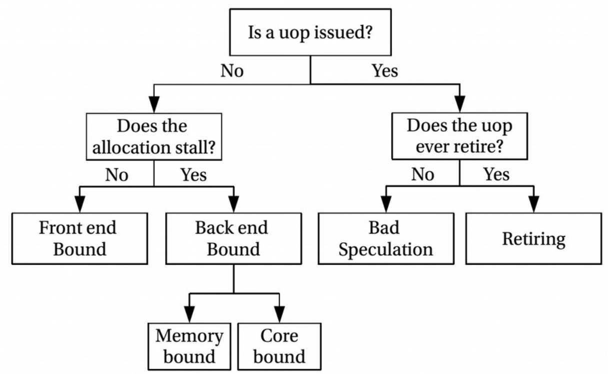
Advanced Optimization Techniques

ICTP Trieste 2014

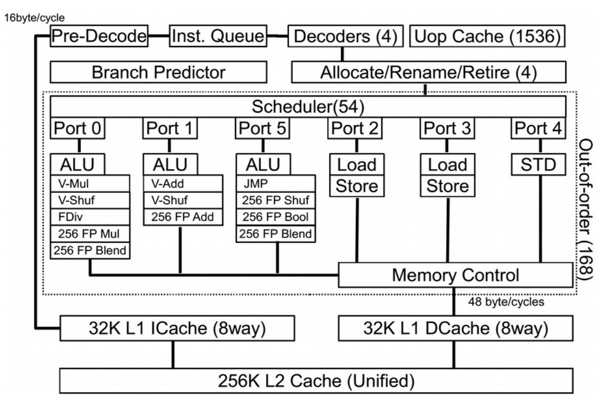
Dr. Christopher Dahnken
Intel GmbH

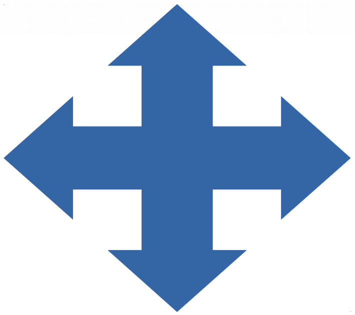
Outline

Method



CPU





Code

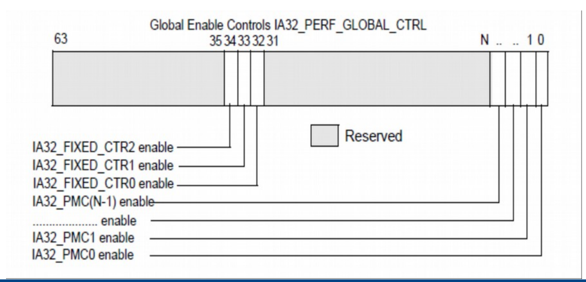
```
!$OMP SECTION
!   tsend=dclock()
 if(iblock.lt.(nblocks)) then
    nxti=m_of_i(iblock+1)
    nxtj=n_of_i(iblock+1)
    nxtk=k_of_i(iblock+1)

    nxt_buffsize_m=buffersize(ms,bm,nxti)
    nxt_index_m=(nxti-1)*bm+1

    nxt_buffsize_n=buffersize(ns,bn,nxtj)
    nxt_index_n=(nxtj-1)*bn+1

    nxt_buffsize_k=buffersize(ks,bk,nxtk)
    nxt_index_k=(nxtk-1)*bk+1
```

Measurement

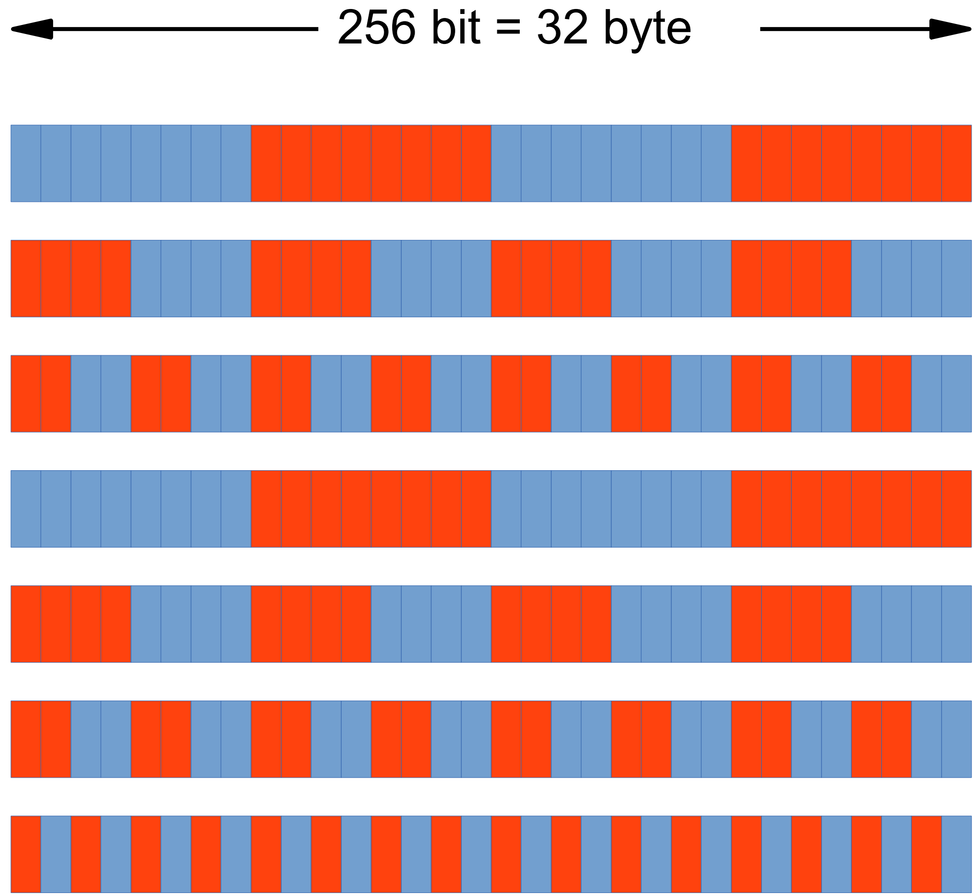


Vectorization: AVX Programming

Overview

- In this module we will have a closer look at the AVX(2) instruction set
- You will learn to directly access the instructions from your C/C++ source code
- You will learn how to load, process and store data in an explicitly vectorized fashion

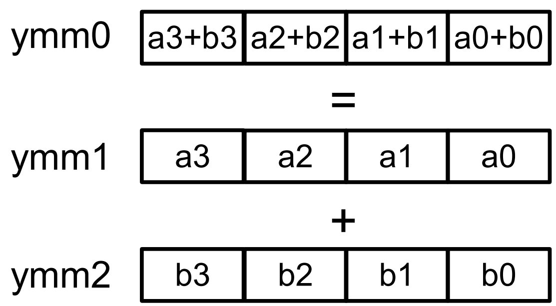
AVX Packed Data Types



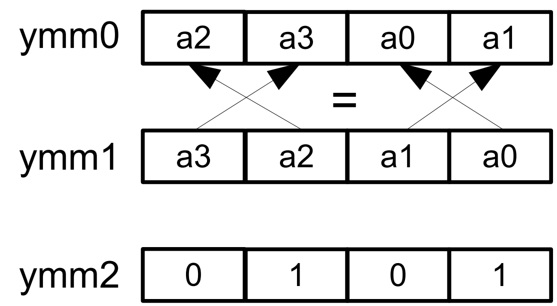
4x	double
8x	float
16x	float16
4x	long
8x	int
16x	short
32x	char

AVX - Examples

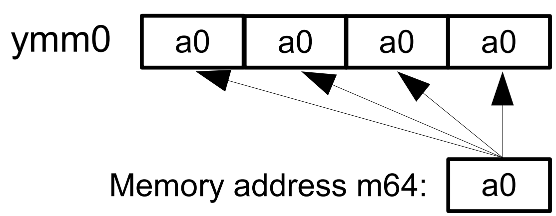
vaddpd ymm0,ymm1,ymm2



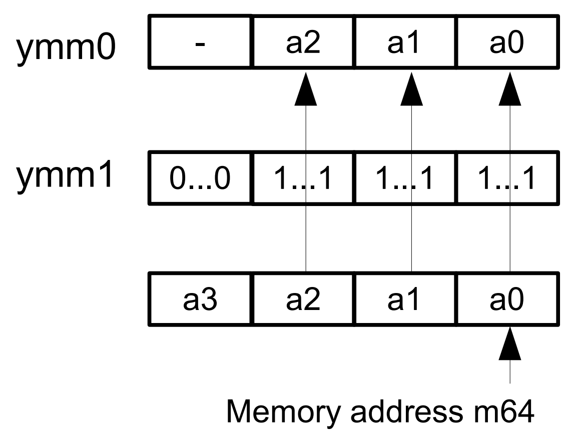
vpermilpd ymm0,ymm1,ymm2



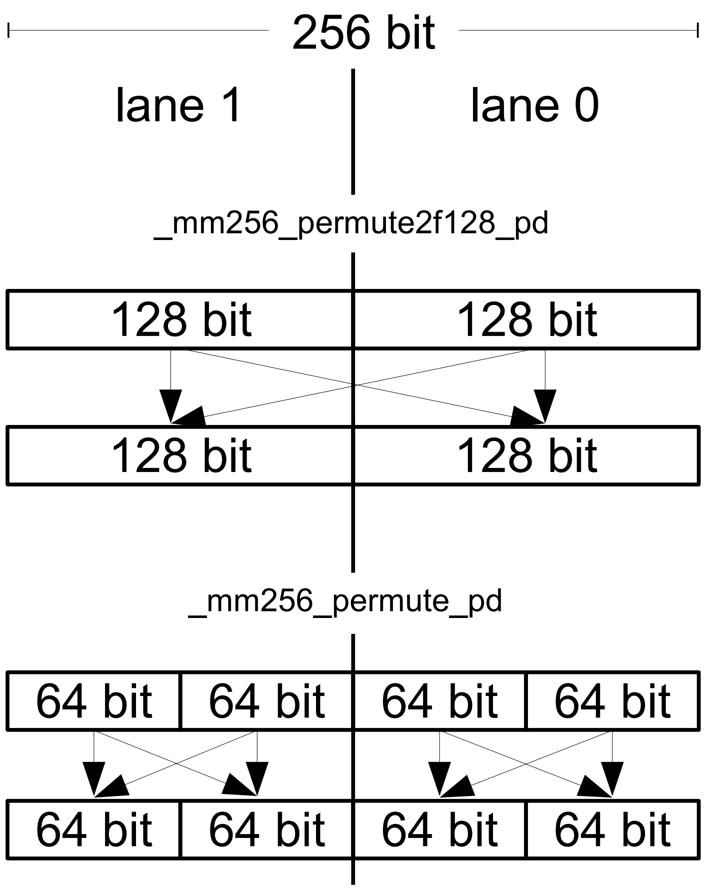
vbroadcastsd ymm0,m64



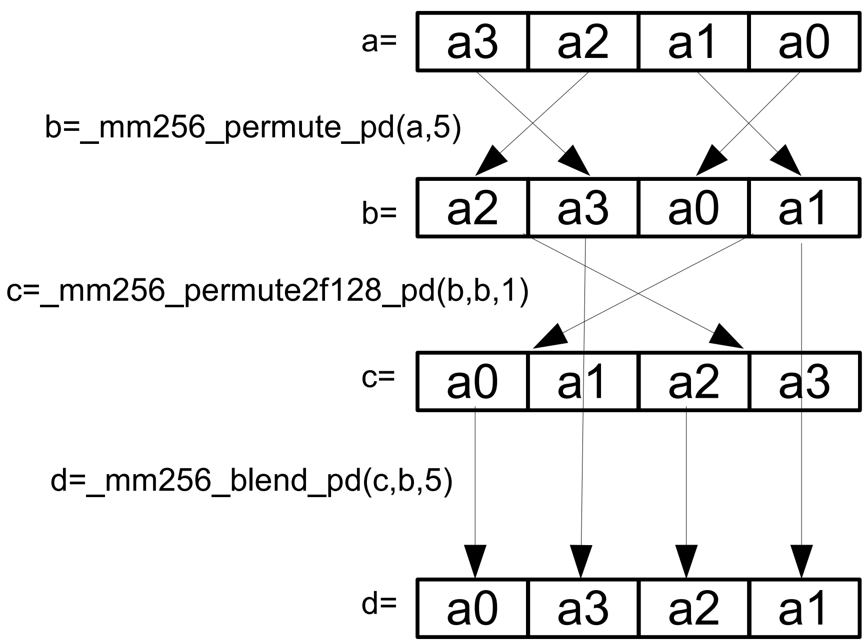
vmaskmovpd ymm0,ymm1, m64



AVX Lanes



Cyclic double vector rotate



AVX

- In 99% of the cases, the compiler does a good job vectorizing code
- Sometimes it can't vectorize or at least can't vectorize it in the way you would want it too
- In these cases it is instructive to program the respective kernel explicitly
- We have two ways of doing that
  - Intrinsics
  - Assembly

AVX Vector data types

Vectors must be declared with the following types

- __m256: a vector with 8 float entries
- __m256d: a vector with 4 double entries
- __m256i: a vector with 4 long or 8 int, etc ...

AVX

- Here we will chose intrinsics - easier than assembly, still very effective

CAVEAT

- This is not intended to write your kernels top-down in intrinsics
- Use this for checking the compiler performance against your expectations

AVX – What is an intrinsic?

- An intrinsic (function) is a function recognized by the compiler and handled specially
- In scope the intrinsic is mostly directly translated into one or more assembly instructions
- There is no way for the compiler to check correctness (well mostly)
- There are still some optimizations the compiler will do when translating, e.g. optimizing the number and use of the registers.
- In order to use intrisics in C/C++, you have to include
  #include "immintrin.h"
  in GCC and ICC

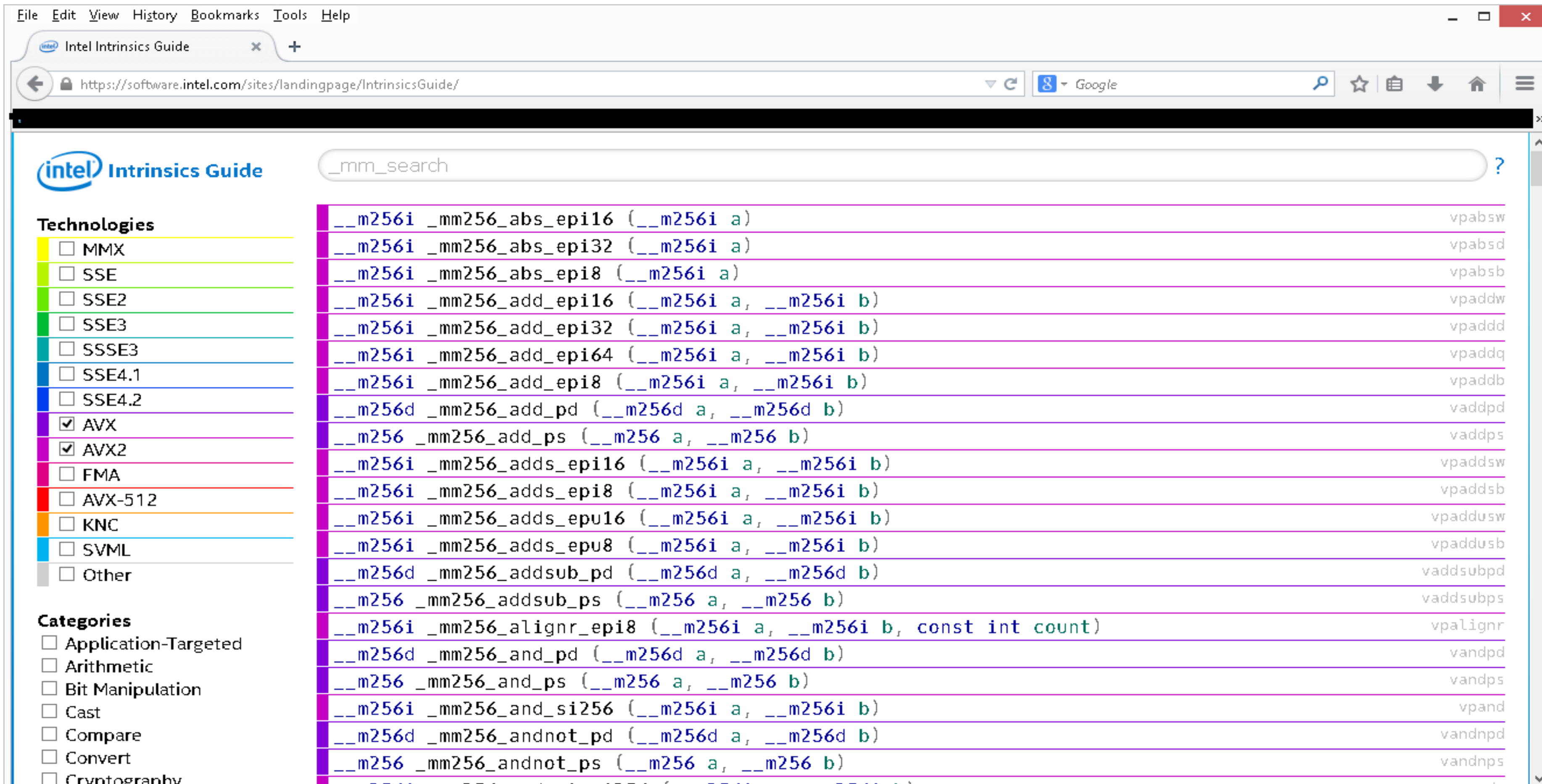
AVX Intrinsics Encoding

```
__m256 a = _mm256_add_pd(__m256 a, __m256 b)
```

Vector length	Functionality:	Type and precision
_mm256	• add	pd: packed double
_mm128	• mul	ps: packed single
	• sub	sd: scalar double
	• load	ss: scalar single
	...	

- Packed is using the full vector
- Scalar is using just the first element

AVX a very useful link



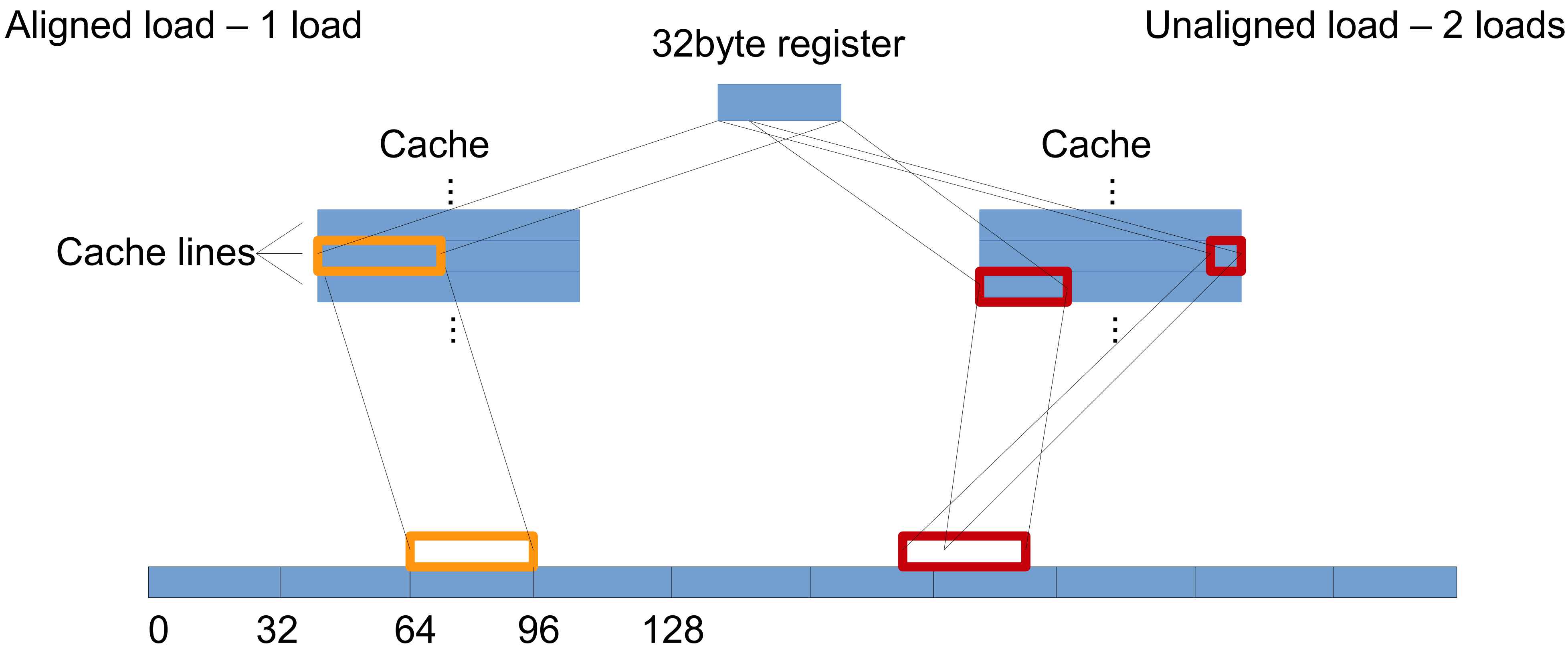
https://software.intel.com/sites/landingpage/IntrinsicsGuide

AVX - Loading data

- First thing we need to do is getting the data into the register
- Explanation of alignment next slide

- **__m256d _mm256_load_pd (double const * mem_addr)**
  Load an 32byte aligned memory location into a 256bit vector
- **__m256d _mm256_loadu_pd (double const * mem_addr)**
  Load an unaligned memory location into a 256bit vector

AVX - Alignment



We can only load full cache lines – an attempt to load an unaligned address with an aligned load causes a general-protection fault. An unaligned load causes 2 loads (since two cache lines are needed). For I/O critical workloads, aligned loads can be extremely beneficial.

AVX – Unaligned loading example

```
#include "immintrin.h"

int size=128;
double* a = new double[size];
__m256 va;
for(int i=0;i<size;i+=4)a[i]=i;

for(int i=0;i<size;i+=4){
    va = _mm256_loadu_pd(&a[i]);
}
```

Unaligned allocation

Unaligned load here

va contents in iteration n				n
3	2	1	0	0
7	6	5	4	1
11	10	9	8	2
⋮				
99	98	97	96	24

AVX – Aligned loading example

```
#include "immintrin.h"

int size=128;
double* a = (double*) _mm_malloc(size,32);
__m256 va;
for(int i=0;i<size;i+=4)a[i]=i;

for(int i=0;i<size;i+=4){
   va = _mm256_load_pd(&a[i]);
}
```

Aligned allocation

Aligned load here

va contents in iteration n				n
3	2	1	0	0
7	6	5	4	1
11	10	9	8	2
⋮				
99	98	97	96	24

AVX – storing data

- When we are done with calculating thing, we might want to save the result back to memory from our vector

- **void _mm256_store_pd (double * mem_addr, __m256d a)**
  Store a 256bit vector into a 32byte aligned memory location
- **void _mm256_storeu_pd (double * mem_addr, __m256d a)**
  Store a 256bit vector into a 32byte unaligned memory location

AVX – Arithmetic

- Arithmetic operations in AVX perform the requested per element. All basic operations are present (plus some more esoteric ones, but let's stick with basics here):

- **__m256d _mm256_add_pd (__m256d a, __m256d b)**
  Adds two vectors (a + b) element-wise and write the results into the destination
- **__m256d _mm256_sub_pd (__m256d a, __m256d b)**
  Subtracts two vectors (a - b) element-wise and write the results into the destination
- **__m256d _mm256_mul_pd (__m256d a, __m256d b)**
  Multiplies two vectors (a * b) element-wise and write the results into the destination
- **__m256d _mm256_div_pd (__m256d a, __m256d b)**
  Divides two vectors (a / b) element-wise and write the results into the destination

AVX – Setting/Broadcasting

- Setting is an example of an intrinsic which is not represented directly by an assembly instruction. Setting allows you to write one value into all entries of a vector, or write different values into the different elements of a vector:

- **__m256d _mm256_broadcast_sd (double const * mem_addr)**
  Writes the double in mem_addr into all the elements of a vector
- **__m256d _mm256_set1_pd (double a)**
  Writes the double a into all the elements of a vector
- **__m256d _mm256_setr_pd (double e3, double e2, double e1, double e0)**
  Writes the elements e0-e3 into the elements 0-4 of a vector

Data Rearrangement

- **__m256d _mm256_permute_pd (__m256d a, int imm)** Perform an **in-lane permute**. The control integer is 0 where you want the data from the first, and one where you want it from the second element in the lane

- **Examples**
- 1010b=10 (a3,a2,a1,a0) Identity
- 0000b=0 (a2,a2,a0,a0) copy the 1st element in both elements
- 1111b=15 (a3,a3,a1,a1) copy the 2nd element in both elements
- 0101b=5 (a2,a3,a0,a1) swap the elements of each lane

Data Rearrangement

- **__m256d _mm256_permute2f128_pd (__m256d a, __m256d a, int imm)** Perform an **cross-lane permute**. The control value is quite complicated, please refer to the documentation. Here, we provide some examples only.

__m256d c = mm256_permute2f128_pd(__m256d a, __m256d b, int m)

- 00010000b=48 (a3,a2,a1,a0) identity c=a
- 00110010b=50 (b3,b2,b1,b0) identity c=b
- 00000001b=1 (a1,a0,a3,a2) swap two lanes of a
- 00100011b=35 (b1,b0,b3,b2) swap two lanes of b
- 00000011b=3 (a1,a0,b3,b2) 1^{st} lane of a, 2^{nd} of b

Data Rearrangement

- **__m256d _mm256_blend_pd (__m256d a, __m256d b, const int m)**
  Copies the elements of a or b into the destination, according to the bits in m. If the bit is 0, take the first (a), if it is 1, take the second source (b)

Examples:

- 0000b=0 (a3,a2,a1,a0) Identity a
- 1111b=15 (b3,b2,b1,b0) Identity b
- 1010b=5 (b3,a2,b1,a0) 4^{th} and 2^{nd} element from b, rest from a
- 0101b=10 (a3,b2,a1,b0) 4^{th} and 2^{nd} element from a, rest from b

Appendix

- A useful function

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

void print256d(__m256d a){
   double x[4];
   _mm256_storeu_pd(&x[0], a);
   printf("%f %f %f %f\n",x[3],x[2],x[1],x[0]);
}
```

Summary

- AVX intrinsics provide a usable interface to the AVX instruction set (much easier than assembly)
- Often, the compiler can achieve similar
- In case, it doesn't hurt to check :-)