14 Multimedia

Oskar Mencer, Stefan Rűger

Lecture 14: Olav Beckmann

oskar@doc.ic.ac.uk • http://www.doc.ic.ac.uk/~oskar
Office: Huxley 422 • Tel: 44-(0207)-594 8268

srueger@doc.ic.ac.uk • http://www.doc.ic.ac.uk/~srueger
Office: Huxley 379 • Tel: 44-(0207)-594 8355

Imperial College
London

Today's Lecture: Implementing Multimedia Computations on Modern Microprocessors

- Giving a processor the label "multimedia" is good business – but what does that mean?
- What can be done, from the point of computer architecture, to support multimedia operations?
  - Instruction set
  - Memory hierarchy
- What do we have to do as programmers to make effective use of processor support for multimedia operations?
- What is the role of the compiler?

What is characteristic of multimedia computations?

- Often inherently parallel
  - Carry out one computation on every pixel of an image, etc.
- May access large volumes of data
  - We may also know that we will not access the same data again soon (e.g. frame-by-frame processing on video)
  - This implies that the data should not be retained in the full memory hierarchy.
- Sometimes need for integer / floating point conversion
  - IEEE floating point format: sign, exponent, mantissa

  - Conversion to/from integer involves bit-level operations
- Regular memory access patterns
- Control flow independent of data values

Multimedia Extensions, by Manufacturer

- Sun: VIS Instruction Set
  - For details, see Sun's June 2002 White Paper
  - Partitions existing 32- and 64-bit floating-point registers to hold multiple short integer values
  - Arithmetic and Logical operations that work in parallel (SIMD) on such registers (see below).
  - Data Access: normally requires 64-bit alignment to load into 64-bit register; special instructions to deal with cases where this is not true.
  - Data Access: block load/store instructions that by-pass caches.
  - Special support for blocked data layout for 3D arrays (more later).
- SIMD (Single Instruction, Multiple Data)
  - SIMD is a form of parallelism; the idea is to apply the same operation to multiple pieces of data
  - There is also SISD, MIMD etc., but less common.

Multimedia Extensions, by Manufacturer

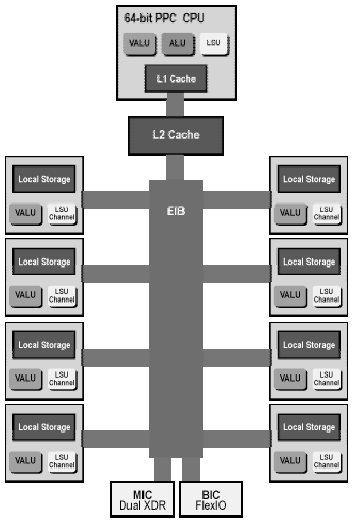
- Intel: MMX / SSE / SSE2 / SSE3
  - New 128-bit registers
  - New instructions for packed (= SIMD) arithmetic, both on floating point and integer data
  - New instructions that control cacheability
- SSE3: New in the "Prescott" version of Pentium 4 Processors
  - x87 to integer conversion, complex arithmetic
  - Video encoding, graphics
  - Thread synchronisation:
    - monitor and mwait (wait on monitor) instructions
    - Will require OS support; Intended to save software idle-spinning when waiting for a monitor

Multimedia Extensions, by Manufacturer

- IBM, Motorola and Apple: AltiVec
  - For more information, see "**AltiVec Technology Programming Interface Manual**"
  - Special-purpose 128-bit (i.e. 16 byte) registers
  - Extensive collection of SIMD instructions. Arithmetic, logic as well as special instructions to support decoding etc.
  - Note that an Apple G5 has a 1-core Power5 processor
  - Apple's gcc compiler has some modifications to generate code for AltiVec instructions.
- AMD: 3DNow! ; also implement some of Intel's instrcutions
  - Range of SIMD, shuffle etc instructions
  - Packed floating point to integer conversions and vice-versa
  - Streaming (cache-bypassing) load/store
  - Fence instructions – these can be used to set "boundaries" for out-of-order execution

Multimedia Extensions, by Manufacturer

- New this month: "Cell" Processor (IBM, Sony, Toshiba)



The CELL Architecture

- General-purpose PowerPC processor with 8 special SIMD cores (processors) on *one* chip
- Picture on the right from arstechnica.com
- Each SIMD core has very simple design in terms of control logic
- Execution units target 128-bit SIMD operations
- SIMD cores have 256K of local memory on-chip, no L1 cache.
- SIMD cores can load and issue 2 instructions at a time
- SIMD cores can communicate via fast "element interface bus"
- Looking forward to programming this?

Code Generation for Multimedia Extensions

- Given a loop like this, how do we generate code that makes use of multimedia extensions:

```
for( i = 0; i < N; ++i ) {
  a[i] = scalar * b[i] + a[i];
}
```

- The problem is that if we just compile this, we get plain, non-multimedia code
- We could write the assembler code by hand
- There are some special compiler-intrinsics that instruct a compiler to generate code using specific instructions
- The best solution would appear to be to rely on the compiler to generate the code
- This is not always possible of the computation to be done cannot be expressed easily in the high-level language

Code Generation for Multimedia Extensions

```
L6:
        fld     %st(0)
        fmuls
        (%ebx,%eax,4)
        fadds
        (%edx,%eax,4)
        fstps
        (%edx,%eax,4)
        incl    %eax
        cmpl    %ecx, %eax
        jl      L6
```

- To the left is normal i368 code generated by gcc
- Underneath: Intel SSE code generated by Intel's C Compiler

```
$B1$11:                         ; Preds $B1$11 $B1$10
        movaps    xmm2, XMMWORD PTR [esi+ebp*4]                 ;4.22
        mulps     xmm2, xmm1                                    ;4.22
        addps     xmm2, XMMWORD PTR [eax+ebp*4]                 ;4.5
        movaps    XMMWORD PTR [eax+ebp*4], xmm2                 ;4.5
        movaps    xmm3, XMMWORD PTR [esi+ebp*4+16]              ;4.22
        mulps     xmm3, xmm1                                    ;4.22
        addps     xmm3, XMMWORD PTR [eax+ebp*4+16]              ;4.5
        movaps    XMMWORD PTR [eax+ebp*4+16], xmm3              ;4.5
        add       ebp, 8                                        ;3.3
        cmp       ebp, ecx                                      ;3.3
        jb        $B1$11        ; Prob 97%                      ;3.3
```


Code Generation for Multimedia Extensions

- Do we notice anything about this code? (Write this down!)

```
$B1$11:                         ; Preds $B1$11 $B1$10
        movaps    xmm2, XMMWORD PTR [esi+ebp*4]                 ;4.22
        mulps     xmm2, xmm1                                    ;4.22
        addps     xmm2, XMMWORD PTR [eax+ebp*4]                 ;4.5
        movaps    XMMWORD PTR [eax+ebp*4], xmm2                 ;4.5
        movaps    xmm3, XMMWORD PTR [esi+ebp*4+16]              ;4.22
        mulps     xmm3, xmm1                                    ;4.22
        addps     xmm3, XMMWORD PTR [eax+ebp*4+16]              ;4.5
        movaps    XMMWORD PTR [eax+ebp*4+16], xmm3              ;4.5
        add       ebp, 8                                        ;3.3
        cmp       ebp, ecx                                      ;3.3
        jb        $B1$11        ; Prob 97%                      ;3.3
```

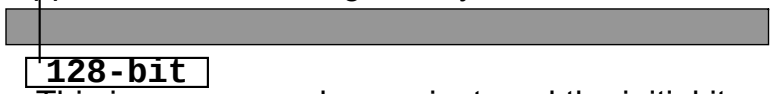

Code Generation for Multimedia Extensions

- Recompile, telling the Intel C Compiler that the arrays are not aliased
- The loop is unrolled, also one of the operands for the add operation is in memory

```
$B1$19:                         ; Preds $B1$19 $B1$18
        movss     xmm1, DWORD PTR [esi+ebp*4]                   ;4.22
        mulss     xmm1, xmm0                                    ;4.22
        addss     xmm1, DWORD PTR [edx+ebp*4]                   ;4.5
        movss     DWORD PTR [edx+ebp*4], xmm1                   ;4.5
        movss     xmm2, DWORD PTR [esi+ebp*4+4]                 ;4.22
        mulss     xmm2, xmm0                                    ;4.22
        addss     xmm2, DWORD PTR [edx+ebp*4+4]                 ;4.5
        movss     DWORD PTR [edx+ebp*4+4], xmm2                 ;4.5
        movss     xmm3, DWORD PTR [esi+ebp*4+8]                 ;4.22
        […]
        mulss     xmm5, xmm0                                    ;4.22
        addss     xmm5, DWORD PTR [edx+ebp*4+16]                ;4.5
        movss     DWORD PTR [edx+ebp*4+16], xmm5                ;4.5
        add       ebp, 5                                        ;3.3
        cmp       ebp, eax                                      ;3.3
        jbe       $B1$19        ; Prob 97%                      ;3.3
                                ; LOE eax edx ebp esi xmm0
```


Code Generation for Multimedia Extensions

- We are not finished yet…
- Most SIMD instructions that can take a memory operand require the address in memory to be aligned, i.e. for 128-bit words, the address in memory has to be 128-bit aligned etc.
- Suppose we have a single array



  - This is easy enough – we just peel the initial iterations of the loop until the rest become aligned.
- However, if we have two arrays with different alignments?
- Having a special malloc() / new that always returns 128-bit aligned addresses *may* help.
  - Why is this not always enough?

Code Generation for Multimedia Extensions

- Consider a 2D image, 24-bit colour, i.e. three colour channels of an unsigned 8-bit number

- Assume the image is 1024x768.
- Where will the second row of the image be aligned?
- Solution: (write this down!)

- Intel and others provide quite complicated routines for allocating data to be used as images for this reason!

Code Generation for Multimedia Extensions

- Consider a 2D image, 24-bit colour, i.e. three colour channels of an unsigned 8-bit number

- What will happen when we apply this convolution (filter)? (Write this down!)

- Solution:
  Saturating arithmetic instructions
- The real problem is that there is no way of representing this in a high-level language; requires assembler or library code.

Code Generation for Multimedia Extensions

- Consider a 2D image, 24-bit colour, i.e. three colour channels of an unsigned 8-bit number

- How about the proximity of memory accesses here? Let's draw out how this is stored in (linear) memory:

Code Generation for Multimedia Extensions

- Consider a 2D image, 24-bit colour, i.e. three colour channels of an unsigned 8-bit number

- One solution is to store this data by blocks