Programming with vector instructions

- **SIMD = Single Instruction stream, Multiple Data stream**
  - From Flynn’s taxonomy: SISD, SIMD, (MISD), MIMD
- **Extensions to the Intel and AMD x86 instruction set for parallel operations on packed integer or floating-point data**
  - data parallelism
  - parallel vector operations
  - applies the same operation in parallel on a number of data items packed into a 64-, 128- or 256-bit vector
  - also supports scalar operations on integer or floating-point values
- **Originally designed to speed up media processing applications**
  - can also be very useful in scientific computations and other numerically intensive applications
- **There are many different versions of SIMD extensions**
  - MMX, SSE, SSE2, SSE3, SSE4, 3DNow!, Altivec, VIS, AVX, AVX2, ...

MMX, SSE and AVX

- **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 – Streaming SIMD Extension 2 was introduced in Pentium 4, 2000
  - also supports double-precision floating point operations
  - later extensions: 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
Vector extensions

- The vector instructions have been introduced and gradually extended in multiple processor generations
  - the SSE extensions together include over 400 instructions
- We will cover programming with the SSE family of vector instructions
  - available in most modern processors
  - uses 16 dedicated registers of length 128 bits
  - programming with AVX is similar, vector length and instruction names differ

![Diagram showing the evolution of SIMD extensions from MMX to AVX]

Characteristics of SIMD operations

- The SIMD extensions were originally designed to speed up multimedia and communication applications
  - graphics and image processing
  - video and audio processing
  - speech recognition, ...
- Can also be used for other data-intensive scientific computations
- Applications can benefit from SIMD processing if they have the following characteristics
  - small integer or floating-point data types
  - small, highly repetitive loops
  - frequent additions, multiplications or other simple operations
  - compute-intense algorithms
  - data-parallelism, can operate on independent values in parallel
SIMD operation

- **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 \( \otimes \) can be a**
  - data movement instruction
  - arithmetic instruction
  - logical instruction
  - comparison instruction
  - conversion instruction
  - shuffle instruction

- **MMX**
  - Introduced in the Pentium processor 1997
  - Uses 8 64-bit MMX registers which are aliased to the x87 floating-point registers
    - can store 1, 2, 4 or 8 packed integer values
  - MMX registers can only hold integer data
    - not memory addresses
    - the general-purpose registers are used for addresses
  - Can not use the x87 floating-point unit and the MMX unit at the same time
    - they share the same set of registers
  - MMX operations are limited to integer values
    - the SSE and AVX extensions also provide operations on floating-point data
  - MMX is seldom used any more in modern processors
    - SSE or AVX is used instead
SSE

- Streaming SIMD Extension
  - introduced with the Pentium III processor 1999
  - supports single-precision floating point operations (not double precision)
- Adds 8 or 16 dedicated 128-bit vector registers to the processor architecture
  - called XMM0 – XMM15
- Parallel operations on packed single precision floating-point values
  - 128-bit packed single precision floating point data type
  - four IEEE 32-bit floating point values packed into a 128-bit field
  - data must be aligned in memory on 16-byte boundaries

SSE2

- Streaming SIMD Extension 2
  - introduced in the Pentium 4 processor 2001
  - extends SSE and replaces the MMX integer vector instructions
- Extends SSE with support for
  - packed double precision floating point-values
  - packed integer values
  - adds over 70 new instructions to the instruction set
- Operates on 128-bit entities in the XMM registers
  - data must be aligned on 16-bit boundaries when stored in memory
  - the restrictions on alignment were later relaxed
    - unaligned load/store instructions were introduced
AVX

- Advanced Vector Extensions
  - introduced in the Sandy Bridge microarchitecture in 2011
- Extends the vector registers to 256 bits
  - called YMM0 – YMM15
  - SSE instructions operate on the lower half of the YMM registers
- Introduces new three-operand instructions
  - one destination and two source operands: \( c = a \odot b \)

- AVX-512 will be supported in the Knights landing microarchitecture
  - scheduled to be shipped in 2015 according to Intel
- The Intel Xeon Phi processor supports a 512-bit vector processing unit
  - not compatible with SSE or AVX

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
**SSE instructions**

- The original SSE extension added 70 new instructions to the instruction set
  - 50 for SIMD single-precision floating-point operations
  - 12 for SIMD integer operations
  - 8 for cache control
  - Later extensions added more instructions
- Supports both packed and scalar single precision floating-point instructions
  - Operations on packed 32-bit floating-point values
    - Packed instructions have a suffix P
  - Operations on a scalar 32-bit floating-point value (the 32 LSB)
    - Scalar instructions have a suffix S
- Also included some 64-bit SIMD integer instructions
  - Replaces similar MMX instructions
  - Operations on packed integer values stored in MMX registers

**Packed and scalar operations**

- Packed SSE operations apply an operation in parallel on 2 or 4 floating-point values
- Scalar SSE operations apply an operation on a single (scalar) floating-point value
- The compilers use scalar SSE instructions for floating-point operations instead of the x87 floating point unit
SSE vector data types

- 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

- There are different ways to use vector instructions in a program

1. 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

2. Express the computation as arithmetic expressions on vector data types
   - declare variables of a vector type
   - express computations as normal arithmetic expression on the vector variables
   - the compiler generates vector instructions for the arithmetic operations

3. Use compiler 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

- 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 cannot 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

Example: SAXPY procedure

- SAXPY (Single-precision Alpha X Plus Y)
  - computes \( Y = \alpha X + Y \), where \( \alpha \) 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)

```c
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 \( Y \)
Using compiler vectorization

- 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

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

- 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

```
void saxpy(int n, float alpha, float *X, float *Y) {
    __m128 *x_ptr, *y_ptr; /* Pointers to vectors */
    int i;
    x_ptr = (__m128 *) X;  /* Set pointers to the data */
    y_ptr = (__m128 *) Y;
    for (i=0; i<n/4; i++) {
        y_ptr[i] = alpha*x_ptr[i] + y_ptr[i]; /* Do the computation */
    }
}
```

Explicit arithmetic operations on vector data types

- Declare variables of vector data types and express the computations with normal arithmetic expressions
  - the arithmetic operations (+, -, *, etc.) are overloaded with the corresponding vector operations

```
void saxpy(int n, float alpha, float *X, float *Y) {
    __m128 *x_ptr, *y_ptr; /* Pointers to vectors */
    int i;
    x_ptr = (__m128 *) X; /* Set pointers to the data */
    y_ptr = (__m128 *) Y;
    for (i=0; i<n/4; i++) {
        y_ptr[i] = alpha*x_ptr[i] + y_ptr[i]; /* Do the computation */
    }
}
```

- Note that the number of operations in the loop now is \( \frac{n}{4} \)
  - \( x_{ptr[i]} \) and \( y_{ptr[i]} \) are vectors of 4 floating-point values
  - the compiler recognizes that \( \alpha \) is a scalar value
Vector data types

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

<table>
<thead>
<tr>
<th>Instruction set</th>
<th>Header file</th>
<th>Registers</th>
<th>Length (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMX</td>
<td>mmintrin.h</td>
<td>MMX</td>
<td>64</td>
</tr>
<tr>
<td>SSE</td>
<td>xmmintrin.h</td>
<td>XMM</td>
<td>128</td>
</tr>
<tr>
<td>SSE2</td>
<td>xmmintrin.h</td>
<td>XMM</td>
<td>128</td>
</tr>
<tr>
<td>SSE3</td>
<td>xmmintrin.h</td>
<td>XMM</td>
<td>128</td>
</tr>
<tr>
<td>SSE4.2</td>
<td>xmmintrin.h</td>
<td>XMM</td>
<td>128</td>
</tr>
<tr>
<td>AVX</td>
<td>nmmintrin.h</td>
<td>YMM</td>
<td>256</td>
</tr>
</tbody>
</table>

- 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

Using compiler 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 __m128, __m128d and __m128i

- 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, ...
**SAXPY with vector intrinsic functions**

```c
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 values from X */
        y_vec = _mm_load_ps(&Y[i]); /* Load 4 values 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 instructions**

- Arithmetic 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 value: 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_sqrt_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
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 $B[i]$

- Example: conditionally assign an integer value to $A[i]$ depending on some other value $B[i]$

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

$$A[i] = (C[i] && mask) \lor (D[i] && \neg mask)$$

where the logical operations AND ($\&\&$), OR ($\lor$) and NOT ($\neg$) are done bitwise

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

Example

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

```c
for (i=0; i<N; i++) {
    A[i] = (B[i] > 0) ? (C[i]+2) : (D[i]+10);
}
```

```c
__m128i zero_vec = _mm_set1_epi32(0); // 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_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(d_vec, mask); // 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:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>z</td>
<td>v</td>
</tr>
<tr>
<td>x</td>
<td>y</td>
<td>z</td>
<td>v</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Structure of arrays:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
|x:| 0 | 1 | 2 | 3 | 4 | 5 | 6 |   ...
|y:| 0 | 1 | 2 | 3 | 4 | 5 | 6 |   ...
|z:| 0 | 1 | 2 | 3 | 4 | 5 | 6 |   ...
|v:| 0 | 1 | 2 | 3 | 4 | 5 | 6 |   ...

Hybrid structure:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>z</td>
<td>z</td>
<td>z</td>
<td>z</td>
</tr>
<tr>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Shuffling data in vectors

- 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(b3, b2, a1, a0) creates a bit mask describing the shuffle operation
  - interleaves values from a and b in the order specified

x: 3 2 1 0
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>c</td>
<td>b</td>
<td>a</td>
</tr>
</tbody>
</table>

y: 3 2 1 0
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>g</td>
<td>f</td>
<td>e</td>
</tr>
</tbody>
</table>

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

result: 3 2 1 0
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>e</td>
<td>d</td>
<td>c</td>
</tr>
</tbody>
</table>

- 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

CPU dispatching

- A disadvantage of using extended instruction sets is that the code will not be compatible with older version of the instruction set
  - the code will not run on older processors
- One solution is to make performance-critical code available in multiple versions
  - for instance one version with AVX vectorization and one version with normal scalar computations
- A CPU dispatching mechanism detects during runtime which capabilities the processor has, and calls the appropriate version of the code
  - the computer manufacturers have their own solutions for CPU dispatching
  - uses the CPUID instruction, which returns a set of values describing the capabilities of the processor
- CPU dispatching mechanisms are not in general portable between different compilers or between different computer manufacturers
  - for instance, code compiled on an Intel architecture may perform less efficiently on an AMD processor