Using Vector Intrinsics for BGQ

Bob Walkup (walkup@us.ibm.com, 914-945-1512)

BGQ has SIMD capability = instructions to process four doubles or floats.

All arithmetic work is done with doubles. QPX mult-add = 8 Flops/cycle.

Vector load and store operations can work with doubles or floats, and include automatic format conversion to/from float. Format is double in registers.

Vector intrinsics are provided by the XL compiler to make it possible for you to explicitly code SIMD instructions. Each vector intrinsic maps to one QPX instruction, callable from C, C++, or Fortran.

Documentation in IBM XL compiler manuals :

C/C++ Compiler Reference
/opt/ibmcmp/vacpp/bg/12.1/doc/en_US/pdf/compiler.pdf

Fortran Language Reference
/opt/ibmcmp/vacpp/bg/12.1/doc/en_US/pdf/compiler.pdf

On Argonne systems, look in /soft/compilers instead of /opt.

http://pic.dhe.ibm.com/infocenter/compbg/v121v141/index.jsp

C example with load, store, and multiply-add operations : doubles

Note: the vector load, vec_ld(0, &x[i]), will grab four contiguous doubles starting at the specified address rounded down to the nearest 32-byte boundary. It is absolutely critical to check or know the alignment.

```
//---------------------------------------
// vector version of y[i] = a*x[i] + y[i]
// where "x" and "y" are 32-byte aligned
//---------------------------------------
#include <stdio.h>    // C++ codes must #include <builtins.h>

#define NPTS 8
static double __attribute__((aligned(32))) x[NPTS], y[NPTS];

int main(int argc, char * argv[])
{
   int i;
   double a = 2.0;
   vector4double av, xv, yv;
   for (i=0; i<NPTS; i++) {
      x[i] = (double) i;
      y[i] = (double) i + 1;
   }
   if ((long) x & 0x1F ) printf("x is not 32-byte aligned\n");
   if ((long) y & 0x1F)  printf("y is not 32-byte aligned\n");
   av = vec_splats(a);            // replicate "a" in four vector slots
   for (i=0; i<NPTS; i+=4) {
      xv = vec_ld(0, &x[i]);      // load four contiguous elements of x[]
      yv = vec_ld(0, &y[i]);      // load four contiguous elements of y[]
      yv = vec_madd(av, xv, yv);  // yv = av*xv + yv
      vec_st(yv, 0, &y[i]);       // store four contiguous elements of y[]
   }
   for (i=0; i<NPTS; i++) printf("y[%d] = %.1lf\n", i, y[i]);
   return 0;
}
```

Fortran example with load, store, and mult-add operations : doubles

Note: the vector load, vec_ld(0, x(i)), will grab four contiguous doubles starting at the specified address rounded down to the nearest 32-byte boundary. It is absolutely critical to check or know the alignment.

```
!---------------------------------------
! vector version of y(:) = a*x(:) + y(:)
! where "x" and "y" are 32-byte aligned
!---------------------------------------
program fmain
  implicit none
  integer i
  integer, parameter :: n = 8
!IBM* align(32, x, y)
  real(8) a, x(n), y(n)
  vector(real(8)) av, xv, yv
  a = 2.0d0
  do i = 1, n
    x(i) = dble(i-1)
    y(i) = dble(i)
  end do
  if (iand(loc(x), z'1F') .ne. 0) print *, 'x is not 32-byte aligned'
  if (iand(loc(y), z'1F') .ne. 0) print *, 'y is not 32-byte aligned'
  av = vec_splats(a)            ! replicate "a" in four vector slots
  do i = 1, n, 4
     xv = vec_ld(0, x(i))       ! load four contiguous elements of x()
     yv = vec_ld(0, y(i))       ! load four contiguous elements of y()
     yv = vec_madd(av, xv, yv)  ! yv = av*xv + yv
     call vec_st(yv, 0, y(i))   ! store four contiguous elements of y()
  end do
  do i = 1, n
    write(*,'(a,i1,a,f4.1)') 'y(', i, ') = ', y(i)
  end do
end
```

C example with load, store, and multiply-add operations : floats

Note: the vector load, vec_ld(0, &x[i]), will grab four contiguous floats starting at the specified address rounded down to the nearest 16-byte boundary. It is absolutely critical to check or know the alignment.

```
//---------------------------------------
// vector version of y[i] = a*x[i] + y[i]
// where "x" and "y" are 16-byte aligned
//---------------------------------------
#include <stdio.h>

#define NPTS 8
static float __attribute__((aligned(16))) x[NPTS], y[NPTS];

int main(int argc, char * argv[])
{
   int i;
   float a = 2.0f;
   vector4double av, xv, yv;
   for (i=0; i<NPTS; i++) {
      x[i] = (float) i;
      y[i] = (float) i + 1;
   }
   if ((long) x & 0xF ) printf("x is not 16-byte aligned\n");
   if ((long) y & 0xF)  printf("y is not 16-byte aligned\n");
   av = vec_lds(0, &a);           // replicate "a" in four vector registers
   for (i=0; i<NPTS; i+=4) {
      xv = vec_ld(0, &x[i]);      // load four contiguous elements of x[]
      yv = vec_ld(0, &y[i]);      // load four contiguous elements of y[]
      yv = vec_madd(av, xv, yv);  // yv = av*xv + yv
      vec_st(yv, 0, &y[i]);       // store four contiguous elements of y[]
   }
   for (i=0; i<NPTS; i++) printf("y[%d] = %.1f\n", i, y[i]);
   return 0;
}
```

Fortran example with load, store, and mult-add operations : floats

Note: the vector load, vec_ld(0, x(i)), will grab four contiguous floats starting at the specified address rounded down to the nearest 16-byte boundary. It is absolutely critical to check or know the alignment.

```
!---------------------------------------
! vector version of y(:) = a*x(:) + y(:)
! where "x" and "y" are 16-byte aligned
!---------------------------------------
program fmain
  implicit none
  integer i
  integer, parameter :: n = 8
!IBM* align(16, x, y)
  real(4) a, x(n), y(n)
  vector(real(8)) av, xv, yv
  a = 2.0
  do i = 1, n
    x(i) = float(i-1)
    y(i) = float(i)
  end do
  if (iand(loc(x), z'F') .ne. 0) print *, 'x is not 16-byte aligned'
  if (iand(loc(y), z'F') .ne. 0) print *, 'y is not 16-byte aligned'
  av = vec_lds(0, a)            ! replicate "a" in four vector slots
  do i = 1, n, 4
     xv = vec_ld(0, x(i))       ! load four contiguous elements of x()
     yv = vec_ld(0, y(i))       ! load four contiguous elements of y()
     yv = vec_madd(av, xv, yv)  ! yv = av*xv + yv
     call vec_st(yv, 0, y(i))   ! store four contiguous elements of y()
  end do
  do i = 1, n
    write(*,'(a,i1,a,f4.1)') 'y(', i, ') = ', y(i)
  end do
end
```

Data alignment is the number one concern.

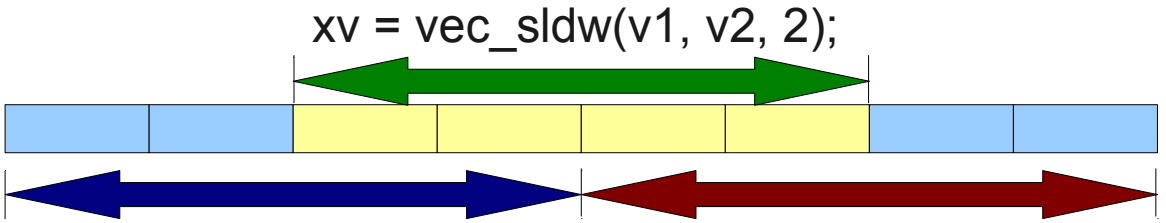
vec_ld() will always grab four contiguous items starting at the specified address rounded down to the nearest 32/16-byte boundary (doubles/floats).

vec_st() will always store four contiguous items starting at the specified address rounded down to the nearest 32/16 byte boundary (doubles/floats).

If there is any doubt about alignment, check the address :

```
C/C++ :  int misaligned = ((long) &x[i]) & 0x1F ;      // 0xF  for floats
Fortran : integer misaligned = iand(loc(x(i)), z'1F')  !  z'F'   for floats
```

To load misaligned data, you have to load two “quads” and then select the appropriate four elements. There are several ways to do this, using shift or permute instructions. The example below illustrates a shift instruction.



First four locations.
v1 = vec_ld(0, &x[i]);

Second four locations.
v2 = vec_ld(0, &x[i+4]);

Vector Shift and Permute Instructions

Shift : xv = vec_sldw(v1, v2, 2); the third argument must be 0,1,2,or 3.
Think of the concatenation of vectors v1 and v2, shifted left by two elements.
Limitation : the third argument must be a constant 0-3, not a variable.

To handle misaligned data at runtime, use : pctl = vec_lvsl(0, &x[i]) :

```
double * x;
vector4double v1, v2, xv, pctl;
v1 = vec_ld(0, &x[i]);            // load 4 doubles at &x[i] rounded down to n*32
v2 = vec_ld(0, &x[i+4]);          // load 4 doubles at &x[i+4] rounded down to n*32
pctl = vec_lvsl(0, &x[i]);        // compute the permute control vector
xv = vec_perm(v1, v2, pctl);  // xv contains x[i], x[i+1], x[i+2], x[i+3]
```

Limitation : the maximum left shift is 3 … useful for loops.

General permutation extracts any four slots from the concatenation of two vectors :

```
double * x;
vector4double v1, v2, xv, pctl;
v1 = vec_ld(0, &x[i]);            // load 4 doubles at &x[i] rounded down to n*32
v2 = vec_ld(0, &x[i+4]);          // load 4 doubles at &x[i+4] rounded down to n*32
pctl = vec_gpci(0abcd);           // compute the permute control vector
xv = vec_perm(v1, v2, pctl);  // xv contains data from slots a,b,c,d
```

Note : a,b,c,d must be numbers in the range 0-7; leading zero for octal constant.
Fortran version : xv = vec_perm(o'abcd') … leading “o” for octal constant.

Fortran Example of Shifts and Permutes : compile with bgxlf90

```
!-----------------------------------------------
! example of vector shift and permute operations
!-----------------------------------------------
program fmain
  implicit none
  integer, parameter :: npts = 8
  integer i, j
  vector(real(8)) v1, v2, xv, pctl
!IBM* align(32,x)
  real(8) x(npts), xa(4)
  equivalence (xv,xa)

  do i = 1, npts
    x(i) = dble(i)
  end do

  v1 = vec_ld(0, x(3))     ! v1 has values x(1-4)
  v2 = vec_ld(0, x(7))     ! v2 has values x(5-8)

  xv = vec_sldw(v1, v2, 2) ! xv has values x(3-6)
  write(6,*) 'shift : xv =', (xa(i), i=1,4)

  pctl = vec_gpci(o'5243')     ! slots are numbered 0-7
  xv = vec_perm(v1, v2, pctl)  ! xv has "x" values from slots 5,2,4,3
  write(6,*) 'gpci  : xv =', (xa(i), i=1,4)

  do j = 1, 5
     pctl = vec_lvsl(0, x(j))   !xv has values x(j-j+3)
     xv = vec_perm(v1, v2, pctl)
     write(6,*) 'lvsl  : xv =', (xa(i), i=1,4)
  end do
end
```

C Example of Shifts and Permutes : compile with bgxlc

```
//-----------------------------------------------
// example of vector shift and permute operations
//-----------------------------------------------
#include <stdio.h>

#define NPTS 8
static double __attribute__((aligned(32))) x[NPTS];

int main(int argc, char * argv[])
{
   int i, j;
   double d;
   vector4double v1, v2, xv, pctl;

   for (i=0; i<NPTS; i++) x[i] = (double) i;

   v1 = vec_ld(0, &x[2]);     // v1 has values x[0-3]
   v2 = vec_ld(0, &x[6]);     // v2 has values x[4-7]

   xv = vec_sldw(v1, v2, 2);  // xv has values x[2-5]
   for (i=0; i<4; i++) printf("shift : xv[%d] = %.1lf\n", i, xv[i]);

   pctl = vec_gpci(05243);    // xv has "x" values from slots 5,2,4,3
   xv = vec_perm(v1, v2, pctl);
   for (i=0; i<4; i++) printf("gpci  : xv[%d] = %.1lf\n", i, xv[i]);

   for (j=0; j<=4; j++) {
      pctl = vec_lvsl(0, &x[j]);  //xv has values x[j-j+3]
      xv = vec_perm(v1, v2, pctl);
      for (i=0; i<4; i++) printf("lvsl  : xv[%d] = %.1lf\n", i, xv[i]);
   }

   return 0;
}
```

More on vector load and store operations

vec_ld(0, &x); can use x = double, float, _Complex double, _Complex float, or long. Alignment constraint is 32-bytes for double or long, 16-bytes for float.

vec_ld2(0, &x); can use x = double or float. Alignment constraint is 16-bytes for double, 8-bytes for float. Loads two contiguous items into QPX registers, values from registers 0 and 1 are replicated in registers 2 and 3.

vec_lds(0, &x); can use x = double, float, _Complex double, or _Complex float. Values are replicated across the vector registers. For the _Complex types, values from registers 0 and 1 are replicated in registers 2 and 3. Alignment constraint is 16-bytes for _Complex double, 8-bytes for _Complex float.

vec_st(av, 0, &x); vector4double av. Can use x = int, long, float, double, _Complex float, _complex double. Alignment constraint is 16-bytes for int and float types, 32-bytes for long and double types.

vec_st2(av, 0, &x); vector4double av. Can use double or float types. Stores two contiguous elements. Alignment constraint is 16-bytes for doubles, 8-bytes for floats.

vec_sts(av, 0, &x); vector4double av. Can use double, float, _Complex double, or _Complex float types. Stores one or two (for _Complex) values. Alignment constraint is 16-bytes for _Complex double, 8-bytes for _Complex float.

There are also signaling versions for these … consult the manuals.

Additional Useful Vector Intrinsics

```
float f;                    // Fortran type is real(4)
double d;                   // Fortran type is real(8)
vector4double av, bv;  // Fortran type is vector(real(8))

av = vec_splats(d);      replicates scalar value “d” across four vector registers

av = vec_splat(bv, 2);  replicates value bv[2] across four vector registers
                                second argument must be 0,1,2,or 3

d = vec_extract(bv, 2);  assigns value of bv[2] to double “d”
                                 second argument should be 0,1,2, or 3.

av = vec_insert(d, bv, 2); vector av = bv with element 2 replaced by value “d”
                                    second argument should be 0,1,2, or 3.
```

---

vec_re(d), vec_res(f) = reciprocal estimate for 1/d or 1/f.
vec_rsqrte(d), vec_rsqrtes(f) = estimate for 1/sqrt(d) or 1/sqrtf(f).

The estimate instructions are good to 14 bits and are useful as inputs into Newton's method for iterative refinement. Two iterations gives full accuracy for doubles, one Newton iteration is enough for floats.

Arithmetic Operations

There is an extensive set of arithmetic operations which use double precision.

Examples : vec_add, vec_sub, vec_mul, vec_madd, vec_msub, vec_nmadd, vec_nmsub, vec_swdiv(_nochk), vec_swsqrt(_nochk), etc.

There are “cross” and “double-cross” versions that can be useful for complex numbers : vec_xmadd, vec_xmul, vec_xxcpnmadd, vec_xxmadd, vec_xxnpmadd.

Other operations support floating-point select, compare, conversion operations, and some logical operations.

Syntax is described in the compiler manuals :

C/C++ Compiler Reference
/opt/ibmcmp/vacpp/bg/12.1/doc/en_US/pdf/compiler.pdf

Fortran Language Reference
/opt/ibmcmp/vacpp/bg/12.1/doc/en_US/pdf/compiler.pdf

On Argonne systems, look in /soft/compilers instead of /opt.

http://pic.dhe.ibm.com/infocenter/compbg/v121v141/index.jsp

DAXPY Example Y(:) = a*X(:) + Y(:)

Version for C : void axpy(int i1, int i2, double a, double * x, double * y)

Simple C-code : for (i=i1; i<i2; i++) y[i] = a*x[i] + y[i];

Problems :

(1) Pointers are passed in, so you can't assume 32-byte alignment.

(2) A compiler will not assume that “x” and “y” are disjoint unless you specify the restrict keyword or use a vendor-specific pragma.

(3) The “x” and “y” pointers may have different alignments.

Approach: We have to load and store the “y” array values, so increment “y” until it is on a 32-byte boundary, and start using vector intrinsics from there. We then have two cases :

(a) The “x” and “y” have the same relative alignment. Use simple vector intrinsics : vec_ld(...x), vec_ld(...y), vec_madd(...), vec_st(...y).

(b) The “x” array has different alignment from “y”. Use vec_lvsl(...x) to compute the required shift. The loop has vec_ld(...x), vec_ld(...y), vec_perm(), vec_madd(), vec_st(...y).

Code to handle different alignments for “x” and “y”

```
void axpy(int is, int ie, double alpha, double * x, double * y)
{
   int i, ioff, misaligned;
   vector4double avec, xv0, xv4, yv0, xvec, pctl;

   avec = vec_splats(alpha);

// increment "i" until y[i] is on a 32-byte boundary
   i = is;
   while ( ((long) &y[i]) & 0x1F ) {
      y[i] = y[i] + alpha*x[i];   i++;
   }
   ioff = i;

   misaligned = ((long) &x[ioff]) & 0x1F;

   if (!misaligned) {
      for (i=ioff; i<ie-3; i+=4) {
         xv0 =  vec_ld(0, &x[i]);
         yv0 =  vec_ld(0, &y[i]);
         vec_st(vec_madd(avec, xv0,  yv0),  0, &y[i]);
      }
   }
   else  {
      pctl = vec_lvsl(0, &x[ioff]);
      xv0 = vec_ld(0, &x[ioff]);
      for (i=ioff; i<ie-3; i+=4) {
         xv4 =  vec_ld(0, &x[i+4]);
         yv0 =  vec_ld(0, &y[i]);
         xvec = vec_perm(xv0,  xv4,  pctl);
         vec_st(vec_madd(avec, xvec, yv0),  0, &y[i]);
         xv0 = xv4;
      }
   }

   if (i < ie ) for (i=i; i<ie; i++) y[i] = y[i] + alpha*x[i];
}
```

Assembly code for DAXPY loop using scalar instructions :

// no disjoint pragma or restrict keywords … pointer aliasing is assumed
for (i=i1; i<i2; i++) y[i] = alpha*x[i] + y[i];

Compiled with -O3 -qsimd=noauto **-qlist -qsource**

```
 0|                                    CL.330:
79|  000560 lfd      C8450028   1      LFL       fp2=(*)double(gr5,40)
79|  000564 stfdu    DC250020   1      STFDU     gr5,(*)double(gr5,32)=fp1
79|  000568 lfd      C8270008   1      LFL       fp1=(*)double(gr7,8)
79|  00056C fmadd    FC20107A   1      FMA       fp1=fp2,fp0,fp1,fcr
79|  000570 lfd      C8450010   1      LFL       fp2=(*)double(gr5,16)
79|  000574 stfd     D8250008   1      STFL      (*)double(gr5,8)=fp1
79|  000578 lfd      C8270010   1      LFL       fp1=(*)double(gr7,16)
79|  00057C fmadd    FC20107A   1      FMA       fp1=fp2,fp0,fp1,fcr
79|  000580 lfd      C8450018   1      LFL       fp2=(*)double(gr5,24)
79|  000584 stfd     D8250010   1      STFL      (*)double(gr5,16)=fp1
79|  000588 lfd      C8270018   1      LFL       fp1=(*)double(gr7,24)
79|  00058C fmadd    FC20107A   1      FMA       fp1=fp2,fp0,fp1,fcr
79|  000590 lfd      C8450020   1      LFL       fp2=(*)double(gr5,32)
79|  000594 stfd     D8250018   1      STFL      (*)double(gr5,24)=fp1
79|  000598 lfdu     CC270020   1      LFDU      fp1,gr7=(*)double(gr7,32)
79|  00059C fmadd    FC20107A   1      FMA       fp1=fp2,fp0,fp1,fcr
 0|  0005A0 bc       4200FFC0   1      BCT       ctr=CL.330,taken=100%(100,0)
```

The loop is unrolled 4x but the same registers are used for each fmadd. The loop iterations are completed one at a time; there are no independent instructions to hide the latency from load to use => poor performance.

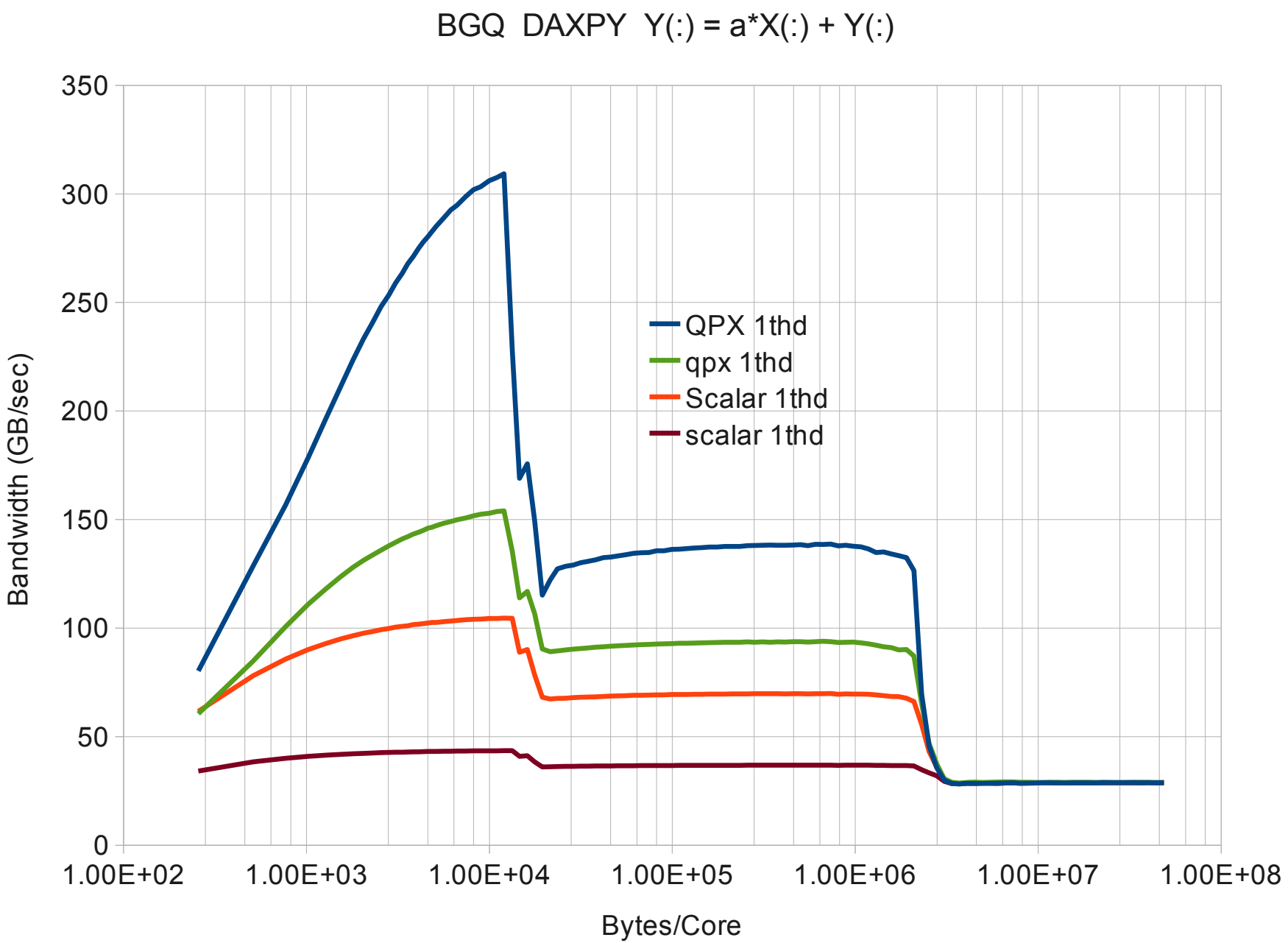
Assembly code for DAXPY loop using scalar instructions :

```
#pragma disjoint(*x, *y)     // could use the restrict keyword
  for (i=i1; i<i2; i++) y[i] = alpha*x[i] + y[i];
```

Compiled with -O3 -qsimd=noauto **-qlist -qsource**

```
 0|                                  CL.167:
81| 000098 lfd      C8A70008   1     LFL       fp5=x[]0(gr7,8)
81| 00009C fmadd    FC40107A   1     FMA       fp2=fp2,fp0,fp1,fcr
81| 0000A0 stfd     D8C50010   1     STFL      y[]0(gr5,16)=fp6
81| 0000A4 lfd      C8250028   1     LFL       fp1=y[]0(gr5,40)
81| 0000A8 lfd      C8C70010   1     LFL       fp6=x[]0(gr7,16)
81| 0000AC fmadd    FC6020FA   1     FMA       fp3=fp4,fp0,fp3,fcr
81| 0000B0 stfd     D8450018   1     STFL      y[]0(gr5,24)=fp2
81| 0000B4 lfd      C8850030   1     LFL       fp4=y[]0(gr5,48)
81| 0000B8 lfd      C8450038   1     LFL       fp2=y[]0(gr5,56)
81| 0000BC fmadd    FCA0097A   1     FMA       fp5=fp1,fp0,fp5,fcr
81| 0000C0 stfdu    DC650020   2     STFDU     gr5,y[]0(gr5,32)=fp3
81| 0000C4 lfd      C8270018   1     LFL       fp1=x[]0(gr7,24)
81| 0000C8 lfdu     CC670020   1     LFDU      fp3,gr7=x[]0(gr7,32)
81| 0000CC fmadd    FCC021BA   1     FMA       fp6=fp4,fp0,fp6,fcr
81| 0000D0 stfd     D8A50008   1     STFL      y[]0(gr5,8)=fp5
81| 0000D4 lfd      C8850020   1     LFL       fp4=y[]0(gr5,32)
 0| 0000D8 bc       4200FFC0   1     BCT       ctr=CL.167,taken=100%(100,0)
```

The loop is unrolled 4x and a different set of registers is used for each fmadd.
The instructions are pipelined to hide the latency between load and use.



Measurement using all 16 cores, with one thread per core.

Vector Intrinsic code for DAXPY, co-aligned case :

```
vector4double avec;

avec = vec_splats(alpha);

if (!misaligned) {
  for (i=ioff; i<ie-15; i+=16) {
    vec_st(vec_madd(avec, vec_ld(0,&x[i]),    vec_ld(0,&y[i])),    0, &y[i]);
    vec_st(vec_madd(avec, vec_ld(0,&x[i+4]),  vec_ld(0,&y[i+4])),  0, &y[i+4]);
    vec_st(vec_madd(avec, vec_ld(0,&x[i+8]),  vec_ld(0,&y[i+8])),  0, &y[i+8]);
    vec_st(vec_madd(avec, vec_ld(0,&x[i+12]), vec_ld(0,&y[i+12])), 0, &y[i+12]);
  }
}
```

The loop was unrolled in an attempt to get better instruction scheduling.

You can chain calls to vector intrinsics. That can eliminate the need for temporary vector variables, but you are more at the mercy of the compiler for register assignments and instruction scheduling. You need to check the assembly code using **-qlist -qsource**. In this case the compiler generates inefficient code, using the same registers repeatedly instead of using a separate set of registers for each fmadd, and pipelining the instructions.

Assembly code for daxpy using QPX instructions, aligned case :

Options : -O3 -qstrict -qhot -qsimd=noauto -qdebug=nunroll **-qlist -qsource**

```
 0|                                    CL.214:
43| 00022C qvlfdx   7C484C8E   1      QVLFDX    qr2=x[]0.rns3.(gr8,gr9,0)
43| 000230 qvlfdx   7C664C8E   1      QVLFDX    qr3=y[]0.rns2.(gr6,gr9,0)
43| 000234 qvfmadd  104118BA   1      QVFMADD   qr2=qr3,qr1,qr2,fcr
44| 000238 qvlfdx   7C68548E   1      QVLFDX    qr3=x[]0.rns3.(gr8,gr10,0)
43| 00023C qvstfdx  7C464D8E   1      QVSTFDX   y[]0.rns2.(gr6,gr9,0)=qr2
44| 000240 qvlfdx   7C46548E   1      QVLFDX    qr2=y[]0.rns2.(gr6,gr10,0)
44| 000244 qvfmadd  104110FA   1      QVFMADD   qr2=qr2,qr1,qr3,fcr
45| 000248 qvlfdx   7C685C8E   1      QVLFDX    qr3=x[]0.rns3.(gr8,gr11,0)
44| 00024C qvstfdx  7C46558E   1      QVSTFDX   y[]0.rns2.(gr6,gr10,0)=qr2
45| 000250 qvlfdx   7C465C8E   1      QVLFDX    qr2=y[]0.rns2.(gr6,gr11,0)
45| 000254 qvfmadd  104110FA   1      QVFMADD   qr2=qr2,qr1,qr3,fcr
46| 000258 qvlfdux  7C6864CE   1      QVLFDUX   qr3,gr8=x[]0.rns3.(gr8,gr12,0)
45| 00025C qvstfdx  7C465D8E   1      QVSTFDX   y[]0.rns2.(gr6,gr11,0)=qr2
46| 000260 qvlfdx   7C46648E   1      QVLFDX    qr2=y[]0.rns2.(gr6,gr12,0)
46| 000264 qvfmadd  104110FA   1      QVFMADD   qr2=qr2,qr1,qr3,fcr
46| 000268 qvstfdux 7C4665CE   1      QVSTFDUX  gr6,y[]0.rns2.(gr6,gr12,0)=qr2
 0| 00026C bc       4200FFC0   1      BCT       ctr=CL.214,taken=100%(100,0)
```

The loop is unrolled 4x using QPX instructions, so it processes 16 elements per loop iteration. However, there is no pipelining. Each fmadd and store operation completes before the next one is started => poor performance when using one thread per core.

Vector Intrinsic code for DAXPY, co-aligned case :

```
vector4double avec, xv0, xv4, xv8, xv12, yv0, yv4, yv8, yv12;

avec = vec_splats(alpha);

if (!misaligned) {
    for (i=ioff; i<ie-15; i+=16) {
       xv0  = vec_ld(0, &x[i]);
       yv0  = vec_ld(0, &y[i]);
       xv4  = vec_ld(0, &x[i+4]);
       yv4  = vec_ld(0, &y[i+4]);
       xv8  = vec_ld(0, &x[i+8]);
       vec_st(vec_madd(avec, xv0,  yv0),  0, &y[i]);
       yv8  = vec_ld(0, &y[i+8]);
       vec_st(vec_madd(avec, xv4,  yv4),  0, &y[i+4]);
       yv12 = vec_ld(0, &y[i+12]);
       xv12 = vec_ld(0, &x[i+12]);
       vec_st(vec_madd(avec, xv8,  yv8),  0, &y[i+8]);
       vec_st(vec_madd(avec, xv12, yv12), 0, &y[i+12]);
    }
 }
```

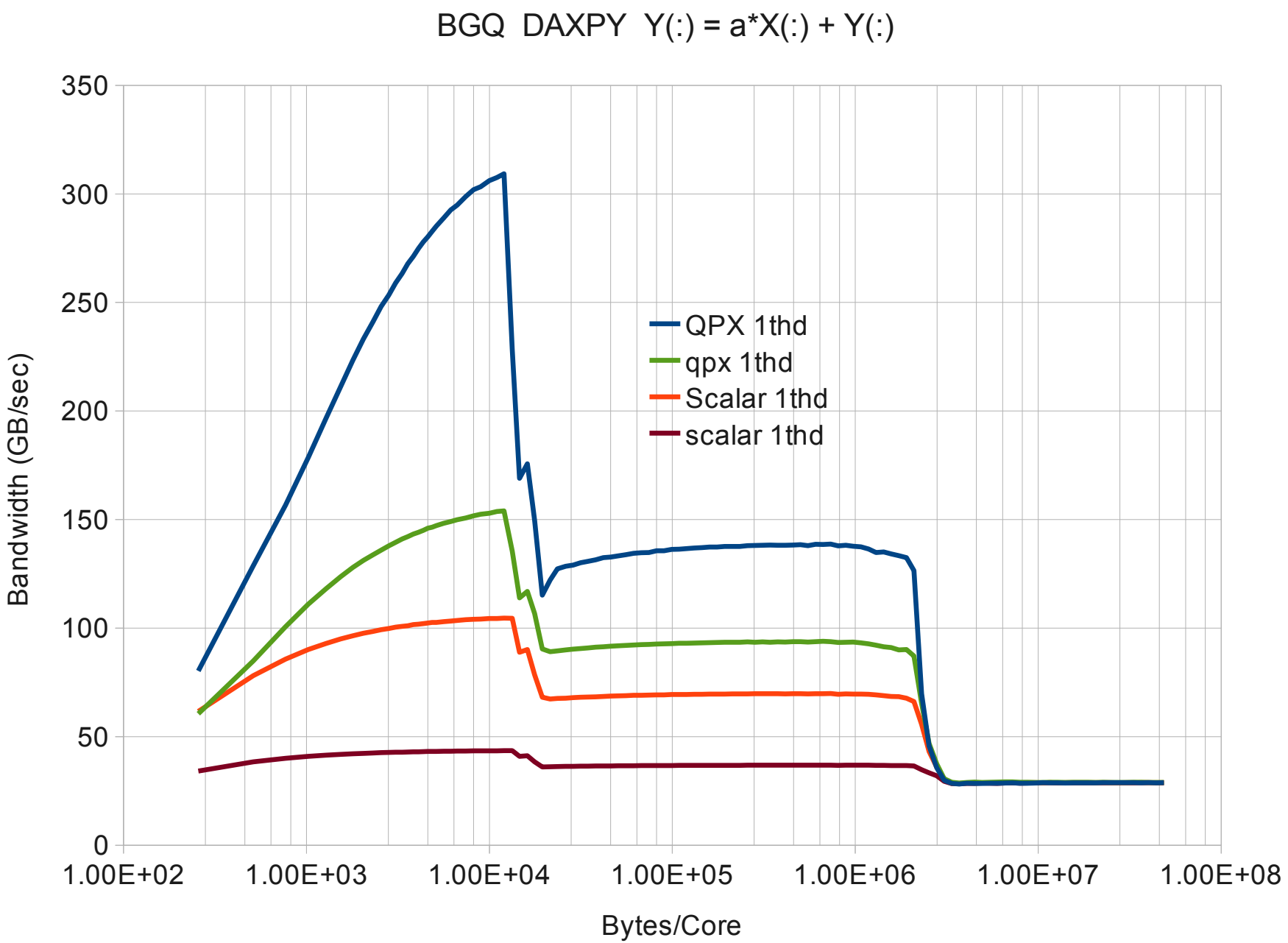
You can experiment with placement of the loads, stores, and fmadds; but the compiler will do the register assignments and instruction scheduling. Check the assembly code using **-qlist -qsource**.

Assembly code for daxpy using QPX instructions, co-aligned case :

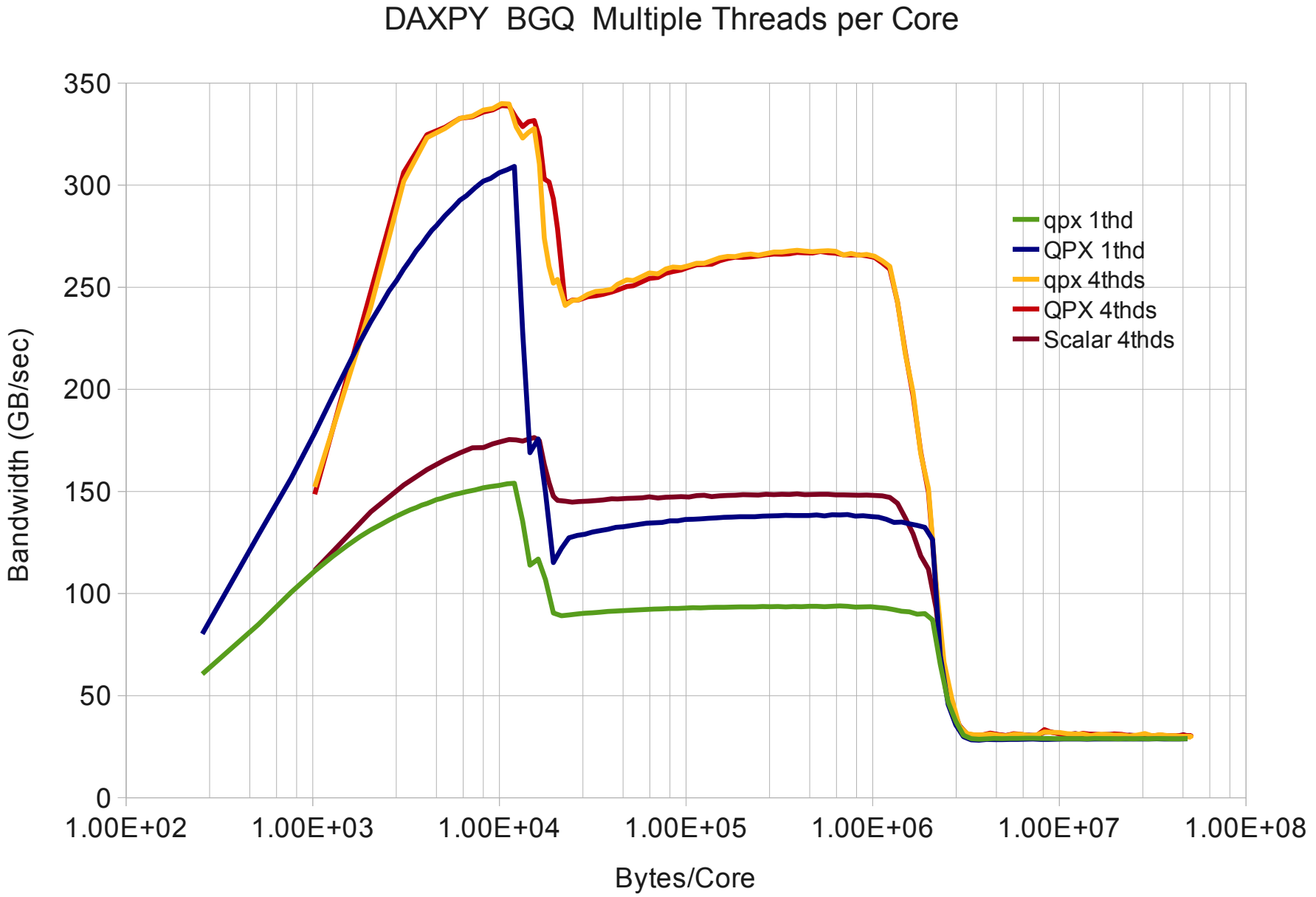
Options : -O3 -qstrict -qhot -qsimd=noauto -qdebug=nunroll **-qlist -qsource**

```
 0|                                  CL.198:
43| 00025C qvlfdx   7D46448E   1      QVLFDX    qr10=x[]0.rns3.(gr6,gr8,0)
47| 000260 qvfmadd  104118BA   1      QVFMADD   qr2=qr3,qr1,qr2,fcr
49| 000264 qvfmadd  1061293A   1      QVFMADD   qr3=qr5,qr1,qr4,fcr
52| 000268 qvfmadd  10C139BA   1      QVFMADD   qr6=qr7,qr1,qr6,fcr
53| 00026C qvfmadd  10E1427A   1      QVFMADD   qr7=qr8,qr1,qr9,fcr
45| 000270 qvlfdx   7C865C8E   1      QVLFDX    qr4=x[]0.rns3.(gr6,gr11,0)
46| 000274 qvlfdx   7CAAEC8E   1      QVLFDX    qr5=y[]0.rns2.(gr10,gr29,0)
48| 000278 qvstfdx  7C4A458E   1      QVSTFDX   y[]0.rns2.(gr10,gr8,0)=qr2
50| 00027C qvstfdx  7C6A5D8E   1      QVSTFDX   y[]0.rns2.(gr10,gr11,0)=qr3
53| 000280 qvstfdx  7CCA658E   1      QVSTFDX   y[]0.rns2.(gr10,gr12,0)=qr6
54| 000284 qvstfdux 7CEAF5CE   1      QVSTFDUX  gr10,y[]0.rns2.(gr10,gr30,0)=qr7
47| 000288 qvlfdx   7CC6648E   1      QVLFDX    qr6=x[]0.rns3.(gr6,gr12,0)
44| 00028C qvlfdx   7C6A448E   1      QVLFDX    qr3=y[]0.rns2.(gr10,gr8,0)
49| 000290 qvlfdx   7CEA648E   1      QVLFDX    qr7=y[]0.rns2.(gr10,gr12,0)
51| 000294 qvlfdx   7D0AF48E   1      QVLFDX    qr8=y[]0.rns2.(gr10,gr30,0)
52| 000298 qvlfdux  7D26F4CE   1      QVLFDUX   qr9,gr6=x[]0.rns3.(gr6,gr30,0)
 0| 00029C qvfmr    10405090   1      LRQV      qr2=qr10
 0| 0002A0 bc       4200FFBC   1      BCT       ctr=CL.198,taken=100%(100,0)
```

The loop is unrolled 4x using QPX instructions, so it processes 16 elements per loop iteration. A different set of registers is used for each fmadd, and there are instructions between load and use to hide latency.



Measurement using all 16 cores, with one thread per core.



Using all hardware threads for computation is the best way to ensure good performance on BGQ. Compiler-generated code is similar to “qpx” case.

Conclusions

You can use vector intrinsics to generate QPX code for BGQ, but great care must be taken to properly handle data alignment issues. The complexity grows significantly with multiple array references and with multi-dimensional arrays.

The suggested strategy for performance optimization on BGQ is to focus first on using enough hardware threads per core. After finding the best choice for threads-per-core, you can then worry about instruction scheduling and QPX code generation.

In many cases using all of the hardware threads for computation can get you close to fundamental performance limits.

Vector intrinsics can be very useful for certain kernels when auto-SIMDization by the compiler is not feasible. Check the compiler listings for hot loops using options -qlist -qsource, and optionally -qreport.