Awareness of Architecture in Programming

First Part of Lesson 6: Vector Instructions

Daniel Jiménez-González

Computer Architecture Department
Technical University of Catalonia

1718-Q2

Part I

Introduction

Lesson 6: Objectives

- At the end of this Lesson, the student should be able:
  - Enumerate the differences between scalar and vector instructions
  - Identify which part of the codes can be vectorized/simdizated
  - Find out if the compiler has been able to vectorize the code
  - Insert vector instructions in the C code using intrinsics
  - Describe the main characteristics of the vector set SSE2

Previous Knowledge

- Pointer Arithmetic
- Castings

Bibliography

- Chapters 4, 5 and 6 and Apendix C of the Intel Optimization Manual
- Autovectorization status of GCC
  - http://gcc.gnu.org/projects/tree-ssa/vectorization.html
- Intel intrinsics guide
  - https://software.intel.com/sites/landingpage/IntrinsicsGuide/
- ARM intrinsics guide
  - https://gcc.gnu.org/onlinedocs/gcc-4.8.2/gcc/ARM-NEON-Intrinsics.html
  - https://developer.arm.com/technologies/neon/intrinsics

Part II

Outline

Outline

Outline

Vector Instructions (1/2)

- Instructions with vector operands
  - SIMD (*Single Instruction Multiple Data*)
- Usually, two input vector registers and one output vector register
- Each vector can be interpreted as a set of different number of elements (1, 2, 4, 8, 16, 32, 64)
- The size of each element is the size of the vector register divided by the number of elements
  - **In parallel**, each i-element of one input register is operated with the i-element of the another input register

Vector Instructions (2/2)

- The processor should have vector registers
  - Each processor ISA defines the size and type of the vector registers
    - The number and type of the elements of the vector register
    - Example: 128 bits organized as 16 bytes, 8 words, 4 double-words or 2 quad-words
- The vector instructions will be executed on a FU for vector computation

Examples (1/3)

- Vectorizable loop
  - If the vector register size is 128 bits, we will be able to code the loop with one vector instruction

```
char A[16], B[16], C[16];

for (i=0; i<16; i++)
    A[i] = B[i] + C[i];
```

Examples (2/3)

- Non vectorizable loop

```
for (i=1; i<16; i++)
    A[i] = A[i-1] + B[i];
```

  - There is a data dependency between iterations
  - It is not possible to execute two iterations in parallel without breaking that data dependency

- Vectorizable Loop

```
for (i=0; i<(N-1); i++)
  A[i] = A[i+1] + B[i];
```

Examples (3/3)

- Vectorizable Loop?

```
for (i=0; i<(N-3); i++) {
  A[i] = A[i+1] + B[i];
  B[i+1] = B[i+2] + B[i+3];
}
```

Examples (3/3)

- Vectorizable Loop?

```
for (i=0; i<(N-3); i++) {
  A[i] = A[i+1] + B[i];
  B[i+1] = B[i+2] + B[i+3];
}
```

  - Yes, but you have to transform the code before vectorizing it

Outline

Vector instructions

- SIMD is very useful for certain type of applications
  - Multimedia Applications, 3D games, bioinformatic applications...
    - Applications that are very used nowadays
- SIMD usage can improve the performance of some applications
  - The vector computation will be faster than the scalar one for some computations
- First x86 processors didn't have SIMD instructions
- In the last 10 years, each new generation of x86 has added SIMD instructions to its ISA
  - Backward compatibility
- Examples of vector instruction Extension ISA : MMX, SSE, SSE2, SSE3, SSE4, 3DNow!,.... AVX (Sandy and Ivy Bridge), AVX2 (Haswell), AVX-512 (MIC)... Neon!.

MMX

- 57 new instructions of Intel Pentium (1997)
  - They are only for integer arithmetic
- There are 8 vector registers of 64 bits: `mm0...mm7`
  - Each `mmx` register can be seen as:
    - Vector of 8 bytes
    - Vector of 4 words
    - Vector of 2 double words
  - Actually, the `mmx` register are Floating Point registers
    - It is not possible to mix scalar FP with MMX code
    - That does not affect the OS (Save FP state, Save MMX state)
- Not sucessful
- Other versions: MMX2 (19 new instructions)

3DNow!

- 21 new instructions in the AMD K6-2 (1998)
  - Most of them are FP but some of them use integer arithmetic
  - AMD needed to improve its FP performance since it was smaller than the Pentium II's one
- Use the same `mmx` register
  - Add two new views to the `mmx` registers:
    - Vector of 2 32-bit single-precision FP
    - A quadword
- It is possible to mix MMX and 3DNow! code
- Other versions: Enhanced 3DNow! (3DNow!+) and 3DNow! Professional

SSE/SSE1

- *Streaming SIMD Extensions*
- 70 new instructinos in the Intel PentiumIII (1999)
  - Most of them are FP but some of them use integer arithmetic
- It is an answer to 3DNow!
- Offers 8 vector registers of 128 bits: `xmm0...xmm7`
  - Specific Register File
  - Each vector contains 4 32-bit single precision FPs
  - The 64-bit processador has 16 vector registers: `xmm0...xmm15`
- That let MMX keep running
- It is possible to mix scalar FP and SSE code

SSE2

- 144 new instructions in the Intel Pentium4 (2001)
  - Integer Arithmetic, FP but also instructions in order to control caches, format conversions, ...
  - MMX instruction versions to work with 128 bit registers
- 64-bit double-precision FP
- 8 vector registers of 128 bits: `xmm0...xmm7`
  - The same as those of the SSE
  - Each register can be seen as:
    - A vector of 16 bytes
    - Vector of 8 words
    - Vector of 4 double-words
    - Vector of 2 quad-words
    - Vector of 4 32-bit single-precision FPs
    - Vector of 2 64-bit double-precision FPs

SSE3

- 13 new instructions in the Intel Pentium4 Prescott (2003)
  - PNI (*Prescott New Instructions*)
  - Include instructions to synchronize threads
- Include the concept of *horitzontal* vector instruction
  - Instruction that work with the values stored in one vector register
  - It was already present in the 3DNow!

SSSE3

- *Supplemental Streaming SIMD Extensions 3*
- 16 new instructions in the Intel Core2 model Meron (2006)
  - Other used names
    - SSE4 (incorrecte)
    - TNI (*Tejas New Instructions*)
    - MNI (*Meron New Instructions*)
  - The instructions are more complicated
    - `PMULHRSW` (Packed Multiply High with Round and Scale) treat the sixteen-bit words in registers A and B as signed 15-bit fixed-point numbers between -1 and 1 (eg 0x4000 is treated as 0.5 and 0xa000 as -0.75), and multiply them together with correct rounding.

SSE4

- 54 new intructions in the Intel Core2
  - Scalar product of two vectors, computation of CRC, population count, ...
  - Some intructions are not specific for multimedia applications
  - Look for Intel® SSE4 Programming Reference
- There were two phases
  - SSE4.1: 47 instructions on the Penryn model
  - SSE4.2: 7 instructions on the Nehalem model

AltiVec/Velocity Scale/VMX/VSX

- FP and SIMD instructions
  - Designed by Apple, IBM and Motorola
  - Also, it has instructions to control the cache
- PowerPC processors with this vector instructions:
  - G3 (1997), G4, G5, Cell (used on the Playstation-3), POWER6 (2007), POWER7 (VMX,VSX), POWER8 (VMX,VSX), POWER9(VSX-3)
- 32 vector registers with 128 bits:
  - Vector of 16 bytes
  - Vector of 8 words
  - Vector of 4 double words
  - Vector of 4 32-bit single-precision FPs

AVX, AVX-512

- Extension of the SSE series, comming with Intel's Sandy Bridge and AMD's Bulldozer processors
- Registers of 256 bits (`YMM0 up to YMM15`)
- There are operations to deal with memory alignments
- New Coding System: VEX coding
  - Up to 5 operands will be allowed
  - Several OpCode free, for future extensions
  - VEX coding can support register of up to 1024 bits
  - Compatible with SSE
  - Add 32 and 33 arithmetic and 256-bit non airthmetic instructions respectively
- AVX-512
  - https://software.intel.com/en-us/blogs/2013/avx-512-instructions

Neon

- The NEON instructions support 8-bit, 16-bit, 32-bit, and 64-bit signed and unsigned integers.
- NEON also supports 32-bit single-precision floating point elements , and 8-bit and 16-bit polynomials
- The NEON register bank consists of 32 64-bit registers
  - sixteen 128-bit quadword registers, Q0-Q15
  - thirty-two 64-bit doubleword registers, D0-D31
- Note that ARM also has a vector co-processor called VFPv3. It may share registers with the Neon instructions

Outline

Outline

Which vector extension do we have? (1/4)

- command cat /proc/cpuinfo
  - Intel PentiumM

```
...
flags           : fpu vme de pse tsc msr mce cx8 sep mtrr pge mca cmov pat
clflush dts acpi mmx fxsr sse sse2 ss tm pbe tm2 est
...
```

  - Intel Pentium4 Willamette

```
...
flags           : fpu vme de pse tsc msr pae mce cx8 apic mtrr pge mca cmov pat
pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm
...
```

  - Intel Pentium4 Prescott

```
...
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe nx lm constant_tsc pni
monitor ds_cpl cid cx16 xtpr
```

Which vector extension do we have? (2/4)

- cat /proc/cpuinfo
  - Intel Core2

```
...
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe nx lm constant_tsc pni
monitor ds_cpl vmx est tm2 ssse3 cx16 xtpr lahf_lm
...
```

  - Intel Core2 Duo

```
...
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
 pat pse36 clflush dts acpi mmx fxsr sse
sse2 ss ht tm pbe nx lm constant_tsc arch_perfmon pebs
bts pni monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr sse4_1 lahf_lm
...
```

  - AMD Turion 64

```
...
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush mmx fxsr sse sse2 syscall nx mmxext fxsr_opt lm 3dnowext 3dnow
up pni lahf_lm ts fid vid ttp tm stc
...
```

Which vector extension do we have? (3/4)

- cat /proc/cpuinfo
  - Intel Core i7-2600

```
...
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx rdtscp lm
constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf
pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm sse4_1
sse4_2 x2apic popcnt aes xsave avx lahf_lm ida arat epb xsaveopt
pln pts dts tpr_shadow vnmi flexpriority ept vpid
...
```

  - Intel Core i5-3470

```
...
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe
nx rdtscp lm constant_tsc arch_perfmo n pebs bts xtopology nonstop_tsc aperfmperf
eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3
cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic popcnt
tsc_deadline_tim er aes xsave avx f16c rdrand lahf_lm ida
arat epb xsaveopt pln pts dtherm tpr_shadow vnmi flexpriority
ept vpid fsgsbase smep erms
...
```

Which vector extension do we have? (4/4)

- `cat /proc/cpuinfo`
  - ARM Cortex A9

```
processor        : 0
model name        : ARMv7 Processor rev 0 (v7l)
BogoMIPS        : 1332.01
Features        : half thumb fastmult vfp edsp neon vfpv3 tls vfpd32
CPU implementer        : 0x41
CPU architecture: 7
CPU variant        : 0x3
CPU part        : 0xc09
CPU revision        : 0

...
```

Outline

Compiler

- Some compilers can automatic vectorize code
  - Autovectorizing
  - Examples: `gcc` (4.0 and later), `icc`
- We need to indicate some gcc flags
  - `-ftree-vectorize`: turn on the autovectorization
    - Maybe that is not turn on with -O3 flag
  - `-march=native`: generate specific code
    - `gcc` will use the vector instructions that are in the indicate processor
- With `-ftree-vectorizer-verbose=5` flag gcc shows which loops are vectorized and which not
- http://gcc.gnu.org/projects/tree-ssa/vectorization.html describes which loops are vectorizable

Autovectorization Examples-Intel (1/2)

- Original Code :

```
int A[N], B[N], C[N];

for (i=0; i<n; i++)
        C[i] = A[i] + B[i];
```

- Autovectorized Code (gcc 4.1.2):

```
...
 8048429 :          movdqa 0xfffffee8(%eax,%ebp,1),%xmm0
 8048432 :          paddd  (%eax,%ebx,1),%xmm0
 8048437 :          movdqa %xmm0,0xfffffe68(%eax,%ebp,1)
 8048440 :          add    $0x1,%edx
 8048443 :          add    $0x10,%eax
 8048446 :          cmp    %edx,%ecx
 8048448 :          ja     8048429 <main+0x45>
...
```

Autovectorization Examples-Intel (2/2)

- Original Code:

```
#define MAX 52
int A[N], B[N];
for (i=0; i<n; i++)
   B[i] = (A[i] > MAX ? MAX : 0);
```

- Autovectorized Code (gcc 4.1.2):

```
...
 80483a1:            movdqa 0x8048580,%xmm1
 80483a9:            movdqa 0x804a040(%edx),%xmm0
 80483b1:            pcmpgtd %xmm1,%xmm0
 80483b5:            pand   %xmm1,%xmm0
 80483b9:            movdqa %xmm0,0x804a440(%edx)
 80483c1:            add    $0x1,%ecx
 80483c4:            add    $0x10,%edx
 80483c7:            cmp    %ecx,%ebx
 80483c9:            ja     80483a9 <foo+0x25>
...
      <.rodata.cst16>:  /* To init register with 4 times 52 (=0x34) integer (= 0x34) */
 8048580:   34 00 00 00 34 00 00 00
 8048588:   34 00 00 00 34 00 00 00
```

Autovectorization Examples-ARM (1/2)

- Original Code :

```
int A[N], B[N], C[N];

for (i=0; i<n; i++)
        C[i] = A[i] + B[i];
```

- Autovectorized Code (gcc 4.8.2):

```
...
        .loc 1 10 0 discriminator 2
        vld1.32 q9, [r0]!
        vld1.64 d16-d17, [r2:64]!
        adds    r3, r3, #1
        vadd.i32        q8, q9, q8
        cmp     r4, r3
        vst1.32 q8, [r1]!
        bhi     .L8

...
```

Autovectorization Examples-ARM(2/2)

- Original Code:

```
#define MAX 52
int A[N], B[N];
for (i=0; i<n; i++)
   B[i] = (A[i] > MAX ? MAX : 0);
```

- Autovectorized Code (gcc 4.8.2):

```
...
        vmov.i32        q9, #52  @ v4si
        vmov.i32        q10, #0  @ v4si
...
.L8:
        vld1.64 d16-d17, [r2:64]!
        vcgt.s32        q8, q8, q9
        adds    r3, r3, #1
        cmp     r0, r3
        vbsl    q8, q9, q10
        .loc 1 9 0
        vst1.32 q8, [r1]!
        bhi     .L8
...
```

May help to the compiler

- Avoid conditions, although some cases can be vectorized
- Avoid alising problems (`__restrict__`)
- Don't unroll loops (before doing it, look if the compiler is doing vectorization)
- Use contable loops, with no side-effects
    - No function-calls in the loop (if possible, distribute into a separable loop)
    - No 'break'/'continue'
- Keep the memory access-pattern simple (no indirect accesses, no unknown strides)

Outline

Outline

Include vector instructions to the code

- In some cases, the compiler can not vectorize the code
  - So, we have do it by hand
- How can we introduce the vector instructions?
  - We can mix C and assembler code with the `asm` macro
    - Uff...
  - Using intrinsic functions
    - Allow using vector instructions as if they were function calls
    - Allow vector variables
    - It is more userfriendly, for the programmer and for the compiler
- We will use some intrinsic functions

Intrinsic Function Usage - Intel

- Depending on the compiler version, you will need to specify `-march=native`
- Remember to include the header library with `#include` based on the extension you use:
  - MMX: `mmintrin.h`
  - SSE: `xmmintrin.h`
  - SSE2: `emmintrin.h`
  - SSE3: `pmmintrin.h`
  - SSSE3: `tmmintrin.h`
  - SSE4.1: `smmintrin.h`
  - SSE4.2: `nmmintrin.h`
  - AVX,AVX2: `immintrin.h`
  - 3DNow: `mm3dnow.h`

Programming with SSE2

- Include `emmintrin.h`
- We have two new types of 128 bits
  - `__m128i`: integer vector
    - 16 bytes, 8 words, 4 double-words or 2 quad-words
  - `__m128`: vector of 4 32-bit single-precision floating-point
  - `__m128d`: vector of 2 64-bit double-precision floating-point
- They are not native data types but we can use it
  - in assignment sentences, function returns, parameters
  - within an `struct`, `union`
  - using pointers to those data types
  - within arithmetic expressions
- 16-byte alignment forced by the compiler
  - We can force alignment to 16 bytes for other variables
    - `__attribute__ ((__aligned__(16)))`

Programming with SSE2: Intrinsic names

- Format syntax `_mm_<intrin_op>_<suffix>`
  - `intrin_op`: Basic operation (add, sub, average,...)
  - `suffix`: data types it works with
    - p/ep/s: packed, extended packed, scalar
    - s/d: single-precision/double-precision FP
    - i128: signed 128-bit integer
    - i64/u64: signed/unsigned 64-bit integer
    - i32/u32: signed/unsigned 32-bit integer
    - i16/u16: signed/unsigned 16-bit integer
    - i8/u8: signed/unsigned 8-bit integer
- Examples:
  - `__m128i _mm_add_epi8(__m128i a, __m128i b);`
  - `int _mm_extract_epi16 (__m128i a, int imm);`
- Intrinsics names doesn't match vector instruction pnemotechnics

Outline

Integer Arithmetic SSE2 Intrinsics (1/3)

- add: add
  - Add the elements of two vector registers
  - Intrinsic:
    - `_mm_add_epi8`, `_mm_add_epi16`, `_mm_add_epi32`, `_mm_add_epi64`
- adds/subs: saturated add/sub
  - Saturated add/sub never produces overflow, saturate in the limit value:
    - Example: Integer add of `A` and `B` saturated to 127
    - `result = ((A+B)>127)?  127:  A+B;`
  - Intrinsics
    - `_mm_adds_epi8`/`_mm_subs_epi8`, `_mm_adds_epu8`/`_mm_subs_epu8`
    - `_mm_adds_epi16`/`_mm_subs_epi16`, `_mm_adds_epu16`/`_mm_subs_epu16`

Integer Arithmetic SSE2 Intrinsics (2/3)

- `avg`: average
  - Computes the average of the elements of two vectors
  - `_mm_avg_epu8`, `_mm_avg_epu16`
- `max`/`min`: maximum/minimum
  - Computes the maximum/minium of the elements of two vectors
  - `_mm_max_epu8`/`_mm_min_epu8`, `_mm_max_epi16`/`_mm_min_epi16`
- `madd`: multipy and add
  - `__mm128i _mm_madd_epi16(__mm128i a, __mm128i b);`
    - Multiplies the 8 signed 16-bit integers from `a` by the 8 signed 16-bit integers from `b`. Adds the signed 32-bit integer results pairwise and packs the 4 signed 32-bit integer results.

Integer Arithmetic SSE2 Intrinsics (3/3)

- `mul`: multiply
  - `_mm_mulhi_epi16`/`_mm_mulhi_epu16`
    - Multiplies the 8 signed/unsigned 16-bit integers from a by the 8 signed/unsigned 16-bit integers from b. Packs the upper 16 bits of the 8 signed 32-bit results.
  - `_mm_mullo_epi16` (it the same than the previous one on the less significant bits)
  - `_mm_mul_su32`/`_mm_mul_epu32`
    - Multiply one/two 32-bit unsigned integer and return one/two 64-bit unsigned integer
- `sad`: add absolute value of the difference
  - `_mm_sad_epu8`
    - Computes the absolute difference of the 16 unsigned 8-bit integers from a and the 16 unsigned 8-bit integers from b. Sums the upper 8 differences and lower 8 differences, and packs the resulting 2 unsigned 16-bit integers into the upper and lower 64-bit elements.
    - Usefull for the video decode

Integer binary/shift/compare SSE2 intrinsics

- and/andnot/or/xor: binary operations (bit by bit)
  - _mm_and_si128, _mm_andnot_si128, _mm_or_si128, _mm_xor_si128
- cmpeq/cmpgt/cmplt
  - _mm_cmpeq_epi8, _mm_cmpeq_epi16, _mm_cmpeq_epi32,...
  - The result of each element of the vector will be:
    - 0x00 / 0x0000 / 0x00000000 if the condition is false
    - 0xFF / 0xFFFF / 0xFFFFFFFF if the condition is true
  - Those results can be used as mask at bit level
- sll/sra: shift right/left logic/arithmetic
  - All the elements are shifted the same number of bits
  - _mm_sll_epi16, _mm_sll_epi32, _mm_sll_epi64,...
  - _mm_srl_epi16, _mm_srl_epi32, _mm_srl_epi64,...

Memory access SSE2 intrinsics

- `load`
  - `__m128i _mm_load_si128 (__m128i *p)`
    - Loads 128-bit value. Address `p` **must be 16-byte aligned**
    - If it is not, the program will be aborted
  - `__m128i _mm_loadu_si128 (__m128i *p)`
    - Loads 128-bit value. `p` **may not be 16-byte aligned**.
    - It may be **much** slower than the previous one, depending on the aligned access hardware support
- `store`
  - `void _mm_store_si128 (__m128i *p, __m128i a)`
  - `void _mm_storeu_si128 (__m128i *p, __m128i a)`
  - `void _mm_maskmoveu_si128(__m128i d, __m128i n, char *p)`
    - Conditionally store byte elements of `d` to address `p`. The high bit of each byte in the selector `n` determines whether the corresponding byte in `d` will be stored.

Register access SSE Intrinsics

- `set`: Init the elements of a vector register
  - `_mm_set_epi64/32/16/8`
    - Load the register with 2/4/8/16 data elements of type quad-word/double-word/word/byte
  - `_mm_set1_epi64/32/16/8`
    - Load all the elements of the register with the same value
  - `_mm_setzero_si128`
    - Init the 128-bit register to 0
- `extract`: Get one element from the register
  - `_mm_extract_epi16`
    - Extracts the selected signed or unsigned 16-bit integer from a and zero extends.

FP arithmetic SSE2 intrinsic

- add/sub/mul/div: add/sub/multiply/divide double-precision FP
  - _mm_add_sd, _mm_add_pd
  - _mm_sub_sd, _mm_sub_pd
  - _mm_mul_sd, _mm_mul_pd
  - _mm_div_sd, _mm_div_pd
- sqrt: double-precision FP square root
  - _mm_sqrt_sd, _mm_sqrt_pd
- min/max: double-precision FP minimum/maximum
  - _mm_min_sd, _mm_min_pd
  - _mm_max_sd, _mm_max_pd

FP binary/compare SSE2 Intrinsics

- `and/andnot/or/xor`: Binary operations bit by bit over FP double-precision
  - `_mm_and_pd`, `_mm_andnot_pd`, `_mm_or_pd`, `_mm_xor_pd`
- `cmpeq/cmplt/cmple/...`
  - *scalar* and *packed* versions
  - The result of each element of the vector will be
    - 0x0000000000000000 if the condition is false
    - 0xFFFFFFFFFFFFFFFF if the condition is true
- `cvt...`: Format conversion
  - Between double-precision, simple precision and integer

Load/Write Floating Point register SSE2 intrinsics

- `load`: load a vector register from memory data
  - `_mm_load_pd`, `_mm_loadl_pd`, `_mm_loadr_pd`, `_mm_loadu_pd`
  - `_mm_load_sd`, `_mm_loadh_pd`, `_mm_loadl_pd`
  - They are different on the number of elements that they charge (one, two, or the same but duplicated), order and alignment
- `store`: write the vector register value to memory
  - Similar to a `load`
- `set`: init the component (or two) of a register
  - `_mm_set_sd`, `_mm_setl_pd`, `_mm_set_pd`, `_mm_setr_pd`, `_mm_setzero_pd`, `_mm_move_sd`

SSE2 Intrinsics

- There are more
  - *Shuffle*
  - *Interleaving*
  - Mask creation
  - ...
- Look at the manual
- You can use other intrinsics as SSE and MMX, SSE4, AVX if your compiler support Intel Core i5-3470.

Intrinsic Function Usage - ARM with Neon

- Depending on the compiler version, you will need to specify `-march=native` and `-mfpu=neon`
- Remenber to include the header library:
  - Include: `#include <arm_neon.h>`
  - There are instrinsics using 64-bit (D-registers) or 128-bit (Q-registers) registers

Programming with Neon

- We have different types (`<basic type>x<number of elements>_t`) depending on the operation to do
  - Integers:
    - `u/int8x8_t`, `u/int8x16_t`, ... `u/int64x1_t`, `u/int64x2_t`
  - Floats:
    - `float32x2_t`, `float32x4_t`, `float64x1_t`, `float64x2_t`
  - Polys:
    - `poly8x8_t`, `poly8x16_t`, `poly16x4_t`, `poly16x8_t`, `poly64x1_t`, `poly64x2_t`
- NEON architecture provides full unaligned support for NEON data access but...
  - it is better if it is aligned to cache line size
  - We can force alignment to 16 bytes for other variables
    - `__attribute__ ((__aligned__(16)))`

Programming with Neon: Intrinsic names

- NEON intrinsics begin with the letter `V`, and their format syntax is:

  `V{<mod>}<op>{<shape>}{<cond>}{<.dt>}(<dest> src1, src2`

  - <mod>: Q,H,D,R... modifiers
  - <op>: Operation
  - <shape>: Shape L, W, N
  - <cond>: Condition bits, used with IT instruction <.dt>: Data type <dest>: Destination src1, src2 : Source operand 1 and 2
- Examples:
  - `int8x16_t vaddq_s8 (int8x16_t a, int8x16_t b)`
  - `uint32_t vget_lane_u32 (uint32x2_t v, const int lane)`
- Intrinsics names doesn't match vector instruction pnemotechnics

Outline

Integer Arithmetic Neon Intrinsics (1/3)

- add: add
  - Add the elements of two vector registers
  - Intrinsic:
    - `vadd_s8`, `vadd_u8`, ...`vaddq_s8`, ...
- adds/subs: saturated add/sub
  - Saturated add/sub never produces overflow, saturate in the limit value:
    - Example: Integer add of `A` and `B` saturated to 127
    - `result = ((A+B)>127)? 127: A+B;`
  - Intrinsics
    - `vqadd_s8`, `vqadd_u8`, ...`vqaddq_s8`, ...
    - `vqsub_s8`, `vqsub_u8`, ...`vqsubq_s8`, ...

Integer Arithmetic Neon Intrinsics (2/3)

- `avg`: No average
- `max`/`min`: maximum/minimum
  - Computes the maximum/minium of the elements of two vectors
  - `vmax_s8`, `vmax_u8`, ...`vmaxq_s8`, ...
  - `vmin_s8`, `vmin_u8`, ...`vminq_s8`, ...
- `madd`: multipy and add
  - `int8x8_t vmla_s8 (int8x8_t a, int8x8_t b, int8x8_t c)`
    - Multiplies the 8 signed 8-bit integers from `a` by the 8 signed 8-bit integers from `b`. Adds the less significant signed 8-bit integer results to `c` and return an 8 signed 8-bit integer register result.

Integer Arithmetic Neon Intrinsics (3/3)

- `mul`: multiply
  - `int32x4_t vmulq_s32 (int32x4_t a, int32x4_t b)`
    - Multiplies the each element (4) of `a` with the corresponding one in `b` and leaves the less significant bits of the result.
  - `uint64x2_t vmull_high_u32 (uint32x4_t a, uint32x4_t b)`
    - Multiply 2 32-bit unsigned integer (2 low elements) and return two 64-bit unsigned integer
- `aba`: absolute value of the difference and accumulate
  - `int32x4_t vabaq_s32 (int32x4_t a, int32x4_t b, int32x4_t c)`
    - Computes the absolute difference of the 4 unsigned 32-bit integers from `b` and the 4 unsigned 32-bit integers from `c`. Sums the absolute values of the results into the elements of the vector of the destination (`a`)
    - Usefull for the video decode

Integer binary/shift/compare Neon intrinsics

- `and/not/or/xor`: binary operations (bit by bit)
  - `vandq_u/s32`, `vmvnq_u/s32`, `vorrq_u/s32`, `veorq_s32`
- `cmpeq/cmpgt/cmplt`
  - `vceqq_u/s32`,...
  - The result of each element of the vector will be:
    - `0x00 / 0x0000 / 0x00000000` if the condition is false
    - `0xFF / 0xFFFF / 0xFFFFFFFF` if the condition is true
  - Those results can be used as mask at bit level
- `sll/sra`: shift logic left/arithmetic right
  - All the elements are shifted (left) the based on the elements of the another vector: e.g. `vshlq_s32`
  - All the elements are shifted (right) the same number of bits (immediate): e.g. `vshrq_n_s32`
- `bit selection`: for instance, `vbslq_s32`

Memory access Neon intrinsics

It would be good for the performance to have an aligned memory pointer

- `load`
  - `int32x4_t vld1q_s32 (int32_t const * ptr)`
- `store`
  - `void vst1q_s32 (int32_t * ptr, int32x4_t val)`

Register access Neon Intrinsics

- `set`: Init one element of a vector register
  - `vsetq_lane_u/s32`
- `extract`: Get one element from the register
  - `vgetq_lane_u/s32`
    - Extracts the selected signed or unsigned 32-bit integer from the vector register

FP arithmetic Neon intrinsic

- `add/sub/mul/div`: add/sub/multiply/divide single/double-precision FP
  - `vaddq_f32`, `vaddq_f64`
  - `vsubq_f32`, `vsubq_f64`
  - `vmultq_f32`, `vmultq_f64`
  - `vdivq_f32`, `vdivq_f64`
- `sqrt`: double-precision FP square root
  - `vsqrtq_f32`, `vsqrt_f64`
- `min/max`: double-precision FP minimum/maximum
  - `vminq_f32`, `vmin_f64`
  - `vmaxq_f32`, `vmax_f64`

FP binary/compare Neon Intrinsics

- `and/not/or/xor`: No intrinsics
- `cmpeq/cmplt/cmple/...`
  - `vceqq_f32`,...
  - The result of each element of the vector will be
    - 0x0000000000000000 if the condition is false
    - 0xFFFFFFFFFFFFFFFF if the condition is true
- `vcvtq...`: Format conversion
  - Between double-precision, simple precision and integer

Load/Write Floating Point register SSE2 intrinsics

It would be good for the performance to have an aligned memory pointer

- load
  - `float32x4_t vld1q_f32 (float32_t const * ptr)`
- store
  - `void vst1q_f32 (float32_t * ptr, float32x4_t val)`

Outline

Example (1/3) - Intel

- Original Loop

```
for (i=0; i<N; i++)
    C[i] = A[i] + B[i];
```

- Vector Code with explicit loads/stores (aligned vectors)

```
for (i=0; i<N; i=i+4) {
    __m128i a, b, c;
    a = _mm_load_si128((__m128i*) &A[i]);
    b = _mm_load_si128((__m128i*) &B[i]);
    c = _mm_add_epi32(a, b);
    _mm_store_si128((__m128i*)&C[i], c);
}
```

Example (2/3) - Intel

- Vector Code using pointers

```
for (i=0; i<N; i=i+4) {
    __m128i *pa, *pb, *pc;

    pa = (__m128i*) &A[i];
    pb = (__m128i*) &B[i];
    pc = (__m128i*) &C[i];
    *pc = _mm_add_epi32(*pa, *pb);
}
```

Example (3/3) - Intel

- Scalar product of two vectors
  - 16-bit integer elements

```
short dotmul(short *A, short *B, int len, int N_iter)
{
    int i;
    short acum = 0;

    for (i=0; i<len; i++) {
        acum += A[i]*B[i]
    }
    return acum;
}

Use:
_mm_set1_epi16,  _mm_add_epi16, _mm_mullo_epi16, __m128i,
_mm_load_si128, _mm_store_si128, _mm_extract_epi16
```

Example (3/3) - Intel

- Scalar product of two vectors
  - 16-bit integer elements

```
short dotmul(short *A, short *B, int len, int N_iter)
{
    int i;
    __m128i acum = _mm_set1_epi16(0), *a, *b;

    for (i=0; i<len; i+=8) {
        a = (__m128i*) &A[i];
        b = (__m128i*) &B[i];
        acum = _mm_add_epi16(acum, _mm_mullo_epi16(*a, *b));
    }
    return _mm_extract_epi16(acum, 0) + _mm_extract_epi16(acum, 1)
         + _mm_extract_epi16(acum, 2) + _mm_extract_epi16(acum, 3)
         + _mm_extract_epi16(acum, 4) + _mm_extract_epi16(acum, 5)
         + _mm_extract_epi16(acum, 6) + _mm_extract_epi16(acum, 7);
}
```

Example (1/2)- ARM

- Original Loop

```
void add_scalar(uint8_t * A, uint8_t * B, uint8_t * C){
    for(int i=0; i<16; i++){
         C[i] = A[i] + B[i];
    }
}
```

- Vector Code

```
void add_neon(uint8_t * A, uint8_t * B, uint8_t *C){
        //Setup a couple vectors to hold our data
        uint8x16_t vectorA, vectorB vectorC;

        //Load our data into the vector's register
        vectorA = vld1q_u8(A);
        vectorB = vld1q_u8(B);

        //Add A and B together
        vectorC = vaddq_u8(vectorA, vectorB);
}
```

Example (2/2) - ARM

- Original Loop

```
for (i=0; i<N; i++)
    C[i] = A[i] + B[i];
```

- Vector Code with explicit loads/stores (aligned vectors)

```
uint8x16_t vectorA, vectorB vectorC;
for (i=0; i<N-15; i=i+16) {
    vectorA = vld1q_u8(&A[i]);
    vectorB = vld1q_u8(&B[i]);
    //Add A and B together
    vectorC = vaddq_u8(vectorA, vectorB);
    vst1q_u8(&C[i], vectorC);
}

for (; i<N; i=i++)
    C[i] = A[i] + B[i];
```

Part III

Epilog

Typical Pitfall

- Don't realize that a loop is vectorizable
- Don't properly align the data
- Don't realize that some vector instruction already do what we want to do
- Don't properly use the signed/unsigned types
- Don't properly use of the pointers
- Think that number of iterations is multiple of the possible number of elements per vector register
- Forget the use of -march flag

Awareness of Architecture in Programming

First Part of Lesson 6: Vector Instructions

Daniel Jiménez-González

Computer Architecture Department
Technical University of Catalonia

1718-Q2