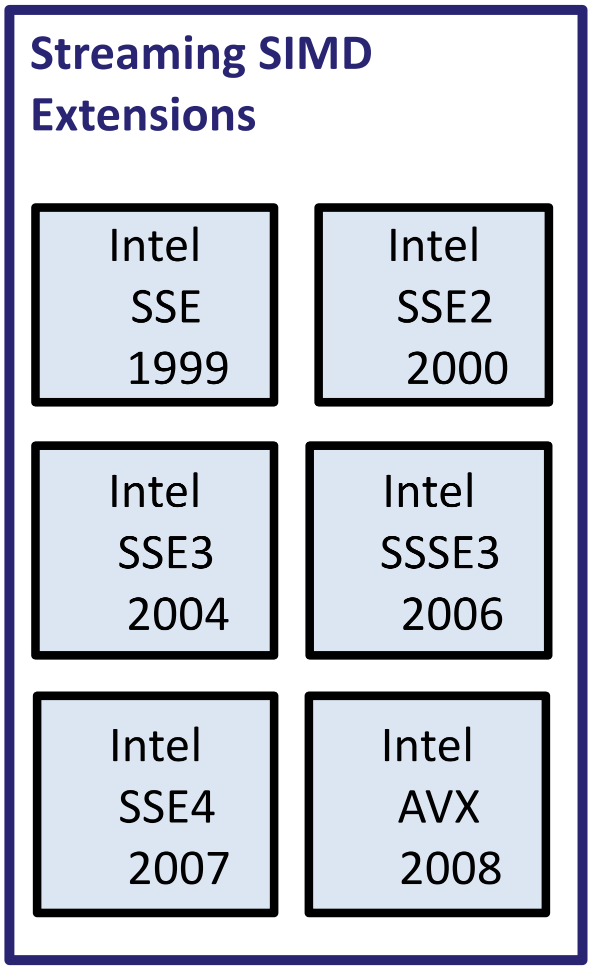
CS 293S SIMD: Single Instruction Multiple Data

Yufei Ding

SIMD: Single Instruction Multiple Data

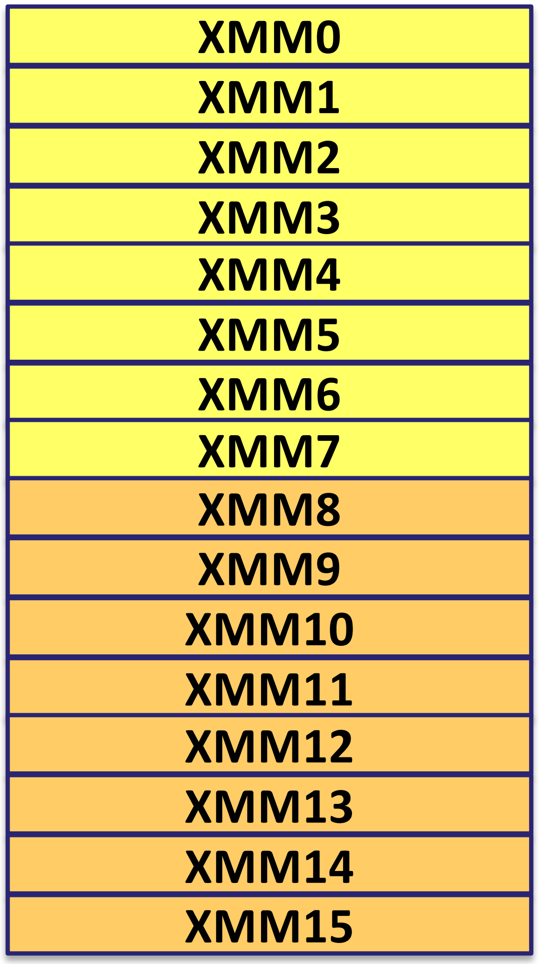
Streaming SIMD Extensions (SSE)



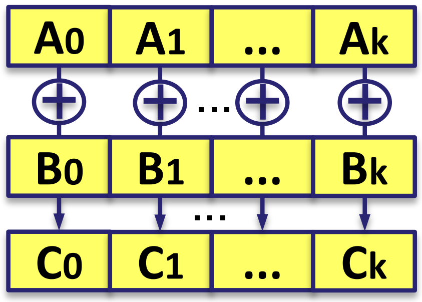
- Extensions to the IA-32 and x86-64 instruction sets for parallel SIMD operations on packed data
- MMX – Multimedia Extensions
  - introduced in the Pentium processor 1993
  - 64-bit vector registers
  - supports only integer operations, not used much any more
- SSE – Streaming SIMD Extension
  - introduced in Pentium III 1999, supported by most modern processors
  - 128 bit vector registers
  - support for single-precision floating point operations
- SSE2, SSE3, SSSE3, SSE4.1, SSE4.2
- AVX – Advanced Vector Extensions
  - announced in 2008, supported in the Intel Sandy Bridge processors, and later – extends the vector registers to 256 bits

SSE vector registers

- SSE introduced a set of new 128-bit vector registers
  - 8 XMM registers in 32-bit mode
  - 16 XMM registers in 64-bit mode
- The XMM registers are real physical registers
  - not aliased to any other registers
  - independent of the general purpose and FPU/MMX registers
- XMM registers can be accessed in 32-bit, 64-bit or 128-bit mode
  - only for operations on data, not addresses
- There is also a 32 bit control and status register, called MXCSR
  - flag and mask bits for floating-point exceptions
  - rounding control bits
  - flush-to-zero bit
  - denormals-are-zero bit



Abstraction of SIMD Extensions



Loop programs

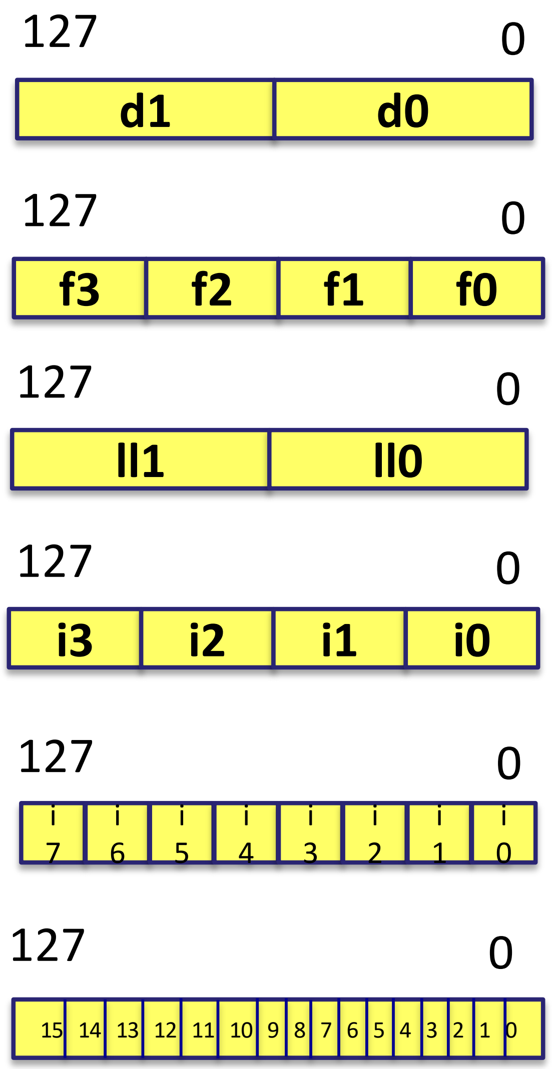
```
for i in (0, m)
   C[i] = A[i] + B[i]
```

More natural to be **vectorized**

- SIMD execution
  - performs an operation in parallel on an array of 2, 4, 8,16 or 32 values, depending on the size of the values
  - data parallel operation
- The operation can be a
  - data movement instruction
  - arithmetic instruction
  - logical instruction
  - comparison instruction
  - conversion instruction
  - shuffle instruction

SSE vector data type

- 2 double precision floating-point values
  - elements are of type *double*
- 4 single precision floating-point values
  - elements are of type *float*
- 2 64-bit integer values
  - elements are of type *long long*
- 4 32-bit integer values
  - elements are of type *int*
- 8 16-bit integer values
  - elements are of type *short int*
- 16 8-bit integer values
  - elements are of type *char*



Programming with vector instructions

- Automatic vectorization by the compiler
  - no explicit vectorized programming is needed, but requires a vectorizing compiler
  - have to arrange the code so that the compiler can recognize possibilities for vectorization
- Use vector intrinsic functions for vector operations
  - functions that implement vector instructions in a high-level language
  - requires detailed knowledge of the vector instructions
  - one function often implements one vector assembly language instruction

Automatic vectorization by compiler

- Requires a compiler with vectorizing capabilities
  - in gcc, vectorization is enabled by *–O3*
  - the Intel compiler, icc, can also do advanced vectorization
- The compiler automatically recognizes loops that can be implemented with vectorized code
  - easy to use, no changes to the program code are needed
- Only loops that can be analyzed and that are found to be suitable for SIMD execution are vectorized
  - does not guarantee that the code will be vectorized
  - has no effect if the compiler can not analyze the code and find opportunities for vector operations
- Pointers to vector arguments should be declared with the keyword *restrict*
  - guarantees that there are no aliases to the vectors
- Arrays that used in vector operations should be 16-byte aligned
  - this will automatically be the case if they are dynamically allocated

Use vector intrinsic functions

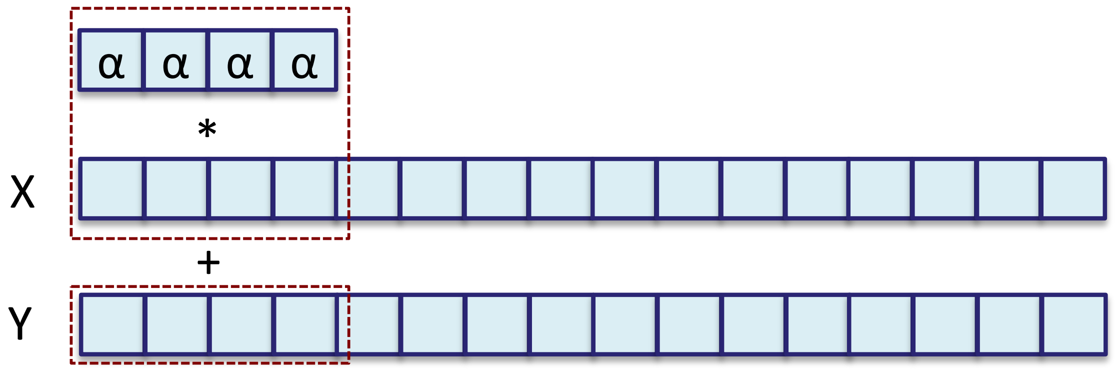
- Functions for performing vector operations on packed data
  - implemented as functions which call the corresponding vector instructions
  - implemented with inline assembly code
  - allows the programmer to use C function calls and variables
- Defines a separate C function for each vector instruction
  - there are also some intrinsic functions composed of several vector instructions
- Vectorized programming with intrinsic functions is very low-level
  - have to exactly specify the operations that should be done on the vector values
- Operate on the vector data types
- Often used for vector operations that can not be expressed as normal arithmetic operations
  - loading and storing of vectors, shuffle operations, type conversions, masking operations, ...

Example: SAXPY

- SAXPY (Single-precision Alpha X Plus Y)
  - computes Y = αX+Y, where α is a scalar value and X and Y are vectors of single-precision type
  - one of the vector operation in the BLAS library (Basic Linear Algebra Subprograms)

```
void saxpy(int n, float alpha, float *X, float *Y) {
    int i;
    for (i=0; i<n; i++)
        Y[i] = alpha*X[i] + Y[i];
}
```

- The vectorized code will do the computation on 4 values at a time
  - multiplies four values of *X* with *alpha*
  - adds the results to the corresponding four values of *Y*



Automatic vectorization by compiler

- Use the compiler switches *–O3* and *-ftree-vectorizer-verbose=1* to see reports about which loops were vectorized
  - a higher value gives more verbose output
  - the verbosity level 2 also prints out reasons why loops are not vectorized

```
gcc -O3 -ftree-vectorizer-verbose=1 saxpy1.c -o saxpy1

Analyzing loop at saxpy1.c:16
Vectorizing loop at saxpy1.c:16
16: created 1 versioning for alias checks.
16: LOOP VECTORIZED.
saxpy1.c:14: note: vectorized 1 loops in function.
```

  - NOTE: In new version of gcc, *-ftree-vectorizer-verbose* is deprecated in favor of *–fopt-info-vec*
- Vector elements should be aligned to 16 bytes
  - access to unaligned vector elements will fail
- Aliasing may prevent the compiler from doing vectorization
  - pointers to vector data should be declared with the *restrict* keyword

Use vector intrinsic functions

```
void saxpy(int n, float alpha, float *X, float *Y) {
  __m128 x_vec, y_vec, a_vec, res_vec;  /* Vector variables */
  int i;
  a_vec = _mm_set1_ps(alpha); /* Vector of 4 alpha values */
  for (i=0; i<n; i+=4) {
    x_vec = _mm_load_ps(&X[i]); /* Load 4 value from X */
    y_vec = _mm_load_ps(&Y[i]); /* Load 4 value from Y */
    res_vec = _mm_add_ps(_mm_mul_ps(a_vec, x_vec), y_vec); /* Compute */
    _mm_store_ps(&Y[i], res_vec);  /* Store the result */
  }
}
```

- Declare vector variables of the appropriate type
- Load data values into the variables
- Do arithmetic operations on the vector variables by calling intrinsic functions
  - it is possible to nest calls to intrinsic functions (they normally return a vector value)
  - can also use normal arithmetic expressions on the vector variables
- Load and store operations require that data is **16-byte aligned**
  - there are also corresponding functions for unaligned load/store: *_mm_loadu_ps* and *_mm_storeu_ps*

Vector data types

- The vector data types are defined in separate header files
  - depends on which vector extension is used

Instruction set	Header file	Registers	Length (bits)
MMX	mmintrin.h	MMX	64
SSE	xmmintrin.h	XMM	128
SSE2	emmintrin.h	XMM	128
SSE3	pmmintrin.h	XMM	128
SSE4.2	nmmintrin.h	XMM	128
AVX	immintrin.h	YMM	256

- When compile, use –msse4.2, -mavx (machine dependent code)
  - Some are default for gcc.
- Vector data types in SSE
  - __m128: four 32-bit floating-point values
  - __m128d: two 64-bit floating-point values
  - __m128i: 16 / 8 / 4 / 2 integer values, depending on the size of the integers

SSE intrinsics

- SSE intrinsic functions have a mnemonic name that tries to describe the operation
  - the function name starts with *_mm*
  - after that follows a name describing the operation: *add, mul, div, load, set*, ...
  - the next character specifies whether the operation is on a packed vector or on a scalar vaue: P stands for Packed and S for Scalar operation
  - the last character describes the data type
    - S – single precision floating point values
    - D – double precision floating point values
- Examples:
  - _mm_load_ps – load packed single-precision floating-point values
  - _mm_add_sd – add scalar double precision values
  - _mm_rsqrt_ps – reciprocal square root of four single-precision fp values
  - _mm_min_pd – minimum of the two double-precision fp values in the arguments
  - _mm_set1_pd – set the two double-precision elements to some value

SSE intrinsics

- Data movement and initialization
  - _mm_load_ps, _mm_loadu_ps, _mm_load_pd, _mm_loadu_pd, etc
  - _mm_store_ps, _mm_storeu_ps ...
  - _mm_setzero_ps
- Arithemetic intrinsics:
  - _mm_add_ss, _mm_add_ps, ...
  - _mm_add_pd, _mm_mul_pd
- More completed list:
  - https://software.intel.com/sites/landingpage/IntrinsicsGuide/#

Vectorizing conditional constructs

- The compiler will typically not be able to vectorize loops containing conditional constructs
- *Example*: conditionally assign an integer value to *A[i]* depending on some other value *B[i]*

```
// A, B, C and D are integer arrays
for (i=0; i<N; i++) {
     A[i] = (B[i] > 0) ? (C[i]) : (D[i]);
}
```

```
for (i=0; i<N; i++) {
     if (B[i] > 0)
        A[i] = C[i];
     else
        A[i] = D[i];
}
```

- This can be vectorized by first computing a Boolean mask which contains the result of the comparison: *mask[i] = (B[i] > 0)*
  - then the assignment can be expressed as A[i] = (C[i] && mask) || (D[i] && ¬mask)
  - where the logical operations AND (&&), OR (||) and NOT (¬) are done bitwise

Example

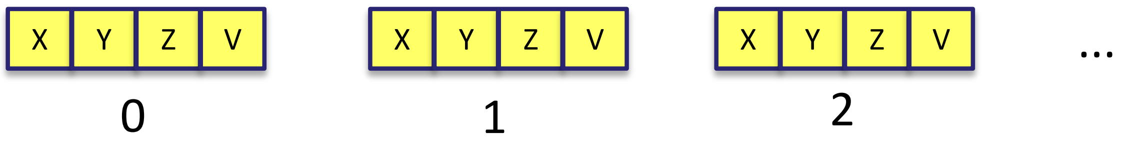
- As an example we vectorize the following (slightly modified) code

```
// A, B, C and D are integer arrays
for (i=0; i<N; i++) {
     A[i] = (B[i] > 0) ? (C[i] + 2) : (D[i] + 10);
}
```

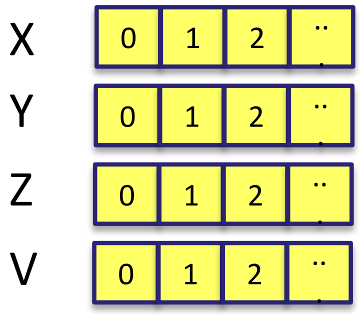
```
__m128i zero_vec = _mm_setzero_epi32(); // Vector of four zeros
__m128i two_vec = _mm_set1_epi32(2); // Vector of four 2's
__m128i ten_vec = _mm_set1_epi32(10); // Vector of four 10's

for (i=0; i<N; i+=4) {
        __m128i b_vec, c_vec, d_vec, mask, result;
        b_vec = _mm_load_si128((__m128i *)&B[i]); // Load 4 elements from B
        c_vec = _mm_load_si128((__m128i *)&C[i]); // Load 4 elements from C
        d_vec = _mm_load_si128((__m128i *)&D[i]); // Load 4 elements from D

        c_vec = _mm_add_epi32(c_vec, two_vec); // Add 2 to c_vec
        d_vec = _mm_add_epi32(d_vec, ten_vec); // Add 10 to d_vec
        mask = _mm_cmpgt_epi32(b_vec, zero_vec); // Compare b_vec to 0
        c_vec = _mm_and_si128(c_vec, mask); // AND c_vec and mask
        d_vec = _mm_andnot_si128(mask, d_vec); // AND d_vec with NOT(mask)
        result = _mm_or_si128(c_vec, d_vec); // OR c_vec with d_vec
        _mm_store_si128((__m128i *)&A[i], result); // Store result in A[i]
}
```

Arranging data for vector operations

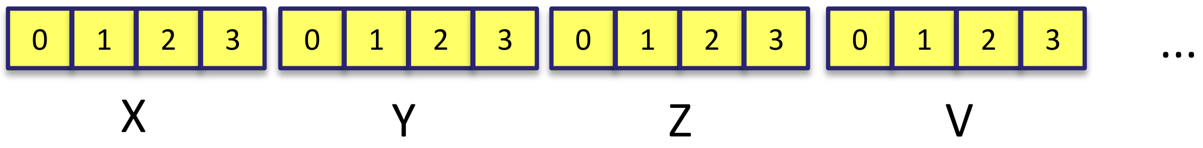
- It is important to organize data in memory so it can be accessed as vectors
  - consider a structure with four elements: x, y, z, v
- Array of structure:



- Structure of arrays:



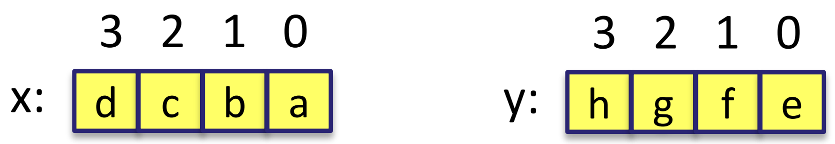
- Hybrid structure:



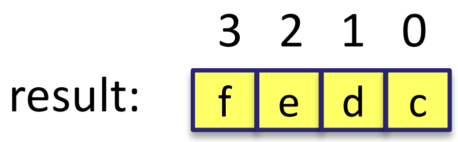
- Rearranging data in memory for vector operation is called *data swizzling*

Arranging data for vector operations

- It is often necessary to rearrange data so that they fit the vector operations
- SSE contains a number of instructions for shuffling the elements of a vector
- Example: *_mm_shuffle_ps (__m128 a, __m128 b, int i)*
  - the integer argument *i* is a bit mask which specifies how the elements of *a* and *b* should be rearranged
  - the macro _MM_SHUFFLE(b_3, b_2, a_1, a_0) creates a bit mask describing the shuffle operation
  - interleaves values from *a* and *b* in the order specified



  - result = _mm_shuffle_ps (x, y, _MM_SHUFFLE(1,0,3,2))



- There is a rich set of instructions for shuffling, packing and moving elements in vectors

Portability

- Explicitly vectorized code is not portable
  - only runs on architectures which support the vector extension that is used
  - code using SSE2 intrinsic functions will not run on processors without SSE2
- Can use conditional compilation in the code
  - the program contains both a vectorized version and a normal scalar version of the same computation
  - use *#ifdef* statements to choose the correct version at compile time
  - if the compiler switch *–msse2* is on, the GCC compiler defines a macro *__SSE2__*

```
#ifdef __SSE2__
// SSE2 version of the code
#else
// Normal scalar version of the code
#endif
```

- A benefit of this is also that a non-vectorized version of the code is available for reference