# Μεταγλωττιστές για Ενσωματωμένα Συστήματα

## Χειμερινό Εξάμηνο 2023-24
## «SIMD Vectorization»

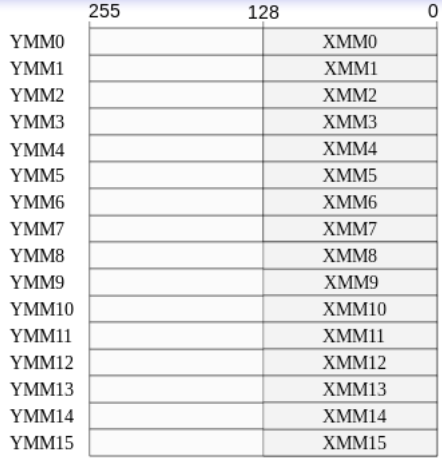
Παναγιώτης Χατζηδούκας

# SIMD (vector instructions)

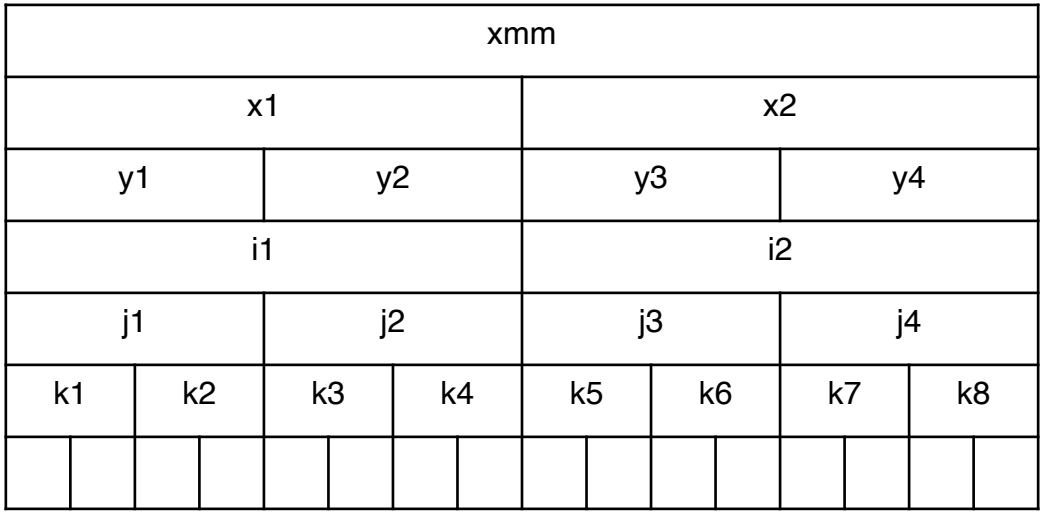
- Recall SIMD: Single Instruction Multiple Data
  - we perform the same operation on many values at once.
- This was pioneered by the vector supercomputers
  - Cray X/MP
- Since 1999 part of all Intel CPUs
  - SSE (Streaming SIMD Extensions)
  - AVX (Advanced Vector Extensions)
- The easiest way of getting parallel speedup

# SIMD registers and operations

- SIMD units contain vector registers
  - 128-bit registers XMM0 - XMM15 for SSE
  - 256-bit registers YMM0-YMM15 for AVX, overlapping the XMM registers



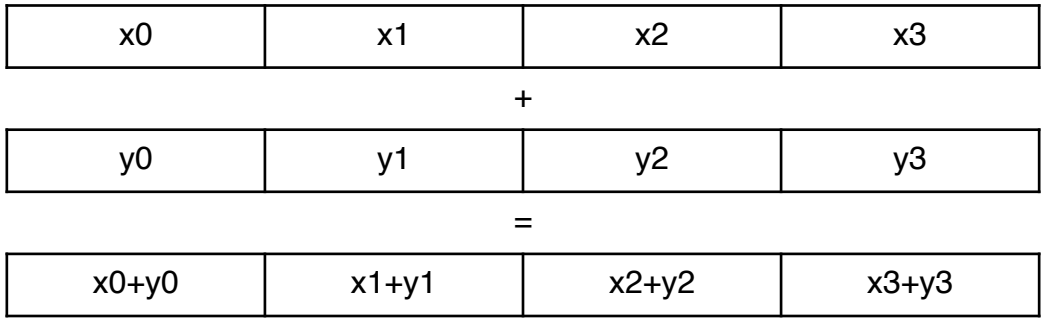

- The SSE XMM registers can store
  - XMM register
  - 2 doubles
  - 4 floats
  - 2 64-bit integer
  - 4 32-bit integers
  - 8 16-bit integer
  - 16 bytes




- AVX register can store 8 float or 4 double, integers since AVX2
- AVX-512 doubles that again

# SIMD vector operations

- SIMD vector operations act on all values in the vector at once
- Example: adding four floats with one “packed floating point” instruction




- Advantages:
  - One instruction instead of 4
  - Memory access can be optimized
  - An easy way to gain speed for almost any code

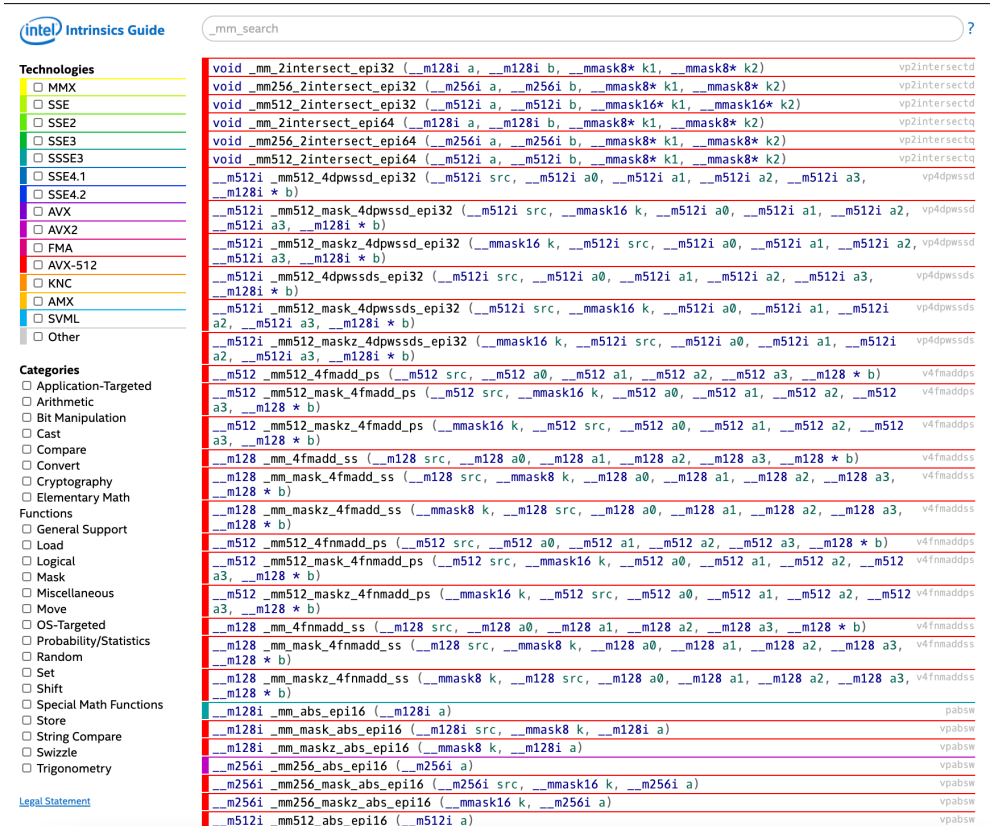
# SSE/AVX versions

- Intel and AMD have introduced more and more SIMD instructions with every new processor generation. The history is complex, and only roughly summarized below

| Generation | Year | First Intel CPU | Main features |
|---|---|---|---|
| SSE | 1999 | Pentium III | |
| SSE2 | 2001 | Pentium 4 | SSE registers can be used together with scalar floating-point registers |
| SSE3 | 2004 | Pentium 4 - Prescott | more instructions, and conversions between floating-point and integer |
| SSE4 | 2006 | Core 2 | More instructions |
| AVX | 2011 | Sandy Bridge | floating point 256 bit registers |
| AVX2 | 2013 | Haswell | integer 256 bit registers |
| AVX-512 | 2017 | Skylake | 512 bit registers |

# SSE/AVX documentation

- The best documentation tool is by Intel, at
  - https://software.intel.com/sites/landingpage/IntrinsicsGuide/
- An excellent documentation tool
  - lists all functions with clear explanation and documentation
  - allows to search for functions



# Review: Caches

- Memory access speed did not keep up with Moore's law
- Are added to speed up memory access
  - Many GByte of slow but cheap DRAM
  - 2-20 MByte of fast L3-Cache
  - 256-512 kByte of faster L2-Cache per core
  - 2x32 - 2x64 kByte of fastest L1-Cache per core (instruction and data cache)
- Data that is read is stored in the caches and kept there until it needs to be evicted because new data is loaded
- Data written to memory is written to the cache and only further to memory if it needs to be evicted (or if we need to synchronize memory access between cores)
- Problems reusing memory will run faster!

# Comparison of memory/cache speeds

- Data for Intel Sandy Bridge CPU

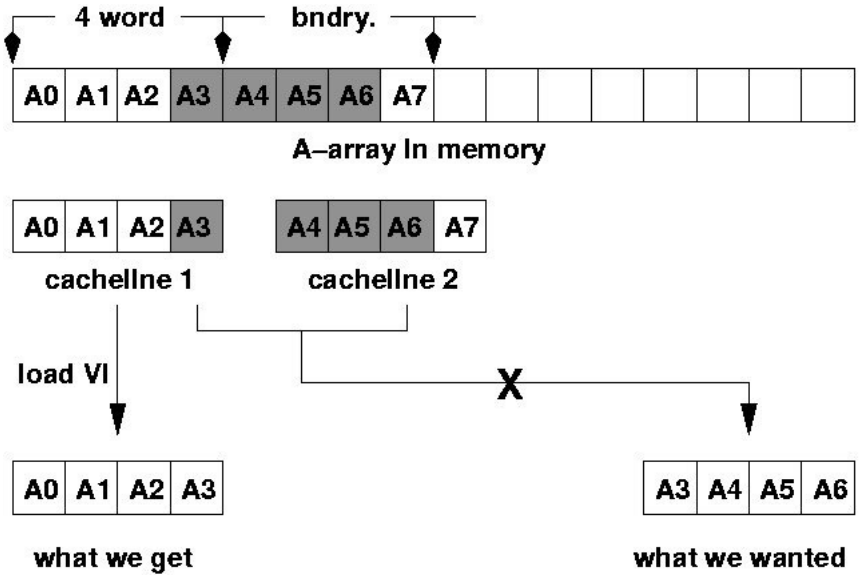
| | Size | Access Time in cycles |
|---|---|---|
| L1 cache | 2x32 KB | 4-5 |
| L2 cache | 256 KB | 12-19 |
| L3 cache | 3-20 MB | 30-50 |
| Memory | Many GB | ~ 300 |

# How does a cache work?

- CPU requests a word (e.g. 4 bytes) from memory
  - A full “cache line” (nowadays typically: 64 bytes) is read from memory and stored in the cache
  - The first word is sent to the CPU
- CPU requests another word from memory
  - Cache checks whether it has already read that part as part of the previous cache line
  - If yes, the word is sent quickly from cache to CPU
  - If not, a new cache line is read
- Once the cache is full, the oldest data is overwritten
- Locality of memory references are important for speed

# Data alignment

- To achieve optimal speed data should be aligned on cache line boundaries.
- Consider what happens if we load one value that is not at the start of a cache line (on an old machine with 16 byte cache lines):



# Alignment

- SSE registers are 16 bytes and need 16-byte alignment
- AVX registers are 32 bytes and need 32-byte alignment
- It is even better to align on cache line boundaries: 64/128 bytes on modern Intel CPUs

# Allocating aligned data

- Aligned memory can be allocated
  - on POSIX (Linux, Unix) systems by calling `posix_memalign`
  - On Windows systems by calling by calling `_aligned_malloc`
- Easiest using an alignment specifier in the declaration

```
float __attribute__((aligned(32))) sse[8];
```

# Memory layout

- Array of Structures (AoS) vs Structure of Arrays (SoA)

```
#define N (128*1024*1024)

typedef struct {
  double x;
  double y;
} point_t ;

point_t points[N];

for (int i = 0; i < N ; ++ i) {
  points[i].x = drand48();
  points[i].y = drand48();
}
```

- The points are stored in a array of structures (AoS)
- Typical object-based approach
- Data access is performed through objects

```
#define N (128*1024*1024)

struct {
  double x[N];
  double y[N];
} points;

for (int i = 0; i < N ; ++ i) {
  points.x[i] = drand48();
  points.y[i] = drand48();
}
```

- The points are stored a structure of arrays (SoA)
- No object abstraction but focus on the data of the points
- Access to  the individual information of the points

# AoS vs SoA

```
#pragma omp parallel for reduction(+:result)
for (int i = 0; i < N; ++i) {
  result += (points[i].x); // AoS
}
```

```
#pragma omp parallel for reduction(+:result)
for (int i = 0; i < N ; ++i) {
  result += (points.x[i]); // SoA
}
```

- Ease of programming and performance
- AoS can cause unecessary memory traffic
  - Pollutes the cache with unused data
- AoS not suitable for SIMD vectorization
  - Data transformation from AoS to SoA must be performed

# When can a loop be vectorized?

- A loop can only be vectorized (or parallelized by threads) if there are no dependencies between the iterations:
  - A linear congruential generator cannot be vectorized since one iteration depends on the previous one. We must wait for it to finish.

```
for (int i=1 ; i<N; ++i)
  rnd[i] = a* rnd[i-1] + c;
```

  - adding vectors by saxpy can be vectorized (no dependencies)

```
for (int i=0 ; i<N; ++i)
  x[i] = a*x[i] + y[i];
```

  - a lagged Fibonacci generator can be vectorized for vector lengths up to min(p,q). Dependencies only beyond a distance min(p,q)

```
for (int i=std::max(p,q) ; i<N; ++i)
  rnd[i] = rnd[i-p] + rnd[i-q];
```

- Vector supercomputers had vector lengths up to 1024 elements.
- SSE has at most 16 bytes and AVX at most 32 bytes
  - the lagged Fibonacci is easier to vectorize

# Detecting dependencies

- Look at every variable in the loop and check whether it might be written or read by another loop iteration. If so there is a dependency.
- Some dependencies can be removed by introducing additional variables

```
for (int i=0; i<N-1; i++) {
  x = (b[i] + c[i])/2;
  a[i] = a[i+1] + x;
}
```

```
for (int i=0; i<N-1; i++)
  a2[i] = a[i+1];

for (int i=0; i<N-1; i++) {
  x = (b[i] + c[i])/2;
  a[i] = a2[i] + x;
}
```

- now both loops can be safely vectorized or parallelized
- Another special case are reductions:

```
double s=0;
for (int i=0; i<N; i++)
  s += x[i]*y[i];
```

- reductions can be vectorized, but it needs special care
- in OpenMP parallelization there is the reduction clause

# Using SIMD instructions

- SIMD instructions can be used through assembly language. Complicated!
- Compilers offer support through intrinsics. Special types and functions that will be mapped directly to registers and SIMD instructions.
- Include the appropriate header or use the header <x86intrin.h> that is available with some compilers to load all headers available depending on the target platform

| | |
|---|---|
| MMX | <mmintrin.h> |
| SSE | <xmmintrin.h> |
| SSE2 | <emmintrin.h> |
| SSE3 | <pmmintrin.h> |
| SSSE3 | <tmmintrin.h> |
| SSE4.1 | <smmintrin.h> |
| SSE4.2 | <nmmintrin.h> |
| SSE4A | <ammintrin.h> |
| AES | <wmmintrin.h> |
| AVX | <immintrin.h> |

- Enable code generation for SSE or AVX with the right compiler switches
- With gcc one option is to use the -msse3, -msse4 or -maxv to enable SSE3, SSE4 or AVX support

# Intrinsics: register data types

- The intrinsics headers define a few datatypes that map directly to SSE or AVX registers. The compiler will place such variables in the registers.
- Note: these start with two underscores!

| | |
|---|---|
| __m128 | 4 floats |
| __m128d | 2 doubles |
| __m128i | Integers of any size |

| | |
|---|---|
| __m256 | 8 floats |
| __m256d | 4 doubles |
| __m256i | integers of any size, AVX2 |

| | |
|---|---|
| __m512 | 16 floats |
| __m512d | 8 doubles |
| __m512i | integers of any size, AVX-512 |

# Intrinsics: naming of operations

- SSE and AVX instructions have a certain naming scheme
  - SSE operations: _mm_name_type
  - AVX operations: _mm256_name_type
  - operations on types shorter than a full register will not modify the higher bits

| type | length in bits | description |
|---|---|---|
| **ss** | 32 | a single float |
| **ps** | 128,256 or 512 | 4, 8, or 16 floats |
| **sd** | 64 | a single double |
| **pd** | 128,256 or 512 | 2, 4, or 8 doubles |
| si64 | 64 | any integers |
| si128 | 128 | any integers |
| si256 | 256 | any integers |
| pi8 | 64 | 8 8-bit integers |
| pi16 | 64 | 4 16-bit integers |
| pi32 | 64 | 2 32-bit integers |
| epi8 | 128,256 or 512 | 16, 32 or 64 8-bit integers |
| epi16 | 128,256 or 512 | 8, 16 or 32 16-bit integers |
| epi32 | 128,256 or 512 | 4, 8 or 16 32-bit integers |
| epi64 | 128,256 or 512 | 2, 4 or 8 64-bit integers |

# A first example: sscal

- Multiply a vector by a scalar, assuming aligned data and a vector length that is a multiple of 4

```
void sscal(int n, float a, float* x)
{
  // load the scale factor four times into a register
  __m128 x0 = _mm_set1_ps(a);
  // loop over chunks of 4 values
  for (int i=0; i<n/4; ++i) {
    __m128 x1 = _mm_load_ps(x+4*i); // aligned (fast) load
    __m128 x2 = _mm_mul_ps(x0,x1);  // multiply
    _mm_store_ps(x+4*i,x2);         // store back aligned
  }
}
```

- We are using four instructions: two loads, a multiplication and a store

# A first example: sscal

- Multiply a vector by a scalar, assuming aligned data, but now arbitrary vector length. We need to do the remaining values by hand

```
void sscal(int n, float a, float* x)
{
  // load the scale factor four times into a register
  __m128 x0 = _mm_set1_ps(a);

  int ndiv4 = n/4;
  // loop over chunks of 4 values
  for (int i=0; i<ndiv4; ++i) {
    __m128 x1 = _mm_load_ps(x+4*i); // aligned (fast) load
    __m128 x2 = _mm_mul_ps(x0,x1);  // multiply
    _mm_store_ps(x+4*i,x2);         // store back aligned
  }

  // do the remaining entries
  for (int i=ndiv4*4 ; i< n ; ++i)
    x[i] *= a;
}
```

# A first example: sscal

- Multiply a vector by a scalar, assuming aligned data, but now arbitrary vector length. We need to do the remaining values by hand

```
void sscal(int n, float a, float* x)
{
  // load the scale factor four times into a register
  __m128 x0 = _mm_set1_ps(a);

  int ndiv4 = n/4;
  // loop over chunks of 4 values
  for (int i=0; i<ndiv4; ++i) {
    __m128 x1 = _mm_load_ps(x+4*i); // aligned (fast) load
    __m128 x2 = _mm_mul_ps(x0,x1);  // multiply
    _mm_store_ps(x+4*i,x2);         // store back aligned
  }

  // do the remaining entries
  int i = ndiv4*4;
  switch (n-i) {
    case 3: x[i+2] *= a;
    case 2: x[i+1] *= a;
    case 1: x[i] *= a;
  }
}
```

# Load /store

- An incomplete summary of load/store instruction

| Instruction | Types | Explanation |
|---|---|---|
| set1 | all | sets all elements to a given value |
| set | all | set each element to a different value |
| setr | all | set in reverse order |
| setzero | pd, ps, si64,si128,si256 | set to zero |
| load1 | pd,ps | load a single value into each element of the register |
| broadcast | pd,ps | same as load1 but much faster (AVX only) |
| load | pd, ps, ss, sd, si128,si256 | load values from memory into a register |
| loadr | pd,ps | load values in reverse order |
| loadu | pd, ps, ss, sd, si128,si256 | load unaligned values from memory (slow!) |
| streamload | si128 | load integer values bypassing the cache |
| store | pd, ps, ss, sd, si128,si256 | store values from register into memory |
| storeu | pd, ps, ss, sd, si128,si256 | store values from register into unaligned memory (slow!) |
| stream | pd, ps, pi,si128,si256 | store values into memory bypassing the cache |

- The streaming loads and stores bypass the cache. This reduces cache eviction, but it is hard to see a difference in many codes

# Prefetch

- Prefetch instruction can be used to hint that some data will be used later and should already be fetched into the cache since they will soon be used

```
void _mm_prefetch (char const *p, int hint)
```

| Hint | meaning |
|---|---|
| _MM_HINT_T0 | prefetch into L1 (and L2 and L3) cache. Use for integer data. |
| _MM_HINT_T1 | prefetch into L2 (an L3) cache. Use for floating point data. |
| _MM_HINT_T2 | prefetch into L3 cache. Use if the cache line is not reused much. |
| _MM_HINT_NTA | prefetch into L2 but not L3 cache. Use if the data is needed only once. |

- Example:

```
// loop over chunks of 4 values
for (int i=0; i<ndiv4; ++i) {
  _mm_prefetch((char*) y+4*i+8,_MM_HINT_NTA ); // prefetch data for two iterations later
  __m128 x1 = _mm_load_ps(x+4*i);              // aligned (fast) load
  __m128 x2 = _mm_mul_ps(x0,x1);               // multiply
  _mm_store_ps(x+4*i,x2);                      // store back aligned
}
```

# Arithmetic floating-point instructions

- An incomplete summary of arithmetic instructions

| Instruction | Explanation |
|---|---|
| add, sub | +, - |
| addsub | - on even, + on odd elements |
| mul, div | *, / |
| ceil | ceil, round up |
| floor | floor, round down |
| round | round, allows specification of rounding policy |
| min | min |
| max | max |
| rcp | reciprocal (inverse) |
| sqrt | sqrt |
| rsqrt | reciprocal (inverse) square root |
| and andnot | bitwise &, &! |
| or xor | bitwise \|, ^ |

# Arithmetic integer instructions

- An incomplete summary of arithmetic instructions

| Instruction | Explanation |
|---|---|
| add, adds | +, adds is saturated add: assigns maximum/minimum if overflow or underflow |
| sub, subs | -, subs is saturated sub: assigns maximum/minimum if overflow or underflow |
| avg | rounded average of x and y: (x+y+1)/2 |
| mul | *, multiplies low words into result of twice the size - ignores every second input value |
| mullo | *, low word of product (result has twice the number of bits) |
| mulhi | *, high word of product (result has twice the number of bits) |
| sign | transfers sign of one integer to another and sets it to zero if “sign” is 0 |
| min, max | min, max |
| and andnot | &, &! |
| or xor | \|, ^ |
| sll, slli | <<, the version ending in i needs an integer constant shift |
| srl,slri | >> for unsigned integers, shifting in 0 bits |

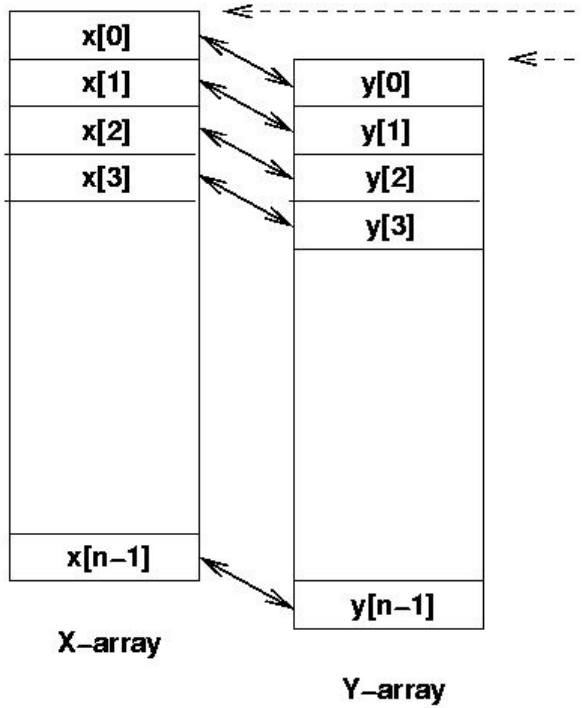
# Comparisons

- An incomplete summary of important comparison instructions

| Instruction | Types | Explanation |
|---|---|---|
| cmpeq, cmpneq | all | x==y , x!=y |
| cmpgt, cmpge | all | x>y, x>=x |
| cmplt, cmple | all | x<y, x<=y |
| cmpngt, cmpnge | floating point | !(x>y), !(x>=x) |
| cmpnlt, cmpnle | floating point | !(x<y), !(x<=y) |
| cmpord, cmpunord | floating point | tests whether the number are ordererd or unordered (e.g. if NaN) |
| test_all_ones | i128 | test if all bits are 1 |
| test_all_zeros | i128 | test if all bits are 0 |
| test_mix_ones_zeros | i128 | test if either all are 0 or all are 1 |

# _axpy operations

- Alignment is trickier with operations involving two vectors
  - Example _axpy:
    $\vec{y} = a\,\vec{x} + \vec{y}$
  - We need both arrays aligned in the same way
  - Two solutions:
    - either always require alignment
    - or code a slow version to use if not aligned



# saxpy

- a vectorized saxpy implementation assuming alignment

```
void saxpy(int n, float a, float* x, float* y)
{
  // load the scale factor four times into a register
  __m128 x0 = _mm_set1_ps(a);

  int ndiv4 = n/4;
  // loop over chunks of 4 values
  for (int i=0; i<ndiv4; ++i) {
    __m128 x1 = _mm_load_ps(x+4*i); // aligned (fast) load
    __m128 x2 = _mm_load_ps(y+4*i); // aligned (fast) load
    __m128 x3 = _mm_mul_ps(x0,x1); // multiply
    __m128 x4 = _mm_add_ps(x2,x3); // add
    _mm_store_ps(y+4*i,x4); // store back aligned
  }

  // do the remaining entries
  for (int i=ndiv4*4; i< n; ++i)
    y[i] += a*x[i];
}
```

# sdot

- a vectorized dot product assuming alignment
- we have to manually do the reduction

```
float sdot(int n, float* x, float* y)
{
  // set the total sum to 0, one sum per vector element
  __m128 x0 = _mm_set1_ps(0.);

  // loop over chunks of 4 values
  int ndiv4 = n/4;
  for (int i=0; i<ndiv4; ++i) {
    __m128 x1 = _mm_load_ps(x+4*i); // aligned (fast) load
    __m128 x2 = _mm_load_ps(y+4*i); // aligned (fast) load
    __m128 x3 = _mm_mul_ps(x1,x2);  // multiply
    x0 = _mm_add_ps(x0,x3);         // add
  }

  // store the 4 partial sums back to aligned memory
  float __attribute__((aligned(32))) tmp[4];
  _mm_store_ps(tmp,x0);

  // do the reduction over the vector elements by hand
  float sum = tmp[0]+tmp[1]+tmp[2]+tmp[3];

  // do the remaining entries
  for (int i=ndiv4*4 ; i< n ; ++i)
    sum += x[i]*y[i];

  return sum;
}
```

https://godbolt.org/z/nbaj7zse3

# Matrix-vector multiplication

- Given that x, y are appropriately aligned and n % 8 == 0

```
#include <string.h>
#include <immintrin.h>

void mxv(int m, int n, float * restrict A, float * restrict x,
         float * restrict y)
{
  __m256 AA, xx, yy;
  int simd_width = 8;

  memset(y, 0, m * sizeof(float));
  for (int i=0; i<m; ++i) {
    yy = _mm256_set_ps(0, 0, 0, 0, 0, 0, 0, 0);

    for (int j=0; j<n; j+=simd_width) {
      AA = _mm256_load_ps(A+i*n+j);
      xx = _mm256_load_ps(x+j);
      yy = _mm256_fmadd_ps(AA, xx, yy); // yy = AA*xx + yy
    }

    y[i] = yy[0]+yy[1]+yy[2]+yy[3]+yy[4]+yy[5]+yy[6]+yy[7];
  }
}
```

https://godbolt.org/z/vW5K4seT1

# Automatic vectorization with gcc

- Modern compilers try to automatically vectorize loops. This can save you time but will sometimes not be as good as vectorization by hand.
- Compiler options for gcc/g++
  - Turn vectorization on: -ftree-vectorize
  - Generate vectorization reports: -Om -ftree-vectorizer-verbose=n

| n | Description |
|---|---|
| 0 | No output at all. |
| 1 | Report vectorized loops. |
| 2 | Also report unvectorized "well-formed" loops and respective reason. |
| 3 | Also report alignment information (for "well-formed" loops). |
| 4 | Like level 3 + report for non-well-formed inner-loops. |
| 5 | Like level 3 + report for all loops. |
| 6 | Print all vectorizer dump information. |

- Further reading
  - GNU documentation
    - http://gcc.gnu.org/projects/tree-ssa/vectorization.html
  - Critical analysis of what autovectorization in gcc can and cannot do:
    - http://locklessinc.com/articles/vectorize/

# Aliasing prevents optimization

- Consider the saxpy operation:

```
void saxpy(int n, float a, float* x, float* y)
{
  for (int i=0; i<n; ++i)
    y[i] += a*x[i];
}
```

- Naïvely it seems this can be vectorized since there are no dependencies: each iteration accesses different elements
- Now consider the following call:

```
float x[1000];
saxpy(999, 1., x, x+1)
```

- Problem: now y=x+1 and we have an “aliasing” problem. The loop becomes

```
for (int i=0; i<n; ++i)
  x[i+1] += a*x[i];
```

- We have potential dependencies! No optimization or vectorization is actually possible unless we prevent aliasing.

# restrict

- Fortran-77 can optimize aggressively since aliasing is forbidden
- Fortran-90 and later, C, C++, ... have pointers and with pointers aliasing becomes a potential problem and prevents many optimizations.
- Solution in C: restrict keyword to declare that pointers are not aliased.

```
void saxpy(int n, float a, float* restrict x, float* restrict y)
{
  for (int i=0; i<n; ++i)
    y[i] += a*x[i];
}
```

- The compiler now assumes no aliasing.
- Note that the compiler does not check for aliasing. The caller must be careful!

# Declaring alignment

- In our manually vectorized code we assumed the absence or presence of alignment. We can also tell this to the compiler:
- gcc/g++ have the following extension

```
__builtin_assume_aligned(variable,alignment);
```

```
void saxpy(int n, float a, float* restrict x, float* restrict y)
{
  __builtin_assume_aligned(x,32);
  __builtin_assume_aligned(y,32);
  for (int i=0; i<n; ++i)
    y[i] += a*x[i];
}
```

# Vectorization with OpenMP

- The `simd` construct (OpenMP 4.0+)
  - `#pragma omp simd`: declares that a loop will be utilizing SIMD

```
float a[8], b[8];
...
#pragma omp simd
for (int n=0; n<8; ++n) a[n] += b[n];
```

  - `#pragma omp declare simd`: indicates a function or procedure that is explicitly designed to take advantage of SIMD parallelism.
    - The compiler may create multiple versions of the same function for different CPU capabilities for SIMD processing
    - `aligned` attribute: hints the compiler that each element listed is aligned to the given number of bytes.

```
#pragma omp declare simd aligned(a,b:16)
void add_arrays(float *__restrict__ a, float *__restrict__ b)
{
  #pragma omp simd aligned(a,b:16)
  for (int n=0; n<8; ++n) a[n] += b[n];
}
```

- The compiler attempts to vectorize regardless of the OpenMP simd directives

# Vectorization with OpenMP

- collapse clause: can be added to bind the vectorization into multiple nested loops

```
#pragma omp simd collapse(2)
for (int i=0; i<4; ++i)
  for (int j=0; j<4; ++j)
    a[j*4+i] += b[i*4+j];
```

- reduction clause: can be used with SIMD just like with parallel loops

```
int sum=0;
#pragma omp simd reduction(+:sum)
for (int n=0; n<1000; ++n) sum += table[n];
```

# Vectorization with OpenMP

- The for simd construct: for and simd can be combined
  - divide the execution of a loop into multiple threads
  - execute the loop slices in parallel using SIMD

```
float sum(float* table)
{
  float result=0;
  #pragma omp parallel for simd reduction(+:result)
  for (int n=0; n<1000; ++n)
    result += table[n];

  return result;
}
```

# References


- Prof. M. Troyer, HPCSE I, ETH Zurich
- https://en.wikipedia.org/wiki/Fast_inverse_square_root
- https://www.mathworks.com/help/simulink/ref_extras/hdlreciprocal.html