IN5050:
Programming heterogeneous multi-core processors

SIMD (and SIMT)

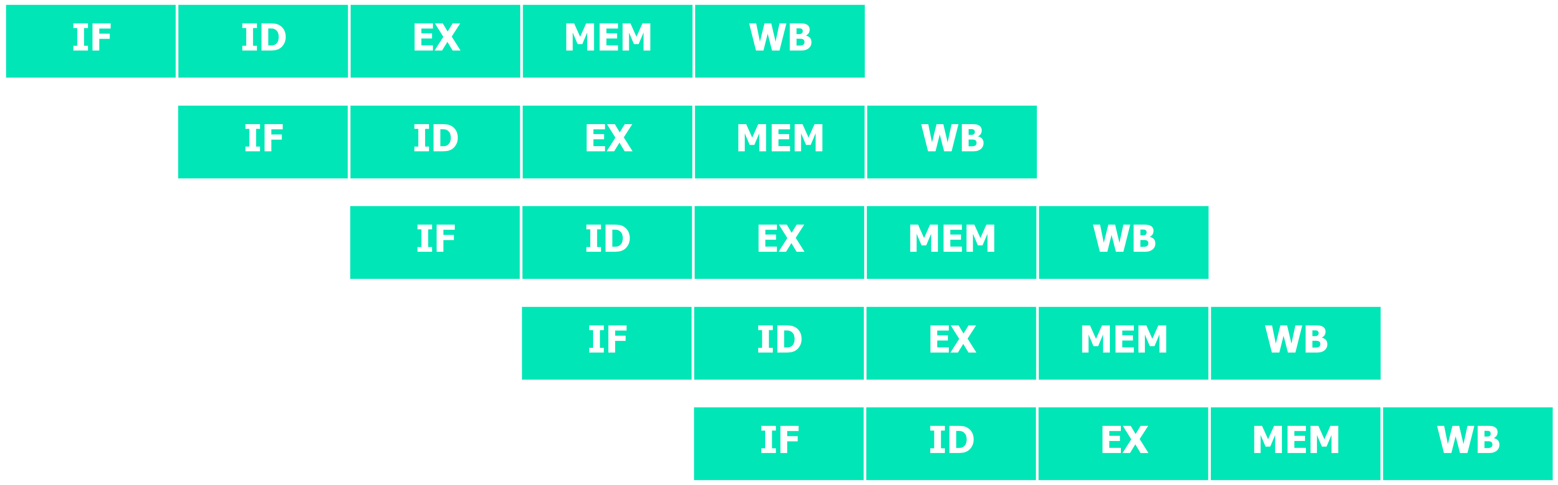
single scull: one is fast

quad scull: many are faster

Single worker digging the ground
2 days to complete
More than 1 worker digging the ground
1 day to complete

Types of Parallel Processing/Computing?

- Bit-level parallelism
  - 4-bit → 8-bit → 16-bit → 32-bit → 64-bit → …
- 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: Scheduling on a multicore

Types of Parallel Processing/Computing?

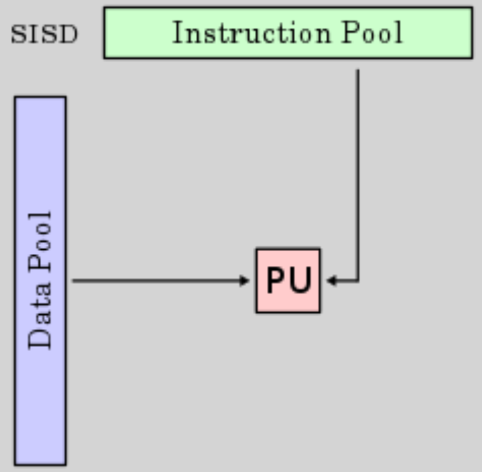
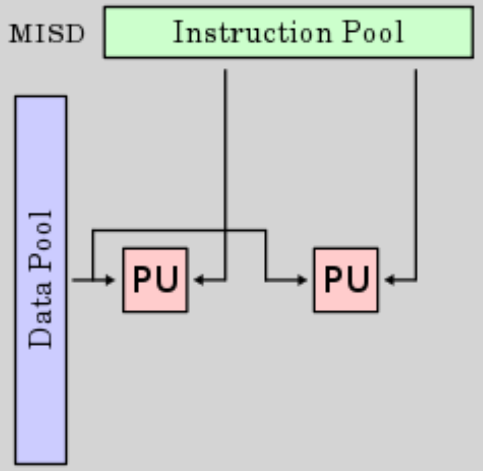
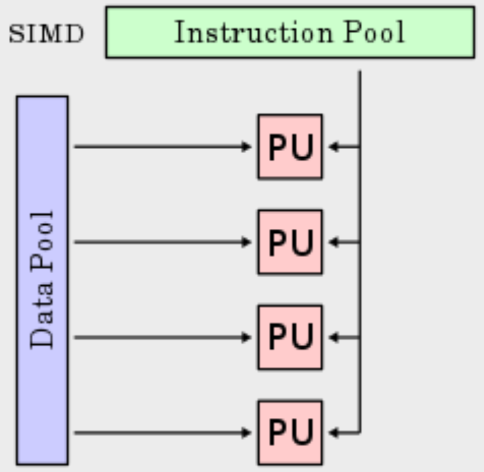
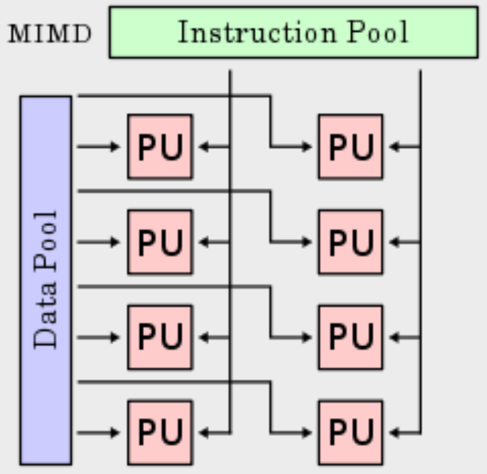
- **Data** parallelism
  - Distribution of data across different parallel computing nodes

  - Data parallelism is achieved when each processor performs the **same** task on different pieces of the data
  - Examples?

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

  - When should we not use data parallelism?

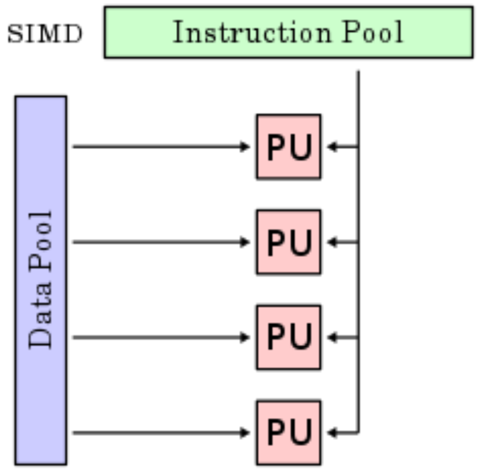
Flynn's taxonomy

	Single instruction	Multiple instruction
Single data		
Multiple data	 SIMD Instruction Pool Data Pool PU PU PU PU	 MIMD Instruction Pool Data Pool PU PU PU PU PU PU PU PU

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) with 72x 4096 bit registers

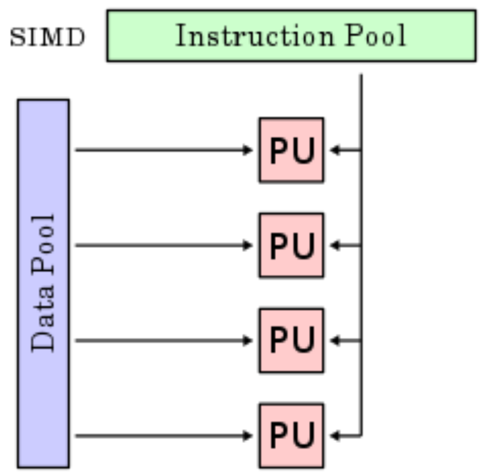
    - Intel
    - ARM
    - …



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)
- …



- Instruction sets:
  - MMX
  - SSE
  - AVX
  - AltiVec
  - 3DNow!
  - NEON
  - …

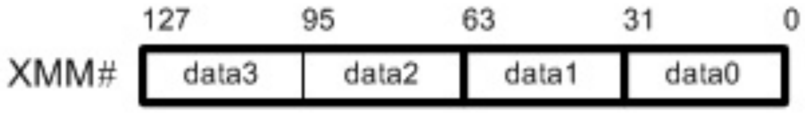
Vector Instruction sets

- **MMX**
  - MMX is officially a meaningless initialism trademarked by Intel; unofficially,
    - MultiMedia eXtension
    - Multiple Math eXtension
    - Matrix Math eXtension
  - Introduced on the “Pentium with MMX Technology” in 1998.
  - SIMD computation processes multiple data in parallel with a single instruction, resulting in significant performance improvement; MMX gives 2 x 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.
  - **3DNow!** was the AMD extension of MMX (closed down in 2010).

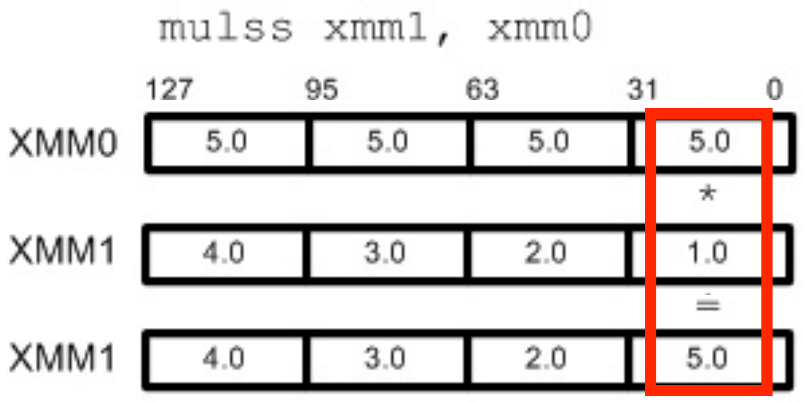
Vector Instruction sets

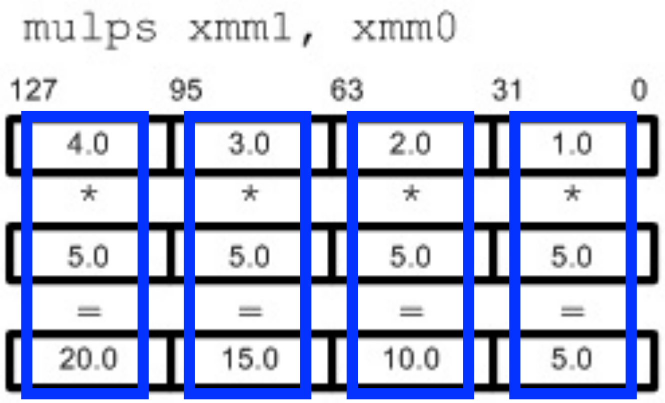
Altivec has same size (128-bit),
has some more advanced instructions,
but has seem to lose the performance battle

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



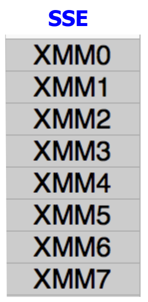
  - Single or packed scalar operations: `__SS` vs `__PS`





Vector Instruction sets

- **AVX**
  - Advanced Vector Extensions (AVX)
  - SIMD; 8 simultaneous 32-bit computations



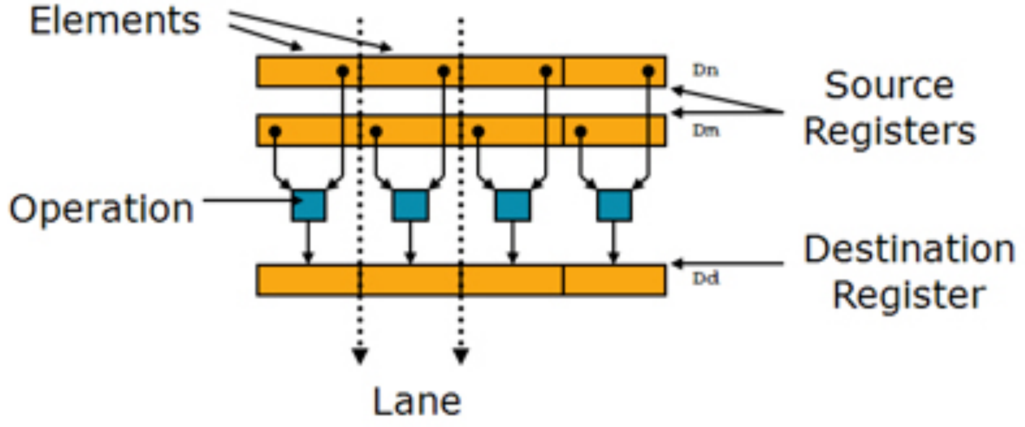
  - A new-256 bit instruction set extension to SSE
  - 16-registers available in x86-64
  - Registers renamed from XMM_i to YMM_i

  - Yet a proposed extension is AXV-512
  - A 512-bit extension to the 256-bit XMM
  - supported in Intel's Xeon Phi x200 (Knights Landing) and Skylake-SP.
  - Will be in future Intel CPUs for desktop (Ice Lake)

Vector Instruction sets

- **NEON**
  - Also known as ”Advanced SIMD Extensions”.
  - Introduced by ARM in 2004 to accelerate media and signal processing
    - NEON can for example execute MP3 decoding on CPUs running at 10 MHz
  - 128-bit SIMD Extension for the ARMv7 & ARMv8.
    - Data types can be: signed/unsigned 8-bit, 16-bit, 32-bit or 64-bit.

More next week – home exam!



SIMD: Why not just “auto-magical”?

- Why should I bother? Libraries, frameworks and compiler optimizations can do the work for me…

- True… but… lot of performance at stake!
  - simplified
    - 4 cores can potentially give a 4x speedup
    - AVX-512 can potentially give an 8x speedup
    - Combined, 32x!!

  - optimizers are good, but sometimes fail to meet your needs

  - learn invaluable understanding of the best use of vectorization!

Vector Multiplication using SSE (and AVX)

Element-wise Vector Multiplication

- Find element-wise product of 2 vectors:

a_1	a_2	a_3	a_4	a_5	a_6	…	a_n
*	*	*	*	*	*		*
b_1	b_2	b_3	b_4	b_5	b_6	…	b_n
=	=	=	=	=	=		=
a_1*b_1	a_2*b_2	a_3*b_3	a_4*b_4	a_5*b_5	a_6*b_6	…	a_n*b_n

```
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

```
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 x 32-bit operations in parallel using the 128-bit registers
→ unroll loop to a 4-element operation

```
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:

```
void vec_eltwise_product_unrolled(vec_t* a, vec_t* b, vec_t* c)
{
        size_t i;
        for (i =
```

```
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];
```

operations on these 4 elements can be performed in parallel

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

```
; 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 elemens
  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.

LOOP decrements ECX and jumps to the address specified by arg unless decrementing ECX caused its value to become zero.

Element-wise Vector Multiplication

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

```
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

```
; 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
```

```
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

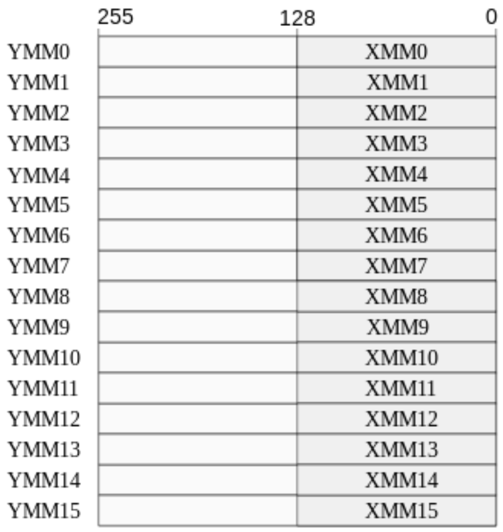
- Use SSE intrinsic functions:

```
; 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 elemens
  loop for-loop
exit:
  popall
  ret
```

```
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_i to YMM_i
  - available from Intel's Sandy Bridge and AMD's Bulldozer prosessors (2011)

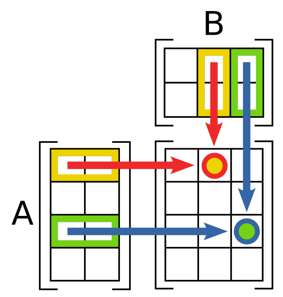


```
void vec_eltwise_product_SSE(vec_t* a, vec_t* b, vec_t* c)
{
          size_t i;
          __m128 va;
          __m128 vb;
          __m128 vc;
          for (i = 0; i < a->size; i+= 4) {
                    va = _mm_loadu_ps(&a->data[i]);
                    vb = _mm_loadu_ps(&b->data[i]);
    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

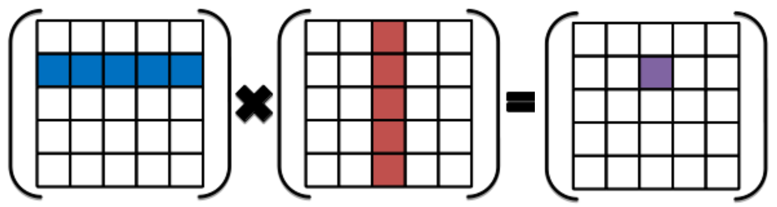
```
void vec_eltwise_product_SSE(vec_t* a, vec_t* b, vec_t* c)
{
          size_t i;
          __m128 va;
          __m128 vb;
          __m128 vc;
          for (i = 0; i < a->size; i+= 4) {
                    va = _mm_loadu_ps(&a->data[i]);
                    vb = _mm_loadu_ps(&b->data[i]);
    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

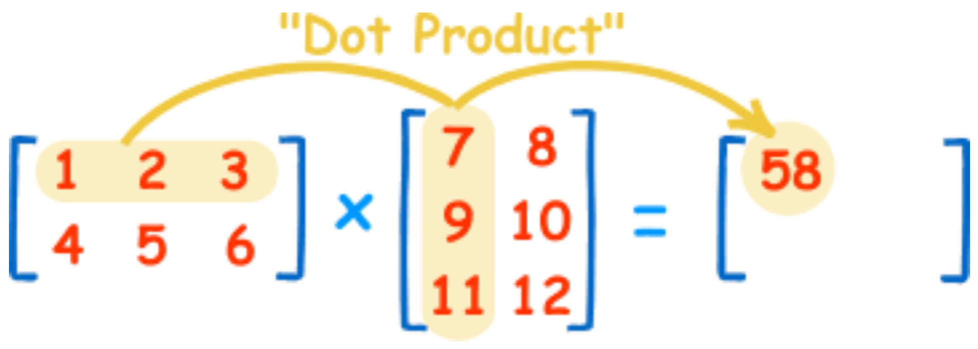
```
→ __m256 va;
→ __m256 vb;
→ __m256 vc;
→ i += 8;
→ va = _mm256_loadu_ps(&a->data[i]);
→ vb = _mm256_loadu_ps(&b->data[i]);
→ vc = _mm256_mul_ps(va, vb);
→ _mm256_storeu_ps(&c->data[i], vc);
```

$$\begin{array}{cccc} & & \begin{array}{cc}\vec{b_1} & \vec{b_2}\\ \downarrow & \downarrow\end{array} & \\ \begin{array}{c}\vec{a_1}\rightarrow\\ \vec{a_2}\rightarrow\end{array} & \begin{bmatrix}1&7\\2&4\end{bmatrix} \cdot & \begin{bmatrix}3&3\\5&2\end{bmatrix} & = \begin{bmatrix}\vec{a_1}\cdot\vec{b_1} & \vec{a_1}\cdot\vec{b_2}\\ \vec{a_2}\cdot\vec{b_1} & \vec{a_2}\cdot\vec{b_2}\end{bmatrix} \\ & A & B & C \end{array}$$



Matrix multiplication using SSE





Matrix Multiplication

$$
\begin{array}{c}
\rightarrow \\ \rightarrow \\ \rightarrow \\ \rightarrow
\end{array}
\begin{bmatrix}
\boxed{1} & \boxed{1} & \boxed{1} & \boxed{1} \\
2 & 2 & 2 & 2 \\
3 & 3 & 3 & 3 \\
4 & 4 & 4 & 4
\end{bmatrix}
\times
\begin{bmatrix}
\boxed{1} \\ \boxed{2} \\ \boxed{3} \\ \boxed{4}
\end{bmatrix}
=
\begin{array}{ccccccc}
1 & + & 2 & + & 3 & + & 4 \\
2 & + & 4 & + & 6 & + & 8 \\
3 & + & 6 & + & 9 & + & 12 \\
4 & + & 8 & + & 12 & + & 16
\end{array}
=
\begin{bmatrix}
10 \\ 20 \\ 30 \\ 40
\end{bmatrix}
$$

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];

   vout[1] =  elts[1][0] * vin[0]  +  elts[1][1] * vin[1] +
              elts[1][2] * vin[2]  +  elts[1][3] * vin[3];

   vout[2] =  elts[2][0] * vin[0]  +  elts[2][1] * vin[1] +
              elts[2][2] * vin[2]  +  elts[2][3] * vin[3];

   vout[3] =  elts[3][0] * vin[0]  +  elts[3][1] * vin[1] +
              elts[3][2] * vin[2]  +  elts[3][3] * vin[3];

   printf("%f %f %f %f\n", 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][2] * vin[2]

   vout[1] =  elts[1][0] * vin[0]  +  elts[1][1] * vin[1] +
              elts[1][2] * vin[2]  +  elts[1][3] * vin[3];

   vout[2] =  elts[2][0] * vin[0]  +  elts[2][1] * vin[1] +
              [2][2] * v           +  elts[2][3] * vin[3];

              [3][0] * v           +  elts[3][1] * vin[1] +
              [3][2] * v           +  elts[3][3] * vin[3];

            f %f\n", v             ], vout[1], vout[2], vout[3]);
}
```

Efficient multiplication,

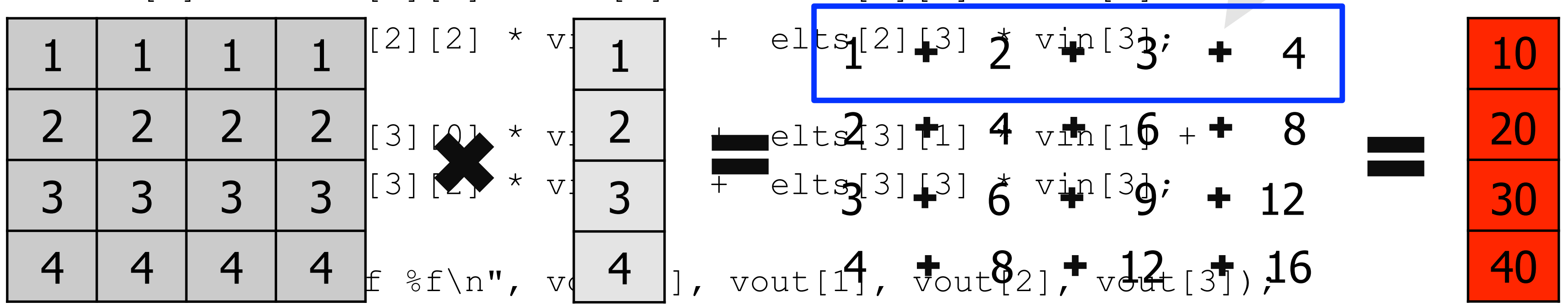
`MULPS` a, b = [A0*B0, A1*B1, A2*B2, A3*B3]

… but no "efficient" way to do the horizontal add:

`HADDPS` a, b = [B0+B1, B2+B3, A0+A1, A2+A3]

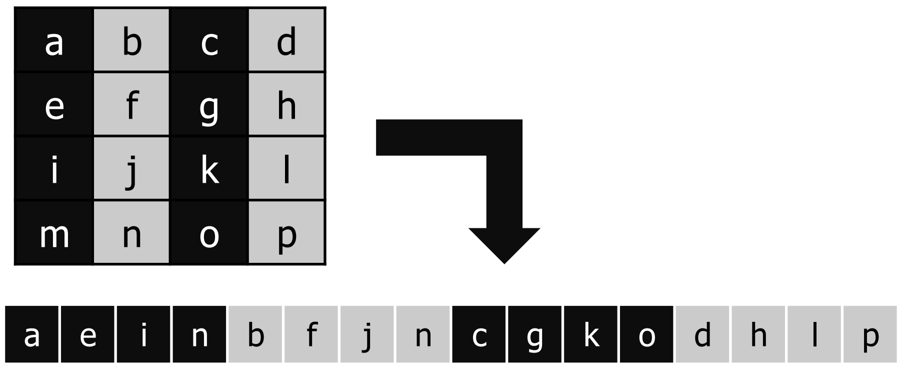
after (first row) multiplication we have x = [1, 2, 3, 4], add:

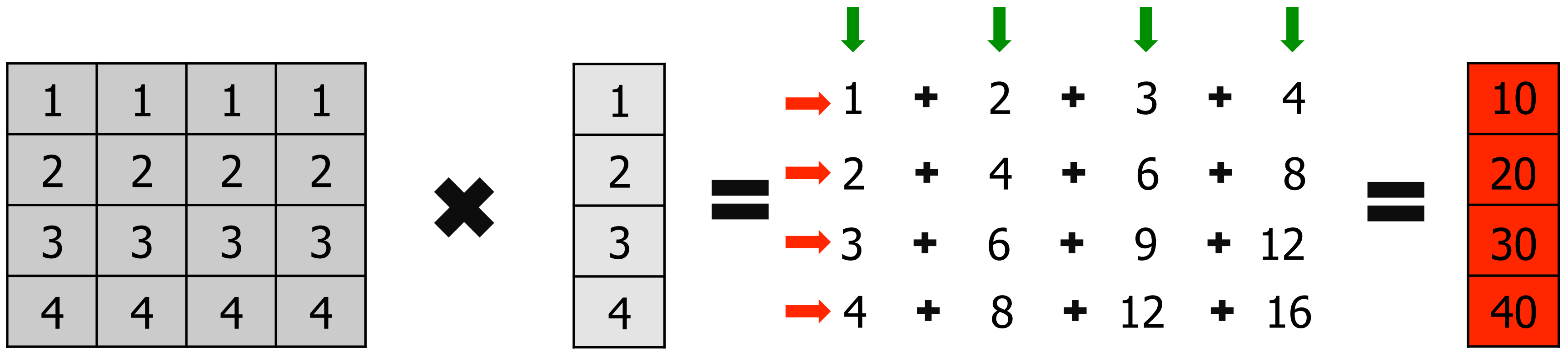
1. y = `__mm_hadd_ps(x, x)` = [1+2, 3+4, 1+2, 3+4]
2. z = `__mm_hadd_ps(y, y)` = [1+2+3+4, 1+2+3+4, 1+2+3+4, 1+2+3+4]
3. move Z0 to destination register



Matrix Multiplication – SSE

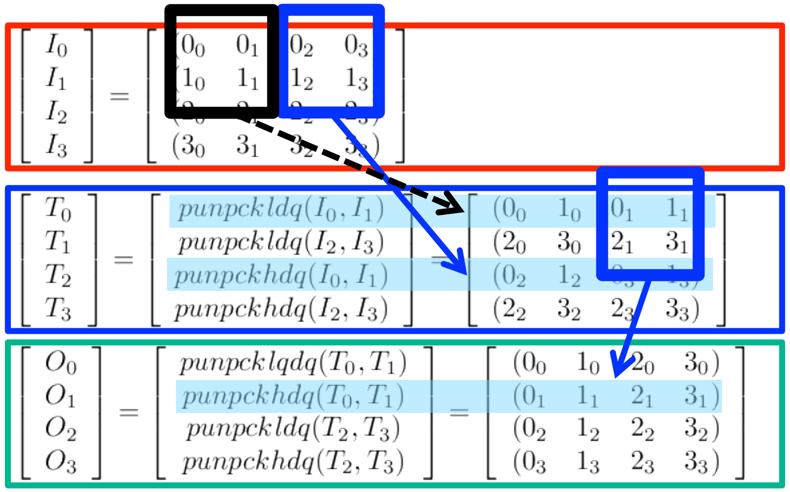
Assuming **elts** in a COLUMN-MAJOR order:





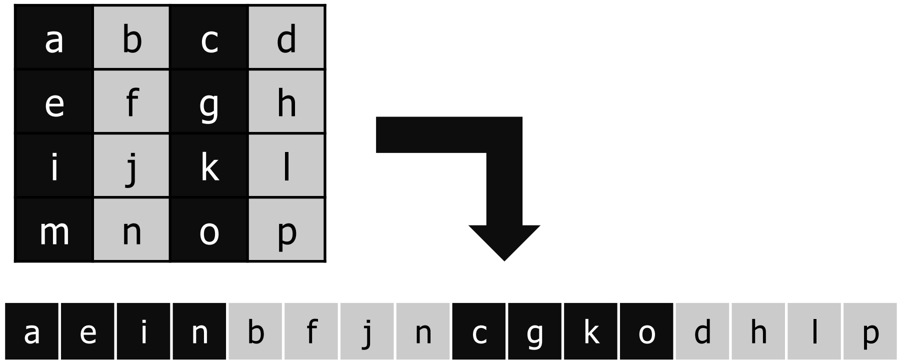
Matrix Multiplication – SSE

Store **elts** in column-major order – *transpose* using **unpack low/high-order data elements**:



$I_0, I_1, I_2, I_3 = (0_0\ 0_1\ 0_2\ 0_3), (1_0\ 1_1\ 1_2\ 1_3), (2_0\ 2_1\ 2_2\ 2_3), (3_0\ 3_1\ 3_2\ 3_3)$
$T_0 = punpckldq(I_0, I_1) = (0_0\ 1_0\ 0_1\ 1_1)$
$T_1 = punpckldq(I_2, I_3) = (2_0\ 3_0\ 2_1\ 3_1)$
$T_2 = punpckhdq(I_0, I_1) = (0_2\ 1_2\ 0_3\ 1_3)$
$T_3 = punpckhdq(I_2, I_3) = (2_2\ 3_2\ 2_3\ 3_3)$
$O_0 = punpcklqdq(T_0, T_1) = (0_0\ 1_0\ 2_0\ 3_0)$
$O_1 = punpckhdq(T_0, T_1) = (0_1\ 1_1\ 2_1\ 3_1)$
$O_2 = punpckldq(T_2, T_3) = (0_2\ 1_2\ 2_2\ 3_2)$
$O_3 = punpckhdq(T_2, T_3) = (0_3\ 1_3\ 2_3\ 3_3)$

Assuming **elts** in a COLUMN-MAJOR order:



```
<load matrix into I0, ..I3 __m128i registers >

/* Interleave values */
__m128i T0 = _mm_unpacklo_epi32(I0, I1);
__m128i T1 = _mm_unpacklo_epi32(I2, I3);
__m128i T2 = _mm_unpackhi_epi32(I0, I1);
__m128i T3 = _mm_unpackhi_epi32(I2, I3);

/* Assigning transposed values back into I[0-3] */
O0 = _mm_unpacklo_epi64(T0, T1);
O1 = _mm_unpackhi_epi64(T0, T1);
O2 = _mm_unpacklo_epi64(T2, T3);
O3 = _mm_unpackhi_epi64(T2, T3);
```

__m128i _mm_unpack**lo**_epi32 (__m128i a, __m128i b);

Interleaves the <u>lower</u> 2 signed or unsigned 32-bit integers in a with the lower 2 signed or unsigned 32-bit integers in b.

__m128i _mm_unpack**hi**_epi32 (__m128i a, __m128i b);

Interleaves the <u>upper</u> 2 signed or unsigned 32-bit integers in a with the upper 2 signed or unsigned 32-bit integers in b.

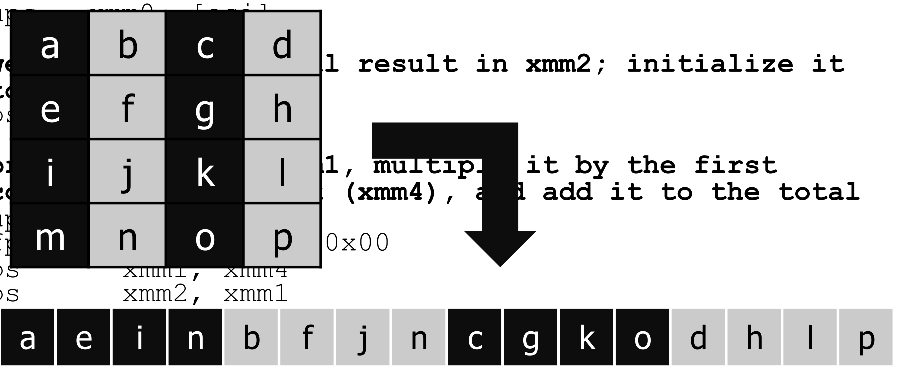
__m128i _mm_unpack**hi/lo**_epi**64** (__m128i a, __m128i b);

Interleaves the <u>upper / lower</u> signed or unsigned 64-bit integers in a with the upper signed or unsigned 64-bit integers in b.

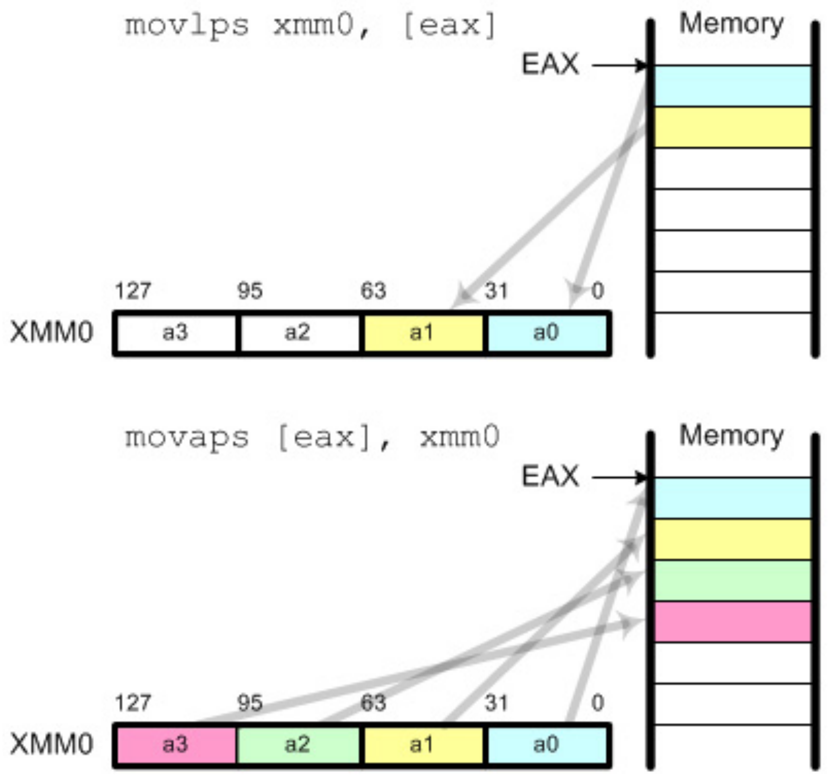
Matrix Multiplication – SSE

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

        // load columns of (transposed) 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.
        movu

        // we                          l result in xmm2; initialize it
        // to
        xorp

        // br                          1, multip   it by the first
        // co                          (xmm4), a   d add it to the total
        movu
        shuf                              0x00
        mulps         xmm1,  xmm4
        addps         xmm2,  xmm1

        // r
        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 **elts** in a COLUMN-MAJOR order:



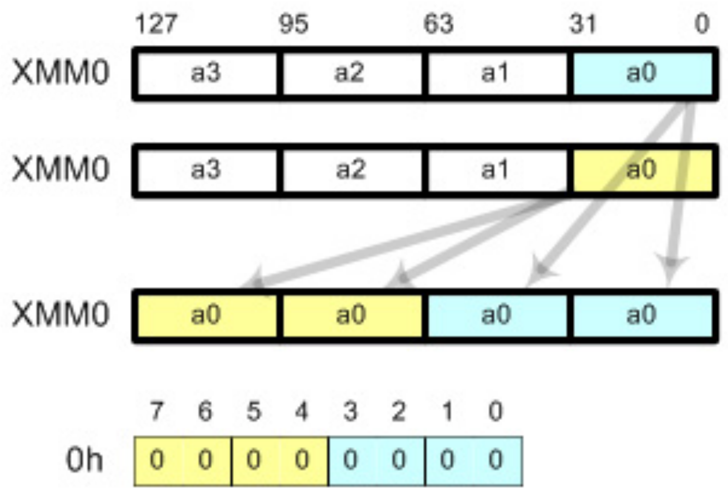
Matrix Multiplication – SSE



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

        // load columns of (transposed) 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
}
```

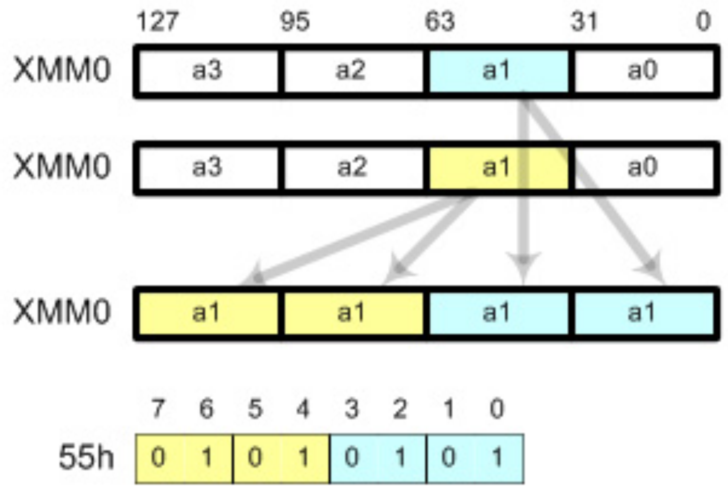
Matrix Multiplication – SSE

shufps xmm0, xmm0, 0h

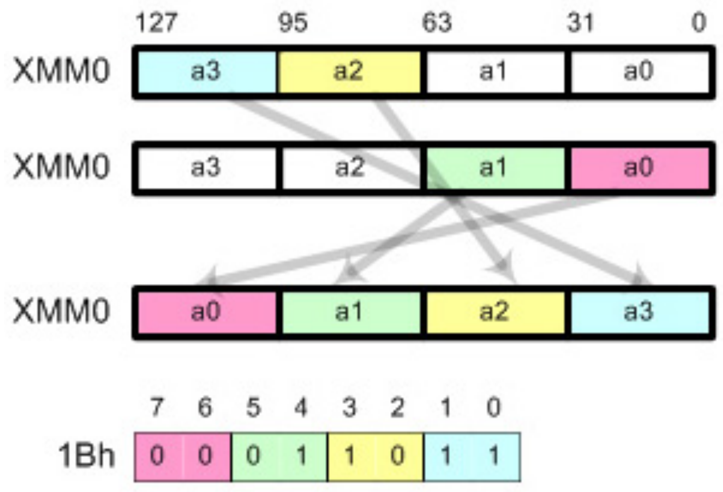


```
__asm {
    mov         esi, VIN
    mov         edi, VOUT

    // load columns of (transposed) 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
}
```

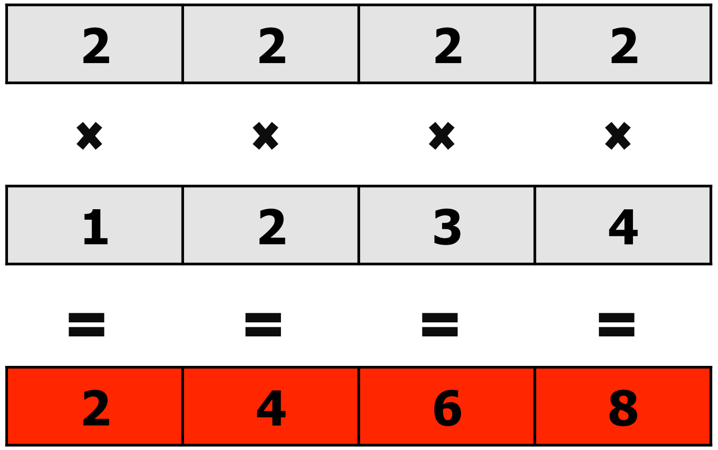
shufps xmm0, xmm0, 55h



shufps xmm0, xmm0, 1Bh

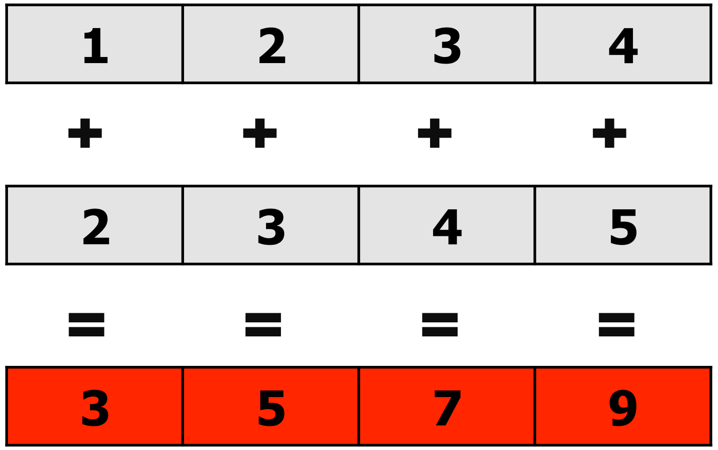


Matrix Multiplication – SSE



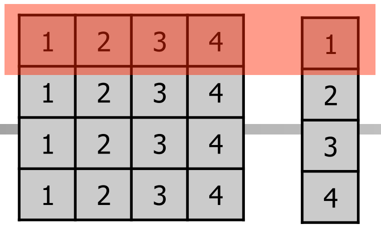
```
__asm {
        mov             esi, VIN
        mov             edi, VOUT

        // load columns of (transposed) 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
}
```

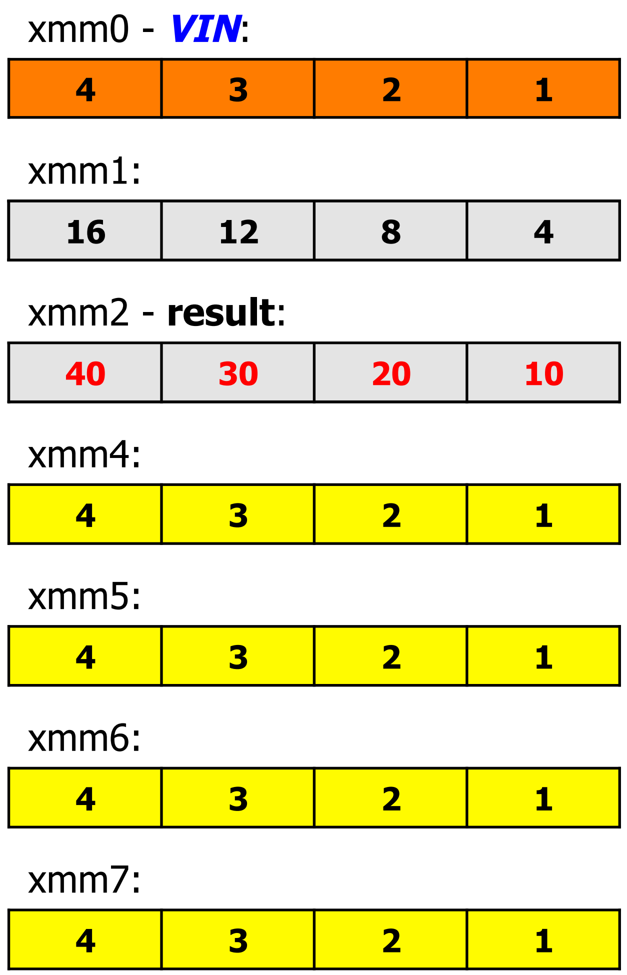
Matrix Multiplication – SSE



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

        // load columns of (transposed) 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

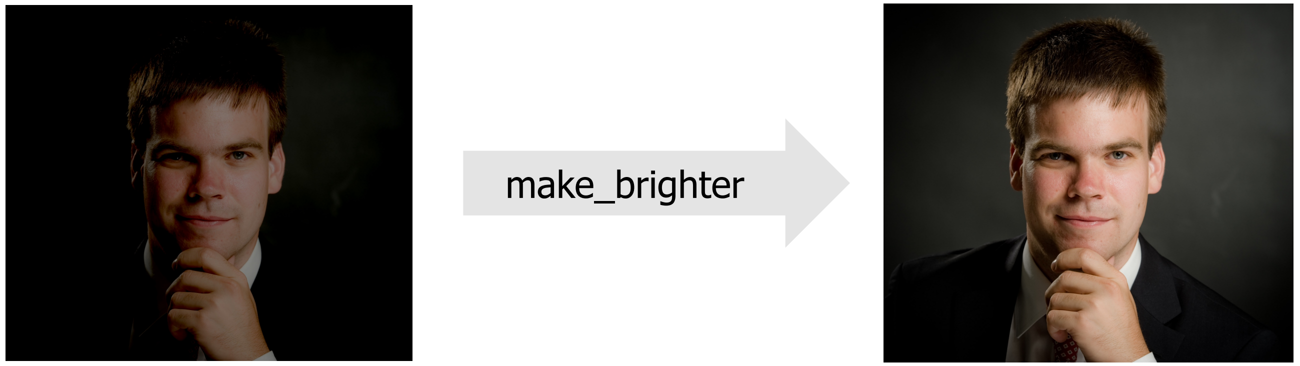




```
__asm {
    mov             esi, VIN
    mov             edi, VOUT

    // load columns of (transposed) 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
}
```

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 (luma - brightness).

Brightness – main

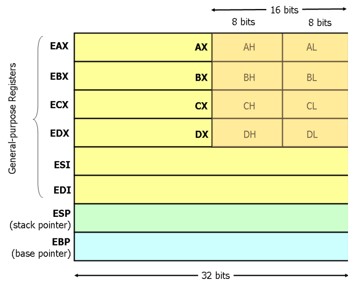
```
<.. includes ..>

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 x 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 >
}
```

Brightness – Naïve C

```
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;
      }
  }
}
```

What is the potential for parallelization using SSE-vectors?

Brightness – SSE ASM



```
; 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           ; xmm1=(v,v,v,v,v,v,v,v,v,v,v,v,v,v,v,v)
  pinsrw xmm1, ax, 0   ; Packed Insert Word
  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, 0xff000000  ; if v was negative,
  jnz dark_loop         ; make darker

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
```

```
;; Fewer instructions ALTERNATIVE
;; using shufps
move ah, al
pinsrw xmm1, ax, 0
pinsrw xmm1, ax, 1
shufps xmm1, xmm1, 0x00
```

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).

PADDUSB (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.

Brightness – SSE Intrinsics

```
void brightness_sse (unsigned char *buffer, int len, int v)
{
  __m128 pixel_vector;
  __m128 value_vector;
  int t;

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

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

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

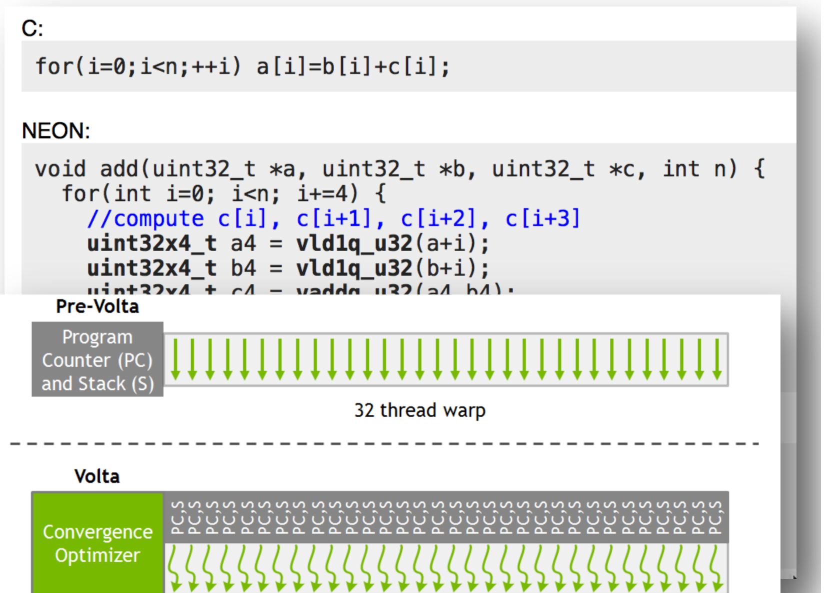
Brightness – SSE Intrinsics

```
void brightness_sse (unsigned char *buffer, int len, int v)
{
  __m128 pixel_vector;
  __m128 value_vector;
  int t;

  if (v > 0) {
    < make v char >
    value_vector = _mm_set1                                  HUFPS, etc..*/

    for (t = 0; t < len; t +=
      pixel_vector = (__int128
      pixel_vector = _mm_adds_
      *((__m128 *)(buffer+t)) =
    }
  } else { % (v <= 0)
    v=-v;
    < make v char >
    value_vector = _mm_set1_epi8(v)

    for (t = 0; t < len; t += 16) {
      pixel_vector = (__int128 *)(bu                       /* MOVDQA  */
      pixel_vector = _mm_subs_epi8(pi       , value_vector); /* PSUBUSB */
      *((__m128 *)(buffer+t)) = pixel_vector;              /* MOVDQA  */
    }
  }
}
```

SIMT
single instruction multiple threads

SIMT

- Single Instruction Multiple Threads (SIMT) ≈ SIMD + multithreading
- Nvidia's CUDA GPU microarchitecture introduced the SIMT execution model where multiple independent threads execute concurrently using a “single instruction”
- Flexibility: SIMD < SIMT < SIM
  - SIMT is more flexible in SIMD in three areas:
    - Single instruction, multiple register sets
    - Single instruction, multiple addresses
    - Single instruction, multiple flow path
- Performance: SIMD > SIMT > SIM
  - Less flexibility usually means higher performance
  - However, massive parallelism in SIMT?
    (2-16x (64x) operations vs. 1000+ threads?)
- SIMT – easier code?
- Now: CUDA = SIMD? SIMT? MIMD? MIMT?

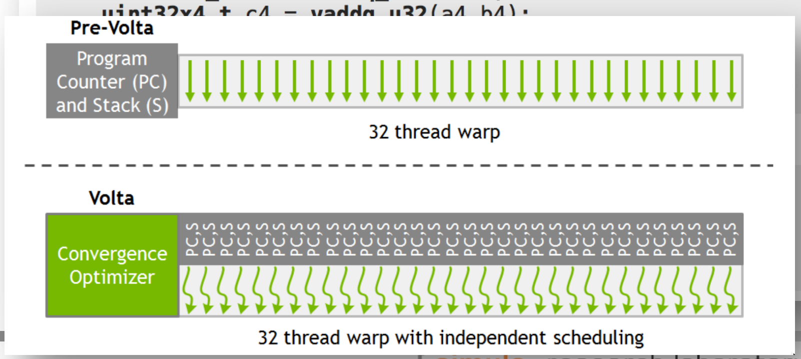


C:

```
for(i=0;i<n;++i) a[i]=b[i]+c[i];
```

NEON:

```
void add(uint32_t *a, uint32_t *b, uint32_t *c, int n) {
  for(int i=0; i<n; i+=4) {
    //compute c[i], c[i+1], c[i+2], c[i+3]
    uint32x4_t a4 = vld1q_u32(a+i);
    uint32x4_t b4 = vld1q_u32(b+i);
    uint32x4_t c4 = vaddq_u32(a4,b4);
```


Pre-Volta
Program Counter (PC) and Stack (S)
32 thread warp
Volta
Convergence Optimizer
PC,S
32 thread warp with independent scheduling

SUMMARY

Summary

- Vector instructions (or intrinsic functions) can be used for SIMD operations – single operation on multiple data elements
- SIMT is more or less SIMD + multithreading
- Tomorrow,
  video coding and a practical example using SSE/AVX
- Next week,
  ARM and NEON (to be used in your home exam)

Some SSE/AVX Resources

- SSE instructions:
  https://msdn.microsoft.com/en-us/library/t467de55(v=vs.90).aspx

- Intrinsics:
  https://software.intel.com/en-us/node/582251