

Performance Optimization CV Winter Camp 2020: SIMD/Autovectorization

Kolpakova Marina, Mikhail Novozhilov

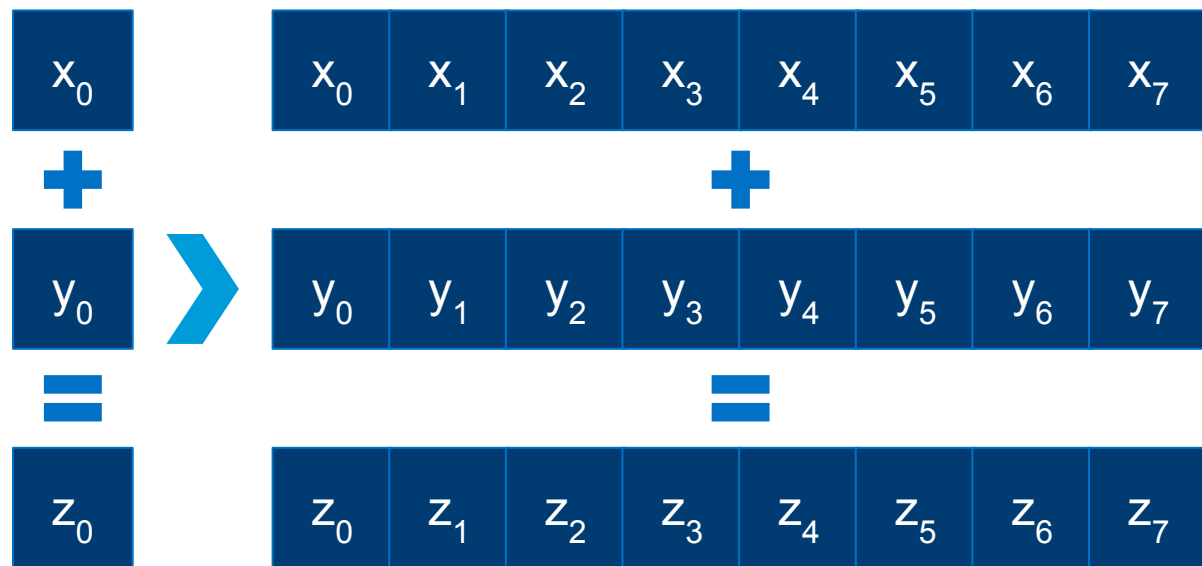
Internet of Things Group

SIMD/Autovectorization

Introduction

Single Instruction Multiple Data

SIMD means processing $W_{\text{SIMD}}/\text{sizeof}(\text{element})$ **more** data elements per clock cycle at the **same** time and power budget!



Historical perspective - vector processors

Vector Processors is a father of modern SIMD pipes

- Process the data in vectors (also called **chains**)
- Each element in a vector (called **lane**) is independent on any other
- Introduced in 1976, dominated for HPC in the 1980s
- Designed to expose DLP for increasing instruction throughput
- Features **deep pipelines**, **wide execution units**, not necessary the same width (**batch** length) as size of vector in elements (**vector** length)
- Is most efficient for simple memory patterns, but gather/scatter is possible too
- Needs **wide memory interfaces** to saturate execution units
- Has **large vector register** file while cache is not a strict requirement
- Example is Cray

Historical perspective - lessons learned

- They aren't used in generic processors design
 - Not efficient for generic latency-limited workloads
 - Not every workload can be formulated in terms of vectors
 - Deep pipelines comes with huge penalty for misprediction
- Precursors of most designs of first GPUs
- Vector pipes with short vector length (8-16 bytes) called SIMD pipes are widely integrated in modern CPU to accelerate most demanding loops
 - IA - SSE (128 bit), AVX (256 bit)
 - ARM - NEON (64, 128 bit), ASIMD (64, 128 bit)

Objectives to SIMD

1. **Processes more elements per clock**
 - Helps with compute bound kernels
2. **Less LD/ST instructions and requests to memory subsystem (caches)**
 - Helps if kernel is bound by number of generated transactions/ store queue
3. **Helps with instructions that aren't present in scalar ISA**
 - Eg: min/max for IA, Int->float conversion for ARM-v7
4. **Makes use of short data types efficient**
 - Less bytes the data needs, more speedup from SIMD is gained

SIMD won't benefit much

- Ineffective data walk, cache conflicts
 - Eg: resize area to scale more than 4-6
- Memory bound kernel
 - Eg: binary threshold from quiz
- Too much overhead instructions on scalarization
 - Eg: version 2 of binary threshold from quiz
- IA specifics: Too much overhead from use AVX 256 bit
 - Comes with need of full width barrel shifting or permutations

SIMD/Autovectorization

Autovectorization: why do we even care?

Why do we care?

```
void
threshold (Mat& src, Mat& dst, int threshold, uint8_t valmax) {
    int dcols = dst.cols;
    for (int row = 0; row < dst.rows; row++) {
        const uint8_t *psrc = src.ptr<uint8_t>(row);
        uint8_t *pdst = dst.ptr<uint8_t>(row);

        for (int col = 0; col < dcols; col++)
            pdst[col] = psrc[col] > threshold ? valmax : 0;
    }
}
```

Why do we care?

```
void
threshold (Mat& src, Mat& dst, int threshold, int valmax) {
    int dcols = dst.cols;
    for (int row = 0; row < dst.rows; row++) {
        const uint8_t *psrc = src.ptr<uint8_t>(row);
        uint8_t *pdst = dst.ptr<uint8_t>(row);

        for (int col = 0; col < dcols; col++)
            pdst[col] = psrc[col] > threshold ? valmax : 0;
    }
}
```

Why do we care?

```
void
threshold (Mat& src, Mat& dst, uint8_t threshold, uint8_t valmax) {
    int dcols = dst.cols;
    for (int row = 0; row < dst.rows; row++) {
        const uint8_t *psrc = src.ptr<uint8_t>(row);
        uint8_t *pdst = dst.ptr<uint8_t>(row);

        for (int col = 0; col < dcols; col++)
            pdst[col] = psrc[col] > threshold ? valmax : 0;
    }
}
```

Quiz: why results are different?

Time SKL-W for 2592 X 2048 8 bit

NAME	CALLS	TIME (ms/call)	WALLTIME (ms)
threshold v0	100	1.146	114.558
threshold v1	100	1.154	115.385
threshold v2	100	0.594	59.363

Compiled with GCC 5.4 with the same flags (-O3 -msse4.2 -std=c++11)

Version 1

Different types for threshold and max value => if-converted using **native register type** (long 64 bit) => no vector analogy => leave it scalar

.L870:

```
movzbl    (%r10,%rax), %r8d
cmpl      %edx, %r8d
movl      %ebx, %r8d
cmovg     %ecx, %r8d // conditional move if greater
movb      %r8b, (%r11,%rax)
addq      $1, %rax
cmpl      %eax, %r9d
jg        .L870
```

1.146 ms for SKL-W

Version 2

Threshold and max value is 32 bit integers => if-convert => decide to use 32 bit => find vector analogy => vectorize with promotion bytes to int

```
.L875:
    movq    -32(%rsp), %rdx
    addl    $1, -56(%rsp)
    movdqa  (%rdx,%rax), %xmm0
    movq    -24(%rsp), %rdx
    pmovzxbw %xmm0, %xmm6
    psrldq   $8, %xmm0
    pmovzxbw %xmm0, %xmm0
    pmovzxwd %xmm6, %xmm1
    psrldq   $8, %xmm6
    pmovzxwd %xmm6, %xmm6
    pcmpgtd  %xmm4, %xmm1
    pcmpgtd  %xmm4, %xmm6
    pand     %xmm3, %xmm1
    pand     %xmm3, %xmm6
    pand     %xmm2, %xmm1
    pand     %xmm2, %xmm6
    packusdw %xmm6, %xmm1
    pmovzxwd %xmm0, %xmm6
    psrldq   $8, %xmm0
    pmovzxwd %xmm0, %xmm0
    pcmpgtd  %xmm4, %xmm6
    pand     %xmm5, %xmm1
    pand     %xmm4, %xmm0
    pand     %xmm3, %xmm6
    pand     %xmm3, %xmm0
    pand     %xmm2, %xmm6
    pand     %xmm2, %xmm0
    packusdw %xmm0, %xmm6
    movdqa  %xmm1, %xmm0
    movdqa  %xmm6, %xmm1
    pand     %xmm5, %xmm1
    packuswb %xmm1, %xmm0
    movups  %xmm0, (%rdx,%rax)
    movl    -56(%rsp), %edx
    addq    $16, %rax
    cmpl    %ebx, %edx
    jb     .L875
```

1.154 ms for SKL-W, too much overhead to work

Version 3

Threshold and max value is unsigned 8 bit => if converted => can be computed with saturating subtraction => 0.594 ms for SKL-W

.L875:

```
movq    -24(%rsp), %rax
addl    $1, %r12d
movdqa  (%rax,%rbp), %xmm0
movq    -48(%rsp), %rax
psubusb %xmm1, %xmm0 // threshold > psrc[i] => threshold - psrc[i] > 0
pcmpeqb %xmm3, %xmm0 // saturate<uint8_t>(threshold-psrc[i]) == 0
pandn   %xmm2, %xmm0
movups  %xmm0, (%rax,%rbp)
addq    $16, %rbp
cmpl    %r11d, %r12d
jb      .L875
```

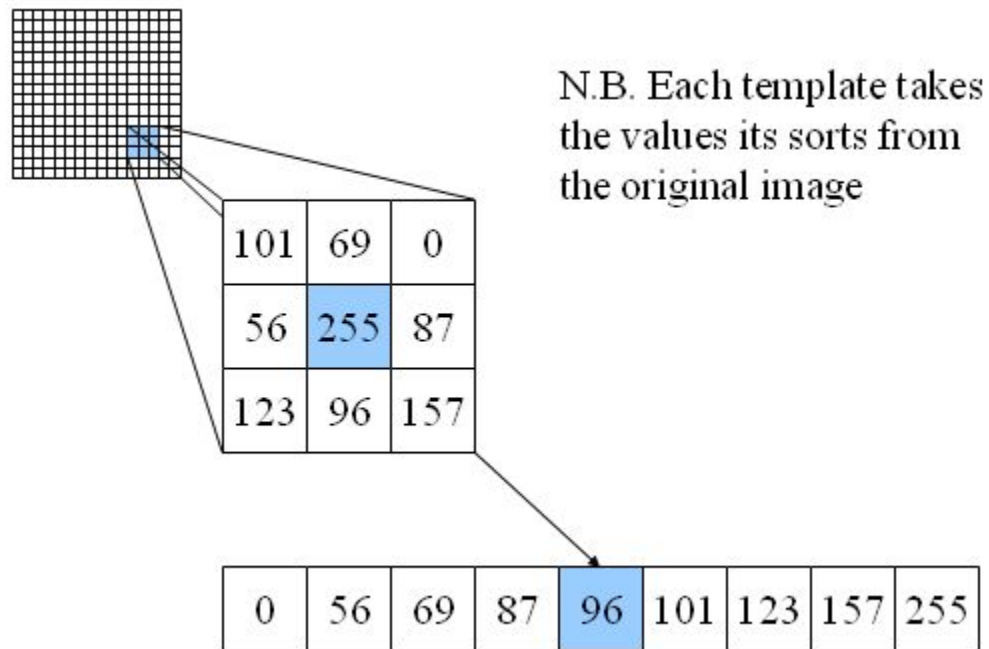
Detailed look

- Streaming kernel 8 bit => 8 bit, resolution 2592 × 2048 pixels
- $2592 \times 2048 \times 1_{\text{byte / pixel}} \times 2_{\text{input+output}} \Rightarrow \mathbf{10.125 \text{ MB}}$ problem size
- Memory bandwidth limit based on SKL-W achievable throughput (17.2 GB/s)
 - $1000 \text{ ms} / 17200 \text{ MB/s} * 10.1265 \text{ MB} \sim 0.588 \text{ ms}$
- Possible speedup from optimizations is $1.146 / 0.588 \text{ ms} \Rightarrow 1.95x$
 - Which is achieved for version 3

SIMD/Autovectorization

Don't overcomplicate

Median Blur



Vectorized with little help - median blur 3x3

- Use optimal sorting network from
“Fast median search: an ANSI C implementation” by Nicolas Devillard

// horizontal pass

```
psort(p1,p2); psort(p4,p5); psort(p7,p8);
```

```
psort(p0,p1); psort(p3,p4); psort(p6,p7);
```

```
psort(p1,p2); psort(p4,p5); psort(p7,p8);
```

// vertical pass

```
psort_max(p0,p3); psort_min(p5,p8);      psort(p4,p7);
```

```
psort_max(p3,p6); psort_max(p1,p4); psort_min(p2,p5);
```

```
psort_min(p4,p7);      psort(p4,p2); psort_max(p6,p4);
```

```
psort(p4, p2);
```

Vintage optimization techniques

Initially proposed version of psort, psort_min and psort_max

```
// Wow! tricky things to avoid comparisons.
static int min_(const int& x, const int& y) {
    return y + ((x - y) & ((x - y) >> (sizeof(int) * CHAR_BIT - 1)));
}
static int max_(const int& x, const int& y) {
    return x - ((x - y) & ((x - y) >> (sizeof(int) * CHAR_BIT - 1)));
}
#define psort(a, b)      { int tmp = a; a = min_(a,b); b = max_(tmp,b); }
#define psort_min(a, b) { a = min_(a,b); }
#define psort_max(a, b) { b = max_(a,b); }
```

Nice try!

- GCC 5.4 -O3 -msse4.2 auto-vectorized it with integer width
 - 801 instruction to produce 16 bytes of output
 - 2592×2048 image SKL-W time is **23.434** ms
- Should replace “obsolete optimization” with naive call to c++ standard library

```
#define psort(a, b)      { int tmp = a; a = std::min<uint8_t>(a,b); \
                        b = std::max<uint8_t>(tmp,b); }

#define psort_min(a, b) { a = std::min<uint8_t>(a,b); }
#define psort_max(a, b) { b = std::max<uint8_t>(a,b); }
```

- Auto-vectorized with 8 bit width
 - 56 instructions to produce 16 bytes of output => 14.3 times less than previous
 - 2592×2048 image SKL-W time is **1.749** ms, **speedup 13.4x**
 - 94% from instruction shrink ratio

So naive to be fast - 56 instructions

```
movdqu (%r11,%rax), %xmm10
movdqu (%r10,%rax), %xmm7
movdqa %xmm10, %xmm5
movdqu (%r12,%rax), %xmm1
pmaxub %xmm7, %xmm10
pminub %xmm7, %xmm5
movdqa %xmm5, %xmm11
movdqu 0(%rbp,%rax), %xmm3
movdqa %xmm1, %xmm7
movdqu (%r15,%rax), %xmm14
pminub %xmm3, %xmm7
pmaxub %xmm3, %xmm1
movdqa %xmm7, %xmm12
movdqu (%r14,%rax), %xmm0
movdqa %xmm14, %xmm3
movdqa (%rsi,%rax), %xmm4
pminub %xmm0, %xmm3
pmaxub %xmm0, %xmm14
movdqa %xmm3, %xmm15
movdqu (%rbx,%rax), %xmm6
pmaxub %xmm4, %xmm11
movdqa %xmm11, %xmm13
pminub %xmm5, %xmm4
movdqu 0(%r13,%rax), %xmm2
pmaxub %xmm6, %xmm12
pmaxub %xmm10, %xmm13
movdqa %xmm12, %xmm8
pminub %xmm11, %xmm10
pminub %xmm7, %xmm6
pmaxub %xmm2, %xmm15
movdqa %xmm15, %xmm9
movdqa %xmm15, %xmm0
pminub %xmm1, %xmm8
pmaxub %xmm12, %xmm1
pminub %xmm13, %xmm1
pminub %xmm14, %xmm9
pmaxub %xmm14, %xmm0
pminub %xmm1, %xmm0
movdqa %xmm9, %xmm1
pmaxub %xmm6, %xmm4
pminub %xmm3, %xmm2
pmaxub %xmm4, %xmm2
pminub %xmm8, %xmm1
pmaxub %xmm9, %xmm8
pmaxub %xmm10, %xmm1
pminub %xmm8, %xmm1
movdqa %xmm1, %xmm8
pminub %xmm0, %xmm8
pmaxub %xmm1, %xmm0
pmaxub %xmm8, %xmm2
pminub %xmm2, %xmm0
movups %xmm0, (%r8,%rax)
```

SIMD/Autovectorization

Vectorizer: how to use?

Vectorization-aware coding

- Evaluate compiler output
 - Does it really vectorize where you think it should?
- Know what makes vectorizable at all
 - “for” loops that meet certain constraints
- Evaluate execution performance and know where vectorization will help
 - Estimate limitations of your kernel
 - Compare to achievable throughput, instruction/byte ratio
- Know data access patterns to maximize efficiency
 - Improve memory access patterns and locality
 - Remove constructs that make vectorization impossible/impractical
 - Encourage/force vectorization when compiler doesn't, but should (by tweaks)

Basic requirements of vectorizable loops

- **Countable at runtime**
 - Number of loop iterations is known before loop executes
 - No conditional termination (break)
- Must be the **innermost** loop if nested
- **No function calls**
 - Pure Inlinable functions are allowed
 - Basic math is allowed if build-in is present: pow(), sqrt(), sin(), etc
- Have **single control flow**
 - 'if' statements are allowed when they can be scalarized / if-converted
 - No switch statements

Countable with no side effect - no function calls

- Not vectorizable

```
for (int i = 0; i < len; i++)  
    // non-hoistable loop invariant  
    y[i] = x[i] * exp( sqrt(len/2.));
```

- Vectorizable

```
double alpha = exp( sqrt(len/2.))  
for (int i = 0; i < len; i++)  
    y[i] = x[i] * alpha;
```

Countable with no side effect - no control flow

- Not vectorizable

```
for (int i = 0; i < len; i++) {  
    if (x[i] == 42) break;  
    y[i] = x[i];  
}
```

- The second is vectorizable

```
int m = len;  
for (int i = 0; i < len; i++)  
    if (x[i] == 42) {  
        m = i;  
        Break;  
    }  
for (int i = 0; i < m; i++)  
    y[i] = x[i];
```

Countable with no side effect - no control flow

.L5:

```
    addq    $4, %rax
    movl    (%rdi,%rax), %ecx
    cmpl    $42, %ecx
    je      .L1
```

.L4:

```
    cmpq    %rax, %rdx
    movl    %ecx, (%rsi,%rax)
    jne     .L5
```

.L18:

```
    addq    $4, %rcx
    cmpl    $42, -4(%rcx)
    je      .L17
```

.L16:

```
    addl    $1, %eax
    cmpl    %eax, %edx
    jne     .L18
    leal    -4(%rcx), %r9d
```

.L24:

```
    movdqa  0(%rbp,%rax), %xmm0
    addl    $1, %r11d
    movups  %xmm0, (%r10,%rax)
    addq    $16, %rax
    cmpl    %r9d, %r11d
    jnb     .L24
```

Obstacle: complex access pattern

- Indirect addressing

```
for (int i = 0; i < len; i++) {  
    b[a[i]] = x;  
}
```

- Strided pattern with unknown stride

```
for (int i = 0; i < len; i+=5) {  
    b[i] = x;  
}
```

Autovectorizer's trajectory

1. Performs scalar loop optimizations
 - Inlining, scalarization, if-conversion, hoisting, etc
2. Performs data range checks
 - Estimates width of used data types
3. Unrolls by vector size
 - Eg: 8x16 bit for 128 bit SSE
4. Looks for vectorization opportunities
 - Checks for possible aliasing
 - Checks for possible dependencies (flow and anti)
5. Estimates vectorization overhead and decides about profitability
 - Apply cost model
6. Finally, vectorizes, if it's profitable

Obstacle: saturating arithmetic

Accumulate square (from OpenVX) with saturation

$$\text{acc}_{x,y} = \text{saturate_cast}<\text{int16_t}>((\text{uint16_t})\text{acc} + (\text{uint16_t})(\text{src}_{y,x} \times \text{src}_{y,x}) \gg \text{shift})$$

```
static void accumulateSqr(const cv::Mat& src, cv::Mat& dst, int shift) {  
    for (int32_t row = 0; row < src.rows; row++) {  
        const uint8_t* psrc = src.ptr<uint8_t>(row);  
        const uint8_t* psrc_end = psrc + src.cols;  
        uint16_t* pdst = dst.ptr<uint16_t>(row);  
  
        for (; psrc < psrc_end; psrc++, pdst++)  
            *pdst = std::min<uint32_t>(SHRT_MAX, *pdst + ((*psrc * *psrc) >> shift));  
    }  
}
```

Obstacle: saturating arithmetic

- Full HD image $1920 \times 1080 \times (2+1) = 6$ MB problem size
 - 0.3125 to be throughput bound
- Compile with ICC17 -march=core-avx2
 - 0.51 ms
- Vectorizer report detected possible psrc, pdst aliasing
 - different types => no strict aliasing rule

remark #25228: Loop multiversioned for Data Dependence

remark #15300: LOOP WAS VECTORIZED

- Autovectorizer generated code promoting to 32-bit, then it performs arithmetic

Accumulate square - compiler version

```
vmovdqu    (%rdi,%r14), %ymm11
vmovdqu    (%r12,%rdi,2), %ymm6
vmovdqu    32(%r12,%rdi,2), %ymm7
vpmovzxbw  %xmm11, %ymm13
vpmullw    %ymm13, %ymm13, %ymm14
vpmulhw    %ymm13, %ymm13, %ymm15
vextracti128 $1, %ymm11, %xmm12
vpunpcklwd %ymm15, %ymm14, %ymm5
vpunpckhwd %ymm15, %ymm14, %ymm3
vpmovzxbw  %xmm12, %ymm0
vpmullw    %ymm0, %ymm0, %ymm2
vpmulhw    %ymm0, %ymm0, %ymm1
vperm2i128 $32, %ymm3, %ymm5, %ymm11
vperm2i128 $49, %ymm3, %ymm5, %ymm3
```

```
vpsrad     %xmm10, %ymm11, %ymm13
vpsrad     %xmm10, %ymm3, %ymm5
vpunpcklwd %ymm1, %ymm2, %ymm4
vpunpckhwd %ymm1, %ymm2, %ymm2
vextracti128 $1, %ymm6, %xmm0
vpmovzxwd  %xmm6, %ymm12
vpmovzxwd  %xmm0, %ymm6
vpadd      %ymm13, %ymm12, %ymm14
vpadd      %ymm5, %ymm6, %ymm0
vpmnud     %ymm9, %ymm14, %ymm15
vpmnud     %ymm9, %ymm0, %ymm3
vpand      %ymm8, %ymm15, %ymm1
vpand      %ymm8, %ymm3, %ymm5
// and the same number instructions
// to pack data back
```

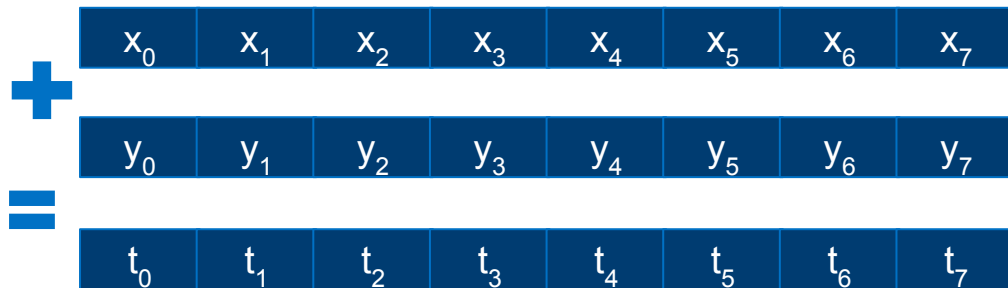
Obstacle: saturating arithmetic

- Kernel is still compute bound
- To force ICC generate 16 bit math, saturating addition should be used since
 - $(\text{uint16_t})\text{acc} \subset \{0, 65536\}$
 - $(\text{uint16_t})(\text{src}_{y,x} \times \text{src}_{y,x}) \gg \text{shift} \subset \{0, 65536\}, \text{shift} \geq 0$
 - $\text{Sum} \subset \{0, 2 \times 65536\}$, which is $\geq \text{SHRT_MAX} = 32768$
 - Formula can be rewritten as

$\text{acc}_{x,y} = \text{saturate_cast} < \text{int16_t} > (\text{adds}((\text{uint16_t})\text{acc}, (\text{uint16_t})(\text{src}_{y,x} \times \text{src}_{y,x}) \gg \text{shift}))$

- Compiler unable to generate this since saturating math is not present in c++
- Should vectorize manually using this formula

Obstacle: saturating arithmetic



$$z_i = (t_i < \text{USHRT_MAX}) ? (0 < t_i) ? t : 0 : \text{USHRT_MAX}$$



Obstacle: saturating arithmetic

- Time SKL-W for full HD 0.32 ms
 - Kernel is very close to memory limit
 - AVX vectorization wouldn't give any improvement
- Compiler vectorizes code as is
 - hardly recognizes constructions that aren't present in programming language
 - even though they present in vector ISA (Eg. saturating arithmetics)
 - While FMA is usually supported by compilers

Obstacle: loop dependencies

- Vectorization changes the order of computation compared to sequential case
- Compiler must be able to prove that vectorization will produce correct result
- Need to consider independence of unrolled loop operations
 - depends on vector width
- Compiler performs dependency analysis
 - forward dependence write after read (WAR)=> vectorizable

```
for (int i=1; i<len; i++)  
    a[i] = a[i+1] + b[i];
```
 - Backward dependence read after write (RAW) => not vectorizable

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

Obstacle: loop dependencies - explanation WAR

`a = {0,1,2,3,4}, b= {5,6,7,8,9}`

Scalar application

```
a[0]= a[1]+b[0]→a[0] = 1+5→a[0] = 6
a[1]= a[2]+b[1]→a[1] = 2+6→a[1] = 8
a[2]= a[3]+b[2]→a[2] = 3+7→a[2] = 10
a[3]= a[4]+b[3]→a[3] = 4+8→a[3] = 12
```

`a = {6, 8, 10, 12 , 4}`

`a = {0,1,2,3,4}, b= {5,6,7,8,9}`

Vector application for `i={0,1,2,3}`

```
a[i+1] = load{1,2,3,4}
b[i]    = load{5,6,7,8}
{1,2,3,4}+{5,6,7,8} = {6,8,10,12}
a[i] = store{6, 8, 10, 12}
```

`a = {6, 8, 10, 12, 4}`

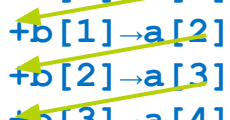
VECTORIZABLE!

Obstacle: loop dependencies - explanation RAW

`a = {0,1,2,3,4}, b= {5,6,7,8,9}`

Scalar application

`a[1]=a[0]+b[0]→a[1] = 0+6→a[1] = 6`
`a[2]=a[1]+b[1]→a[2] = 6+7→a[2] = 13`
`a[3]=a[2]+b[2]→a[3] =13+8→a[3] = 21`
`a[4]=a[3]+b[3]→a[4] =21+9→a[4] = 30`



`a = {0, 6, 13, 21 , 30}`

`a = {0,1,2,3,4}, b= {5,6,7,8,9}`

Vector application for `i={1,2,3,4}`

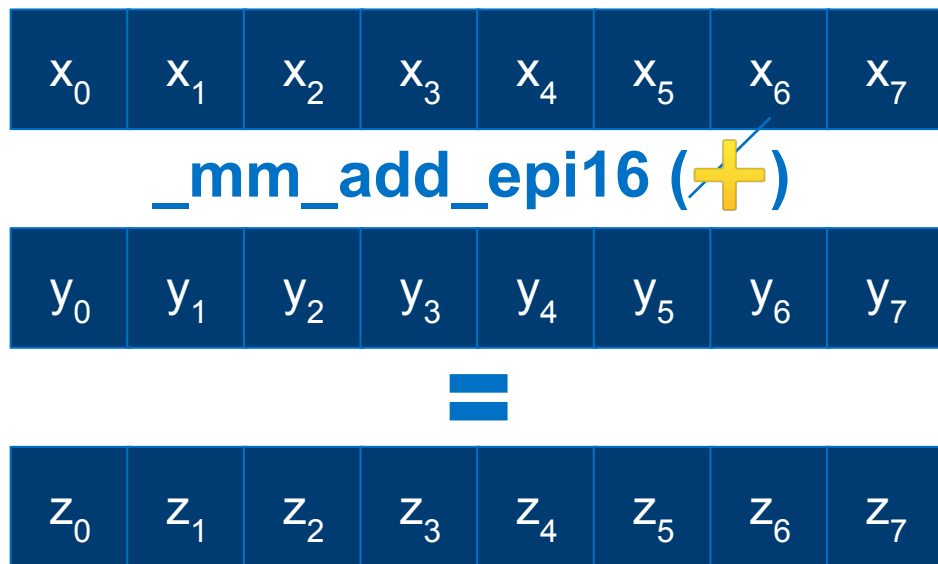
`a[i-1] = load{0,1,2,3}`
`b[i] = load{5,6,7,8}`
`{0,1,2,3}+{6,7,8,9} = {6,8,10,12}`
`a[i] = store{6, 8, 10, 12}`

`a = {0, 6, 8, 10, 12}`

NON VECTORIZABLE!

Intro: Intrinsic functions

- **Intrinsics** are functions that likely to be mapped into **one assembly instruction**
- Provide an opportunity to fully **control optimization** (select instructions, use more complicated vectorization schemes)
- **Register allocation** is still a **compiler** responsibility



SSE/AVX intrinsics notation

- No strict typization (exact types are deduced from instructions)
 - Special types to hold single (`__m{128,256}`) and double (`__m{128,256}d`) floating point numbers and integers (`__m{128,256}i`)
 - Cast between type is not an operation and free (Eg: `_mm_castps_si128` fp->int)
- Not fully orthogonal ISA
 - created on customer demand
 - Eg: No unsigned comparison instructions
 - Eg: `_mm_maddubs_epi16` is for 8 bit input only, while `_mm_mullohi` for 16 bit
- Usual intrinsic naming is `_mm<256>_<h><name><hi,lo,r,s>_<data_type>`
- Macros `_MM_SHUFFLE` - generates permute masks (imm8)
- Little-endian

Intrinsic latency and throughput

- For auto vectorization programmer only indirectly impacts instruction selection
 - Tune code, choose data types, help if-conversion
- Manual vectorization allows to select instructions
 - Not directly mapped on operation supported by HL language
 - Eg: saturating arithmetic, integer multiply and select high-half, etc
 - Choose less latent instruction with highest throughput
 - Eg: `_mm256_blend_epi32` ($\frac{1}{3}$) vs `_mm256_blendv_epi8` (1)

`_mm256_blend_epi32` vs `_mm256_blendv_epi8`

// one bit of immediate is a selector

```
_mm256_blend_epi32(a, b, 0b01010101);
```

// 4 bytes of vector constant is a selector

```
_mm256_blendv_epi8(a, b, _mm256_set_epi8(  
    0,0,0,0, -1,-1,-1,-1, 0,0,0,0, -1,-1,-1,-1,  
    0,0,0,0, -1,-1,-1,-1, 0,0,0,0, -1,-1,-1,-1,  
    0,0,0,0, -1,-1,-1,-1, 0,0,0,0, -1,-1,-1,-1,  
    0,0,0,0, -1,-1,-1,-1, 0,0,0,0, -1,-1,-1,-1)) );
```

Instruction timings

Look into instruction timings before choosing right instruction

- The Intel Intrinsics Guide
 - Interactive page with filtering by generation and timings
 - No timing for intrinsic usually mean that it decomposes into more than one hardware instructions or involves memory
- Agner Fog's instruction tables
 - Small (relative to IA spec) pdf with most useful information
- Manually run mubench
 - If you target widely unavailable architecture

