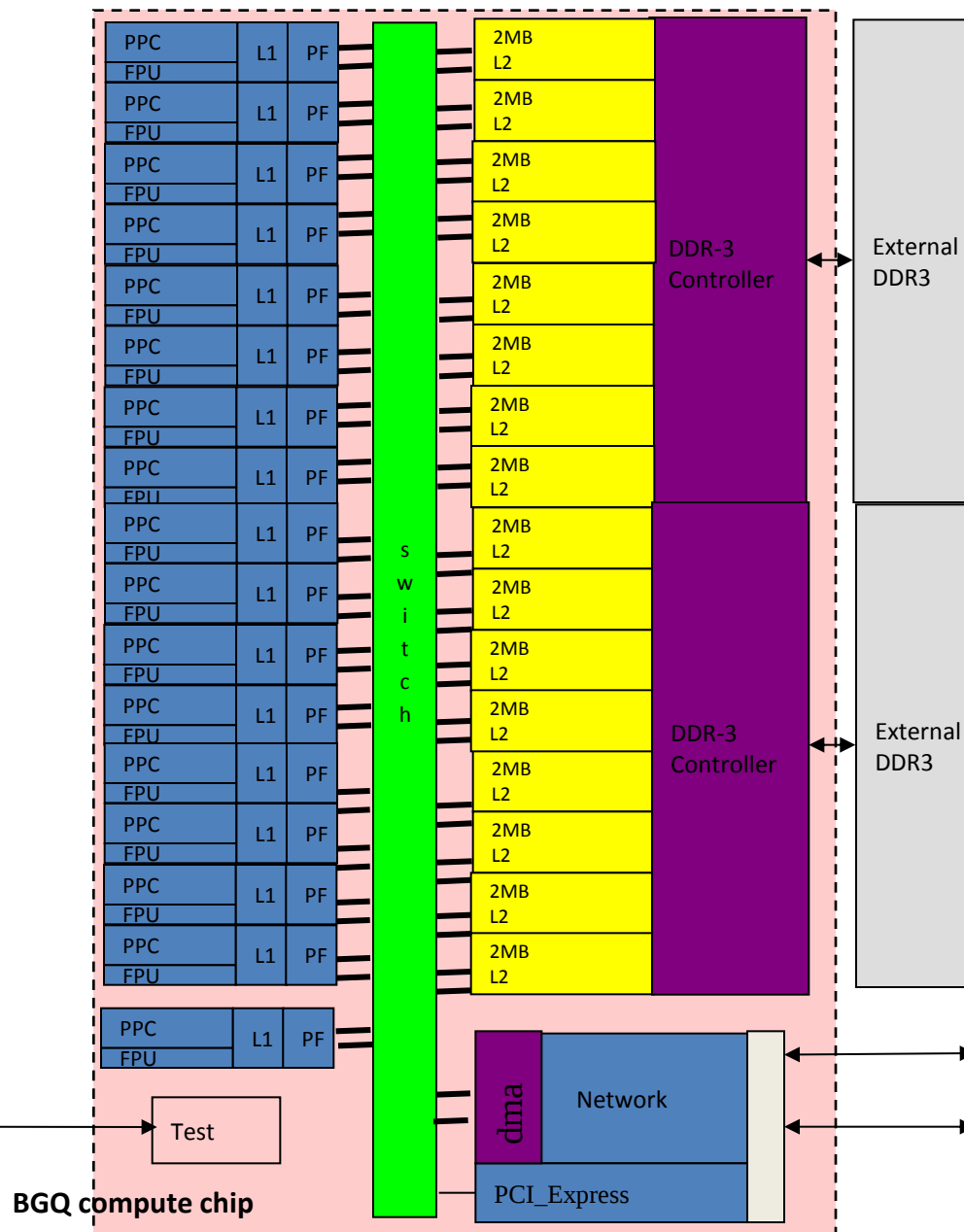


Vectorization using QPX intrinsic

3 February 2014 | Stefan Krieg

BGQ Chip architecture



- 16+1 core SMP
 - Each core 4 way hardware threaded
- Transactional memory and thread level speculation
- Quad float point unit on each core
 - 204.8 GF peak node
- Frequency target of (1.6) GHz
- 563 GB/s bisection bandwidth to shared L2 (BGL at LLNL has 700 GB/s system bisection)
- 32 MB shared L2 cache
- 42.6 GB/s DDR3 bandwidth
 - (2 channels each with chip kill protection)
- 10 intrarack interprocessor links each at 2.0GB/s
- 1 I/O link at 2.0 GB/s
- 4-8 GB memory/node
- ~30 Watts chip power

2 GB/s I/O link (to I/O subsystem)

10*2GB/s Intrarack (5-D torus)

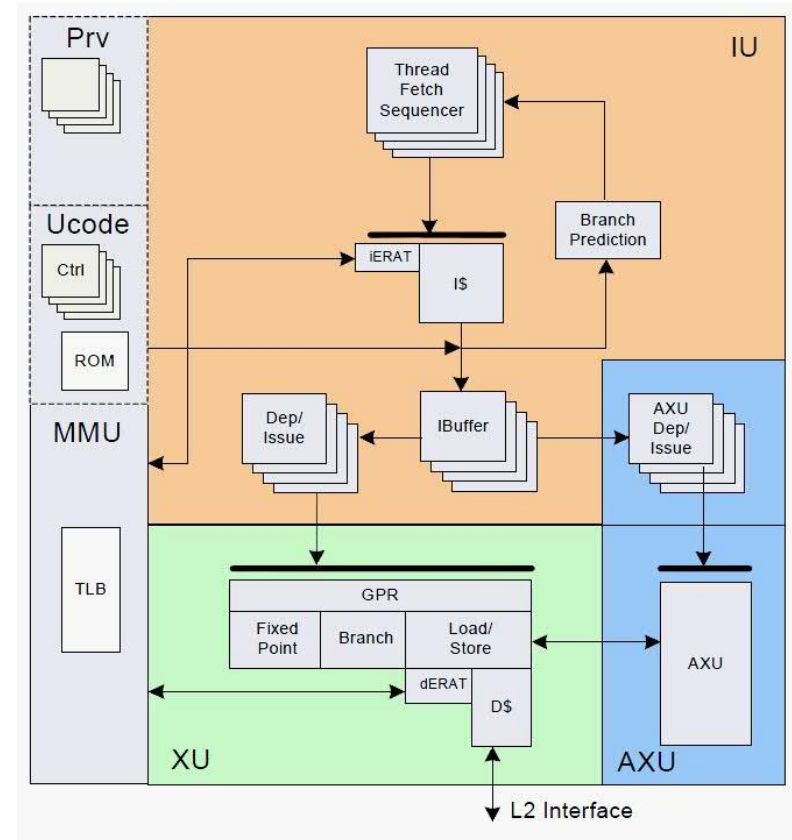
** chip I/O shares function with PCI_Express

BGQ compute chip

3. Februar 2014

A2 core

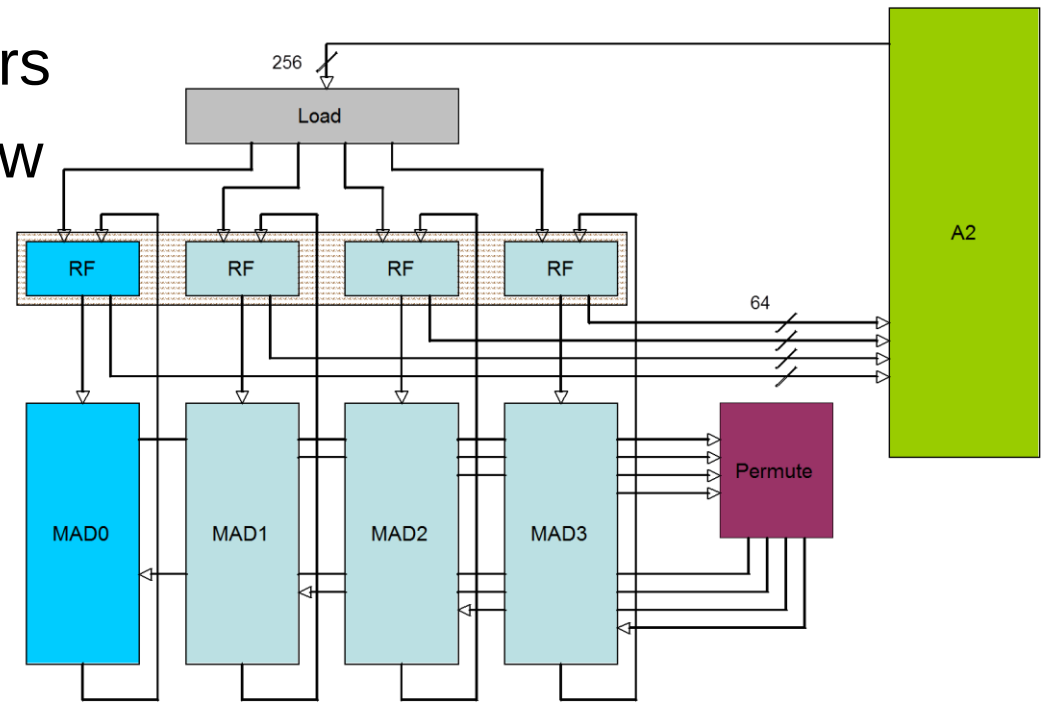
- Embedded processor core
- 64-bit Power ISA
- 4-way simultaneously multi-threaded (SMT)
- In-order dispatch, execution, completion
- Execution unit (XU)
 - *32x4x64-bit general purpose registers*
 - *Dynamic branch prediction*
- Auxiliary execution unit (AXU)
 - *BG/Q: Quad-FPU*



© IBM

Quad Floating-Point Processing Unit

- ISA extension: Quad Processing eXtension (QPX)
- 4 64-bit pipelines, SIMD processing
 - *Vector operands*
- 32x4x256 bit registers
- Multiply-Add Dataflow
 - *Can process in each cycle, e.g., $\pm [(A * B) \pm C]$*
- Peak performance
4 FMA / cycle
→ 12.8 GFlops
at 1.6 GHz



© IBM

BG/Q (twin) complex multiply

- Complex multiplication: $A * B = T$

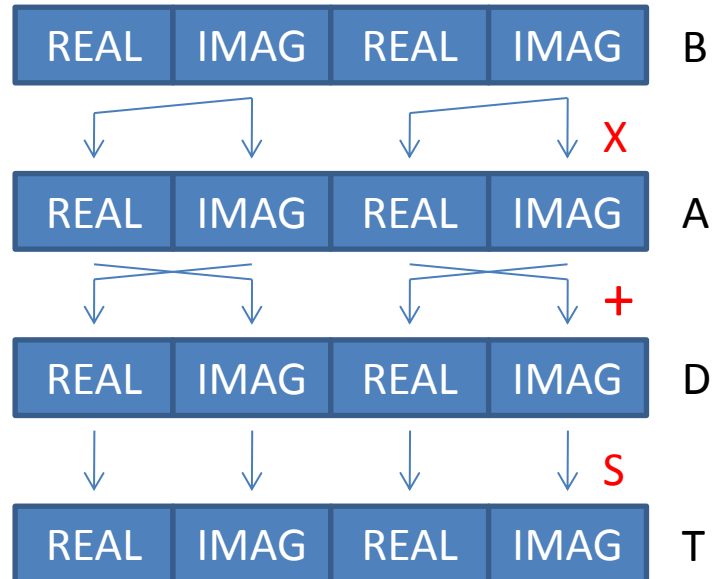
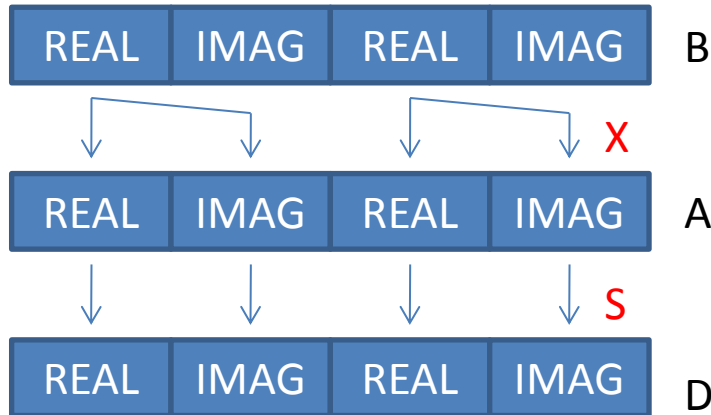
$$\text{Re}(T) = \text{Re}(A) * \text{Re}(B) - \text{Im}(A) * \text{Im}(B)$$

$$\text{Im}(T) = \text{Im}(A) * \text{Re}(B) + \text{Re}(A) * \text{Im}(B)$$

- Same with complex numbers in “complex” register:

$$\begin{array}{ccccccc}
 \begin{array}{|c|c|} \hline \text{REAL} \\ \hline \text{IMAG} \\ \hline \end{array} & = & \begin{array}{|c|c|} \hline \text{REAL} \\ \hline \text{IMAG} \\ \hline \end{array} & * & \begin{array}{|c|c|} \hline \text{REAL} \\ \hline \text{REAL} \\ \hline \end{array} & - & \begin{array}{|c|c|} \hline \text{IMAG} \\ \hline \text{REAL} \\ \hline \end{array} & * & \begin{array}{|c|c|} \hline \text{IMAG} \\ \hline \text{IMAG} \\ \hline \end{array} \\
 T & & A & & rB & & aA & & iB
 \end{array}$$

BG/Q (twin) complex multiply



2 instructions for
2 complex multiplies



No instructions to use 2,3 with 0,1 → 2nd “stream”
or.. forget about 2,3 ??

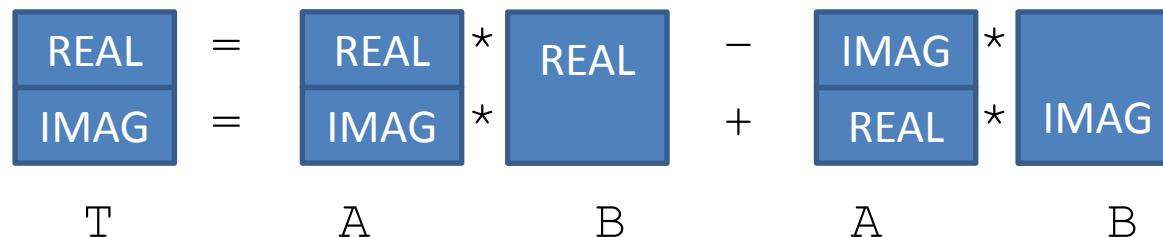
BG/Q (twin) complex multiply

- Complex multiplication: $A * B = T$

$$\text{Re}(T) = \text{Re}(A) * \text{Re}(B) - \text{Im}(A) * \text{Im}(B)$$

$$\text{Im}(T) = \text{Im}(A) * \text{Re}(B) + \text{Re}(A) * \text{Im}(B)$$

- Same with complex numbers in “complex” register:



XL C intrinsics

- New data type: `vector4double`
- Example:

```
double r[4] = {1.0, 2.0, 3.0, 4.0};  
vector4double v1 = (vector4double) {1.0, 2.0, 3.0, 4.0};  
vector4double v2, v3;  
v2 = vec_ld(0L, &r[0]);      /* load from array */  
v3 = vec_add(v1, v2);         /* add two vectors */  
vec_st(v3, 0L, &r[0]);       /* Store result */
```

- No overloaded operators. However:

```
vector4double *v1ptr = &v1;  
vector4double v4 = *v1ptr;  
vector4double v5 = v4;
```

XL C intrinsics

- **Also:**

```
vector4double v1 = (vector4double) {1., 2., 3., 4.};  
double d1, d2, d3, d4;  
d1 = v1[0];           // d1=1.  
d2 = v1[1];           // d2=2.  
d3 = v1[2];           // d3=3.  
d4 = v1[3];           // d4=4.
```

- **And:**

```
sizeof(v1); // vector type size: 32  
sizeof(&v1); // address of vector size: 8  
__alignof__(v1); // vector type alignment: 32  
__alignof__(&v1); // address of vector alignment: 8
```

XL C intrinsics

- Alignment: loads may truncate addresses.

`d = vec_ld(a, b);`

b+0	b+1	b+2	b+3
-----	-----	-----	-----

- Truncated to 32 byte alignment if `b` is of type
 - `long*`, `unsigned long*`, `long long*`,
`unsigned long long*`
 - `double*`, `_Complex double*`
- Truncated to 16 byte alignment if `b` is of type
 - `float*`, `_Complex float*`
- In case of misalignment, an exception (SIGBUS) is generated by

`d = vec_lda(a, b);`

XL C intrinsics

- Alignment: loads may truncate addresses.

`d = vec_ld2(a, b);`



- Truncated to 16 byte alignment if `b` is of type
 - `double*`
- Truncated to 8 byte alignment if `b` is of type
 - `float*`
- In case of misalignment, an exception (SIGBUS) is generated by

`d = vec_ld2a(a, b);`

XL C intrinsics

- Alignment: loads may truncate addresses.

```
d = vec_lds(a, b);
```



- Truncated to 16 byte alignment if `b` is of type

- `double*, _Complex double*`



- Truncated to 8 byte alignment if `b` is of type

- `float*, _Complex float*`

- In case of misalignment, an exception (SIGBUS) is generated by

```
d = vec_ldsa(a, b); // only _Complex types!
```

XL C intrinsics

- Basic operations:

```
vector4double v1,v2,v3,v4;
```

```
v1 = vec_ld(0,&ptr); //ptr = *{_Complex}{double, float}
```

```
v1 = vec_lds(0,&ptr); //ptr = *{_Complex}{double, float}
```

```
v1 = vec_ld2(0,&ptr); //ptr = *{double, float}
```

```
v1 = vec_add(vec_madd(v2,v3,v4), v1);
```

```
vector4double pattern = vec_gpci(01230);
```

```
v = vec_perm(v,v,pattern);
```

```
vector4double p0167 = (vector4double){2.00,2.25,3.50,3.75};
```

- (Twin) complex multiply:

```
T = vec_xxnpmadd(A, B, vec_xmul(B, A)); // T = A*B
```

```
T = vec_xxcpnmadd(B, A, vec_xmul(A, B)); // T = adj(A)*B
```

XL C intrinsics

- Some useful things:

```
d = vec_splat(a, b);      //splat a[b%4]
d = vec_spalts(a);        //splat a
d = vec_sel(a, b, c);      //d[0]=(c[0]>=0)?a[0]:b[0];...
d = vec_promote(a, b);     //d[b%4]=a; rest undefined
d = vec_insert(a, b, c);   //d=b; d[c%4]=a;
d = vec_extract(a, b);     //d=a[b%4];
d = vec_rsp(a);            //double -> single precision
```

XL C intrinsics

- Special ops:
 - Reciprocal estimate. Guaranteed rel. precision 1/256

```
d = vec_re(a);    //double prec. reciprocal estimate
```

```
d = vec_res(a);   //single prec. reciprocal estimate
```

- Reciprocal square root. Guaranteed rel. precision 1/32

```
d = vec_rsqrt(a); //double prec. reciprocal sqrt
```

```
d = vec_rsrts(a); //single prec. reciprocal sqrt
```

Going from P to Q

- In many cases there is a 1 to 1 mapping from BG/P to BG/Q
- Obvious examples:
 - `vec_madd = __fpmadd`
 - `vec_sub = __fpsub`
- Less obvious ones:
 - `vec_xmadd = __fxcpmadd`
 - `vec_xxadd = __fxcxma`
 - `vec_xxcnadd = __fxcxnsma`
 - `vec_xxnpadd = __fxcxnpma`
- Argument mapping may not be straightforward.
- A list is available.

Assembly

- If you want the full “experience” (question is: WHY?)
- “Easy” alternative to “pure” asm: gcc inline asm (also understood by XL C)
- Mind 6 cycle FP pipeline length
- 4 HW threads may help with scheduling (remember: 1 instruction/cycle/hw thread)
- Objdump does not show QPX assembly but “.long 0x...”
- Same with bgxlc -S
- -qlist -qsoure will give you the mapping “.long 0x...” to QPX

Assembly

669 0007D8 qvfxcpn 11CBF486 1	QVFXCPNMA qr14=qr30,qr11,qr18,fcr	.long 0x11cbf486
669 0007DC qvlfdx 7E41AC8E 1	QVLFDX qr18=#qvSPILL18(gr1,gr21,0)	.long 0x7e41ac8e
669 0007E0 qvfxmadd 11B56A92 1	QVFXMADD qr13=qr13,qr21,qr10,fcr	.long 0x11b56a92
669 0007E4 addi 3AA00300 1 LI	gr21=768	addi r21,r0,768
669 0007E8 qvfxmadd 10B329D2 1	QVFXMADD qr5=qr5,qr19,qr7,fcr	.long 0x10b329d2
0 0007EC addi 39290004 1 AI	gr9=gr9,4	addi r9,r9,4
669 0007F0 qvfxmadd 11E98D52 1	QVFXMADD qr15=qr17,qr9,qr21,fcr	.long 0x11e98d52
0 0007F4 addi 39080004 1 AI	gr8=gr8,4	addi r8,r8,4
669 0007F8 qvfxmadd 1229DD92 1	QVFXMADD qr17=qr27,qr9,qr22,fcr	.long 0x1229dd92
0 0007FC addi 38E70004 1 AI	gr7=gr7,4	addi r7,r7,4
669 000800 qvfxmadd 118664D2 1	QVFXMADD qr12=qr12,qr6,qr19,fcr	.long 0x118664d2
669 000804 qvfxmadd 11C67512 1	QVFXMADD qr14=qr14,qr6,qr20,fcr	.long 0x11c67512
669 000808 qvfxcpn 10A72CC6 1	QVFXCPNMA qr5=qr5,qr7,qr19,fcr	.long 0x10a72cc6
669 00080C qvfxcpn 11AA6D46 1	QVFXCPNMA qr13=qr13,qr10,qr21,fcr	.long 0x11aa6d46
669 000810 qvfxnpm 11F57A56 1	QVFXNPMA qr15=qr15,qr21,qr9,fcr	.long 0x11f57a56
669 000814 qvfxnpm 12368A56 1	QVFXNPMA qr17=qr17,qr22,qr9,fcr	.long 0x12368a56
669 000818 qvfxnpm 11936196 1	QVFXNPMA qr12=qr12,qr19,qr6,fcr	.long 0x11936196
669 00081C qvlfdx 7E61AC8E 1	QVLFDX qr19=#qvSPILL19(gr1,gr21,0)	.long 0x7e61ac8e
669 000820 qvfxnpm 11D47196 1	QVFXNPMA qr14=qr14,qr20,qr6,fcr	.long 0x11d47196
0 000824 addi 3AA00180 1 LI	gr21=384	addi r21,r0,384
669 000828 qvfxmadd 10B42A12 1	QVFXMADD qr5=qr5,qr20,qr8,fcr	.long 0x10b42a12

Assembly

```
#define SingleLoadNU(si,sb,tgt) \
do { \
asm volatile("qvlfsx %0,%1,%2" :: "i" (tgt), "b" (si), "r" (sb) : "memory"); \
} while(0)

#define SingleLoadNU(si,sb,tgt) \
do { \
asm volatile("qvstfsx %2,%0,%1" :: "b" (si), "r" (sb), "i" (tgt) : "memory"); \
} while(0)

#define VECTOR_qvfadd(op_a,op_b,tgt) \
do { \
asm volatile("qvfadd %0,%1,%2" :: "i" (tgt), "i" (op_a), "i" (op_b) ); \
}while(0);

#define VECTOR_fxcpmadd(tgt, op_a, op_b, op_c) VECTOR_qvfxmadd(op_a,op_b,op_c,tgt)
#define VECTOR_qvfxmadd(op_a,op_b,op_c,tgt) \
do { \
asm volatile("qvfxmadd %0,%1,%2,%3" :: "i" (tgt), "i" (op_a), "i" (op_b), "i" (op_c) ); \
}while(0);
```

Assembly

```
SingleLoadNU(in0, 10, OUT_P(0,0,0));  
SingleLoadNU(in0, 11, OUT_P(0,1,0));  
SingleLoadNU(in0, 12, OUT_P(0,2,0));  
SingleLoadNU(in0, 13, OUT_P(0,0,1));
```

```
VECTOR_fxcpmadd( 22, 0, 10, 30);  
VECTOR_fxcpmadd( 23, 1, 10, 30);  
VECTOR_fxcpmadd( 24, 2, 10, 30);  
VECTOR_fxcpmadd( 25, 0, 13, 30);  
SingleLoadNU(in0, 14, OUT_P(0,1,1));  
VECTOR_fxcpmadd( 26, 1, 13, 30);  
VECTOR_fxcpmadd( 27, 2, 13, 30);  
SingleLoadNU(in0, 15, OUT_P(0,2,1));
```

Concluding remarks

- Intrinsic achieve high efficiency
- Inline assembly works for both gcc and XL C
- XL C intrinsics (vector built in functions) reference:
<http://www-01.ibm.com/support/docview.wss?uid=swg27027065&aid=1>
- QPX reference:
[http://domino.research.ibm.com/library/cyberdig.nsf/papers/BBD8BB0F809822E885257A2F00511C69/\\$File/rc25291.pdf](http://domino.research.ibm.com/library/cyberdig.nsf/papers/BBD8BB0F809822E885257A2F00511C69/$File/rc25291.pdf)