Vector processing: x86, SSE

September 2, 2013
Overview

- Types of parallel computations
- Vector processors
- x86
- SSE examples
- Home exam 1
Types of Parallel Processing/Computing?

- **Bit-level parallelism**
  - 4-bit $\rightarrow$ 8-bit $\rightarrow$ 16-bit $\rightarrow$ 32-bit $\rightarrow$ 64-bit $\rightarrow$ ...

- **Instruction level parallelism**
  - classic RISC pipeline
    (fetch, decode, execute, memory, write back)
Types of Parallel Processing/Computing?

- Task parallelism
  - different operations are performed concurrently
  - task parallelism is achieved when the processors execute different threads (or processes) on the same or different data
  - examples?
Types of Parallel Processing/Computing?

- Data parallelism
  - distribution of data across different parallel computing nodes
  - data parallelism is achieved when each processor performs the same task on different pieces of the data
  - examples?

```plaintext
for each element a
    perform the same (set of) instruction(s) on a
end
```

- when should we not use data parallelism?
Flynn's taxonomy

<table>
<thead>
<tr>
<th>Single instruction</th>
<th>Multiple instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single data</strong></td>
<td><strong>Multiple data</strong></td>
</tr>
<tr>
<td>SISD</td>
<td>SIMD</td>
</tr>
<tr>
<td>Instruction Pool</td>
<td>Instruction Pool</td>
</tr>
<tr>
<td>Data Pool</td>
<td>Data Pool</td>
</tr>
<tr>
<td>PU</td>
<td>PU</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MISP</th>
<th>MIMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Pool</td>
<td>Instruction Pool</td>
</tr>
<tr>
<td>Data Pool</td>
<td>Data Pool</td>
</tr>
<tr>
<td>PU</td>
<td>PU</td>
</tr>
<tr>
<td>PU</td>
<td>PU</td>
</tr>
<tr>
<td>PU</td>
<td>PU</td>
</tr>
<tr>
<td>PU</td>
<td>PU</td>
</tr>
</tbody>
</table>
Vector processors

A vector processor (or array processor)

- CPU that implements an instruction set containing instructions that operate on one-dimensional arrays (vectors)

- Example systems?
  - Cray-1 (1976)
    - 4096 bits registers (64x64-bit floats)
  - IBM
    - POWER with ViVA (Virtual Vector Architecture) 128 bits registers
    - Cell – SPE 128 bit registers
  - SUN
    - UltraSPARC with VIS 64-bit registers
  - NEC
    - SX-6/SX-9 (2008) - Earth simulator 1/2 with 4096 bit registers (used different ways)
      - up to 512 nodes of 8/16 cores (8192 cores)
      - each core
        - has 6 parallel instruction units
        - shares 72 4096-bit registers
Vector processors

People use vector processing in many areas...

- Scientific computing
- Multimedia Processing (compression, graphics, image processing, ...)
- Standard benchmark kernels (Matrix Multiply, FFT, Convolution, Sort)
- Lossy Compression (JPEG, MPEG video and audio)
- Lossless Compression (Zero removal, RLE, Differencing, LZW)
- Cryptography (RSA, DES/IDEA, SHA/MD5)
- Speech and handwriting recognition
- Operating systems (memcpy, memset, parity, ...)
- Networking (checksum, ...)
- Databases (hash/join, data mining, updates)
- ...
Vector processors

- Instruction sets?
  - MMX
  - SSE (several extensions)
  - AVX
  - Altivec
  - 3DNow!
  - VIS
  - MDMX
  - FMA
  - ...

![Diagram of SIMD and Instruction Pool]
Special instructions...

MMX

- MMX is officially a meaningless initialism trademarked by Intel; unofficially,
  - MultiMedia eXtension
  - Multiple Math eXtension
  - Matrix Math eXtension

- SIMD (Single Instruction, Multiple Data) computation processes multiple data in parallel with a single instruction, resulting in significant performance improvement; MMX gives 2 32-bit computations at once.

- MMX defined 8 “new” 64-bit integer registers (mm0 ~ mm7), which were aliases for the existing x87 FPU registers – reusing 64 (out of 80) bits in the floating point registers.
Special instructions...

- **SSE**
  - Streaming SIMD Extensions (SSE)
  - SIMD; 4 32-bit simultaneous computations

- SSE defines 8 new 128-bit registers (xmm0 ~ xmm7) for single-precision floating-point computations. Since each register is 128-bit long, we can store total 4 of 32-bit floating-point numbers (1-bit sign, 8-bit exponent, 23-bit mantissa/fraction).

- Single or packed scalar operations: __SS vs __PS
x86 – heterogeneous?
Intel Nehalem

Nehalem microarchitecture

- quad-pipe associative Instruction Cache 32 KB, 128-entry TLB-4K, 7 TLB-24M per thread
- Branch Prediction, data and mode, loop, indirect jump
- Prefetch Buffer (16 Bytes)
- Decoder & Instruction Length Decoder
- Instruction Queue 18:86 Instructions Alignment MacroOp Fusion
- Complex Decoder, Simple Decoder, Simple Decoder, Simple Decoder
- Decoded Instruction Queue (28 µOP entries)
- MicroOp Fusion
- Loop Stream Decoder
- 2 x Retirement Register File
- 2 x Register Allocation Table (PAT)
- 2 x Register Buffer (128 entry) fused
- Reservation Station (128-entry used)
- AGU, Store, Load, Add, Unit
- IntegraALU, MUL, Add, Move
- SSE ADD, Move
- FP ADD, MUL
- Memory Order Buffer (MORB)
- Octuple associative Data Cache 32 KB, 64-entry TLB-8K, 32-entry TLB-24M

University of Oslo
INF5063 – Carsten Griwodz, Håvard Espeland, Håkon Stensland, Preben N. Olsen, Pål Halvorsen

[ simula . research laboratory ]
Thus, an x86 is definitely parallel and has heterogeneous (internal) cores (hidden) complicating factors – **out of order execution**
Out of order execution

- Beginning with Pentium Pro, **out-of-order-execution** has improved the micro-architecture design
  - execution of an instruction is delayed if the input data is not yet available

- Instructions are split into micro-operations (μ-ops)
  - **ADD EAX, EBX**  
    - % Add content in EBX to EAX
    - simple operation which generates only 1 μ-ops
  
  - **ADD EAX, [mem1]**  
    - % Add content in mem1 to EAX
    - operation which generates 2 μ-ops:
      1) load mem1 into ( unnamed ) register, 2) add
  
  - **ADD [mem1], EAX**  
    - % Add content in EAX to mem1
    - operation which generates 3 μ-ops:
      1) load mem1 into ( unnamed ) register, 2) add, 3) write result back to memory
Out of order execution

Example – \((\text{mem1} \times 5) + \text{mem2} \rightarrow \text{mem3}\):

```assembly
mov eax, [mem1]
mul eax, 5
add eax, [mem2]
mov [mem3], eax
```

Ordered execution:
1) fetch mem1
2) move mem1 to EAX
3) multiply EAX with 5
4) fetch mem2
5) add mem2 to EAX
6) move EAX to mem3

Out-of-ordered execution:
1) fetch mem1
2) move mem1 to EAX
   3a) fetch mem2
   3b) multiply EAX with 5
4) add mem2 to EAX
5) move EAX to mem3
may be a nice, "easy" introduction to more complex heterogeneous architectures
Element-wise Vector Multiplication using SSE
Element-wise Vector Multiplication

- Find element-wise product of 2 vectors:

<table>
<thead>
<tr>
<th></th>
<th>a_1</th>
<th>a_2</th>
<th>a_3</th>
<th>a_4</th>
<th>a_5</th>
<th>a_6</th>
<th>...</th>
<th>a_n</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>b_1</td>
<td>b_2</td>
<td>b_3</td>
<td>b_4</td>
<td>b_5</td>
<td>b_6</td>
<td>...</td>
<td>b_n</td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>=</td>
<td>=</td>
<td>=</td>
<td>=</td>
<td>=</td>
<td>=</td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>a_1*b_1</td>
<td>a_2*b_2</td>
<td>a_3*b_3</td>
<td>a_4*b_4</td>
<td>a_5*b_5</td>
<td>a_6*b_6</td>
<td>...</td>
<td>a_n*b_n</td>
<td></td>
</tr>
</tbody>
</table>

```c
void vec_eltwise_product(vec_t* a, vec_t* b, vec_t* c)
{
    size_t i;
    for (i = 0; i < a->size; i++) {
        c->data[i] = a->data[i] * b->data[i];
    }
}
```
Element-wise Vector Multiplication

- Unroll loop

```c
void vec_eltwise_product(vec_t* a, vec_t* b, vec_t* c)
{
    size_t i;
    for (i = 0; i < a->size; i++) {
        c->data[i] = a->data[i] * b->data[i];
    }
}
```

SSE can take 4 32-bit operations in parallel using the 128-bit registers → unroll loop to a 4-element operation

```c
void vec_eltwise_product_unrolled(vec_t* a, vec_t* b, vec_t* c)
{
    size_t i;
    for (i = 0; i < a->size; i+=4) {
        c->data[i+0] = a->data[i+0] * b->data[i+0];
        c->data[i+1] = a->data[i+1] * b->data[i+1];
        c->data[i+2] = a->data[i+2] * b->data[i+2];
        c->data[i+3] = a->data[i+3] * b->data[i+3];
    }
}
```
Element-wise Vector Multiplication

Use SSE assembly instructions:

```c
void vec_eltwise_product_unrolled(vec_t* a, vec_t* b, vec_t* c)
{
    size_t i;
    for (i = 0; i < a->size; i +=4) {
        c->data[i+0] = a->data[i+0] * b->data[i+0];
        c->data[i+1] = a->data[i+1] * b->data[i+1];
        c->data[i+2] = a->data[i+2] * b->data[i+2];
        c->data[i+3] = a->data[i+3] * b->data[i+3];
    }
}
```

```assembly
void vec_eltwise_product_SSE:
    pushall
    mov ebp, esp
    mov edi, [ebp+12] ; a
    mov ebx, [ebp+16] ; b
    mov eax, [ebp+20] ; c
    mov ecx, SIZE_OF_VECTOR ; counting i down
    shr ecx, 2 ; use 4-increment of I
    xor esi, esi ; array index = 0
    for-loop:
        movups xmm1, [edi + esi] ; read in a
        movups xmm2, [ebx + esi] ; read in b
        muls xmm1, xmm2 ; multiply
        movups [eax + esi], xmm1 ; result back in c
        add esi, 4 ; next 4 elements
    loop for-loop
    exit:
        popall
        ret
```

MOVUPS (Move Unaligned Packed Single-Precision Floating-Point Values) moves four packed single-precision floating-point numbers from the source operand (second operand) to the destination operand (first operand).

MULPS (Packed Single-Precision Floating-Point Multiply) performs a SIMD multiply of the four packed single-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the packed single-precision floating-point results in the destination operand.

Operations on these 4 elements can be performed in parallel.

 MOVUPS (Move Unaligned Packed Single-Precision Floating-Point Values) moves four packed single-precision floating-point numbers from the source operand (second operand) to the destination operand (first operand)
Element-wise Vector Multiplication

- Use SSE intrinsic functions:

  ```
  void vec_eltwise_product_SSE(vec_t * a, vec_t * b, vec_t * c) {
    size_t i;
    _m128 va;
    _m128 vb;
    _m128 vc;
    for (i = 0; i < a->size; i+= 4) {
      va = _mm_loadu_ps(&a->data[i]);
      vb = _mm_loadu_ps(&b->data[i]);
      vc = _mm_mul_ps(va, vb);
      _mm_storeu_ps(&c->data[i], vc);
    }
  }
  ```

- intrinsic SSE functions exist, e.g., for gcc on Intel Linux:

  ```
  void vec_eltwise_product_unrolled(vec_t * a, vec_t * b, vec_t * c) {
    switch (a->vec_size) {
      case SIZE_OF_VECTOR: {
        vec_eltwise_product_SSE(a, b, c);
        break;
      } ...
    }
  }
  ```

  - `dst = __mm_loadu_ps(src)`  →  `movups dst, src`
  - `src1 = __mm_mul_ps(src1, src2)`  →  `mulps src1, src2`
  - `src1 = __mm_add_ps(src1, src2)`  →  `addps src1, src2`

- which can be used without any (large/noticeable) performance loss
Element-wise Vector Multiplication

- Use SSE intrinsic functions:

```c
void vec_eltwise_product_unrolled (a, b, c)

vec_eltwise_product_SSE:
pushall
mov ebp, esp
mov edi, [ebp+12] ; a
mov ebx, [ebp+16] ; b
mov eax, [ebp+20] ; c
mov ecx, SIZE_OF_VECTOR ; counting i down
shr ecx, 2 ; use 4-increment of i
xor esi, esi ; array index = 0
for-loop:
    movups xmm1, [edi + esi] ; read in a
    movups xmm2, [ebx + esi] ; read in b
    mulps xmm1, xmm2 ; multiply
    movups [eax + esi], xmm1 ; result back in c
    add esi, 4 ; next 4 elements
loop for-loop
exit:
popall
ret
```

```c
void vec_eltwise_product_SSE(vec_t* a, vec_t* b, vec_t* c) {
    size_t i;
    __m128 va;
    __m128 vb;
    __m128 vc;
    for (i = 0; i < a->size; i+= 4) {
        va = _mm_loadu_ps(&a->data[i]);
        vb = _mm_loadu_ps(&b->data[i]);
        vc = _mm_mul_ps(va, vb);
        _mm_storeu_ps(&c->data[i], vc);
    }
}
```
Element-wise Vector Multiplication

- **SSE vs AVX (Advanced Vector Extensions)**
  - AVX is similar to SSE, but has twice the width of the registers: 256 bit
  - renamed registers (now 16) from XMM\textsubscript{i} to YMM\textsubscript{i}
  - available from Intel's Sandy Bridge and AMD's Bulldozer processors (2011)

```c
void vec_eltwise_product_SSE(vec_t * a, vec_t * b, vec_t * c) {
    size_t i;
    __m128 va;
    __m128 vb;
    __m128 vc;
    for (i = 0; i < a->size; i+= 4) {
        va = _mm_loadu_ps(&a->data[i]);
        vb = _mm_loadu_ps(&b->data[i]);
        vc = _mm_mul_ps(va, vb);
        _mm_storeu_ps(&c->data[i], vc);
    }
}
```

Translated code:

```c
void vec_eltwise_product_SSE(vec_t* a, vec_t* b, vec_t* c) {
    size_t i;
    __m256 va;
    __m256 vb;
    __m256 vc;
    for (i = 0; i < a->size; i+= 8) {
        va = _mm256_loadu_ps(&a->data[i]);
        vb = _mm256_loadu_ps(&b->data[i]);
        vc = _mm256_mul_ps(va, vb);
        _mm256_storeu_ps(&c->data[i], vc);
    }
}
```
Matrix multiplication using SSE
Matrix Multiplication

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
2 & 2 & 2 & 2 \\
3 & 3 & 3 & 3 \\
4 & 4 & 4 & 4 \\
\end{array}
\times
\begin{array}{c}
1 \\
2 \\
3 \\
4 \\
\end{array}
= \begin{array}{cccc}
1 + 2 + 3 + 4 \\
2 + 4 + 6 + 8 \\
3 + 6 + 9 + 12 \\
4 + 8 + 12 + 16 \\
\end{array}
= \begin{array}{c}
10 \\
20 \\
30 \\
40 \\
\end{array}
\]
Matrix Multiplication - C

#include <stdio.h>

float elts[4][4] = {1,1,1,1,2,2,2,2,3,3,3,3,4,4,4,4};
float vin[4] = {1,2,3,4};
float vout[4];

void main(void)
{
    vout[0] = elts[0][0] * vin[0] + elts[0][1] * vin[1] +
              elts[0][2] * vin[2] + elts[0][3] * vin[3];


    printf("%f %f %f %f
", vout[0], vout[1], vout[2], vout[3]);
}
Matrix Multiplication – SSE

#include <stdio.h>

float elts[4][4] = {1,1,1,1,2,2,2,2,3,3,3,3,4,4,4,4};
float vin[4] = {1,2,3,4};
float vout[4];

void main(void)
{
    vout[0] = elts[0][0] * vin[0] + elts[0][1] * vin[1] + elts[0][2] * vin[2] + elts[0][3] * vin[3];
    printf("%.1f %.1f %.1f %.1f\n", vout[0], vout[1], vout[2], vout[3]);
}

Assuming elts in a COLUMN-MAJOR order:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
</tr>
<tr>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
</tr>
<tr>
<td>m</td>
<td>n</td>
<td>o</td>
<td>p</td>
</tr>
</tbody>
</table>

1 2 3 4
10 20 30 40
Matrix Multiplication – SSE

```asm
__asm { 
    mov    esi, VIN
    mov    edi, VOUT

    // load columns of matrix into xmm4-7
    mov    edx, ELTS
    movups xmm4, [edx]
    movups xmm5, [edx + 0x10]
    movups xmm6, [edx + 0x20]
    movups xmm7, [edx + 0x30]

    // load \textit{v} into xmm0.
    movups xmm0, [esi]

    // we'll store the final result in xmm2; initialize it
    xorps  xmm2, xmm2

    // broadcast \textit{x} into xmm1, multiply it by the first
    // column of the matrix (xmm4), and add it to the total
    movups xmm1, xmm0
    shufps xmm1, xmm1, 0x00
    mulps  xmm1, xmm4
    addps  xmm2, xmm1

    // repeat the process for \textit{y}, \textit{z} and \textit{w}
    movups xmm1, xmm0
    shufps xmm1, xmm1, 0x55
    mulps  xmm1, xmm5
    addps  xmm2, xmm1

    movups xmm1, xmm0
    shufps xmm1, xmm1, 0xAA
    mulps  xmm1, xmm6
    addps  xmm2, xmm1

    movups xmm1, xmm0
    shufps xmm1, xmm1, 0xFF
    mulps  xmm1, xmm7
    addps  xmm2, xmm1

    // write the results to vout
    movups [edi], xmm2
}
```

Assuming \textit{elts} in a COLUMN-MAJOR order:
__asm {
    mov    esi, VIN
    mov    edi, VOUT

    // load columns of matrix into xmm4-7
    mov    edx, ELTS
    movups xmm4, [edx]
    movups xmm5, [edx + 0x10]
    movups xmm6, [edx + 0x20]
    movups xmm7, [edx + 0x30]

    // load v into xmm0.
    movups xmm0, [esi]

    // we'll store the final result in xmm2; initialize it
    // to zero
    xorps  xmm2, xmm2

    // broadcast x into xmm1, multiply it by the first
    // column of the matrix (xmm4), and add it to the total
    movups xmm1, xmm0
    shufps xmm1, xmm1, 0x00
    mulps  xmm1, xmm4
    addps  xmm2, xmm1

    // repeat the process for y, z and w
    movups xmm1, xmm0
    shufps xmm1, xmm1, 0x55
    mulps  xmm1, xmm5
    addps  xmm2, xmm1

    movups xmm1, xmm0
    shufps xmm1, xmm1, 0xAA
    mulps  xmm1, xmm6
    addps  xmm2, xmm1

    movups xmm1, xmm0
    shufps xmm1, xmm1, 0xFF
    mulps  xmm1, xmm7
    addps  xmm2, xmm1

    // write the results to vout
    movups [edi], xmm2
}

127  95  63  31  0
XMM0 a3 a2 a1 a0
 EAX
 movlps xmm0, [eax]
 Memory

127  95  63  31  0
XMM0 a3 a2 a1 a0
 EAX
 movaps [eax], xmm0
 Memory
Matrix Multiplication – SSE

```asm
__asm {
    mov    esi, VIN
    mov    edi, VOUT

    // load columns of matrix into xmm4-7
    mov    edx, ELTS
    movups xmm4, [edx]
    movups xmm5, [edx + 0x10]
    movups xmm6, [edx + 0x20]
    movups xmm7, [edx + 0x30]

    // load v into xmm0
    movups xmm0, [esi]

    // we'll store the final result in xmm2; initialize it
    // to zero
    xorps xmm2, xmm2

    // broadcast x into xmm1, multiply it by the first
    // column of the matrix (xmm4), and add it to the total
    movups xmm1, xmm0
    shuffle xmm1, xmm1, 0x00
    mulps xmm1, xmm4
    addps xmm2, xmm1

    movups xmm1, xmm0
    shuffle xmm1, xmm1, 0x55
    mulps xmm1, xmm5
    addps xmm2, xmm1

    movups xmm1, xmm0
    shuffle xmm1, xmm1, 0xAA
    mulps xmm1, xmm6
    addps xmm2, xmm1

    movups xmm1, xmm0
    shuffle xmm1, xmm1, 0xFF
    mulps xmm1, xmm7
    addps xmm2, xmm1

    // write the results to vout
    movups [edi], xmm2
}
```
Matrix Multiplication – SSE

__asm {
    mov  esi, VIN
    mov  edi, VOUT

    // load columns of matrix into xmm4-7
    mov  edx, ELTS
    movups xmm4, [edx]
    movups xmm5, [edx + 0x10]
    movups xmm6, [edx + 0x20]
    movups xmm7, [edx + 0x30]

    // load v into xmm0.
    movups xmm0, [esi]

    // we'll store the final result in xmm2; initialize it
    // to zero
    xorps xmm2, xmm2

    // broadcast x into xmm1, multiply it by the first
    // column of the matrix (xmm4), and add it to the total
    mulps  xmm1, xmm4
    addps  xmm2, xmm1

    // repeat the process for y, z and w
    movups xmm1, xmm0
    shufps xmm1, xmm1, 0x00
    mulps  xmm1, xmm4
    addps  xmm2, xmm1

    movups xmm1, xmm0
    shufps xmm1, xmm1, 0x55
    mulps  xmm1, xmm5
    addps  xmm2, xmm1

    movups xmm1, xmm0
    shufps xmm1, xmm1, 0xAA
    mulps  xmm1, xmm6
    addps  xmm2, xmm1

    movups xmm1, xmm0
    shufps xmm1, xmm1, 0xFF
    mulps  xmm1, xmm7
    addps  xmm2, xmm1

    // write the results to vout
    movups [edi], xmm2
}

1 1 1 1
x x x x
1 2 3 4
= = = =
Matrix Multiplication – SSE

```asm
__asm {
    mov    esi, VIN
    mov    edi, VOUT

    // load columns of matrix into xmm4-7
    mov    edx, ELTS
    movups xmm4, [edx]
    movups xmm5, [edx + 0x10]
    movups xmm6, [edx + 0x20]
    movups xmm7, [edx + 0x30]

    // load v into xmm0.
    movups xmm0, [esi]

    // we'll store the final result in xmm2; initialize it
    // to zero
    xorps xmm2, xmm2

    // broadcast x into xmm1, multiply it by the first
    // column of the matrix (xmm4), and add it to the total
    movups xmm1, xmm0
    shufps xmm1, xmm1, 0x00
    mulps xmm1, xmm4
    addps xmm2, xmm1

    // repeat the process for y, z and w
    movups xmm1, xmm0
    shufps xmm1, xmm1, 0x55
    mulps xmm1, xmm5
    addps xmm2, xmm1

    movups xmm1, xmm0
    shufps xmm1, xmm1, 0xAA
    mulps xmm1, xmm6
    addps xmm2, xmm1

    movups xmm1, xmm0
    shufps xmm1, xmm1, 0xFF
    mulps xmm1, xmm7
    addps xmm2, xmm1

    // write the results to vout
    movups [edi], xmm2
}
```
Matrix Multiplication – SSE

```asm
__asm {
    mov   esi, VIN
    mov   edi, VOUT

    // load columns of matrix into xmm4-7
    mov   edx, ELTS
    movups xmm4, [edx]
    movups xmm5, [edx + 0x10]
    movups xmm6, [edx + 0x20]
    movups xmm7, [edx + 0x30]

    // load v into xmm0.
    movups xmm0, [esi]

    // we'll store the final result in xmm2; initialize it to zero
    xorps xmm2, xmm2

    // broadcast x into xmm1, multiply it by the first column of the matrix (xmm4), and add it to the total
    movups xmm1, xmm0
    shufps xmm1, xmm1, 0x00
    mulps xmm1, xmm4
    addps xmm2, xmm1

    // repeat the process for y, z and w
    movups xmm1, xmm0
    shufps xmm1, xmm1, 0x55
    mulps xmm1, xmm5
    addps xmm2, xmm1

    ...  // similar operations for xmm6 and xmm7

    // write the results to vout
    movups [edi], xmm2
}
```
Changing picture brightness using SSE
Brightness

- Make a simple program that change the brightness of a picture by adding a brightness value between -128 and 127 to each pixel. A positive value makes the picture brighter, a negative value darker.

- Assume it is a one byte pixel in gray scale, or a YUV picture where you operate on the Y-part only.
int main (int argc, char *argv[]) {
    < ... more variables ... >

    unsigned char *bw_image;
    int image_len, bright_value;

    /* open input and output files */
    bright_value = atoi( argv[???] ); /* a value between -128 and 127 */
    image_len = atoi( argv[???] ) * atoi( argv[???] ); /* picture width * picture height */
    bw_image = (unsigned char *) malloc(image_len + 15); /* +15 to allow 16-byte alignment */

    if (((long)bw_image % 16) != 0) bw_image += 16 - ((long)bw_image % 16); /* align */

    /* read picture to into memory with first pixel at "bw_image" */

    if (bright_value != 0) brightness( bw_image, image_len, bright_value );

    /* write picture back to file */
    /* free memory, close descriptors */
}
void brightness (unsigned char *buffer, int len, int v) {
    int t, new_pixel_value;

    if (v > 0) {
        for (t = 0; t < len; t++) {
            /* make brighter */
            new_pixel_value = buffer[t] + v;
            if (new_pixel_value > 255) new_pixel_value = 255;
            buffer[t] = new_pixel_value;
        }
    } else {
        for (t = 0; t < len; t++) {
            /* make darker */
            new_pixel_value = buffer[t] + v;
            if (new_pixel_value < 0) new_pixel_value = 0;
            buffer[t] = new_pixel_value;
        }
    }
}
void brightness_sse_asm(unsigned char *image, int len, int v)

brightness_sse_asm:
pushall
mov ebp, esp
mov edi, [ebp+12] ; unsigned char *image
mov ecx, [ebp+16] ; int len
mov eax, [ebp+20] ; int v in [-128, 127]
test eax, 0x80000000 ; check if v is negative
jz bright_not_neg
xor al, 255 ; make al abs(v)
inc al ; add 1

bright_not_neg:
shr ecx, 4 ; len = len / 16 (shift right 4)
mov ah, al
pinsrw xmm0, ax, 0
pinsrw xmm1, ax, 1
pinsrw xmm1, ax, 2
pinsrw xmm1, ax, 3
pinsrw xmm1, ax, 4
pinsrw xmm1, ax, 5
pinsrw xmm1, ax, 6
pinsrw xmm1, ax, 7
test eax, 0xf0000000 ; if v was negative, ; make darker
jnz dark_loop

bright_loop:
movdqa xmm0, [edi] ; move aligned double quadword
paddusb xmm0, xmm1 ; packed add unsigned
movdqa [edi], xmm0
add edi, 16 ; ptr = ptr + 16
loop bright_loop ; while (count>0)
jmp exit

dark_loop:
movdqa xmm0, [edi]
psubusb xmm0, xmm1
movdqa [edi], xmm0
add edi, 16 ; ptr=ptr+16
loop dark_loop ; while (count>0)
exit:
popall
ret

Brightness – SSE ASM

PINSRW (Packed Insert Word) moves the lower word in a 32-bit integer register or 16-bit word from memory into one of the 8 word locations in destination MMX/SSE register. The insertion is done in such a way that the 7 other words from the destination register are left untouched.

MOVDQA (move aligned double) Moves a double quadword from the source operand (second operand) to the destination operand (first operand).

PADDSUB (Packed Add Unsigned with Saturation) instruction adds packed unsigned byte integers. When an individual byte result is beyond the range of an unsigned byte integer (that is, greater than 0xFF), the saturated value of 0xFF is written to the destination operand.

PSUBUSB (Packed Subtract Unsigned with Saturation) instruction subtracts packed unsigned byte integers. When an individual byte result is less than zero (a negative value), the saturated value of 0x00 is written to the destination operand.
void brightness_sse (unsigned char *buffer, int len, int v)
{
  __m128 pixel_vector;
  __m128 value_vector;
  int t;

  if (v > 0) {
    value_vector = _mm_set1_epi8( v ); /* PINSRW, SHUFPS, etc..*/

    for (t = 0; t < len; t += 16) {
      pixel_vector = (__int128 *)(buffer+t); /* MOVQ */
      pixel_vector = _mm_adds_epi8(pixel_vector, value_vector); /* PADDUSB */
      *(__m128 *)(buffer+t)) = pixel_vector; /* MOVQ */
    }
  } else { % (v <= 0)
    v=-v;
    value_vector = _mm_set1_epi8(v);

    for (t = 0; t < len; t += 16) {
      pixel_vector = (__int128 *)(buffer+t); /* MOVQ */
      pixel_vector = _mm_subs_epi8(pixel_vector, value_vector); /* PSUBUSB */
      *(__m128 *)(buffer+t)) = pixel_vector; /* MOVQ */
    }
  }
}
Summary

- The x86 is at first glance "homogeneous", but has several heterogeneous micro-cores able to perform different types of operations

- MMX/SSE/AVX instructions (or intrinsic functions) can be used for SIMD operations – single operation on multiple data elements

- Next, more SSE examples on Thursday