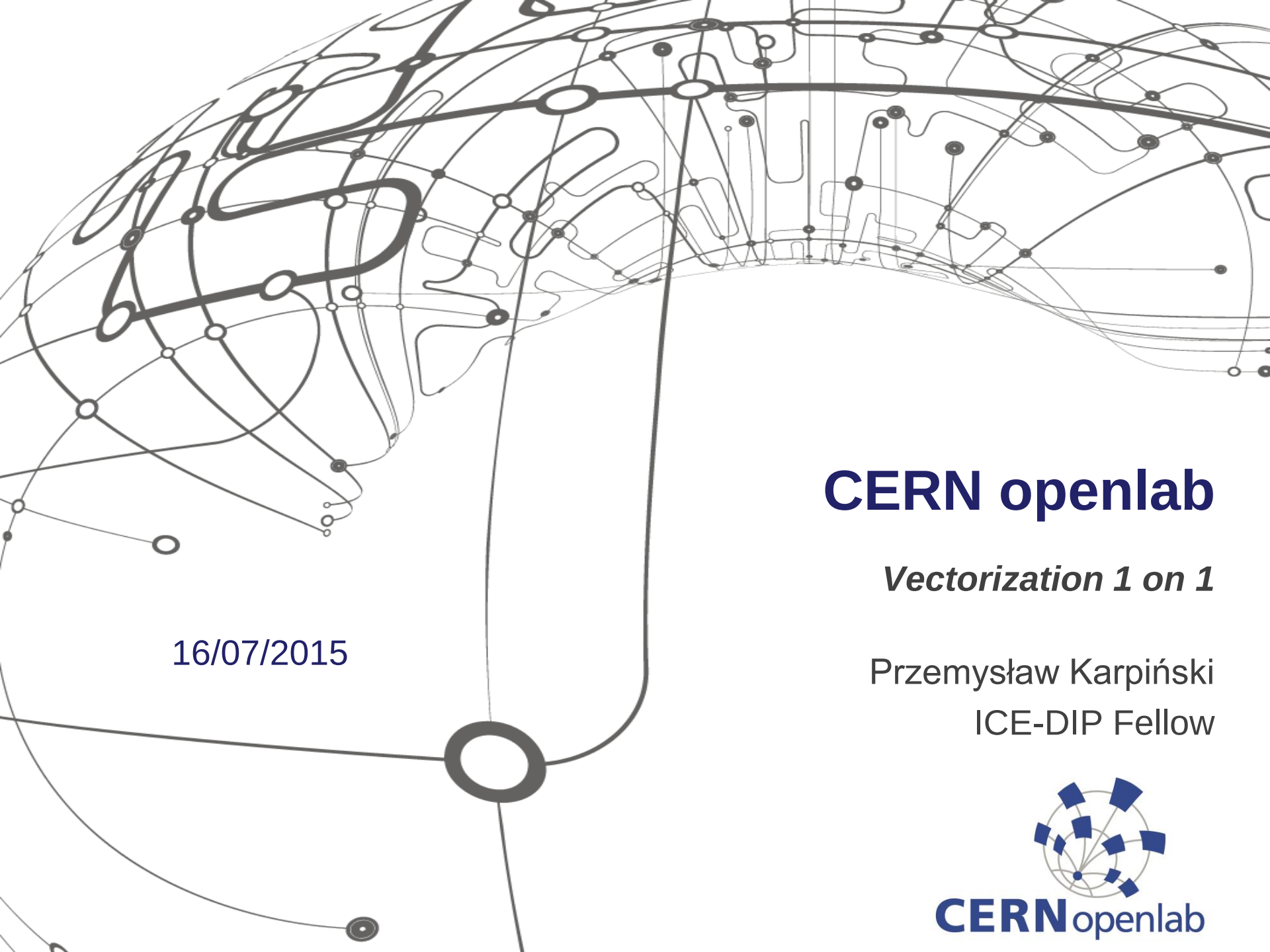


CERN openlab

Vectorization 1 on 1

16/07/2015

Przemysław Karpiński
ICE-DIP Fellow

CERN openlab

This research project has been supported by a Marie Curie Early European Industrial Doctorates Fellowship of he European Community's Seventh Framework Programme under contract number (PITN-GA-2012-316596-ICE-DIP)"

Agenda

- What is a SIMD vector?
- Why vectorization? Why now?
- Vectorization in x86
- Vectorizing code: „The simplest way”
- Some Problems

What is a SIMD vector?

Flynn's Taxonomy

	Single Data (Per Instruction)	Multiple Data (Per Instruction)
Single Instruction (Per cycle)	SISD	**SIMD**
Multiple Instructions (Per cycle)	MISD	MIMD

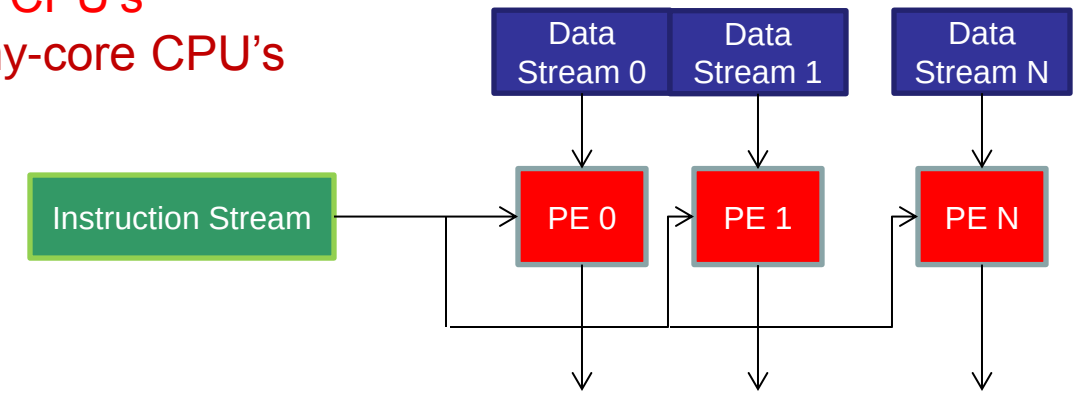
Where we can find these models:

SISD – Old uni-core x86 CPU's

MISD – No ‚real' architecture implements this model...

SIMD – GPUs, modern CPU's

MIMD – New multi/many-core CPU's



x86_64 registers

General purpose registers:	Special purpose:
RAX	Accumulator for operands and results data
RBX	Pointer to data in the DS segment
RCX	Counter for string and loop operations
RDX	I/O pointer
RSI	Pointer to data int the segment pointed to by the DS register; source pointer for string operations
RDI	Pointer to data (or destination) in the segment pointed to by the ES register; destination pointer for string operations
RSP	Stack pointer (int the SS segment)
RBP	Pointer to data on the stack (int the SS segment)
R8-R15	available in 64b modeo only

Special purpose registers	Description
CS,DS,SS,ES,FS,GS	Segment registers
EFLAGS	Program Status and Control Register
RIP	Instruction pointer
ST0-ST7	X87 FPU registers
CR0-CR4	Control registers
GDTR,LDTR,IDTR, LTR, STR	System table pointer registers
DR0-DR7	Debug Registers
MSR	Model Specific Registers

Vector registers	Instruction Set
MMX0-7	MMX
XMM0-15	SSEX, AVX
YMM0-31	AVX2
ZMM0-31	AVX512, IMCI
k0-7	AVX512

Each vector register REPRESENTS multiple data streams!

Why vectorization? Why now?

Simplest performance model

FLOPS (***FL****oating* ***P****oint* ***O****perations* ***P****er* ***S****econd)*

S – number of sockets per platform
C – number of cores per socket
T – number of hardware threads per core
F – CPU frequency [Hz]
N – number of cycles per FLO
V – number of elements per vector

Scalar platforms:

$$FLOPS_{scalar} = S * C * T * F / N$$

Vector platforms:

$$FLOPS_{scalar} = S * C * T * F * \boldsymbol{V} / N$$

S, C & T – cost a lot

F & N – not getting any better

V is cheaper and yields better return

Simplest performance model

$$FLOPS_{scalar} = S\ x\ C\ x\ T\ x\ F\ x\ N\ x\ \boldsymbol{V}$$

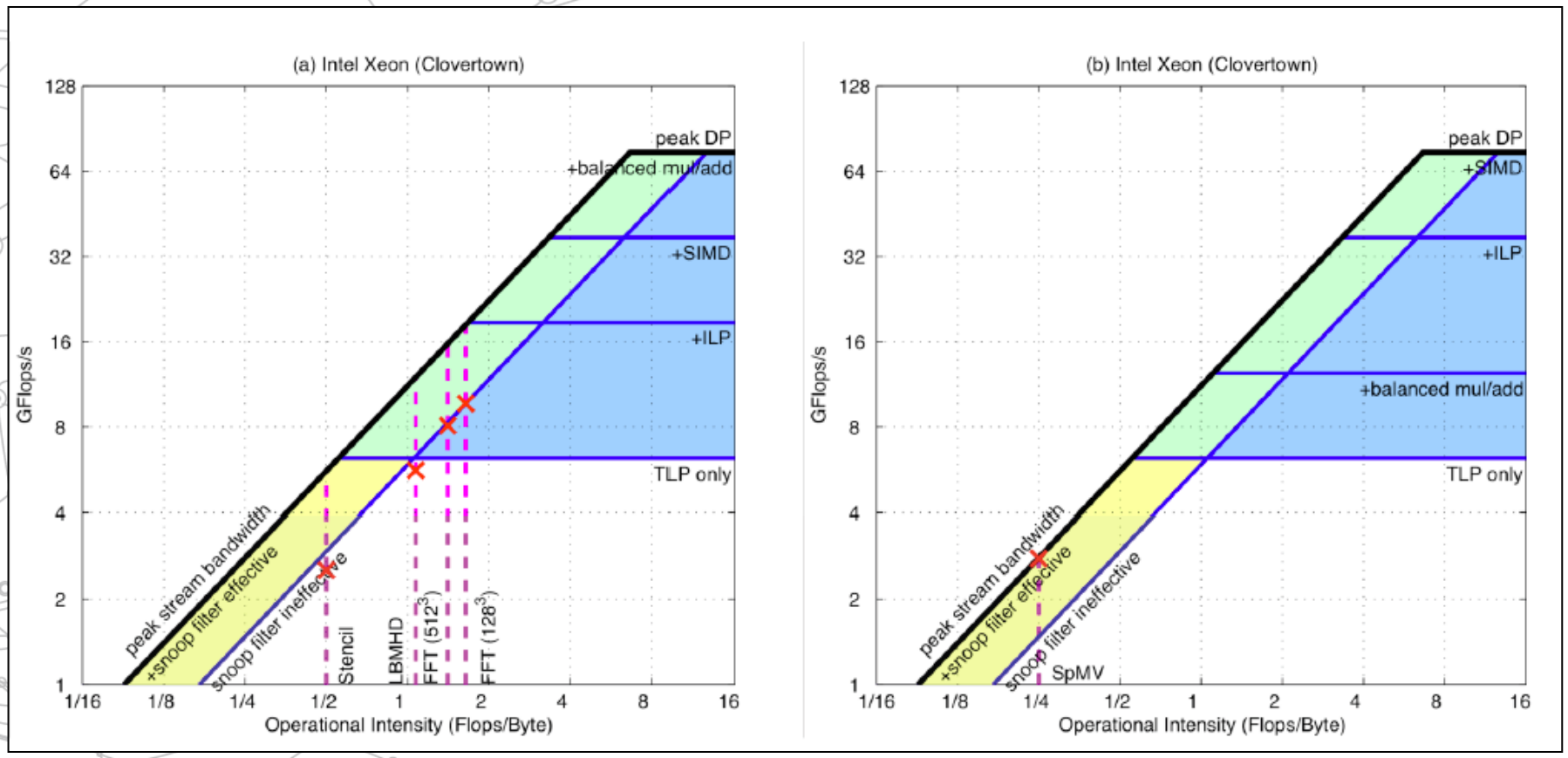
Some conditions for this to be true:

1) Compute task partitionable into multiple threads (**S + C + T**)
2) CPU clock is steady (**F**)
3) # of cycles per instruction is time invariant (**N)**
4) Cycles per vector instruction == cycles per scalar instruction (**V)**

Additional considerations:

5) Data access (cache subsystem)
6) Data dependencies
7) Conditional branching

Roofline models



Roofline Model for Intel Xeon Processor
(From: S.W. Williams, A. Waterman, D. A. Patterson, „Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures")

Vectorization in x86

x86 SIMD extensions

Instruction set:	Vector size:	scalars in vector:						Registers naming:
		Int				Float		
		8b	16b	32b	64b	32b	64b	
MMX	64b	8	4	2	-	-	-	MM0-MM7
SSE	64b(int) 128b(float)	8(MMX)	4(MMX)	2(MMX)	-	4	-	XMM0-7
SSE2	128b(int) 128b(float)	16	8	4	1(MMX)	4	2	XMM0-7
SSE3	128b(int) 128b(float)	16	8	4	1(MMX)	4	2	XMM0-7
SSSE3	128b(int) 128b(float)	16	8	4	1(MMX)	4	2	XMM0-15
SSE4.1	128b(int) 128b(float)	16	8	4	1(MMX)	4	2	XMM0-15
SSE4.2	128b(int) 128b(float)	16	8	4	1(MMX)	4	2	XMM0-15
AVX	128b(int) 256b(float)	16(XMM)	8(XMM)	4(XMM)	1(XMM)	8	4	YMM0-15
AVX2	256b	32	16	8	2	8	4	YMM0-15
KNCNI/IMCI/K1OM	512b	64	32	16	8	16	8	ZMM0-31/k0-7
AVX512	512b	64/32/16	32/16/8	16/8/4	8/4/2*	16/8/4	8/4/2	ZMM0-31/k0-7

x86 SIMD extensions

Categories of instructions:

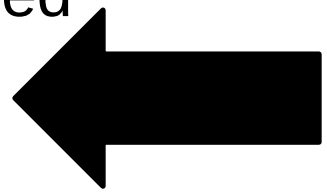
- Load/Store:
- Arithmetic: addition, multiplication, fma, etc.
- Logical: and, or, and-not, etc.
- Bit manipulation: shift, zero-count
- Compare: LT, LE, GT, GE
- Type convertion: round, float->int, int->float etc.
- Math functions: exp, ceil, rand, etc.

- Gather/Scatter
- Initialization: setzero, set1, set.
- Swizzle, blend, breadcast, insert

Operations in all sorts of flavours: float, double, int32, masked, unmasked and with implicit operands

How to use vectorization?

1) Auto-vectorization:
   - Compiler optimizes code to generate vector instructions
   - User gives „hints” like **#pragma ivdep**
   - „Minor” code modifications required
2) Language extensions:
   - Intel(R) Cilk(TM) Plus
   - Code modernization required
3) Explicit vectorization:
   - Inline-assembly
   - Compiler intrinsics



This presentation

CERN openlab

Vectorizing code:

„The simplest way”

Inline assembly (GCC)

1) In Intel(R) ISA reference find interesting instruction:

VEX.NDS.128.66.0F.WIG FE /r VPADDD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add packed doubleword integers from *xmm2, xmm3/m128* and store in *xmm1*.

2) Write inline assembly....

```
int arr1[8] = {1, 2, 3, 4, 5, 6, 7, 8};
int arr2[8] = {90, 100, 110, 120, 130, 140, 150, 160};
int res[8] = {0, 0, 0, 0, 0, 0, 0, 0};

asm("movdqa %0, %%xmm0" :: "m"(*arr1));
asm("movdqa %0, %%xmm1" :: "m"(*arr2));
asm("vpaddd  %%xmm0, %%xmm1, %%xmm0" : :);
asm("movdqa %%xmm0, %0" : "=m"(*res) :);
```

3) Compile with GCC

```
$ g++ source.cpp –std=c++11 –O0 –fno-inline -mavx
```

Intrinsics (GCC)

1) In Intel(R) ISA reference find interesting instruction:

VEX.NDS.128.66.0F.WIG FE /r VPADDD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add packed doubleword integers from *xmm2, xmm3/m128* and store in *xmm1*.

2) In Intel(R) intrinsics reference find interesting intrinsic...:

```
__m256i _mm256_add_epi32 (__m256i a, __m256i b)                    vpaddd
```

Synopsis

```
__m256i _mm256_add_epi32 (__m256i a, __m256i b)
#include "immintrin.h"
Instruction: vpaddd ymm, ymm, ymm
CPUID Flags: AVX2
```

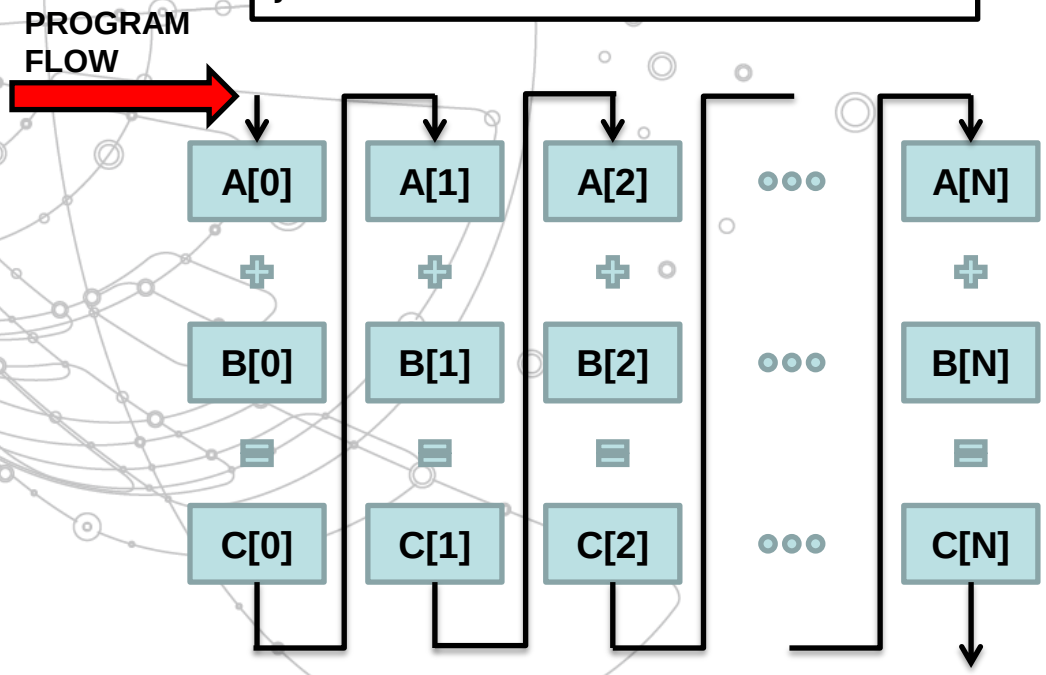
3) Write what you need...

```
#include <immintrin.h>
...
int arr1[8] = {1, 2, 3, 4, 5, 6, 7, 8};
int arr2[8] = {90, 100, 110, 120, 130, 140, 150, 160};
int res[8] = {0, 0, 0, 0, 0, 0, 0, 0};

 __m128i xmm0, xmm1, xmm2;
 xmm0 = _mm_loadu_si128((__m128i const*)arr1);
 xmm1 = _mm_loadu_si128((__m128i const*)arr2);
 xmm2 = _mm_add_epi32(xmm0, xmm1);
 _mm_storeu_si128((__m128i *)res, xmm2);
```

Loop Peeling

Scalar operations

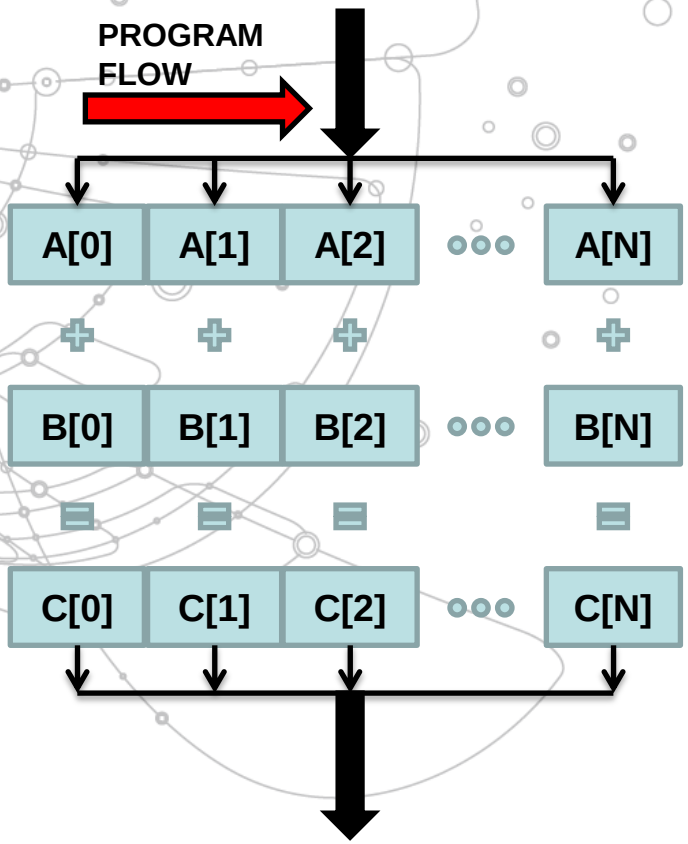
```
unsigned int N = ...

int A[N] = {...};
int B[N] = {...};
int C[N];

for(unsigned int i = 0; i < N; i++)
{
    C[i] = A[i] + B[i];
}
```




```
g++ 4.8.2
(-std=c++11 -O0 -fno-inline -S -fverbose-asm -g)
.LBB6:
  .loc 1 42 0
  movl $0, -24(%rbp) #, i
  jmp .L7 #
.L8:
  .loc 1 44 0 discriminator 2
  movl -24(%rbp), %eax      # i, D.54304
  leaq 0(,%rax,4), %rdx     #, D.54304
  movq -56(%rbp), %rax      # C, tmp114
  addq %rdx, %rax           # D.54304, D.54305
  movl -24(%rbp), %edx      # i, D.54304
  leaq 0(,%rdx,4), %rcx     #, D.54304
  movq -40(%rbp), %rdx      # A, tmp115
  addq %rcx, %rdx           # D.54304, D.54305
  movl (%rdx), %ecx         # *_39, D.54303
  movl -24(%rbp), %edx      # i, D.54304
  leaq 0(,%rdx,4), %rsi     #, D.54304
  movq -48(%rbp), %rdx      # B, tmp116
  addq %rsi, %rdx           # D.54304, D.54305
  movl (%rdx), %edx         # *_43, D.54303
  addl %ecx, %edx           # D.54303, D.54303
  movl %edx, (%rax)         # D.54303, *_36

.loc 1 42 0 discriminator 2
  addl $1, -24(%rbp)        #, i

.L7: .loc 1 42 0 is_stmt 0 discriminator 1
  movl -24(%rbp), %eax      # i, tmp117
  cmpl -28(%rbp), %eax      # N, tmp117
  jb .L8                    #,
```

Scalar vs. SIMD

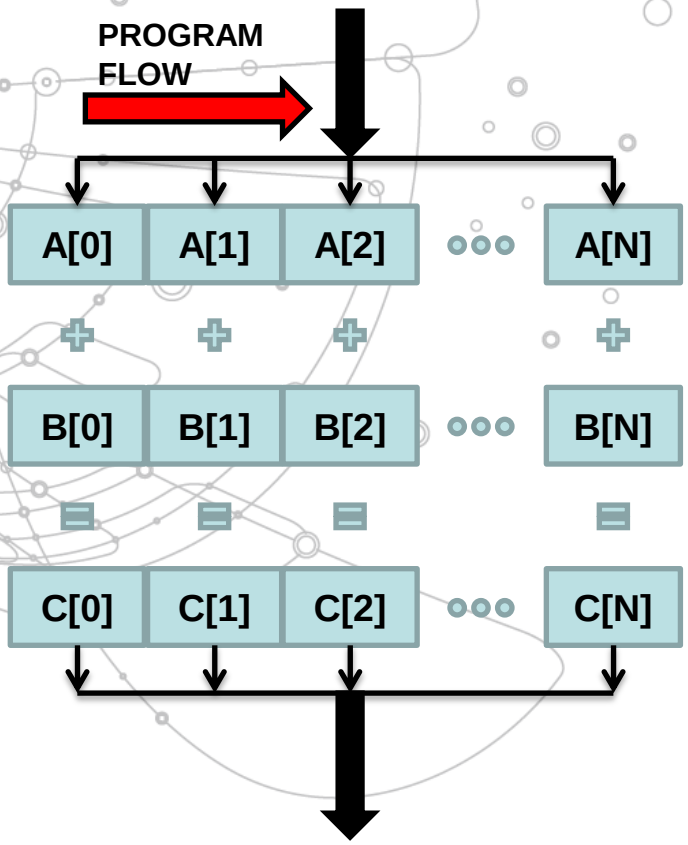
```
72: int N_full_vectors = N/8;
73:
74: __m256i ymm_mask = _mm256_set1_epi32(0xFFFFFFFF);
75: for(int i = 0; i < N_full_vectors; i++)
76: {
77:  int offset = i * 8;
78:
79:  __m256i ymm_a, ymm_b, ymm_c;
80:  ymm_a = _mm256_maskload_epi32((const int*)&A[offset], ymm_mask);
81:  ymm_b = _mm256_maskload_epi32((const int*)&B[offset], ymm_mask);
82:  ymm_c = _mm256_add_epi32(ymm_a, ymm_b);
83:  _mm256_maskstore_epi32(&C[offset], ymm_mask, ymm_c);
84:}
```




```
ICC 15.0.2
(-std=c++11 -O0 -inline-level=0 -S -xCORE-AVX2)
..B19.37:                       # Preds ..B19.36
  movl      -484(%rbp), %eax      #72.26
  sarl      $2, %eax              #72.28
  shrl      $29, %eax             #72.28
  addl      -484(%rbp), %eax      #72.28
  sarl      $3, %eax              #72.28
  movl      %eax, -452(%rbp)      #72.24
  vpcmpeqd  %ymm0, %ymm0, %ymm0   #74.24
  vmovdqu   %ymm0, -272(%rbp)     #74.24
  vmovdqu   -272(%rbp), %ymm0     #74.24
  vmovdqu   %ymm0, -240(%rbp)     #74.22
  movl      $0, -448(%rbp)        #75.15
  # LOE rbx rbp rsp r12 r13 r14 r15 rip
..B19.38:                       # Preds ..B19.39..B19.37
  movl      -448(%rbp), %eax      #75.20
  movl      -452(%rbp), %edx      #75.24
  cmpl      %edx, %eax            #75.24
  jge       ..B19.40  # Prob 50%  #75.24
  # LOE rbx rbp rsp r12 r13 r14 r15 rip
..B19.39:                       # Preds ..B19.38
  movl      -448(%rbp), %eax      #77.22
  imull     $8, %eax, %eax        #77.26
  movl      %eax, -444(%rbp)      #77.20
  movl      -448(%rbp), %eax      #80.17
  movslq    %eax, %rax            #80.17
  imulq     $4, %rax, %rax        #80.17
  addq      -416(%rbp), %rax      #80.17
  vmovdqu   -240(%rbp), %ymm0     #80.17
  vpmaskmovd (%rax), %ymm0, %ymm0 #80.17
  vmovdqu   %ymm0, -208(%rbp)     #80.17
  vmovdqu   -208(%rbp), %ymm0     #80.17
  vmovdqu   %ymm0, -176(%rbp)     #80.9
```

Scalar vs. SIMD

```
72: int N_full_vectors = N/8;
73:
74: __m256i ymm_mask = _mm256_set1_epi32(0xFFFFFFFF);
75: for(int i = 0; i < N_full_vectors; i++)
76: {
77:  int offset = i * 8;
78:
79:  __m256i ymm_a, ymm_b, ymm_c;
80:  ymm_a = _mm256_maskload_epi32((const int*)&A[offset], ymm_mask);
81:  ymm_b = _mm256_maskload_epi32((const int*)&B[offset], ymm_mask);
82:  ymm_c = _mm256_add_epi32(ymm_a, ymm_b);
83:  _mm256_maskstore_epi32(&C[offset], ymm_mask, ymm_c);
84:}
```



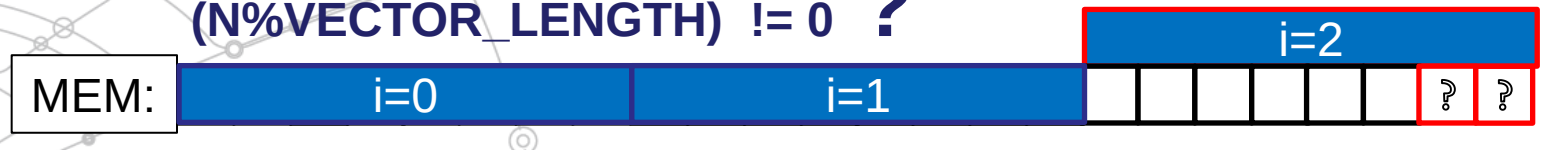

```
ICC 15.0.2
(-std=c++11 -O0 -inline-level=0 -S -xCORE-AVX2)
..B19.37:                       # Preds ..B19.36
  movl      -484(%rbp), %eax      #72.26
  sarl      $2, %eax              #72.28
  shrl      $29, %eax             #72.28
  addl      -484(%rbp), %eax      #72.28
  sarl      $3, %eax              #72.28
  movl
  vpcm
  vmov
  vmov
  vmov
  movl
  # LO
..B19.
  movl
  movl
  cmpl
  jge
  # LO
..B19.
  movl
  imul
  movl
  movl
  movs
  imul
  addq
  vmov
  vpma
  vmov
  vmov
  vmov
```

```
  movl      -448(%rbp), %eax          #81.17
  movslq    %eax, %rax                #81.17
  imulq     $4, %rax, %rax            #81.17
  addq      -400(%rbp), %rax          #81.17
  vmovdqu   -240(%rbp), %ymm0         #81.17
  vpmaskmovd (%rax), %ymm0, %ymm0     #81.17
  vmovdqu   %ymm0, -144(%rbp)         #81.17
  vmovdqu   -144(%rbp), %ymm0         #81.17
  vmovdqu   %ymm0, -112(%rbp)         #81.9
  vmovdqu   -176(%rbp), %ymm0         #82.17
  vmovdqu   -112(%rbp), %ymm1         #82.17
  vpaddd    %ymm1, %ymm0, %ymm0       #82.17
  vmovdqu   %ymm0, -80(%rbp)          #82.17
  vmovdqu   -80(%rbp), %ymm0          #82.17
  vmovdqu   %ymm0, -48(%rbp)          #82.9
  movl      -448(%rbp), %eax          #83.9
  movslq    %eax, %rax                #83.9
  imulq     $4, %rax, %rax            #83.9
  addq      -384(%rbp), %rax          #83.9
  vmovdqu   -240(%rbp), %ymm0         #83.9
  vmovdqu   -48(%rbp), %ymm1          #83.9
  vpmaskmovd %ymm1, %ymm0, (%rax)     #83.9
  movl      $1, %eax                  #75.40
  addl      -448(%rbp), %eax          #75.40
  movl      %eax, -448(%rbp)          #75.40
  jmp       ..B19.38      # Prob 100% #75.40
```

Problem 1: loop peeling

What happens if

(N%VECTOR_LENGTH) != 0 ?




```
72: int N_full_vectors = N/8;
73:
74: __m256i ymm_mask = _mm256_set1_epi32(0xFFFFFFFF);
75: for(int i = 0; i < N_full_vectors; i++)
76: {
77:  int offset = i * 8;
78:
79:  __m256i ymm_a, ymm_b, ymm_c;
80:  ymm_a = _mm256_maskload_epi32((const int*)&A[i],
                                  ymm_mask);
81:  ymm_b = _mm256_maskload_epi32((const int*)&B[i],
                                  ymm_mask);
82:  ymm_c = _mm256_add_epi32(ymm_a, ymm_b);
83:  _mm256_maskstore_epi32(&C[i], ymm_mask, ymm_c);
84:}
```

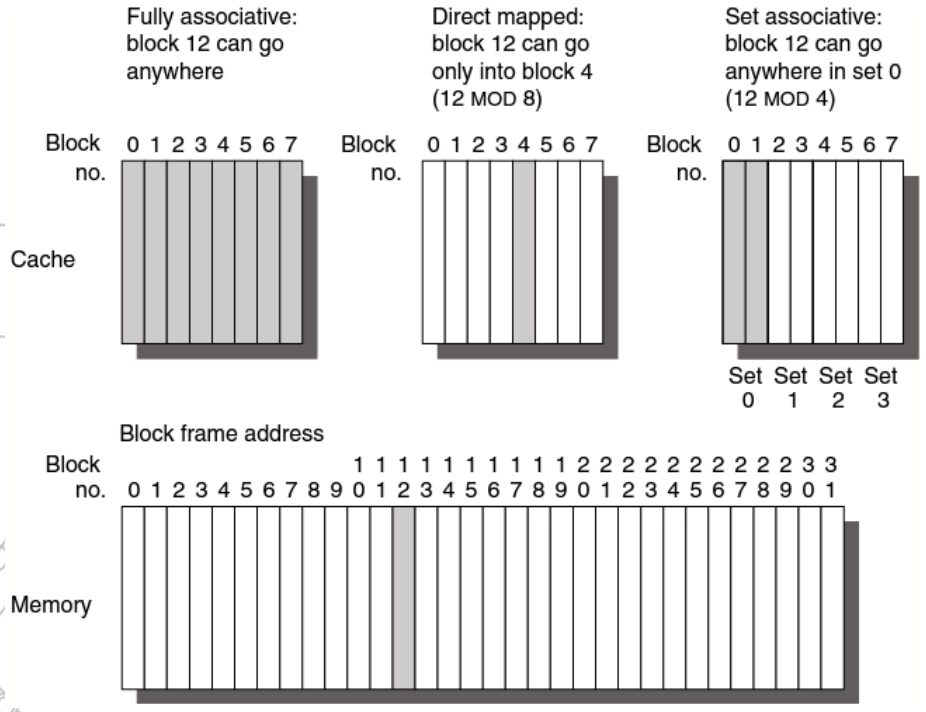
```
const unsigned int N = ...

SCALAR_T A[N] = {...};
SCALAR_T B[N] = {...};
SCALAR_T C[N];

for(unsigned int i = 0; i < N; i++)
{
    C[i] = A[i] + B[i];
}
```

```
72: int N_full_vectors = N/8;
73:
74: __m256i ymm_mask = _mm256_set1_epi32(0xFFFFFFFF);
75: for(int i = 0; i < N_full_vectors; i++)
76: {
77:  int offset = i * 8;
78:
79:  __m256i ymm_a, ymm_b, ymm_c;
80:  ymm_a = _mm256_maskload_epi32((const int*)&A[i], ymm_mask);
81:  ymm_b = _mm256_maskload_epi32((const int*)&B[i], ymm_mask);
82:  ymm_c = _mm256_add_epi32(ymm_a, ymm_b);
83:  _mm256_maskstore_epi32(&C[i], ymm_mask, ymm_c);
84:}
85
86:// Peeling off the reminder of loop iterations
87:for(int i = 8*N_full_vectors; i < N; i++)
88:{
89:  C[i] = A[i] + B[i];
90:}
```

Cache utilization

Cache subsystem

How cache works?



From: J. L. Hennessy, D. A. Patterson, „Computer Architecture: A quantitative Approach (5'th edition)"

Cache subsystem

Intel(R) Xeon(R) CPU
E3-1280 v3 @ 3.60GHz (AVX2 supported)

Cache Level	Cache size		Cache line size		# cache lines		Associativity	
	Data	Instruction	Data	Instruction	Data	Instruction	Data	Instruction
L0	32kB	32kB	64B	64B	512	512	8-way set	8-way set
L1	256kB		64B		4096		8-way set	
L2 (shared)	8192kB		64B		131072		16-way set	

Number of „entities” that fit into cache:

Cache Level	Scalar		Vector	
	32b (unpacked Int8, unpacked int16, int 32, float)	64b (int64, double)	128b (SSEx, AVX)	256b (AVX2)
L0	8192	4096	2048	1024
L1	65536	32768	16384	8192
L2 (shared)	2097k (524k per core)	2097k (262k per core)	524k (131k per core)	262k (65k per core)

Remember: a *cache-miss penalty* is higher for vectors!!!

Data alignment

Processors are designed so that data is efficiently loaded only from certain memory addresses.

	Value of 3 low-order bits of byte address							
Width of object	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1 byte (byte)	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned
2 bytes (half word)	Aligned		Aligned		Aligned		Aligned	
2 bytes (half word)		Misaligned		Misaligned		Misaligned		Misaligned
4 bytes (word)	Aligned				Aligned			
4 bytes (word)		Misaligned				Misaligned		
4 bytes (word)			Misaligned				Misaligned	
4 bytes (word)				Misaligned				Misaligned
8 bytes (double word)	Aligned							
8 bytes (double word)		Misaligned						
8 bytes (double word)			Misaligned					
8 bytes (double word)				Misaligned				
8 bytes (double word)					Misaligned			
8 bytes (double word)						Misaligned		
8 bytes (double word)							Misaligned	
8 bytes (double word)								Misaligned

Figure A.5 Aligned and misaligned addresses of byte, half-word, word, and double-word objects for byte-addressed computers. For each misaligned example some objects require two memory accesses to complete. Every aligned object can always complete in one memory access, as long as the memory is as wide as the object. The figure shows the memory organized as 8 bytes wide. The byte offsets that label the columns specify the low-order 3 bits of the address.

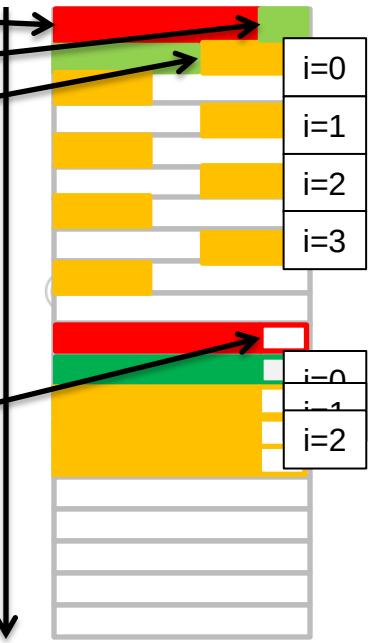
From: J. L. Hennessy, D. A. Patterson, „Computer Architecture: A quantitative Approach (5'th edition)"

Problem 2: Cache misalignment

Direct mapped cache

```
struct pseudo_aligned_foo
{
  int a[13]; //13x4B ->52B
  int b[13]; //13x4B ->52B
  int c[13]; //13x4B ->52B
}
```

Cache Line Width: 64B




```
pseudo_aligned_foo list[1000];

for(int i = 0; i < 1000; i++
{
  vec512 x = masked_load(&list[i].c, 0x1FFF);
    ... // Do something with x
}
```

Even if compiler only loads ‚c' from each list, we still have 2 cache-lines used per loop!!!

```
struct aligned_foo
{
    int a[13]; //13x4B ->52B
    int reserved_a[3];
    int b[13]; //13x4B ->52B
    int reserved_b[3];
    int c[13]; //13x4B ->52B
    int reserved_c[3];
}
```

More cache occupied, but cache utilisation reduced!!!

KNC: all vector load/store operations have to be aligned!!! Misalignment results in segfault.

Controlling alignment

Compiler specific & OS specific...

```
struct static_aligned_foo
{
 __attribute__ ((aligned(64))) int a[13+3];
 __attribute__ ((aligned(64))) int b[13+3];
 __attribute__ ((aligned(64))) int c[13+3];
}
```

```
struct dynamic_aligned_foo
{
    int* a;
    int* b;
    int* c;

    dynamic_aligned_foo()
    {
        posix_memalign( &a, 64, (13+3)*4);
        posix_memalign( &b, 64, (13+3)*4);
        posix_memalign( &c, 64, (13+3)*4);
    }
}
```

The C++11 way:

```
struct static_aligned_foo_cpp_11
{
    alignas(64) int a;
    alignas(64) int b;
    alignas(64) int c;
}

struct dynamic_aligned_foo_cpp_11
{
    int* a;
    int* b;
    int* c;

    dynamic_aligned_foo_cpp_11()
    {
        std::align( &a, 64, (13+3)*4);
        std::align( &b, 64, (13+3)*4);
        std::align( &c, 64, (13+3)*4);
    }
}
```

Conditional execution

Conditional execution

```
const unsigned int N = ...

SCALAR_T A[N] = {...};
SCALAR_T B[N] = {...};
SCALAR_T C[N];
Bool     mask[N] = {...};

for(unsigned int i = 0; i < N; i++)
{
    if(mask[i] == true)
    {
        C[i] = A[i] + B[i];
    }
    else
    {
        C[i] = 42;
    }
}
```

There is
NO SUCH THING
as
„if(vector ==...)”

Problem 3: conditional execution

```
const unsigned int N = ...

SCALAR_T A[N] = {1, 2,   3,  4,  5,  6,  7,  8};
SCALAR_T B[N] = {9, 10, 11, 12, 13, 14, 15, 16};
SCALAR_T C[N];
__mmask8 ymm_blend_mask = 0x63;

__mmask8 YMM_MASK = 0xFF;
__m256i YMM_42   = _mm256_set1_epi32(42);
for(int i = 0; i < N/8; i++)
{
    int offset = i * 8;
    __m256i   ymm_a, ymm_b, ymm_c, ymm_temp;
    ymm_a = _mm256_maskload_epi32((const int*)&A[offset], YMM_MASK);
    ymm_b = _mm256_maskload_epi32((const int*)&B[offset], YMM_MASK);
    ymm_temp = _mm256_add_epi32(ymm_a, ymm_b);
    ymm_c = _mm256_blendv_epi32(ymm_temp, YMM_42, ymm_blend_mask);
    _mm256_maskstore_epi32(&C[offset], YMM_MASK, ymm_c);
}
```

```
vec blendv(vec vec0,
           vec vec1,
           mask_vec mask)
{
  vec temp;
  if(mask.elem[i] == ,1')
    temp.elem[i] = vec0.elem[i];
  else
    temp.elem[i] = vec1.elem[i];
  return temp;
}
```

```
i = 1
ymm_temp       : { 10, 12, 14, 16,   18, 20, 22, 24}
ymm_blend_mask : {  1,  1,  0,  0,    0,  1,  1,  0}
C[0:7]         : { 10, 12, 42, 42,   42, 20, 22, 42}
```

Memory access patterns

Random Memory Access

*A **histogram** is a graphical representation of the distribution of numerical data. It is an estimate of the probability density function (PDF) of a continuous variable /.../.*

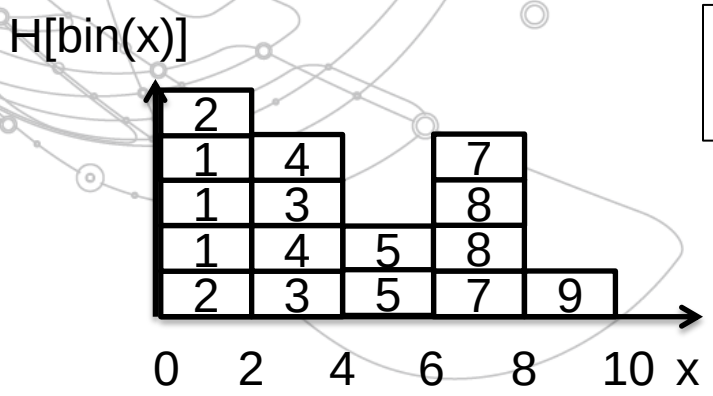
To construct a histogram, the first step is to „bin" the range of values – that is, divide the entire range of values into a series of small intervals – and then count how many values fall into each interval.

-- Wikipedia definition --

Input data:

X = { 3, 7, 8, 2, 4, 1, 5, 1, 1, 8, 3, 2, 4, 9, 5, 7}

Histogram with bins of width 2 ({1, 2}, {3,4}, {5,6}, etc.):



Histogram in memory:

H[5] = {5, 4, 2, 3, 1}

X traversed in a ‚linear' manner,
H accessed RANDOMLY

Problem 4: non-linear access pattern

Gather operation: „load values from memory using arbitrary offsets”

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 90 /r VPGATHERDD *xmm1, vm32x, xmm2*	RMV	V/V	AVX2	Using dword indices specified in *vm32x*, gather dword values from memory conditioned on mask specified by *xmm2*. Conditionally gathered elements are merged into *xmm1*.

Pseudocode for gather:

```
vec8_epi32 gather_epi32(
    const int32 * X,
    int[8] indices)
{
    vec8_epi32 vec;

    for(int i = 0; i < 8; i++)
    {
        vec[i] = X[indices[i]];
    }
    return vec;
}
```

Scatter is just reverse of that.

```
void gather_epi32(
    vec8_epi32 vec,
    const int32 * X,
    int[8] indices)
{
    vec8_epi32 retval;

    for(int i = 0; i < 8; i++)
    {
        X[indices[i]] = vec[i]
    }
}
```

Calculating Histogram

```
int X[N];
int histogram[128/2]; // range of values are 1-127, we use 2-value bins

__m256i YMM_MASK = _mm256_set1_epi32(0xFFFFFFFF);
__m256i YMM_1    = _mm256_set1_epi32(1);      // just ,1's vector
__m256i YMM_BIN_WIDTH = _mm256_set1_epi32(2); // width of a bin

for(int i = 0; i < N/8; i++)
{
    int offset = i * 8;
    __m256i ymm_x, ymm_hist, ymm_temp0, ymm_temp1;

    // Load values from X into vector
    ymm_x = _mm256_maskload_epi32(&X[offset], YMM_MASK);

    // Values from X are in range [0-128].
    // We will calculate for bins of width 2
    ymm_temp0 = _mm256_add_epi32(ymm_x, YMM_BIN_WIDTH);
    ymm_temp1 = _mm256_div_epi32(ymm_temp1, YMM_BIN_WIDTH);

    // ymm_temp0 now holds offsets of bins
    // gather from histogram
    ymm_hist = _mm256_i32gather_epi32(&histogram[0], ymm_temp1, 1);

    // increment – there is no ,++' operation in vector instruction set
    ymm_temp0 = _mm256_add_epi32(ymm_hist, YMM_1);

    // scatter back to histogram
    // NO CAN DO!!! NO SCATTER ON AVX2...
    // ...
}
```

Problem 5: Be careful of what ISA has to offer...

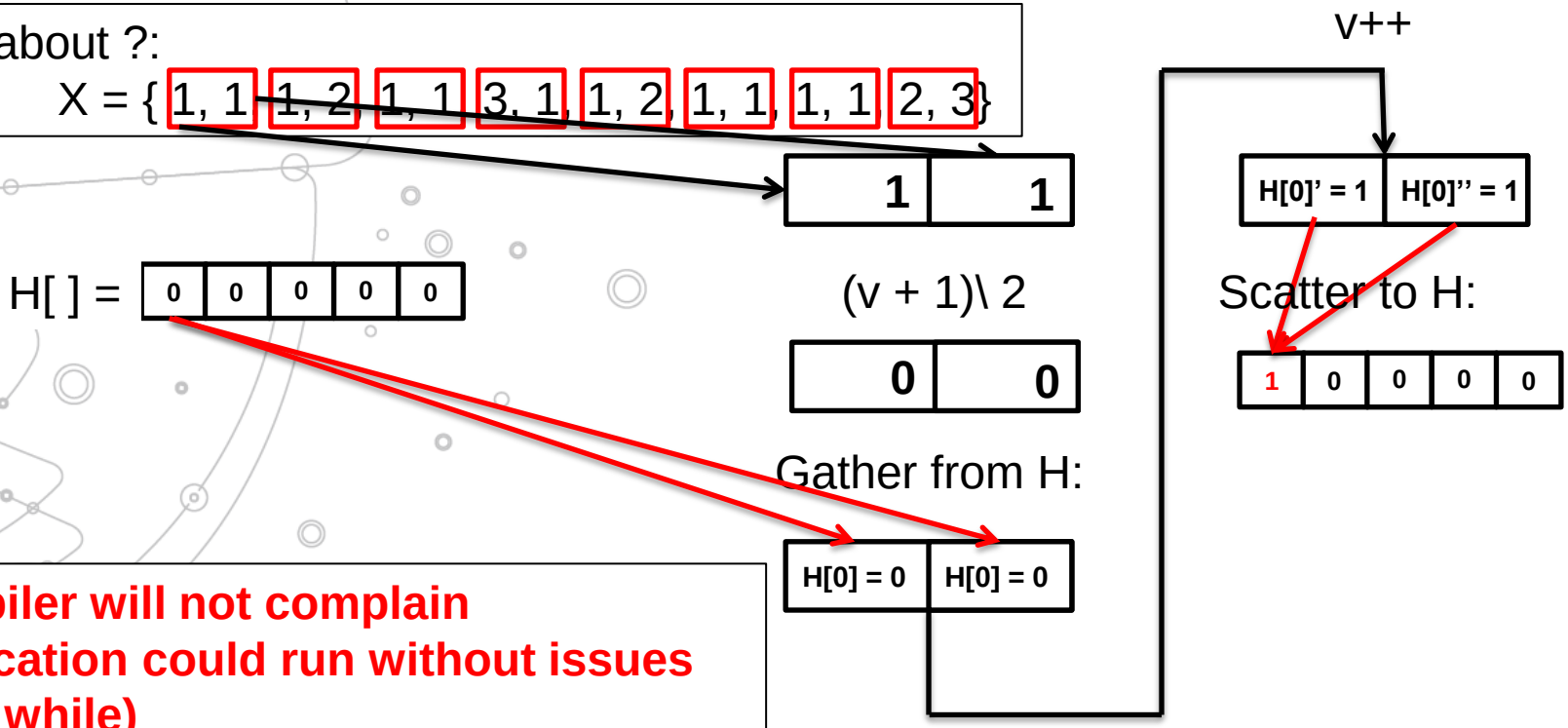
Yet another problem

Previous input data:

X = { 3, 7, 8, 2, 4, 1, 5, 1, 1, 8, 3, 2, 4, 9, 5, 7}

What about ?:

X = { 1, 1, 1, 2, 1, 1, 3, 1, 1, 2, 1, 1, 1, 1, 2, 3}



- **Compiler will not complain**
- **Application could run without issues (for a while)**
- **No way of saying what happens when this situation occurs**

Data dependencies

Data dependencies

Try vectorizing this scalar code:

```
int x[32] = {1, 2, ..., 32}
int y[33] = {0, 0, ..., 0}

for(int i = 1; i < 33; i++)
{
   y[i] = x[i-1] + y[i-1];
}
```

```
int x[32] = {1, 2, ..., 32}
int y[33] = {0, 0, ..., 0}

const int VEC_LEN = 4;

for(int i = 1; i < 33; i+=VEC_LEN)
{
   vec_i32 v0 = load(&x[i-1]);
   vec_i32 v1 = load(&y[i-1]);

   vec_i32 v2 = v0 + v1 // !?
   store(&y[i], v2);
}
```

Problem 5: Data dependencies

```
int x[32] = {1, 2, ..., 32}
int y[33] = {0, 0, ..., 0}

const int VEC_LEN = 4;

for(int i = 1; i < 33; i+=VEC_LEN)
{
   vec_i32 v0 = load(&x[i-1]);
   vec_i32 v1 = load(&y[i-1]);

   vec_i32 v2 = v0 + v1
   store(&y[i], v2);
}
```

V2 depends on result generated in the same iteration!!!

```
i = 0
  v0 = {x[0], x[1], x[2], x[3]}
     = {   1,    2,    3,    4}
  v1 = {y[0], y[1], y[2], y[3]}
     = {   0,    0,    0,    0}

  v2 = {   1,    2,    3,    4}
```

```
Expected:
  y[1] = x[0] + y[0] = 1 + 0 = 1
  y[2] = x[1] + y[1] = 2 + 1 = 3
  y[3] = x[2] + y[2] = 3 + 3 = 6
  y[4] = x[3] + y[3] = 4 + 6 = 10
```

Problem 5: Data dependencies

```
int x[32] = {1, 2, ..., 32}
int y[33] = {0, 0, ..., 0}

for(int i = 4; i < 33; i++)
{
   y[i] = x[i-4] + y[i-4];
}
```

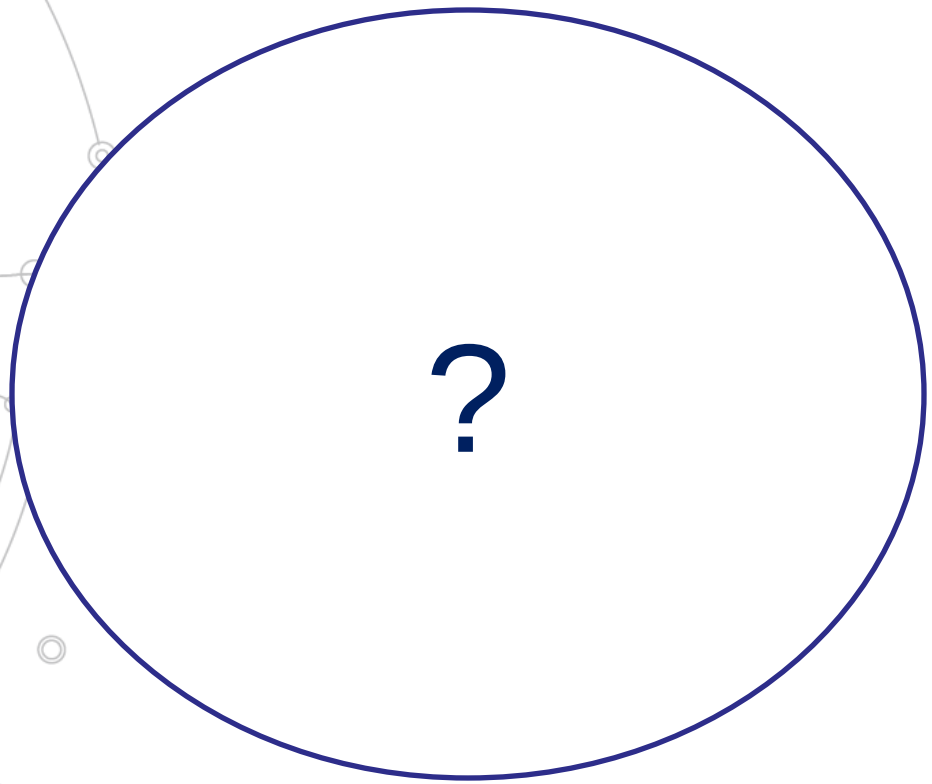
```
Expected:
i=4:  y[4]  = x[0] + y[0] = 1 + 0 = 1
i=5:  y[5]  = x[1] + y[1] = 2 + 0 = 2
i=6:  y[6]  = x[2] + y[2] = 3 + 0 = 3
i=7:  y[7]  = x[3] + y[3] = 4 + 0 = 4
i=8:  y[8]  = x[4] + y[4] = 5 + 1 = 6
i=9:  y[9]  = x[5] + y[5] = 6 + 2 = 8
i=10: y[10] = x[6] + y[6] = 7 + 3 = 10
...
```

```
int x[32] = {1, 2, ..., 32}
int y[33] = {0, 0, ..., 0}

const int VEC_LEN = 4;

for(int i = 4; i < 33; i+=VEC_LEN)
{
   vec_i32 v0 = load(&x[i-4]);
   vec_i32 v1 = load(&y[i-4]);

   vec_i32 v2 = v0 + v1 // OK!
   store(&y[i], v2);
}
```

Golden rules

1) Keep your structures aligned!!!
   - Make sure that all members of a structure are aligned.
   - The best rule for alignment is to align according to vector size.
   - For non-KNC instruction sets at least align to cache-line boundaries
   - This rule applies also for high performance scalar code, although it will not crash your app (at least not always...)
2) Keep track of how many vector registers you are using
   - If you have to re-load value from L0, there is a big chance it is not there anymore!!!
   - Too long instruction list will require you to manually re-use temporary registers
3) Remember about data dpendencies
   - Not all algorithms can be re-written!!!
   - Using vectors with length smaller than stride is a fair solution!!!
4) Be carefull of what operations ISA has to offer
   - When using intrinsics, remember there are some intrinsics which don't have representation in ISA
   - ISA extensions add new operations for already existing vector types. Re-compiling with flag for newer ISA could solve the problem
5) SIMD vectorization is expensive but can give significant speedup
   - Additional vector load/store and gather/scatter operations can be very expensive. Try doint only one load/store per data piece.
   - Vector extensions are not as mature as scalar extensions. Some operations are not supported and require additional instructions to execute.
6) You can only vectorize innermost loops
   - This is a intricacy of SIMD arithmetics. No approach on x86 is able to pass over that

Questions



Mini FAQ

Q: How can I check which ISA extensions my cpu is supporting?

A: You can check that

1) *Statically*
   1) *on n linux:*
      $ cat /proc/cpuinfo | grep flags
   2) *on windows:*
      On windows you have to use some tools, like this one:
      - *http://www.cpuid.com/softwares/cpu-z.html*

2) *Dynamically:*
   1) *you can just invoke CPUID instruction with EAX=07H and decode specific flags:*

2.7 CPUID INSTRUCTION

CPUID—CPU Identification

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F A2	CPUID	Valid	Valid	Returns processor identification and feature information to the EAX, EBX, ECX, and EDX registers, as determined by input entered in EAX (in some cases, ECX as well).

From: „Intel(R) Architecture Instruction Set Extensions Programming Reference"

Mini FAQ

Reading CPUID from software level:

```
#if defined (_MSC_VER)
#include <intrin.h>
#endif

void readCPUID(int *cpuidbuff, int function, int subleaf_index)
{
#if defined (_MSC_VER)
    __cpuidex(cpuidbuff, function, subleaf_index);
#elif defined (__GNUC__) || defined (__INTEL_COMPILER)
    int a, b, c, d;
    __asm("cpuid" : "=a"(a), "=b"(b), "=c"(c), "=d"(d) : "a"(function), "c"(subleaf_index) : );
    cpuidbuff[0] = a;
    cpuidbuff[1] = b;
    cpuidbuff[2] = c;
    cpuidbuff[3] = d;
#endif
}
```

Mini FAQ

Q: Can I use AVX2 instructions with SSEx registers?

A: General answer is NO. AVX extends instruction set for SSE, AVX2 adds 256b registers and interpretation of given instruction for such operand. Except of ***conversion*** *operations, no register type mixing is allowed.*

The main reason is different instruction encoding:

PMULDQ—Multiply Packed Doubleword Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 28 /r PMULDQ xmm1, xmm2/m128	RM	V/V	SSE4_1	Multiply packed signed doubleword integers in xmm1 by packed signed doubleword integers in xmm2/m128, and store the quadword results in xmm1.
VEX.NDS.128.66.0F38.WIG 28 /r VPMULDQ xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Multiply packed signed doubleword integers in xmm2 by packed signed doubleword integers in xmm3/m128, and store the quadword results in xmm1.
VEX.NDS.256.66.0F38.WIG 28 /r VPMULDQ ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Multiply packed signed doubleword integers in ymm2 by packed signed doubleword integers in ymm3/m256, and store the quadword results in ymm1.
EVEX.NDS.512.66.0F38.W1 28 /r VPMULDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Multiply packed signed doubleword integers in zmm2 by packed signed doubleword integers in zmm3/m512/m64bcst, and store the quadword results in zmm1 using writemask k1.

From: „Intel(R) Architecture Instruction Set Extensions Programming Reference"

```
__m128i __mm_mul_epi32 (__m128i a, __m128i b)
```

```
No need for such intrinsic in AVX!
```

```
__m256i __mm256_mul_epi32 (__m256i a, __m256i b)
```

```
__m512i __mm512_mul_epi32 (__m256i a, __m256i b)
```

Mini FAQ

Q: Can I mix KNCNI/IMCI/k1om with AVX512

A: NO. IMCI and AVX512 are completely different ISA! AVX512 builds on AVX2 and SSE2. IMCI operates only on 512b length vectors. IMCI does not provide any conversion operators to 256b vectors.

Opcode	Instruction	Description
MVEX.NDS.512.66.0F.W0 FE /r	vpaddd zmm1 {k1}, zmm2, S_{i32}(zmm3/m_t)	Add int32 vector zmm2 and int32 vector S_{i32}(zmm3/m_t) and store the result in zmm1, under write-mask.

From: „Intel(R) Xeon Phi(TM) coprocessor Instruction Set Architecture Reference Manual"

EVEX.NDS.512.66.0F.W0 FE /r VPADDD *zmm1 {k1}{z}, zmm2,* zmm3/m512/m32bcst	FV	V/V	AVX512F	Add packed doubleword integers from *zmm2, zmm3/m512/m32bcst* and store in *zmm1* using writemask k1.

From: „Intel(R) Architecture Instruction Set Extensions Programming Reference"

Mini FAQ

Q: I can't find `__m512i __mm512_add_epi64 (__m512i a, __m512i b)` ***on KNC…***

A: There is no such intrinsic because there is no relating instruction! I cannot really come up with better explanation than one in which there was not enough time for product delivery:

1) 64b scalar registers are present
2) 32b vector instructions are present
3) 64b floating point vectors are present
4) 512b bitwise operations are present for both float32 and float64

Mini FAQ

Q: I found that there is an instruction described in „Intel(R) ISA Reference Manual" called <put_instruction_name_here>, but I can't find the intrinsic in „Intel(R) Intrinsic Guide"

A: Some instructions support multiple encoding combination. Most of the operations that are supported (and make sense) are available as intrinsics. Mistakes in documentation happen, so some operations might not be present in the guide.

If you can't find an intrinsic in intel reference guide, try searching <immintrin.h> header (and subsequent headers for given ISA extension). You can also try guessing name of the intrinsic and its operands.

If the intrinsic is still not there, you can always implement your own version using inline assembly. If the instruction is described in ISA reference, this should definitely work.

If it doesn't – try searching on IDZ (www.software.intel.com/en-us/forum). This might be a known issue.

References

- https://software.intel.com/sites/landingpage/IntrinsicsGuide/ - list of vector intrinsics
- http://www.agner.org/optimize/instruction_tables.pdf - tables of instructions with latencies
- „Intel(R) 64 and IA-32 Architectures Software Developer's Manual"
- „Intel(R) Xeon Phi(TM) Coprocessor Instruction Set Architecture Reference Manual" (https://software.intel.com/sites/default/files/forum/278102/327364001en.pdf)
- „Intel(R) Xeon Phi(TM) Coprocessor System Software Developers Guide" (http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-system-software-developers-guide.html)
- S. W. Williams, A. Waterman, D. A. Patterson, „Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures" (http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-134.html)
- John L. Hennessy, D. A. Patterson, „Computer architecture: Quantitative approach, 5th edition", Elsevier ISBN: 978-0-12-383872-8, Waltham (USA) 2012

Appendix A: Useful gcc commands

```
1) Verify GCC version!!!
$ gcc –version
gcc (GCC) 4.8.2
Copyright (C) 2013 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

2) Compile with c++ 11 standard enabled:
$ g++ main.cpp –o main.out -std=c++11

3) Compile with optimizations turned ON/OFF
$ g++ main.cpp –o main.out -std=c++11 –O0  // Turn off optimizations
$ g++ main.cpp –o main.out -std=c++11 –O2  // Turn on optimizations

4) Compile with inlining turned ON/OFF
$ g++ main.cpp –o main.out -std=c++11 –fno-inline // Turn off function inlining
$ g++ main.cpp –o main.out -std=c++11 –finline \
                                      –finline-atomics \
                                      –finline-functions \
                                      –finline-functions-called-once // inline everything

5) Generate debug symbols:
$ g++ main.cpp –o main.out -std=c++11 –g

6) Dump assembly:
$ g++ main.cpp -std=c++11 –S –fverbose-asm -g
```