Blue Gene/Q Tuning Early Experience

Vitali Morozov

ALCF Performance Engineering

Argonne Leadership Computing Facility

Leap to Petascale Workshop,

Argonne National Laboratory,

May 22, 2012

Blue Gene/Q Compilers

- XL compilers
  - Fortran 77, 90, 95, 2003, 2008
  - C89, C99, C++98, Standard C++ 2003
  - IBM Extensions
  - OpenMP 3.1
- GNU compiler collection
  - version 4.4.6 – gcc, g++, gfortran
- LLVM/clang
  - Unsupported, see Hal Finkel's presentation for details
- MPI wrappers

Blue Gene/Q MPI wrappers

- Set Include and library paths
  - for MPICH2-1.4.1p1, PAMI, and MUSPI
- **gcc**: fine-grain locking, error checking, asserts
- **gcc.legacy**: coarse-grain locking, error checking, asserts
- **xl**: mpich with XL, fine-grain locking, error checking, asserts
- **xl.legacy**: mpich with XL, coarse-grain locking, error checking, asserts
- **xl.ndebug**: xl without error checking and asserts
- **xl.legacy.ndebug:** xl.legacy without error checking and asserts

Blue Gene/Q Softenv keys

```
morozov@cetuslac1:/home/morozov$ /soft/environment/softenv-1.6.2/bin/softenv
SoftEnv version 1.6.2


<skipped>


These are the macros available:

* P @default                         Default Software Stack
* P @ibm-compilers-apr2012           IBM BGP C/C++ & Fortran Compilers April 2012
* P @ibm-compilers-default
  P @ibm-compilers-feb2012           IBM BGP C/C++ & Fortran Compilers February
                                     2012 PTF

These are the keywords explicitly available:

* P +bgqdriver                       Blue Gene /Q Driver
  P +bgqdriver-V1R1M0                Driver V1R1M0

    +mpiwrapper-gcc                  gcc wrappers and toolchain
    +mpiwrapper-gcc.legacy           gcc.legacy wrappers and toolchain
    +mpiwrapper-xl                   xl wrappers and toolchain
    +mpiwrapper-xl.legacy            xl.legacy wrappers and toolchain
    +mpiwrapper-xl.legacy.ndebug     xl.legacy.ndebug wrappers and toolchain
    +mpiwrapper-xl.ndebug            xl.ndebug wrappers and toolchain
```

Blue Gene/Q Hello World

```
#include <mpi.h>
#include <stdio.h>
#include <omp.h>

int main( int argc, char **argv )
{
   int iam, nt, rank, nprocs, provided;

   MPI_Init_thread( &argc, &argv,

           MPI_THREAD_MULTIPLE, &provided  );
   if ( provided != MPI_THREAD_MULTIPLE )        MPI_Abort
( MPI_COMM_WORLD, 1 );

   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   MPI_Comm_size( MPI_COMM_WORLD, &nprocs);

#pragma omp parallel default(shared) private( iam, nt )
{
   iam = omp_get_thread_num();
   nt  = omp_get_num_threads();

   if ( iam == 0 )
   {
      printf( "Rank %d/%d thread %d/%d Hello world!\n",

            rank, nprocs, iam, nt );
   }
}

   MPI_Finalize();

   return 0;
}
```

```
/bgsys/drivers/ppcfloor/comm/xl/bin/mpixlc_r  \

            -O3 -qnohot -qsimd=noauto -qsmp=omp:noauto \

           -o hello hello.c
```

```
#!/bin/bash -x

export PROG=hello
export NODES=128
export RANKS_PER_NODE=2

export NPROCS=$((NODES*RANKS_PER_NODE))
export \

OUTPUT=N${NODES}_R${RANKS_`PER_NODE}_hello_mpi_${NPROCS}_ranks

rm -f core.* ${OUTPUT}.cobaltlog ${OUTPUT}.error ${OUTPUT}.output

qsub -A Performance -n $NODES --mode c${RANKS_PER_NODE} -t 0:10:00 \

--env BG_SHAREDMEMSIZE=32:OMP_NUM_THREADS=16 -O $OUTPUT \

$PROG
```

```
Rank 178/256 thread 0/16 Hello world!
Rank 151/256 thread 0/16 Hello world!
Rank 143/256 thread 0/16 Hello world!
Rank 251/256 thread 0/16 Hello world!
Rank 179/256 thread 0/16 Hello world!
Rank 135/256 thread 0/16 Hello world!
Rank 79/256 thread 0/16 Hello world!
...
```

IBM XL Compilers - general advice

-O0 -qsmp=omp:noopts:noauto -qfloat=nofold

- removes obviously redundant code, suppress folding FP instructions, preserves debugging information

-O2 -qsmp=omp:noopts:noauto

- as -O0, combines common expression, propagates constants, removes dead code, optimizes variable usage, improves instruction flow, loop unrolling, moving loop invariants…

-O3 -qsmp=omp:noauto

- as -O2, improved loop scheduling, HOT for loops, inlining within compilation unit, merges adjacent memory operations …

DAXPY: XL Compilers at O0, O2

-O0

```
40:   ld      r3,232(r1)
44:   lwa     r0,128(r1)
48:   rldicr  r4,r0,3,60
4c:   lfdx    f0,r3,r4
50:   lfd     f1,136(r1)
54:   ld      r5,216(r1)
58:   lfdx    f2,r5,r4
5c:   fmul    f3,f1,f2
60:   fmadd   f0,f1,f2,f0
64:   stfdx   f0,r3,r4
68:   lwa     r3,128(r1)
6c:   addi    r0,r3,1
70:   stw     r0,128(r1)
74:   lwa     r0,128(r1)
78:   lwa     r3,208(r1)
7c:   cmpw    r0,r3
80:   blt     40 <.daxpy_c+0x40>
```

-O2

```
6c:   fmadd   f4,f0,f1,f2
70:   lfd     f2,24(r6)
74:   fmadd   f5,f0,f6,f3
78:   lfd     f1,8(r4)
7c:   lfd     f3,32(r6)
80:   lfdu    f6,16(r4)
84:   addi    r3,r3,2
88:   stfd    f4,8(r6)
8c:   stfdu   f5,16(r6)
90:   bdnz    6c <.daxpy_c+0x6c>
```

```
int daxpy( int N, double *X,
double *a, double *Y )
{
   int i;
   double b = *a;

#pragma disjoint (*X, *Y )
   for ( i = 0; i < N; i++ )
      Y[i] = Y[i] + b * X[i] ;

   return i;
}
```

Line 50: b is loaded every iteration to f1

Line 48: the offset is calculated for X[], Y[]

Lines 40, 6c, 70: index i is kept on stack

Lines 48, 6c, 7c, 80: loop is expensively organized

Lines 6c, 74: b is kept in f0 as temp

Lines 70, 78, 7c, 80, 88, 8c: optimized memory accesses

Line 84: index i is kept locally in r3

Lines 48, 90: loop is unrolled 2

DAXPY: XL Compilers at high opt levels

-O2

```
6c:   fmadd   f4,f0,f1,f2
70:   lfd     f2,24(r6)
74:   fmadd   f5,f0,f6,f3
78:   lfd     f1,8(r4)
7c:   lfd     f3,32(r6)
80:   lfdu    f6,16(r4)
84:   addi    r3,r3,2
88:   stfd    f4,8(r6)
8c:   stfdu   f5,16(r6)
90:   bdnz    6c <.daxpy_c+0x6c>
```

Lines 6c, 74: b is kept in f0

Optimized memory accesses

Line 84: index i is stored locally in r3

Lines 48, 90: loop is unrolled 2, uses CTR

-O3 -qnohot -qsimd=noauto

-O3 -qnohot –qsimd=auto

```
a8:   lfd     f5,40(r6)
ac:   fmadd   f2,f0,f1,f2
b0:   stfd    f6,16(r6)
b4:   lfd     f1,8(r4)
b8:   lfd     f6,48(r6)
bc:   fmadd   f3,f0,f4,f3
c0:   stfd    f2,24(r6)
c4:   lfd     f4,16(r4)
c8:   lfd     f2,56(r6)
cc:   fmadd   f5,f0,f1,f5
d0:   addi    r3,r3,4
d4:   stfdu   f3,32(r6)
d8:   lfd     f1,24(r4)
dc:   lfd     f3,32(r6)
e0:   fmadd   f6,f0,f4,f6
e4:   stfd    f5,8(r6)
e8:   lfdu    f4,32(r4)
ec:   bdnz    a8 <.daxpy_c+0xa8>
```

Lines ac, bc, cc, e0: b is kept in f0

Same Optimized memory accesses

Line d0: index i is stored locally in r3

Lines d0, ec: loop is unrolled 4, uses CTR

-O3 -qhot=level=0 -qsimd=auto

-O3 -qhot=level=1 -qsimd=auto

```
 fc:   qvlfdx  q6,r8,r28
100:   qvfmadd q2,q1,q3,q2
104:   qvstfdx q7,r8,r11
108:   qvlfdx  q3,r9,r10
10c:   qvlfdx  q7,r8,r29
110:   qvfmadd q4,q1,q5,q4
114:   qvstfdx q2,r8,r12
118:   qvlfdx  q5,r9,r11
11c:   qvlfdx  q2,r8,r30
120:   qvfmadd q6,q1,q3,q6
124:   qvstfdux q4,r8,r27
128:   qvlfdx  q3,r9,r12
12c:   qvlfdx  q4,r8,r27
130:   qvfmadd q7,q1,q5,q7
134:   qvstfdx q6,r8,r10
138:   qvlfdux q5,r9,r27
13c:   bdnz    fc <.daxpy_c+0xfc>
```

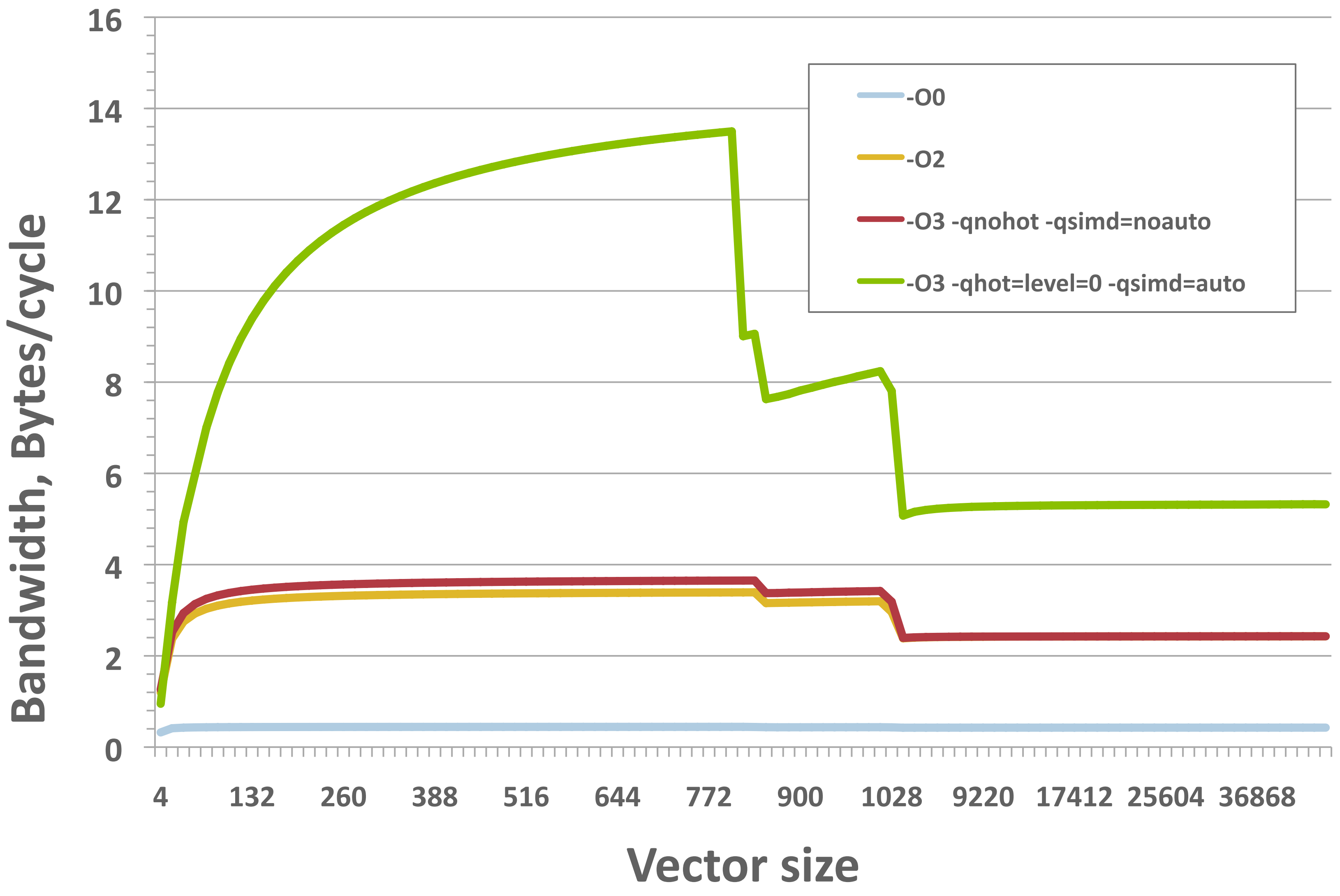
Lines 100,110,120,130: b is kept in q1

QPX Optimized memory accesses

Index I is not stored

Loop is unrolled 4, entirely in CTR

DAXPY: XL Compilers at high opt levels



Advanced features: QPX intrinsics

Fortran

TYPE: VECTOR(REAL(8)) :: A;

Initialization and individual elements:

```
VECTOR( REAL(8) ) :: V
REAL(8) :: C(4)
EQUIVALENCE( V, C )
C(1) = 1.E0; C(2) = 2.E0; C(3)=3.E0; C(4) = 4.E0
! V = ( 1.E0, 2.E0, 3.E0, 4.E0 )
```

Loads and stores

```
INTEGER i; REAL(8) X[N], Y[N], VECTOR( REAL(8) ) V
V = VEC_LD( i, X )
! EA = X+i, V = (EA, EA[1], EA[2], EA[3])
CALL( V, i, Y )
! EA = Y+i, EA[0]=C(1), EA[1]=C(2), EA[2]=C(3), EA[3]=C(4)
```

Unary operations

VEC_ABS, VEC_NEG, VEC_RE, VEQ_RSQRTE

Binary operations

VEC_ADD, VEC_SUB, VEC_MUL, VEC_SWDIV

Multiply-add operations

VEC_MADD, VEC_MSUB, VEC_NMADD, VEC_NMSUB

Special functions

VEC_SEL, VEC_CMPGT, VEC_CMPEQ, VEC_CTID, VEC_FLOOR

C/C++

TYPE: vector4double A;

Initialization:

```
(vector4double)(c1); /* (c1, c1, c1, c1) */
(vector4double)(c1, c2, c3, c4); /* (c1, c2, c3, c4) */
(vector4double) {c1} /* (c1, 0.0, 0.0, 0.0 ) */
```

Individual elements:

```
vector4double v = (vector4double)(1.0,2.0,3.0,4.0);
double d0 = v[0], d1 = v[1], d2 = v[2], d3 = v[3];
```

Loads and stores

```
long i; double *x, *y; vector4double v;
v = vec_ld( i,x ); /* EA = i+x, v = ( *EA, *(EA+8), *(EA+16), *EA+24) ) */
vec_st( v, i, y );
/* EA = i+y; *EA=v[0], *(EA+8)=v[1], *(EA+16)=v[2], *(EA+24)=v[3] */
```

Unary operations

vec_abs, vec_neg, vec_re, vec_rsqrte

Binary operations

vec_add, vec_sub, vec_mul, vec_swdiv

Multiply-add operations

vec_madd, vec_msub, vec_nmadd, vec_nmsub

Special functions

vec_sel, vec_cmpgt, vec_cmpeq, vec_ctid, vec_floor

Advanced features: MASS functions

Available for Fortran and C/C++, single and double precision

Scalar version

acos, acosh, cbrt, erf, exp, expm1, log, pow, rsqrt, sincos, sqrt, …

Vector version

vacos, vacosh, vcbrt, verf, vexp, vexpm1, vlog, vpow, …

SIMD version

operates on VECTOR(REAL(8)) / vector4double types

acosd4, acos4d4, cbrtd4, erfd4, expd4, exmp1d4, logd4, …

Example study – exp function

```
#include <math.h>
int exp_c( int n, float *r2, float m_d, float m_h, float *res0, float *res1 ) {
    for ( int i = 0; i < n; i++ )  {
        res0[i] = exp( -m_d * r2[i] );
        res1[i] = exp( -m_h * r2[i] );
    }
    return 0;
}
```

Regular C: 20 lines of code

```
#include <massv.h>
int exp_v( int n, float *r2, float m_d, float m_h, float *res0, float *res1 ) {
    int I;  float p0[n], p1[n];

    for ( i = 0; i < n; i++ )  {
        p0[i] = -m_d * r2[i];
        p1[i] = -m_h * r2[i];
    }
    vsexp( res0, p0, &n );
    vsexp( res1, p1, &n );
    return 0;
}
```

C with MASSV: 25 lines of code

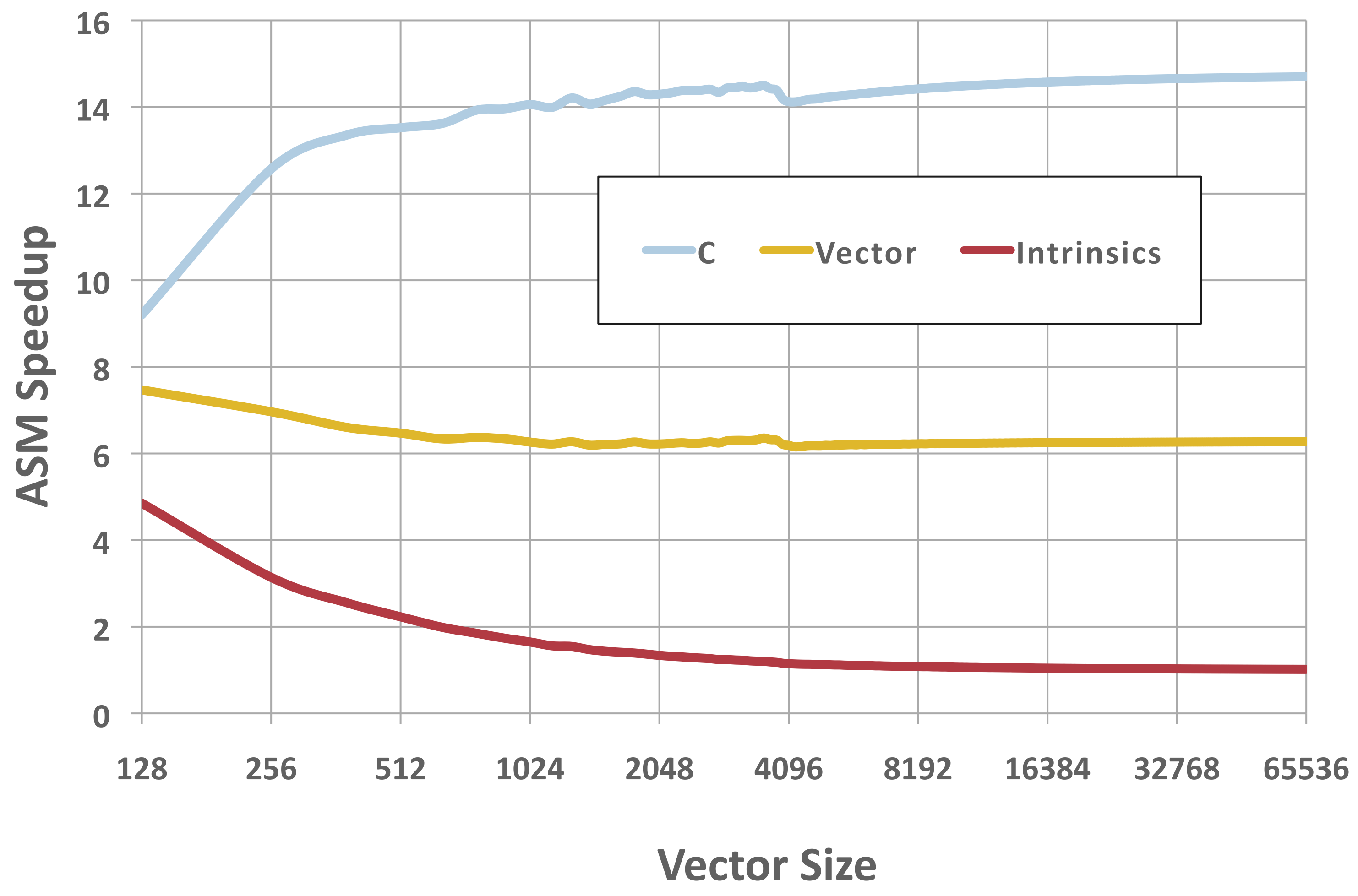
```
vector4double x0, x1, x2, x3, x4, x5, x6, x7, f0, f1, g0, g1, h0, h1;
vector4double i0, i1, j0, j1, k0, k1, l0, l1, m0, m1, s0, s1, s2, s3;

for ( i = 0, j = 0; i < n; i = i + 16, j = j + 64 ) {
        __dcbt( (void *)&r2[i+32] );

        s0  = vec_ld( j    , r2 ); … s3  = vec_ld( j+48, r2 );

        x0  = vec_mul( mmd, s0 ); … x7  = vec_mul( mmh, s3 );
        f0 = vec_madd( x0, a5, a4 ); … m0 = vec_madd( x7, a5, a4 );
        f1 = vec_madd( x0, f0, a3 ); … m1 = vec_madd( x7, m0, a3 );
        f0 = vec_madd( x0, f1, a2 ); … m0 = vec_madd( x7, m1, a2 );
        f1 = vec_madd( x0, f0, a1 ); … m1 = vec_madd( x7, m0, a1 );
        f0 = vec_madd( x0, f1, a0 ); … m0 = vec_madd( x7, m1, a0 );
        f1 = vec_mul( f0, f0 ); … m1 = vec_mul( m0, m0 );
        f0 = vec_mul( f1, f1 ); … m0 = vec_mul( m1, m1 );
        f1 = vec_re( f0 ); … m1 = vec_re( m0 );
        f0 = vec_nmsub( f0, f1, a0 ); … m0 = vec_nmsub( m0, m1, a0 );
        f0 = vec_madd( f1, f0, f1 ); … m0 = vec_madd( m1, m0, m1 );

        vec_st( f0, j    , res0 ); … vec_st( m0, j+48, res1 );
    }
```

C with vector intrinsics: 150 lines of code

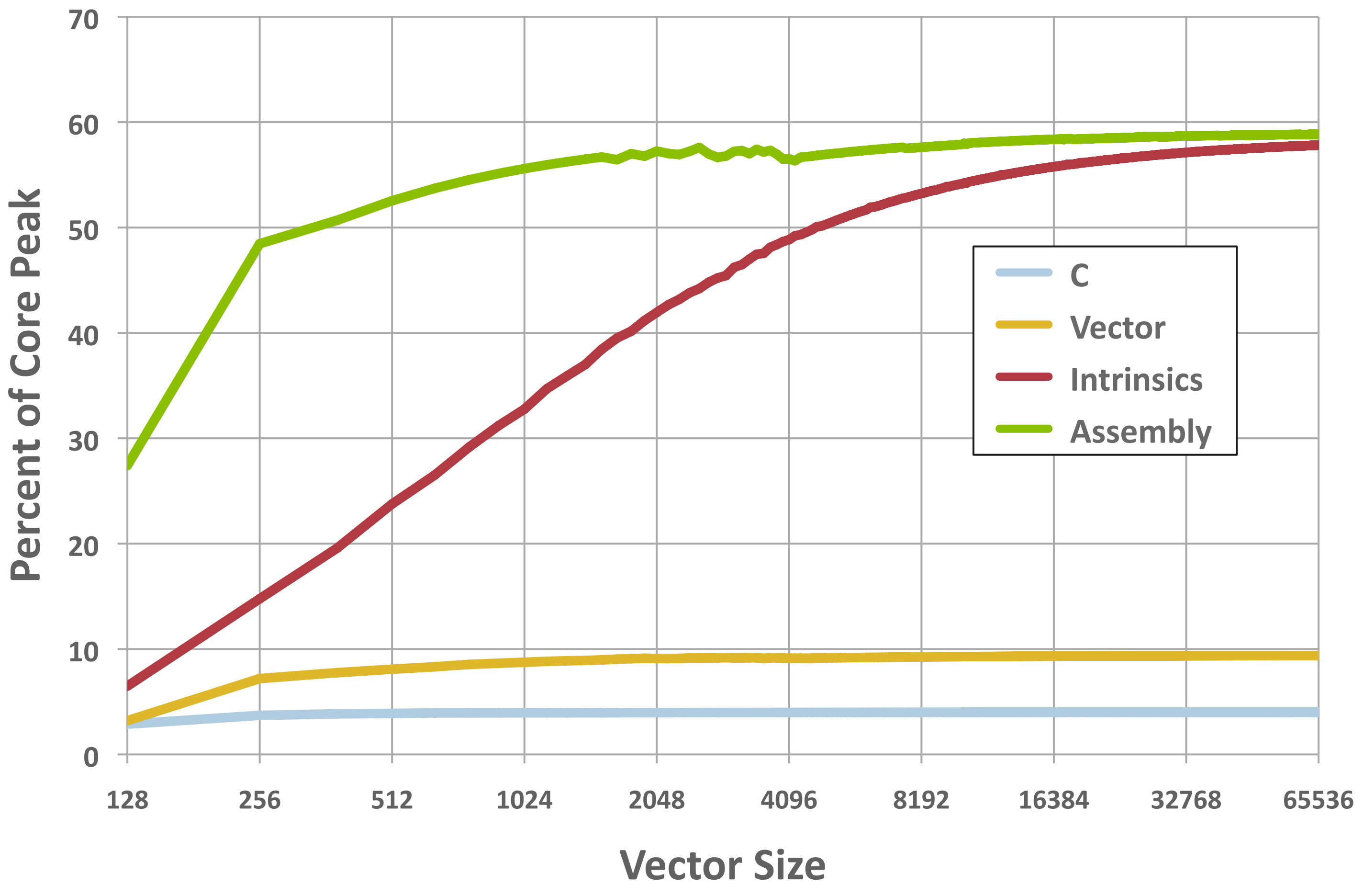
```
LOOP0:
        dcbt    4, 6
        qvlfsx  11,4,11
        ....
        qvlfsux 15,4, 5
        qvfmul  12,11, 1    # x0_unroll1 = (double)r2[i] * m_d;
        ....
        qvfmul  24,15, 2    # x7_unroll1 = (double)r2[i+3] * m_h;
        qvfmadd 11,12, 8,7  # f0 = a4 + x0 * a5;
        ....
        qvfmadd 23,24, 8,7  # f6 = a4 + x0 * a5;
        ....
        qvfmadd 23,24,23,6  # f6 = a3 + x0 * f6;
        ....
        qvfmadd 23,24,23,5  # f6 = a2 + x0 * f6;
        ....
        qvfmadd 23,24,23,4  # f6 = a1 + x0 * f6;
        ....
        qvfmadd 23,24,23,3  # f6 = a0 + x0 * f6;
        qvfmul  11,11,11    # f0 = f0 * f0;
        ....
        qvfmul  23,23,23    # f6 = f6 * f6;
        qvfmul  11,11,11    # f0 = f0 * f0;
        ....
        qvfmul  23,23,23    # f6 = f6 * f6;
        qvfre   12,11       # f1 = __fre( f0 );
        ....
        qvfre   24,23       # f7 = __fre( f6 );
        bc      16, 0, LOOP0
```

Assembly: 245 lines of code

Exp function - speedup



Exp function – Single core performance



A2 Core peak: 12.8 GFlops

Advanced features: Prefetch

L1P prefetch unit in A2 core

- 32 lines, 128 bytes each (4KB per core)
- Approximately 10 streams can simultaneously be prefetched
- Running 4 threads per core may lead to L1P contention

L1P prefetch buffer can be controlled

- #include <spi/include/l1p/sprefetch.h>
- L1P_SetStreamPolicy(L1P_stream_optimistic);

Prefetch policies

- L1P_confirmed_or_dcbt (default): prefetch only if increasing address detected or dcbt issued (default depth 3)
- L1P_stream_confirmed: prefetch only if increasing address detected
- L1P_stream_optimistic: always prefetch the next line
- L1P_stream_diasable: no stream prefetching detected

List prefetch

- L1P_PatternConfigure(), L1P_PatternStart(), L1P_PatternStop(), L1P_PatternUnconfigure()

Advanced features: TM and SE

Transactional memory

- Critical set of instructions are grouped together
- Each block is considered atomic (does not see other memory transactions)
- Conflicts are resolved by hardware (either all accepted or all rejected)
- L2 multi versioning is used
- Suggested use: rare store conflicts between threads
- #pragma tm_atomic{} or !TM$ tm_atomic … !TM$ end tm_atomic; -qtm
- Hard to justify, high overhead

Speculative execution

- Runs sequential portion of code in parallel
- Automatic resolution of conflicts in hardware
- L2 multi versioning is used
- Suggested use: rare store conflicts between threads
- No examples yet

Advanced features: Special functions

How to get memory utilization (and available memory)?

```
#include <spi/include/kernel/memory.h>
uint64_t heapmax, stack;
Kernel_GetMemorySize( KERNEL_MEMSIZE_HEAPMAX, &heapmax);
Kernel_GetMemorySize( KERNEL_MEMSIZE_STACKAVAIL, &stack);
```

How to obtain timebase register?

```
#include <hwi/include/bbqc/A2_inlines.h>
uint64_t t1;
t1 = GetTimeBase();
```

How to obtain FlopRate/CacheUtilization/MemoryBandwidth/MPI

~morozov/HPM/lib/libmpihpm.a or libmpihpm_smp.a

Memory footprint

MPI / node	Memory/rank	% of node memory
1	16207 MB	98.9%
2	8105 MB	98.9%
4	4036 MB	98.5%
8	2017 MB	98.5%
16	984 MB	96.1%
32	475 MB	92.8%
64	235 MB	91.8%

MPI only code

MPI / node	OMP threads	Memory/ rank	% of node memory
1	64	15725 MB	96.0%
2	32	7869 MB	96.1%
4	16	3921 MB	95.8%
8	8	1963 MB	95.8%
16	4	961 MB	93.8%
32	2	467 MB	91.2%
64	1	189 MB	73.8%

MPI-OMP code, 8MB stack size

Will vary depending on

- program text and data segments
- environment variables
- number of MPI ranks per node
- number of threads running per rank

Example of MPIHPM printout - counters

```
======================================================================
Hardware counter report for BGQ  - sum for node <0,0,0,0,0>.
cores in use = 16, active threads per core = 4.
======================================================================
----------------------------------------------------------------
mpiAll, call count = 1, avg cycles = 1614382912420, max cycles = 1614384660429 :
  -- Counter values summed over processes on this node ----
0     363337714240   Committed Load Misses
0    4503495333619   Committed Cacheable Loads
0     331334183955   L1p miss
0   15401665830287   All XU Instruction Completions
0    1875748056796   All AXU Instruction Completions
0    2580440435539   FP Operations Group 1
  -- L2 counters (shared for the node) -----------------
100  2572275654042  L2 Hits
100    28480148014  L2 Misses
100    22718245965  L2 lines loaded from main memory
100    12745048391  L2 lines stored to   main memory


Derived metrics for code block "mpiAll" averaged over process(es) on node <0,0,0,0,0>:
Instruction mix:  FPU = 10.86 %,  FXU = 89.14 %
Instructions per cycle completed per core = 0.6689
Per cent of max issue rate per core = 59.63 %
Total weighted GFlops for this node = 2.557
Loads that hit in L1 d-cache =  91.93 %
                  L1P buffer =   0.71 %
                  L2 cache   =   6.72 %
                  DDR        =   0.63 %
DDR traffic for the node: ld = 1.801, st = 1.011, total = 2.812 (Bytes/cycle)
```

Example of MPIHPM printout – MPI profile

```
Data for MPI rank 0 of 8192
Times and statistics from MPI_Init() to MPI_Finalize().
-----------------------------------------------------------------
MPI Routine                    #calls     avg. bytes      time(sec)
-----------------------------------------------------------------
MPI_Comm_size                    2352            0.0          0.009
MPI_Comm_rank                    6553            0.0          0.007
MPI_Ssend                       19438        67546.6          5.976
MPI_Isend                          90            4.0          0.001
MPI_Recv                          100           24.0          0.002
MPI_Irecv                       21272        81110.0          0.053
MPI_Sendrecv                        4            4.0          0.000
MPI_Sendrecv_replace           294912           18.0         23.156
MPI_Wait                          106            0.0          0.001
MPI_Waitall                      3464            0.0          2.319
MPI_Bcast                        1145         2232.5          0.127
MPI_Barrier                     11265            0.0        158.969
MPI_Gather                          1            4.0          0.000
MPI_Scan                            6            4.7          0.001
MPI_Allgather                     251            4.0          0.342
MPI_Reduce                        113           94.7          3.043
MPI_Allreduce                    3112         3337.9        271.763
MPI_Alltoall                       43            4.0          1.280
MPI_Alltoallv                       5          302.4          0.252
MPI_File_close                      5            0.0          0.174
MPI_File_open                       5            0.0          1.344
MPI_File_set_view                 160            0.0          0.126
MPI_File_sync                      26            0.0          0.975
MPI_File_write_at                  84         2303.1          0.417
MPI_File_write_at_all              80        52459.8        159.992
-----------------------------------------------------------------
total communication time = 467.301 seconds.
total elapsed time       = 1008.990 seconds.
heap memory used         = 118.492 MBytes.

total MPI-IO time = 163.028 seconds.
```