Compiling for Power6

Damien Bonaventure
IBM Toronto Lab
damien@ca.ibm.com

© 2009 IBM Corporation

Agenda

- Power6 specific tuning
- Interesting Optimizations
- VMX Exploitation
- Optimization Reports and Examples

© 2009 IBM Corporation

Architecture and Tuning

- Two of the most important compilation flags when targeting Power6
- The architecture flag (e.g -qarch=pwr6) specifies what instructions the compiler is allowed to use.
- The tuning flag (e.g -qtune=pwr6) specifies the microarchitecture for which the compiler will tune the code
- Code compiled for a specific architecture can only be run on that specific machine, otherwise an illegal instruction trap may result.
- Running a binary compiled with an incorrect -qtune sub-option will not cause runtime errors, but may cause sub-optimal performance.

© 2009 IBM Corporation

Power5 / Power6 differences (summary)

- Power6 executes instructions in order
  - Helps to reach high clock rate, but potentially more stalls
- Store Queue has to be managed to prevent load/store stalls
  - Careful arrangement of stores can get the bandwidth back in
- Power6 does not do store forwarding
  - High cost for store and reload
- Fixed point multiplies are done in the floating point unit
  - Extra cost can be mitigated by grouping them
- Compiler technology is key to extracting performance from the P6

© 2009 IBM Corporation

Compiling for Power6

- New -qarch suboptions for Power6:

-qarch=pwr6e	- Generate all P6 instructions
-qarch=pwr6	- Generate all except for raw-mode only instructions

- Some P6 instructions are only available when the P6 is in “raw mode”

mffgpr, mftgpr:	move between float and integer registers
stfdp:	store float double pair

- Using -qarch=pwr6 will ensure that your binaries continue to run on upcoming processors, while -qarch=pwr6e may provide additional performance.

© 2009 IBM Corporation

Ganging Fixed Point Multiplies

- Integer multiplies need to be handled specially on the Power6
  - These execute in the floating point unit (FPU)
  - Each multiply takes 17 cycles, and stalls dispatch for 1 cycle, but this can be mitigated if they are next to each other within a group
- Adjacent multiplies take an additional 2 cycles only
  - mullw,mullw,mullw,add,sub = min of 23 (17 + 2 + 2 + 2) cycles
  - mullw,add,mullw,sub,mullw = min of 51 (17 + 2 + 17 + 2 + 17) cycles
- Group independent multiplies together in groups that are as long as possible.

© 2009 IBM Corporation

Store Queue Management (cont.)

- Use 1 store float double pair (stfdp) instruction to replace 2 store float double (stfd) instructions, when possible
  - A stfdp takes up just 1 STQ entry
  - Rules: (1) the 2 stfd's store consecutive registers (2) the address of the stfdp must be 16-byte aligned
- Currently done in function prologues only

-qarch=pwr5

```
    | 000000                              PDEF   pov::pov_shellout(SHELLTYPE)
 466|                                     PROC
   0| 006A20 mfspr     7C0802A6   1       LFLR       gr0=lr
   0| 006A24 stfd      DBE1FFF8   1       STFL       #stack(gr1,-8)=fp31
   0| 006A28 stfd      DBC1FFF0   1       STFL       #stack(gr1,-16)=fp30
```

-qarch=pwr6

```
    | 000000                              PDEF       pov::pov_shellout(SHELLTYPE)
 466|                                     PROC
   0| 006A20 mfspr     7C0802A6   1       LFLR       gr0=lr
   0| 006A24 stfdp     F7C1FFF0   1       STFQ       #stack(gr1,-16)=fp30,fp31
```


© 2009 IBM Corporation

Store/Re-Load Cost

- Power6 has no store forwarding: relatively high cost for loading from a storage location that was recently stored into
  - e.g. type conversions, parameter passing
- Two new instructions are available to transfer data between floating point and general purpose registers.
  mffgpr, mftgpr
- These will be generated automatically by the compiler for type conversions only if -qarch=pwr6e is used
  - These insns have not been added to the PPC architecture
  - Builtins available for user code

© 2009 IBM Corporation

Other Power6 Specific Changes

- Pair long running instructions (fdiv, fsqrt) so that they are in the same dispatch group. They execute in parallel. If they are in different dispatch groups, they may execute serially

- Use special group-ending NOP instruction to force early termination of dispatch groups, when needed
  - ORI 1,1,0 special on Power6, regular NOP on Power5

- P6 aware loop unrolling:
  - Unroll loops enough to cover the 5 cycle redirect penalty
  - Unroll to find store grouping opportunities

© 2009 IBM Corporation

Prefetch Enhancements for P6

- Exploit the 16 streams available on Power6 (only 8 on P4/P5)
- Support new store stream prefetch
  - Compiler automatically determines when prefetch insertion is profitable
  - and inserts calls to prefetch stores
- Exploit both L1 and L2 touch instructions
  - Compiler automatically determines if data is more likely to be needed in L1
  - or L2 and inserts the prefetch required.
- Exploit prefetch depth control
  - Compiler tries to fetch further ahead
  - Tricky to get right, may compete with immediately needed lines in L1

© 2009 IBM Corporation

VMX Exploitation

- Support the VMX unit on P6
  - User directed via explicit intrinsic programming
  - Automatically generate VMX code by loop analysis, and efficiently handling alignment constraints
  - Both of the above are fully optimized and scheduled for P6
- Symbolic debug support at opt0
- SIMDization reports to tell users which loops were SIMDized, which were not, and what prevented SIMDization
- More on this later …

© 2009 IBM Corporation

Balanced Tuning (-qtune=balanced)

- This is the default tuning target since XL C/C++ V9.0 and XLF 11.1
- We try to balance the competing optimization priorities of Power5 and Power6
  - Insert special group ending NOP when required, on P5 this acts just like a regular NOP
  - Have “loads only” and “stores only” groups when possible
  - Group fixed point multiplies together in a sequence

© 2009 IBM Corporation

Interesting Optimizations

© 2009 IBM Corporation

Matmul Idiom Recognition

- We used to shipped some matmul functions in libxlopt
  - Users could insert explicit calls to these routines in their code
  - The libxlopt versions would automatically call the equivalent ESSL functions if ESSL was installed on the system.
- In V9/11.1, the compiler recognizes some limited loop nest patterns as matrix multiplies and automatically generates calls to matmul functions in libxlopt or ESSL.
  - sgemm, dgemm
- The loop nest can be interchanged in any order, example next slide

© 2009 IBM Corporation

Matmul Idiom Recognition (cont.)

```
subroutine foo(M,N,L,A,B,C,D)

   ...
   do i=1,M
     do j=1,N
       do k=1,L
          C(i,j) = C(i,j) + A(i,k)*B(k,j)
      end do
     end do
   end do

   do i=1,M
     do k=1,L
      do j=1,N
         D(i,j) = D(i,j) + A(i,k)*B(k,j)
      end do
     end do
   end do

   return
   end
```

```
.foo:
        mfspr   r0,LR
        stfd    fp31,-8(SP)
        stfd    fp30,-16(SP)
        st      r31,-20(SP)
        ....
        lfs     fp31,0(r11)
        stfd    fp31,136(SP)
        stfd    fp31,144(SP)
        bl      .dgemm{PR}
        ....
        st      r26,68(SP)
        st      r0,72(SP)
        l       r9,172(SP)
        bl      .dgemm{PR}
        oril    r0,r0,0x0000
        l       r12,248(SP)
        lfd     fp31,232(SP)
        ...
```


© 2009 IBM Corporation

MASS Enhancements

- Mathematical Acceleration SubSystem is a library of highly tuned, machine specific, mathematical functions available for download from IBM
  - Contains both scalar and vector versions of many (mostly trig.) functions
  - Trades off very limited accuracy for greater speed
  - The compiler tries to automatically vectorize scalar math functions and generate calls to the MASS vector routines in libxlopt
  - Failing that, it tries to inline the scalar MASS routines
  - Failing that, it generates calls to the scalar routines instead of those in libm

- New Power6 tuned library with 60+ routines.
  - single and double precision versions of:
    - sin(), cos(), log(), exp(), acos() etc..
- More info: http://www-306.ibm.com/software/awdtools/mass/

© 2009 IBM Corporation

MASS example

```
subroutine mass_example (a,b)
   real     a(100), b(100)
   integer          i

   do i = 1, 100
      a(i) = sin(b(i))
   enddo;
end subroutine mass_example
```

-O3 –qhot –qarch=pwr6

```
SUBROUTINE mass_example (a, b)
  @NumElements0 = int(100)
    CALL __vssin_P6 (a, b, &@NumElements0)
  RETURN
END SUBROUTINE mass_example
```

Aliasing prevents vectorization:

```
void c_example(float *a, float *b)
{
    for (int i=0; i < 100; i++)
    {
       a[i] = sin(b[i]);
       b[i] = (float) i;
    }
}
```

```
void c_example(float *a, float *b)
{
    @CIV0 = 0;
    do {
       a[@CIV0] = __xl_sin(b[@CIV0]);
       b[@CIV0] = (float) @CIV0;
       @CIV0 = @CIV0 + 1;
    } while ((unsigned) @CIV0 < 100u);
    return;
}
```


© 2009 IBM Corporation

VMX Exploitation

© 2009 IBM Corporation

VMX exploitation

- **User directed**
  - Vector data types and routines available for C, C++ and Fortran
  - Programmer manually re-writes program, carefully adhering to alignment constraints
- **Automatic SIMD Vectorization (SIMDization)**
  - The compiler automatically identifies parallel operations in the scalar code and generates SIMD versions of them.
  - The compiler performs all analysis and transformations necessary
  - to fulfill alignment constraints.
  - Programmer assistance may improve generated code

© 2009 IBM Corporation

User directed VMX

- Data types:

  C/C++: vector float, vector int, vector unsigned char

  Fortran: vector(real(4)), vector (integer), vector(unsigned(1))

- VMX intrinsics

  vec_add(), vec_sub(), vec_ld(), vec_st(), etc.

  The Fortran VMX intrinsic names are the same as those of C/C++

- Symbolic debug (gdb, dbx) support at no-opt.

- Fully optimized at -O2 and above with suite of classical optimizations such as dead code removal, loop invariant code motion, software pipelining and Power6 instruction scheduling

© 2009 IBM Corporation

Example: Fortran VMX intrinsics

```
subroutine xlf_madd (a,b,c,x)
    vector(real(4))     a(100), b(100), c(100)
    vector(real(4))     x(100)
    integer             i

    do i = 1, 100
      x(i) = vec_madd(a(i), b(i), c(i))
    enddo;
end subroutine xlf_madd
```

```
                CL.5:
VLQ       vr0=a[](gr4,gr7,0)
VLQ       vr1=b[](gr5,gr7,0)
VLQ       vr2=c[](gr6,gr7,0)
VMADDFP   vr0=vr0-vr2,nj
VSTQ      x[](gr3,gr7,0)=vr0
AI        gr7=gr7,16
BCT       ctr=CL.5,,100,0
```

Compile options:

xlf -O2 -qarch=pwr6 -qlist -c

Additionally, compiling on AIX requires -qenablevmx

© 2009 IBM Corporation

Automatic SIMDization

- Automatically generate VMX code
  - Compiler attempts to extract parallelism from a variety of sources:
    - Largely within innermost loops
    - Can also handle some non-loop statements
  - Handles the various constraints in the source as well as hardware
  - The transformation framework is mostly machine independent
    Currently targeting: CELL, BlueGene, PPC970 and P6
  - Handle potentially complex data alignment problems, automatically
  - Large set of source assertions available to the programmer
  - Provides detailed information on SIMDized loops

© 2009 IBM Corporation

Successful Simdization

Extract Parallelism

loop level

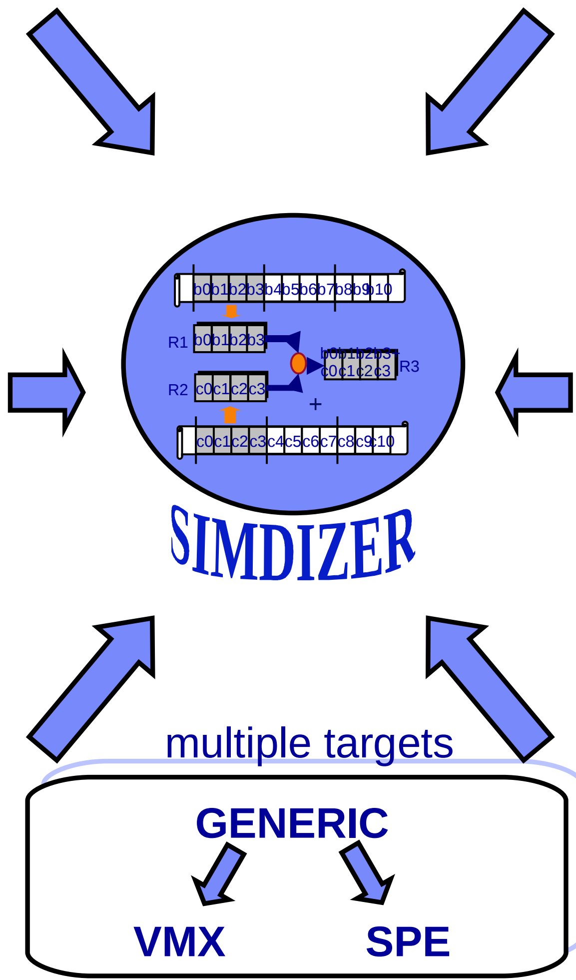
```
for (i=0; i<256; i++)
  a[i] =
```

basic-block level

```
a[i+0] =
a[i+1] =
a[i+2] =
a[i+3] =
```

entire short loop

```
for (i=0; i<8; i++)
  a[i] =
```



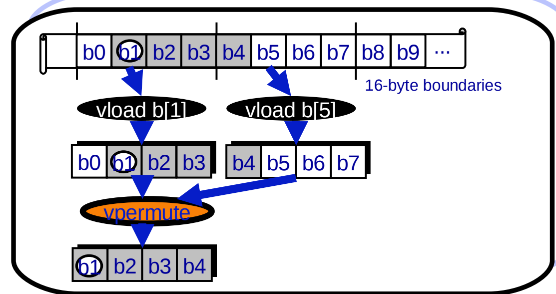
multiple targets

GENERIC

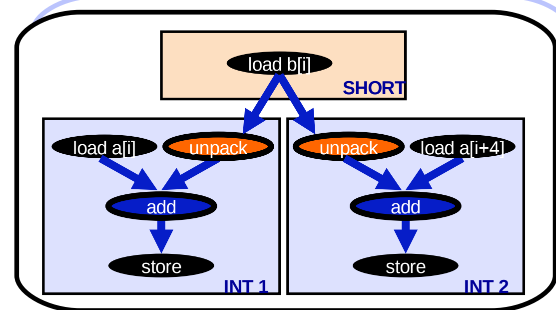
VMX **SPE**

Satisfy Constraints

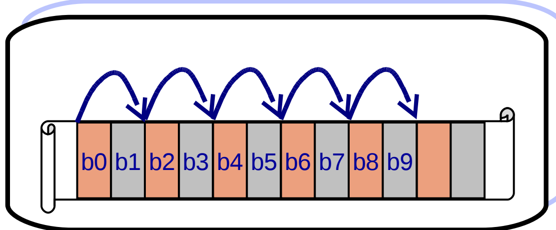
alignment constraints



data size conversion



non stride-one



© 2009 IBM Corporation

Coding choices that impact SIMDization

- How loops are organized
  - Loop must be countable, preferably with literal trip count
  - Only innermost loops are candidates for simdization, except when nested loops have a short literal iteration count
  - Loops with control flow are harder to simdize. Compiler tries to remove control flow, but not always successful
- How data is accessed and laid out in memory
  - Data accesses should preferably be stride-one
  - Layout the data to maximize aligned accesses
  - Prefer use of arrays to pointer arithmetic
- Dependences inherent to the algorithm
  - Loops with inherent data dependences are not simdizable
  - Avoid pointers; pointer aliasing may impede transformations

© 2009 IBM Corporation

Assisting the compiler to perform auto-SIMD

- Loop structure
  - Inline function calls inside innermost loops
  - Automatically (-O5 more aggressive, use inline pragma/directives)
- Data alignment
  - Align data on 16-byte boundaries

    __attribute__((aligned(16))

  - Describe pointer alignment

    _alignx(16, pointer)

  - Can be placed anywhere in the code, preferably close to the loop
  - Use -O5 (enables inter-procedural alignment analysis)
- Pointer aliasing
  - Refine pointer aliasing #pragma disjoint(*p, *q) or restrict keyword
  - Use -O5 (enables interprocedural pointer analysis)

© 2009 IBM Corporation

Compiler options for VMX code generation

- For programs with VMX intrinsics:

  C/C++: -qaltivec -qarch=pwr6

  Fortran: -qarch=pwr6

- Automatic SIMD vectorization:

  Optimization level -O3 -qhot or higher and -qarch=pwr6

- -q[no]enablevmx - Compiler is allowed to generate VMX instructions

  AIX defaults to -qnoenablevmx (must be explicitly turned on by user)

  Linux defaults to -qenablevmx

© 2009 IBM Corporation

Did we SIMDize the loop?

- The -qreport option produces a list of high level transformation performed by the compiler
  - Everything from unrolling, loop interchange, SIMD transformations, etc.
  - Also contains transformed “pseudo source”
- All loops considered for SIMDization are reported
  - Successful candidates are reported
  - If SIMDization was not possible, the reasons that prevented it are also provided
- Can be used to quickly identify opportunities for speedup

© 2009 IBM Corporation

Example – SIMD problems reported

```
extern int *b, *c;

int main()
{
    for (int i=0; i<1024; ++i)
        b[i] = b[i] - c[i-1];
}
```

```
1586-535 (I) Loop (loop index 1) at d.c <line 9> was not SIMD vectorized because the aliasing-induced dependence
prevents SIMD vectorization.
1586-536 (I) Loop (loop index 1) at d.c <line 9> was not SIMD vectorized because it contains memory references  with
non-vectorizable alignment.
1586-536 (I) Loop (loop index 1) at d.c <line 11> was not SIMD vectorized because it contains memory references
((char *)b + (4)*((@CIV0 + 1))) with non-vectorizable alignment.
1586-543 (I) <SIMD info> Total number of the innermost loops considered <"1">. Total  number of the innermost loops
SIMD vectorized <"0">.
```

```
     5 |   long main()
           {
     9 |     @ICM.b0 = b;
             if (!1) goto lab_5;
             @CIV0 = 0;
             __prefetch_by_stream(1,((char *)@ICM.b0  + (0 - 128) + (4)*(@CIV0 + 2)))
             __iospace_lwsync()
    11 |     @ICM.c1 = c;
     9 |     do {   /* id=1 guarded */ /* ~4 */
               /* region = 8 */
               /* bump-normalized */
    11 |       @ICM.b0[@CIV0] = @ICM.b0[@CIV0] - @ICM.c1[@CIV0 - 1];
     9 |       @CIV0 = @CIV0 + 1;
             } while ((unsigned) @CIV0 < 1024u);    /* ~4 */
           lab_5:
             rstr = 0;
    14 |     return rstr;
           } /* function */
```


© 2009 IBM Corporation

Example: correcting SIMD inhibitors

```
extern int * restrict b, * restrict c;


int main()
{
  /* __alignx(16, c);     Not strictly required since compiler   */
  /* __alignx(16, b);     inserts runtime alignment check        */

   for (int i=0; i<1024; ++i)
      b[i] = b[i] - c[i];
}
```

```
586-542 (I) Loop (loop index 1 with nest-level 0 and iteration count 1024) at d_good.c <line 9>
was SIMD vectorized.
1586-542 (I) Loop (loop index 2 with nest-level 0 and iteration count 1024) at d_good.c <line 9>
was SIMD vectorized.
1586-543 (I) <SIMD info> Total number of the innermost loops considered <“2">. Total number of the innermost loops
SIMD vectorized <“2”>.

     7 |  long main()
          {
            @ICM.b0 = b;
            @ICM.c1 = c;
     9 |    @ICMB = (0 - 128);
            @ICM4 = (long) @ICM.c1 & 15;
            @CSE2 = (long) @ICM.b0;

            . . .
```

 © 2009 IBM Corporation

Other examples of SIMD messages

- Loop was not SIMD vectorized because it contains operation which is not suitable for SIMD vectorization.
- Loop was not SIMD vectorized because it contains function calls.
- Loop was not SIMD vectorized because it is not profitable to vectorize.
- Loop was not SIMD vectorized because it contains control flow.
- Loop was not SIMD vectorized because it contains unsupported vector data types
- Loop was not SIMD vectorized because the floating point operation is not vectorizable under -qstrict.
- Loop was not SIMD vectorized because it contains volatile reference

© 2009 IBM Corporation

IBM

Q & A

© 2009 IBM Corporation