / instruction-parallel vs. data-parallel / instruction-serial

Application Partitioning



Main app on PPE

Individual functions on SPE

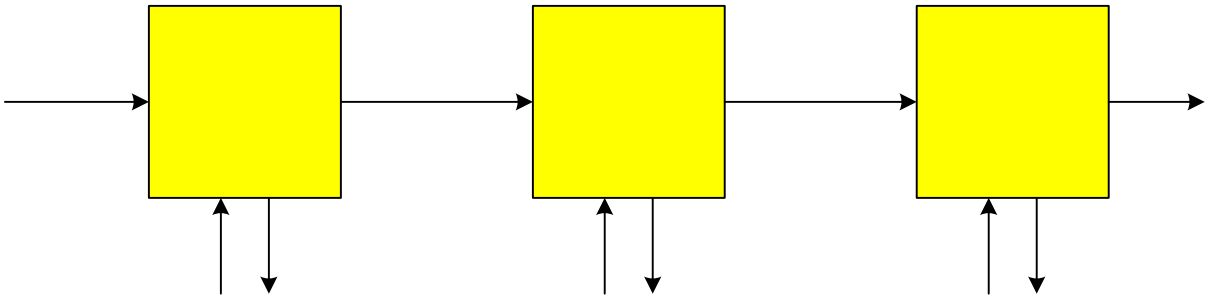
Most of app code distributed

Among SPEs

“PPC-Centric” & “SPE-Centric” Models

- **“PPC-Centric”:**
  - an offload model
  - main line application code runs in PPC core
  - individual functions extracted and offloaded to SPEs
  - SPUs wait to be given work by the PPC core
- **“SPE-Centric”:**
  - most of the application code distributed among SPEs
  - PPC core runs little more than a resource manager for the SPEs (e.g. maintaining in main memory control blocks with work lists for the SPEs)
  - SPE fetches next work item (what function to execute, pointer to data, etc.) from main memory (or its own memory) when it completes current work item

A Pipelined Approach



- **Data-serial / instruction-parallel**
- **Example: three function groups, so three SPEs**
- **Dataflow is unidirectional**
- **Synchronization is important**
  - time spent in each function group should be about the same
  - but may complicate tuning and optimization of code
- **Main data movement is SPE-to-SPE**
  - can be push or pull
- **Disadvantages**
  - Difficulty of load balancing
  - Increase in data movements

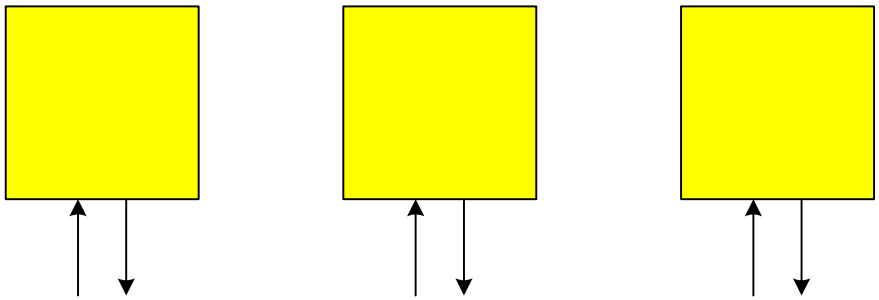
INPUT

FUNCTION GROUP 0 (SPE 0)

LOCAL STATE (TO/FROM MAIN MEM)

(TO

A Data-Partitioned Approach

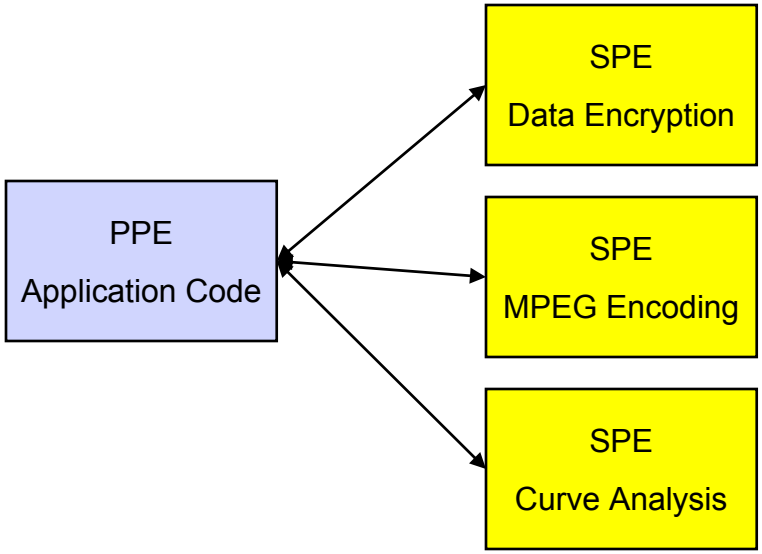


- **Data-parallel / instruction-serial**
- **Example: data blocks partitioned into three sub-blocks, so three SPEs**
- **May require coordination among SPEs between functions**
  - e.g. if there is interaction between data sub-blocks
- **Essentially all data movement is SPE-to main memory or main memory-to-SPE**

Function 0 in each SPE
- then -
Function 1 in each SPE
- then -
Function 2 in each SPE
- etc.

SPE 0

A Services Approach



- **Different functions offered as a service**
- **Fixed static allocation should be avoided**
- **Services should be virtulized and managed on a demand-initiated basis**

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