HPCSE II

Vectorization with SIMD instructions

SIMD (vector instructions)

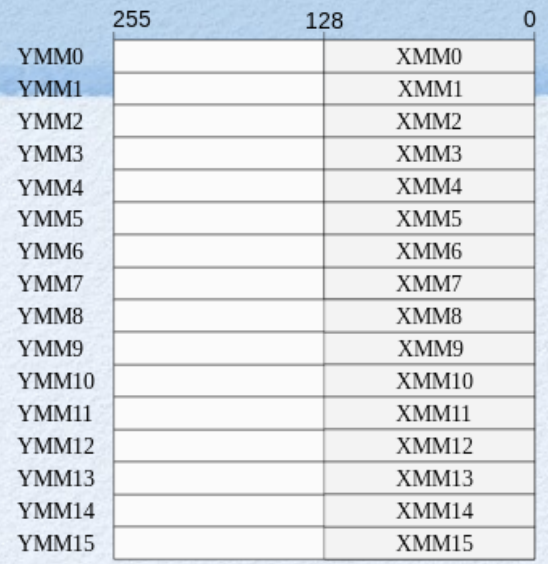
- Recall SIMD: **S**ingle **I**nstruction **M**ultiple **D**ata
- we perform the same operation on many values at once.

- This was pioneered by the vector supercomputers (e.g. Cray X/MP at ETH)

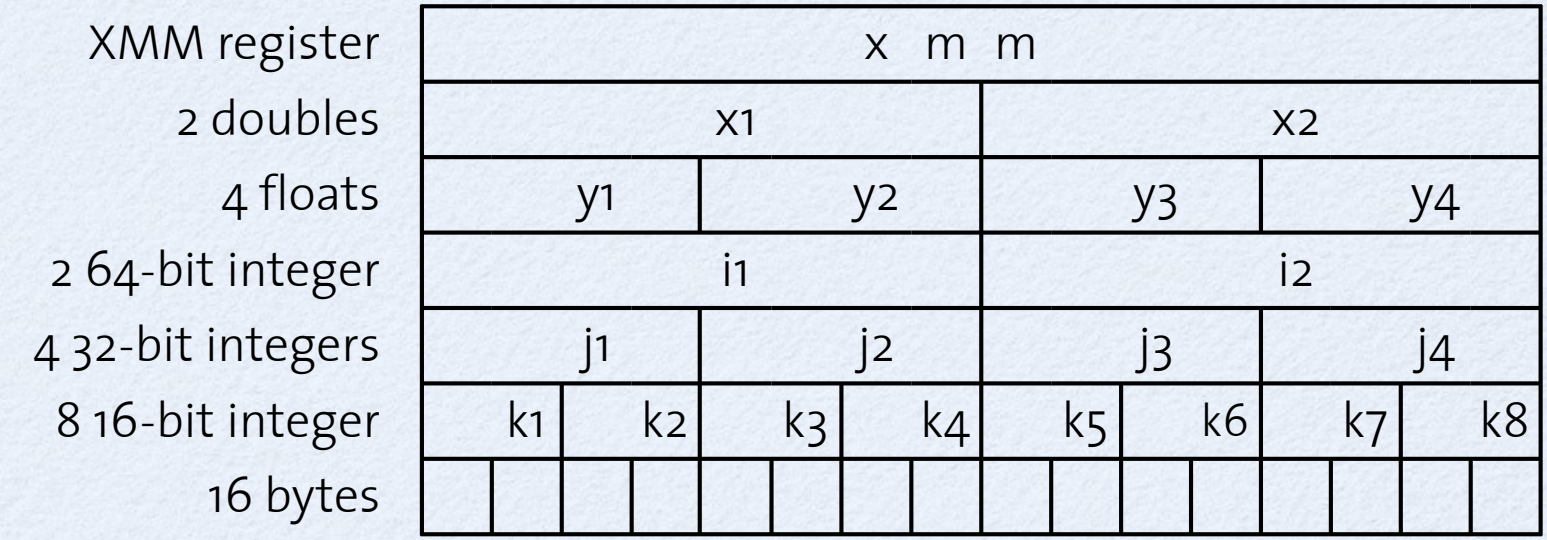
- Since 1999 part of all Intel CPUs
  - SSE (Streaming SIMD Extensions)
  - AVX (Advanced Vector Extensions)

- The easiest way of getting parallel speedup

SIMD registers and operations



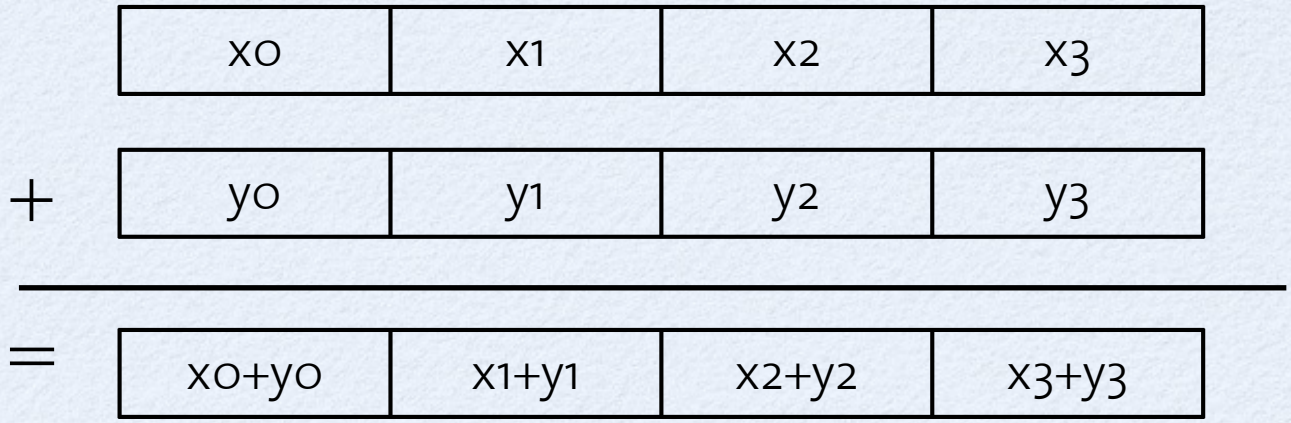
- SIMD units contain vector registers
  - 128-bit registers XMM0 - XMM15 for SSE
  - 256-bit registers YMM0-YMM15 for AVX, overlapping the XMM registers
- The SSE XMM registers can store



- AVX register can store 8 float or 4 double, integers since AVX2

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

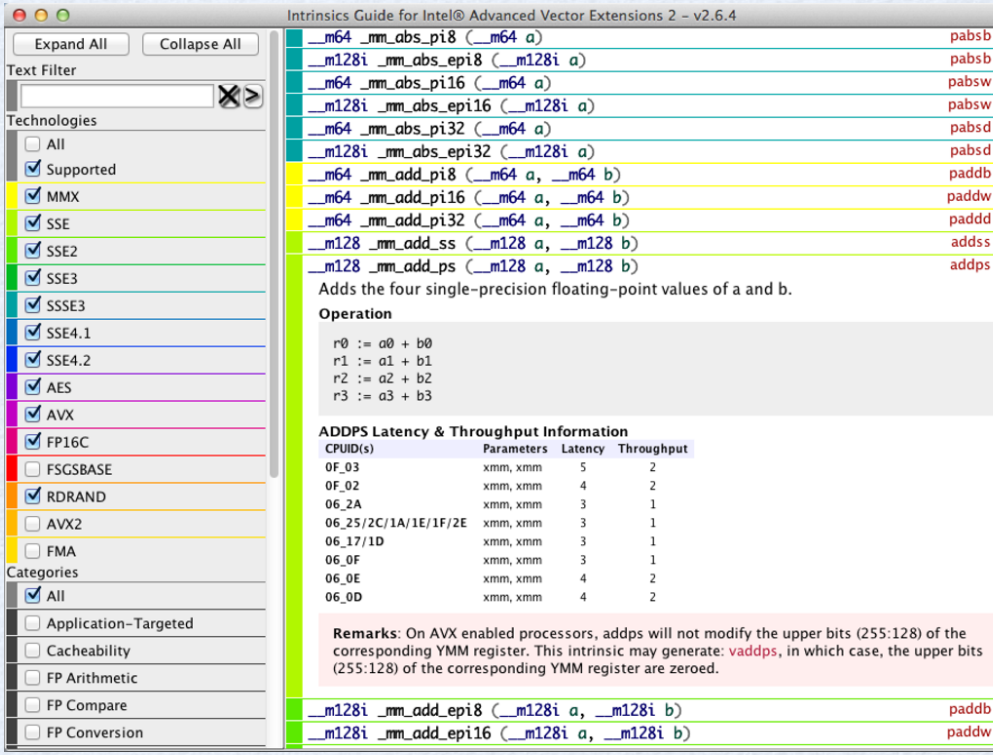
Generation	Year	First Intel CPUs	main features
SSE	1999	Pentium III	
SSE2	2001	Pentium 4	SSE registers can be used together with scalar floating point registers
SSE3	2004	Pentium 4 - Prescott	more instructions, and conversions between floating point and integer
SSE4	2006	Core 2	more instructions
AVX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

- Homework: determine which SIMD instructions your machine supports

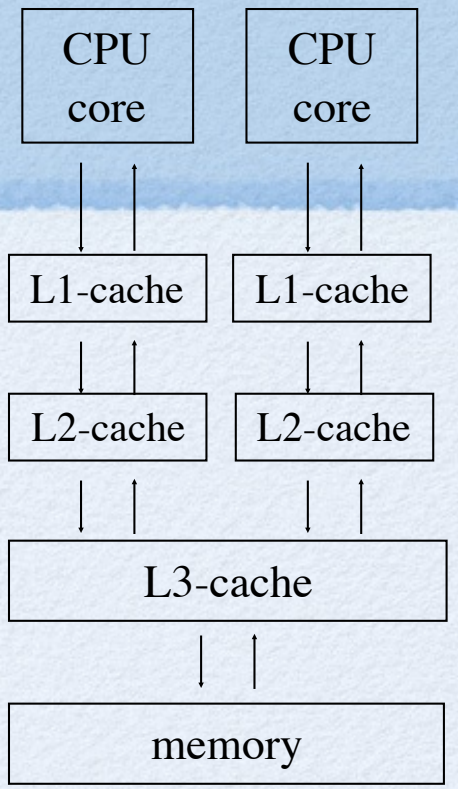
SSE/AVX documentation

- 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
  - can filter functions depending on SSE support



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

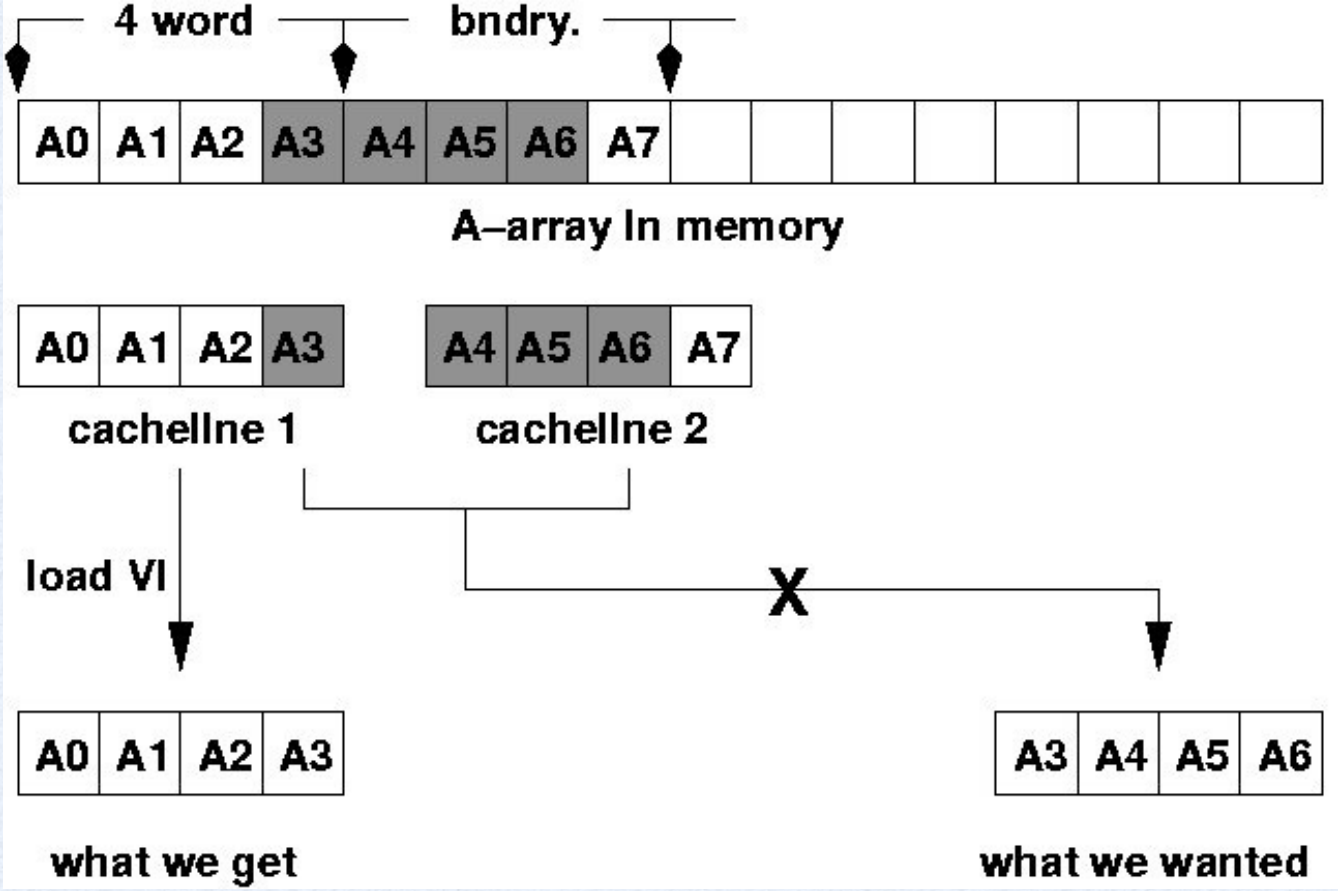
	Size	**Access time in cycles**
L1 cache	2x32 kB	4-5
L2 cache	256 kB	12-19
L3 cache	3-20 MB	30-50
Memory	many GB	≈ 300

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 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
    - C++03 with gcc, clang, icc
    - C++03 with MSVC
    - C++11

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

```
float  __declspec(align(32)) sse[8];
```

```
float  alignas(32) sse[8];
```

- In C++11 we can declare alignment for a data type:

```
// every object of these types will be aligned to 32-byte boundary
struct alignas(32) avx_double
{
  double data[4];
};

template <class T>
struct alignas(32) avx_t
{
  T data[32/sizeof(T)];
};
```

- g++ 4.7 and MSVC11 do not support alignas. We provide a workaround in alignas.hpp

Allocating aligned data with allocators

- For C++ containers we need an aligned allocator.
  - Recall the usually ignored second template parameter of standard containers:
  - Allocators are used to allocate and free the memory for a container.

```
template<class T, class Alloc = std::allocator<T> > class vector;
```

  - Potential use of allocators:
    - allocate memory for small objects in a fast pool (boost::pool_allocator)
    - allocate specially aligned memory (used here!)
  - We provide an aligned allocator in the git repository, and will discuss it now.

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 have to 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 and AVX2	<immintrin.h>

- Enable code generation for SSE or AVX with the right compiler switches
  - Homework: study your favorite's compilers manual to find the switches
  - With g++ or clang++ 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

Intrinsics: naming of operations

- SSE and AVX instructions have a certain naming scheme
  - SSE operations: _mm_*name*_*type*
  - AVX operations: _mm256_*name*_*type*

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

- operations on types shorter than a full register will not modify the higher bits
- 256 bit integer operations are available from AVX2

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;
  }
}
```

the switch statement may be faster than the for loop

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 use:

```
// 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
}
```

- You'll need to play with it and see if it helps

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, srli	>> for unsigned integers, shifting in 0 bits
sra, srai	>> for signed integer, shifting in the sign bit

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

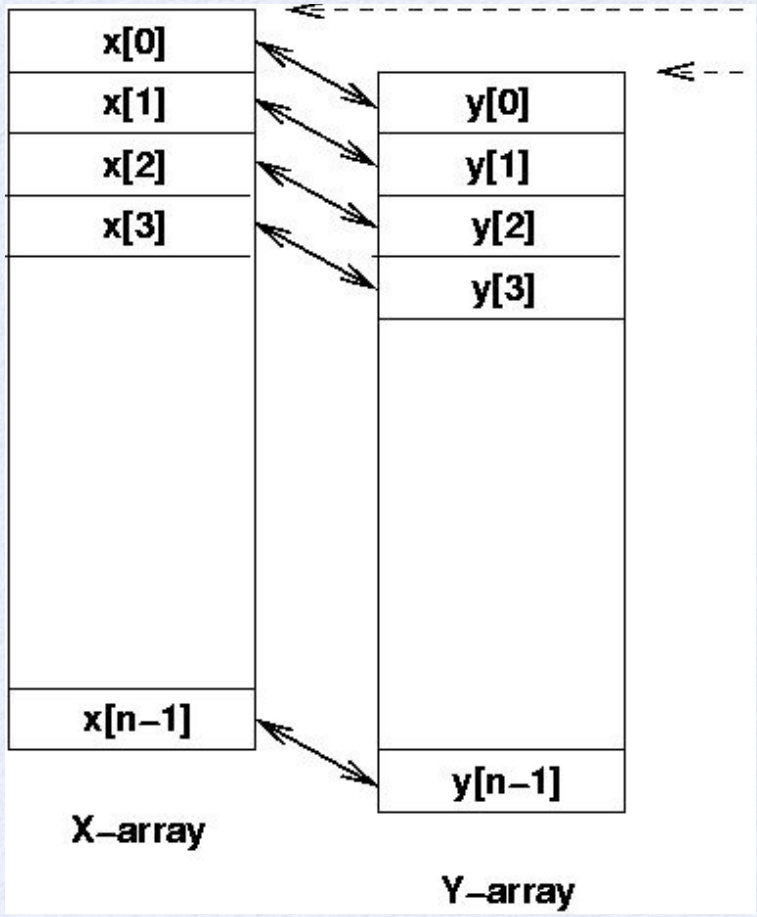
_axpy operations

- Alignment is trickier with operations involving two vectors
  - Example _axpy

$$\vec{y} = \alpha\vec{x} + \vec{y}$$

  - 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);

  // we assume alignment
  assert(((std::size_t)x) % 16 == 0 && ((std::size_t)y) % 16 == 0);

  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.);

  // we assume alignment
  assert(((std::size_t)x) % 16 == 0 && ((std::size_t)y) % 16 == 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 alignas(16) 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;
}
```

Mixing SSE and AVX

- Be careful when mixing SSE and AVX instructions:
  - Manual claims that SSE instructions do not touch the higher bits of the AVX registers
  - What actually happens is that if the higher bits are nonzero they get stored to memory if you call an SSE instruction and get reloaded when you cal an AVX instruction. SLOW!!!!
- Solution: call **_mm256_zeroupper** to clear the upper bits before switching from AVX to SSE
- Be extra careful:
  - all instructions starting with _mm256_... are AVX
  - all SSE instructions start with _mm_...
  - **some** instructions starting with _mm_ are also AVX, e.g. broadcast. You need to look at the documentation tool to check!

Warning about mixing AVX and SSE

- Original sscal:

```
void sscal(int n, float a, float* x)
{
  // load the scale factor four times into a register
  __m128 x0 = _mm_load1_ps(&a);

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

- Naively optimize sccal using broadcast:

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

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

SLOW!
we mix AVX and SSE

Warning about mixing AVX and SSE

- Original sscal:

```
void sscal(int n, float a, float* x)
{
  // load the scale factor four times into a register
  __m128 x0 = _mm_load1_ps(&a);

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

- Naively optimize sccal using broadcast:

```
void sscal(int n, float a, float* x)
{
  // load the scale factor four times into a register
  __m128 x0 = _mm_broadcast_ss(&a); // an AVX instruction!
  _mm256_zeroupper();

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

Now it's fast we clear the higher bits and they will not be stored

Automatic vectorization with g++

- 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 g++
  - Turn vectorization on: **-ftree-vectorize**
  - Generate vectorization reports: **-O*m* -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/

Automatic vectorization with iCC

- Compiler options with iCC
  - Get optimization suggestions: **-guide**
  - Turn vectorization on and generate vectorization reports:
    -qopt-report=*n* -qopt-report-phase=vec

n	description
0	No output at all.
1	Report vectorized loops.
2	Also report unvectorized loops and respective reason.
3	Adds dependency Information
4	Reports only non-vectorized loops
5	Reports only non-vectorized loops and adds dependency info

- Further reading
  - Intel documentation and sample codes:
    http://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers/

Automatic vectorization with clang

- Compiler options with clang
  - Automatic vectorization is on by default!
  - Turn vectorization off: **-fno-vectorize**
  - Turn vectorization on and set vector width: **-mllvm -force-vector-width=*n***
  - Enable vectorization reports:
    - **-Rpass=loop-vectorize** identifies loops that were successfully vectorized.
    - **-Rpass-missed=loop-vectorize** identifies loops that failed vectorization
    - **-Rpass-analysis=loop-vectorize** identifies the statements that caused vectorization to fail.
- Pragmas available to help the compiler vectorize loops
- More information on http://llvm.org/docs/Vectorizers.html

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 has to be careful!**

- No C++ standard support for restrict, but
  - g++ supports a **__restrict__** keyword
  - iCC allows the **restrict** keyword when using the compiler switch **-restrict**.

Declaring alignment

- In our manually vectorized code we assumed the absence or presence of alignment. We can also tell this to the compiler:
  - On **g++** there is a **__builtin_assume_aligned(*variable*,*alignment*);** extension

```
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];
}
```

  - iCC has a pragma to declare alignment for a loop

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

- g++ matches your code against common patterns and vectorizes the loops if there is a match. The vectorization reports can help you find out what was done and how it can be improved.

- Documented with many examples at http://gcc.gnu.org/projects/tree-ssa/vectorization.html
- However, only quite simple loops can be vectorized, as shown here http://locklessinc.com/articles/vectorize/

- Let us time an auto-vectorized loop against our manual SSE and AVX codes

	SSE manual	AVX manual	g++	clang	iCC
vectorized	—	—	35	26	26
not vectorized	27	26	55	26	55

Automatic vectorization on clang

- The clang compiler provides pragmas to further help vectorize loops
- Documented at http://llvm.org/docs/Vectorizers.html and http://clang.llvm.org/docs/LanguageExtensions.html#id20

pragma	**explanation**
clang loop vectorize(enable)	the compiler can ignore potential dependencies and vectorize
clang loop interleave(enable)	the compiler can ignore potential dependencies and interleave iterations, i.e. perform them out of order
clang loop vectorize(disable)	don't vectorize
clang loop interleave(disable)	don't interleave
clang loop vectorize_width(*n*)	vectorize up to *n* iterations
clang loop interleave_count(*n*)	interleave up to *n* iterations
clang loop unroll(full)	can be fully unrolled
clang loop unroll_count(*n*)	unroll up to *n* iterations
clang loop unroll(disable)	don't unroll

Automatic vectorization on iCC

- The Intel compiler provides
  - the **-guide** option to give hints how code can be vectorized
  - a set of pragmas to help the compiler

pragma	**explanation**
ivdep	the compiler can ignore potential dependencies
loop count (*n*) loop count min (*n*)	specify the typical or minimum loop count to help decide whether vectorization is worthwhile
vector always	always attempt to vectorize the loop
novector	don't vectorize the loop
vector aligned	tell the compiler to assume aligned data
vector nontemporal	the data will not be reused , use streaming instructions for data access
simd	always attempt SIMD vectorization
simd vectorlength(*n*)	declare absence of dependencies for *n* iterations
simd reduction (*op:variable)*	a reduction loop (similar to OpenMP)

- Documented with many examples at
  http://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers/

iCC example 1: potential aliasing

- Consider the copying loop in example1[abc].cpp

```
void copy(char *cp_a, char *cp_b, int n)
{
  for (int i = 0; i < n; i++)
    cp_a[i] = cp_b[i];
}
```

- The Intel compiler produces two versions and tests at runtime for aliasing
- You can help either using restrict

```
void copy(char * restrict cp_a, char * restrict cp_b, int n)
{
  for (int i = 0; i < n; i++)
    cp_a[i] = cp_b[i];
}
```

or using the ivdep pragma

```
void copy(char *cp_a, char *cp_b, int n)
{
#pragma ivdep
  for (int i = 0; i < n; i++)
    cp_a[i] = cp_b[i];
}
```

iCC example 2: dependencies

- Loop at the gap.cpp example:

```
void test_scalar_dep(double *A, int n)
{
  double b;
  for (int i=0; i<n; i++) {
    if (A[i] > 0) {b=A[i]; A[i] = 1 / A[i]; }
    if (A[i] > 1) {A[i] += b;}
  }
}
```

- It will not vectorize since there is a dependence on b: the next loop might use the modified value of b. Use guide to learn what can be done:
  remark #30515: (VECT) Assign a value to the variable(s) "b" at the beginning of the body of the loop in line 28. This will allow the loop to be vectorized. [VERIFY] Make sure that, in the original program, the variable(s) "b" read in any iteration of the loop has been defined earlier in the same iteration.

```
void test_scalar_dep(double *A, int n)
{
  for (int i=0; i<n; i++) {
    double b = A[i];
    if (A[i] > 0) {A[i] = 1 / A[i];}
    if (A[i] > 1) {A[i] += b;}
  }
}
```

iCC example 3: reductions

- Look at the simd3.cpp for a reduction example

```
char foo(char *A, int n){
  char x = 0;
#pragma simd reduction(+:x)
  for (int i=0; i<n; i++)
    x = x + A[i];
  return x;
}
```

- In the meantime iCC manages to detect this reduction automatically. The pragma is not needed.
- In the following more complicated reduction in simd4.cpp it is needed:

```
// saturate integers to maximum or minimum of short in reduction

short sat2short(unsigned char *p, char *q, int n) {
  short x = 0;
#pragma simd reduction(+:x)
  for (int i=0; i<n; i++)
    x = std::max(std::min(x + p[i]*q[i],32767),-32768);
  return x;
}
```

iCC example 4: dependencies

- Some dependencies still allow vectorization, as in simd4.cpp:

```
for (int i=0; i<32767; i++) {
    if (i >= 16 && i < 32767) {
      b[i] = b[i-16] - 1;
    }
```

- There are dependencies after 16 iterations. We can still vectorize up to 16 elements, but need to tell the compiler:

```
#pragma simd vectorlength(16)
  for (int i=0; i<32767; i++) {
    if (i >= 16 && i < 32767) {
      b[i] = b[i-16] - 1;
    }
```

- Homework: vectorize if possible:
  - a linear congruential random number generator?
  - a lagged Fibonacci random number generator?

iCC example 5: loop private variables

- Some dependencies can be removed by making the variable private to each iteration, as in simd5.cpp

```
void foo(int *A, int *B, int *restrict C, int n)
{
  int t = 0;

#pragma simd private(t)
  for (int i=0; i<n; i++){
    if (A[i] > 0) {
      t = A[i];
    }
    if (B[i] < 0) {
      t = B[i];
    }
    C[i] = t;
  }
}
```

i_amax

- How would you implement isamax and similar functions that should give the **index** of the largest element?
- Interested students should look at the isamax_sse.cpp example to see some really neat tricks of what you can do with vector instructions.