HPI Hasso Plattner Institut
Digital Engineering · Universität Potsdam

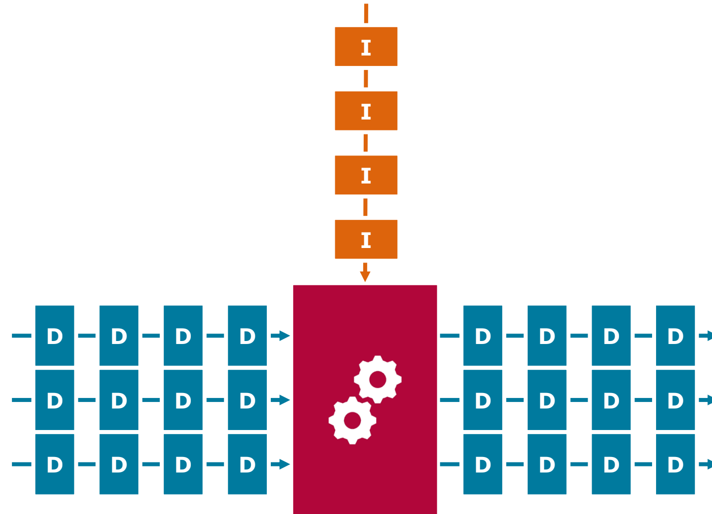
Parallel Programming and Heterogeneous Computing

SIMD: Integrated Accelerators

Max Plauth, *Sven Köhler*, Felix Eberhardt, Lukas Wenzel, and Andreas Polze
Operating Systems and Middleware Group

HPI Hasso Plattner Institut

1 SIMD & AltiVec



Definition SIMD

SIMD ::= **S**ingle **I**nstruction **M**ultiple **D**ata

The same instruction is performed simultaneously on multiple data points (fit for data-level parallelism).

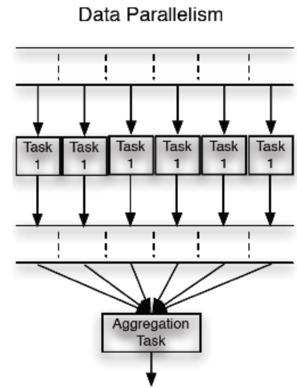
First proposed for ILLIAC IV, University of Illinois (1966).

Today many architectures provide SIMD instruction set extensions.

Intel: MMX, SSE, AVX

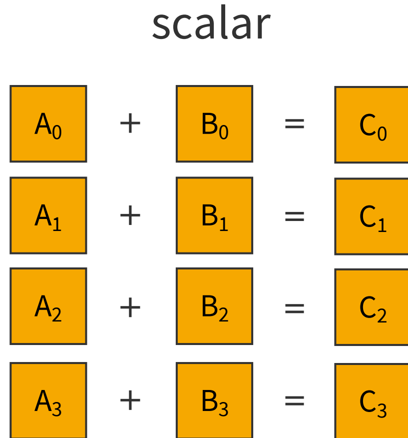
ARM: VPF, NEON, SVE

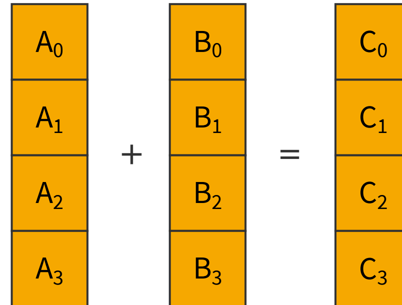
POWER: AltiVec (VMX), VSX



Scalar vs. SIMD

How many instructions are needed to add four numbers from memory?

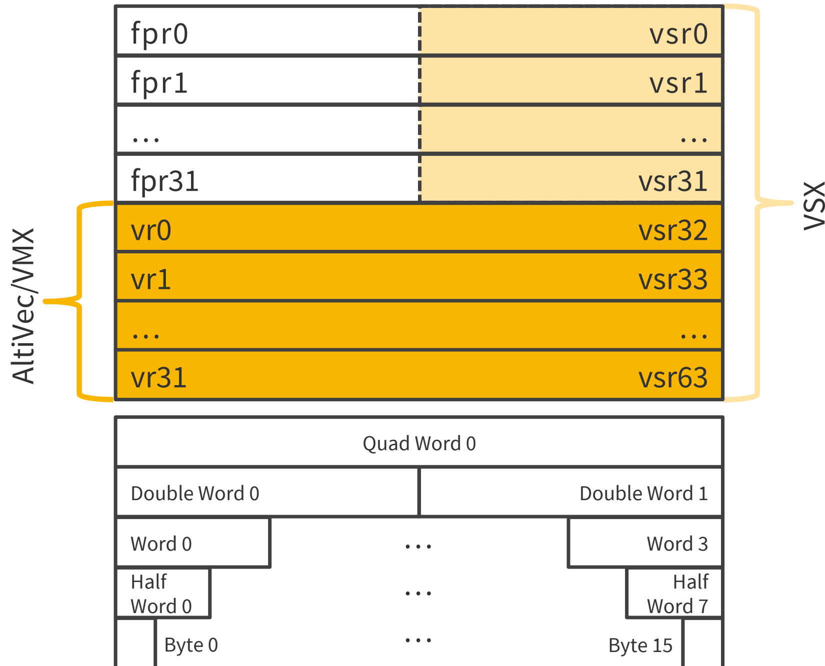




scalar	4 element SIMD
4 additions	1 addition
8 loads	2 loads
4 stores	1 store

Vector Registers on POWER8 (1)

32 vector registers containing 128 bits each.



These are also used by several **coprocessors**:

VSX SHA2 AES …

Vector Registers on POWER8 (2)

32 vector registers containing 128 bits each.

Depending on the instruction they are interpreted as

	16	(un)signed bytes
	8	(un)signed shorts
	4	(un)signed integers of 32bit
	4	single precision floats
	2	(un)signed long integers of 64bit
	2	double precision floats
or	2, 4, 8, 16	logic values

AltiVec Instruction Reference

For all instructions, registers and usage see

PowerISA 2.07(B), chapter 6 & 7

6.7.2 Vector Load Instructions

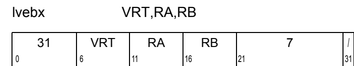
The aligned byte, halfword, word, or quadword in storage addressed by EA is loaded into register VRT.

Programming Note

The *Load Vector Element* instructions load the specified element into the same location in the target register as the location into which it would be loaded using the *Load Vector* instruction.

Load Vector Element Byte Indexed X-form

lvebx VRT,RA,RB



31	VRT	RA	RB	7	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + (RB)
eb  ← EA60:63

VRT ← undefined
if Big-Endian byte ordering then
   VRT8×eb:8×eb+7 ← MEM(EA,1)
else
   VRT120-(8×eb):127-(8×eb) ← MEM(EA,1)
```

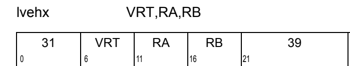
Let the effective address (EA) be the sum (RA|0)+(RB).

Let eb be bits 60:63 of EA.

If Big-Endian byte ordering is used for the storage access, the contents of the byte in storage at address EA are placed into byte eb of register VRT. The remaining bytes in register VRT are set to undefined values.

Load Vector Element Halfword Indexed X-form

lvehx VRT,RA,RB



31	VRT	RA	RB	39	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← (b + (RB)) & 0xFFFF_FFFF_FFFF_FFFE
eb ← EA60:63

VRT ← undefined
if Big-Endian byte ordering then
   VRT8×eb:8×eb+15 ← MEM(EA,2)
else
   VRT112-(8×eb):127-(8×eb) ← MEM(EA,2)
```

Let the effective address (EA) be the result of ANDing 0xFFFF_FFFF_FFFF_FFFE with the sum (RA|0)+(RB).

Let eb be bits 60:63 of EA.

If Big-Endian byte ordering is used for the storage access,

2 C-Interface

```
#include <altivec.h>

gcc -maltivec -mabi=altivec
gcc -mvsx

xlc -qaltivec -qarch=auto
```


HPI Hasso Plattner Institut

Vector Data Types

The C-Interface introduces new keywords and data types:

`vector unsigned char`
`vector signed char`
`vector bool char`
	16x 1 byte	`vector unsigned long`
		`vector signed long`
		`vector double`
			2 x 8 bytes

`vector unsigned short`
`vector signed short`
`vector bool short`
`vector pixel`
	8x 2 bytes		

`vector unsigned int`
`vector signed int`
`vector bool int`
`vector float`
	4x 4 bytes		

`gcc -maltivec`

`gcc -mvsx`

Vector Data Types Initialization, Loading and Storing

```
vector int va = {1, 2, 3, 4};

int data[] = {1, 2, 3, 4, 5, 6, 7, 8};
vector int vb = *((vector int *)data);

int output[4];
*((vector int *)output) = va;

printf("vb = {%d, %d, %d, %d};\n",
       vb[0], vb[1], vb[2], vb[3]);
```

Can be very slow!

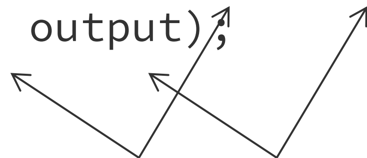
Aligned Addresses

Historically memory addresses required be **aligned at 16 byte** boundaries for efficiency reasons. (Although POWER8 has improved unaligned load/store and modern compilers will support you.)

```
int data[] __attribute__((aligned(16))) = {1, 2, 3, 4,
5, 6, 7, 8};
int *output = aligned_alloc(16, NUM * sizeof(int));
```

(compiler specific)

```
vector int va = vec_ld(0, data);
vec_st(va, 0, output);
```

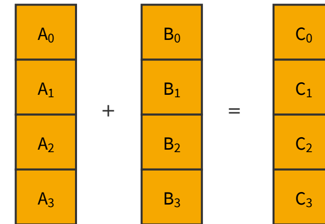


index + address (truncated to 16)

Vector Intrinsics

Operations are available through a rich set[1] of “overloaded functions” (actually intrinsics):

```
vector int va = {4, 3, 2, 1};
vector int vb = {1, 2, 3, 4};
vector int vc = vec_add(va, vb);

vector float vfa = {4, 3, 2, 1};
vector float vfb = {1, 2, 3, 4};
vector float vfc = vec_add(vfa, vfb);
```



[1]https://gcc.gnu.org/onlinedocs/gcc-8.4.0/gcc/PowerPC-AltiVec_002fVSX-Built-in-Functions.html

Vector Intrinsics: Lots of overloads

```
vector signed char vec_add (vector bool char, vector signed char);
vector signed char vec_add (vector signed char, vector bool char);
vector signed char vec_add (vector signed char, vector signed char);
vector unsigned char vec_add (vector bool char, vector unsigned char);
vector unsigned char vec_add (vector unsigned char, vector bool char);
vector unsigned char vec_add (vector unsigned char, vector unsigned char);
vector signed short vec_add (vector bool short, vector signed short);
vector signed short vec_add (vector signed short, vector bool short);
vector signed short vec_add (vector signed short, vector signed short);
vector unsigned short vec_add (vector bool short, vector unsigned short);
vector unsigned short vec_add (vector unsigned short, vector bool short);
vector unsigned short vec_add (vector unsigned short, vector unsigned short);
vector signed int vec_add (vector bool int, vector signed int);
vector signed int vec_add (vector signed int, vector bool int);
vector signed int vec_add (vector signed int, vector signed int);
vector unsigned int vec_add (vector bool int, vector unsigned int);
vector unsigned int vec_add (vector unsigned int, vector bool int);
vector unsigned int vec_add (vector unsigned int, vector unsigned int);
vector float vec_add (vector float, vector float);
vector double vec_add (vector double, vector double);
vector long long vec_add (vector long long, vector long long);
vector unsigned long long vec_add (vector unsigned long long, vector unsigned long long);
```

Attention: **No implicit** conversion!

Also not all types for every operation.

tps://gcc.gnu.org/onlinedocs/gcc-8.4.0/gcc/PowerPC-AltiVec_002fVSX-Built-in-Functions.html

Get Help: Programming Interface Manual

Highly helpful resource:

- Name of operation
- Pseudocode description
- Text description
- Graphical description
- Type table and according assembly instruction

Generic and Specific AltiVec Operations

vec_add

vec_add

Vector Add

d = vec_add(**a**,**b**)

- Integer add:

```
n ← number of elements
do i=0 to n-1
d_i ← a_i + b_i
end
```

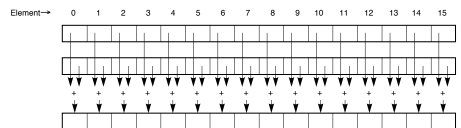
- Floating-point add:

```
do i=0 to 3
d_i ← a_i +_fp b_i
end
```

Each element of a is added to the corresponding element of b. Each sum is placed in the corresponding element of d.

For `vector float` argument types, if VSCR[NJ] = 1, every denormalized operand element is truncated to a 0 of the same sign before the operation is carried out, and each denormalized result element is truncated to a 0 of the same sign.

The valid combinations of argument types and the corresponding result types for `d = vec_add(a,b)` are shown in Figure 4-12, Figure 4-13, Figure 4-14, and Figure 4-15.



d	a	b	maps to
vector unsigned char	vector unsigned char	vector unsigned char	vaddubm d,a,b
	vector unsigned char	vector bool char	
	vector bool char	vector unsigned char	
	vector signed char	vector signed char	

http://www.nxp.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf

Get Help: IBM Knowledge Center

IBM has an online documentation of the extended standard,

not fully implemented by GCC.

IBM Knowledge Center – vec_add

www.ibm.com/support/knowledgecenter/SSGH2K_12.1.0/com.i

IBM Knowledge Center – vec_add

IBM

Marketplace

IBM Knowledge Center

Search Content Products

XL C for AIX > XL C for AIX 12.1.0 > XL C for AIX, V12.1 > Compiler Reference > Compiler built-in functions > Vector built-in functions > vec_add

vec_add Version 12.1.0

Purpose

Returns a vector containing the sums of each set of corresponding elements of the given vectors.

This function emulates the operation on long long vectors.

Syntax

```
d=vec_add(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector float	vector float	vector float
vector double	vector double	vector double

Result value

The value of each element of the result is the sum of the corresponding elements of a and b. For integer vectors and unsigned vectors, the arithmetic is modular.

Parent topic: Vector built-in functions

[Provide feedback]

Contact Us

Some Example Instructions Working on Elements

`vec_add(a, b)`	Add a and b element-wise
`vec_sub(a, b)`	Subtract a and b element-wise
`vec_mul(a, b)`	Multiply a and b element-wise (gcc: float only)
`vec_madd(a, b, c)`	Multiply a and b element-wise and add elements of c
`vec_min(a, b)`	Select element-wise the minimum of a and b
`vec_re(a)`	Compute reciprocals of elements
`vec_sqrt(a)`	Calculate square root of elements
`vec_sr(a, b)`	Right-shift elements of vector a depending on certain bits in b

Conversion of Floating-Point Types

`vec_ctf(a, n)` Divides the elements of integer vector a by 2^n and converts them into floating-point values.

`vec_ctu(a, n)` Multiplies the elements of floating-point vector a by 2^n and converts them into unsigned integers.

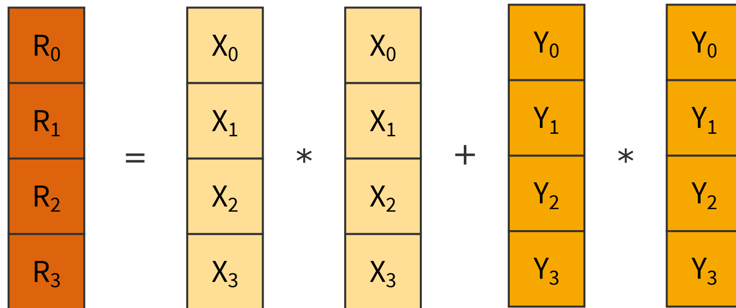
Idea behind this: **Fixed-point numbers** of n digits.

For just plain conversion use $n = 0$.

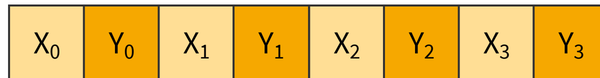
Vector Data Realignment and Permutation (1)

Sometimes memory is not correctly ordered for a certain tasks.

Example: Squared absolute of 2D points ($r^2 = p_x^2 + p_y^2$)

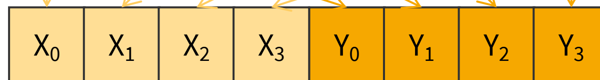


in memory:




```
struct point2d[];
```

in registers:

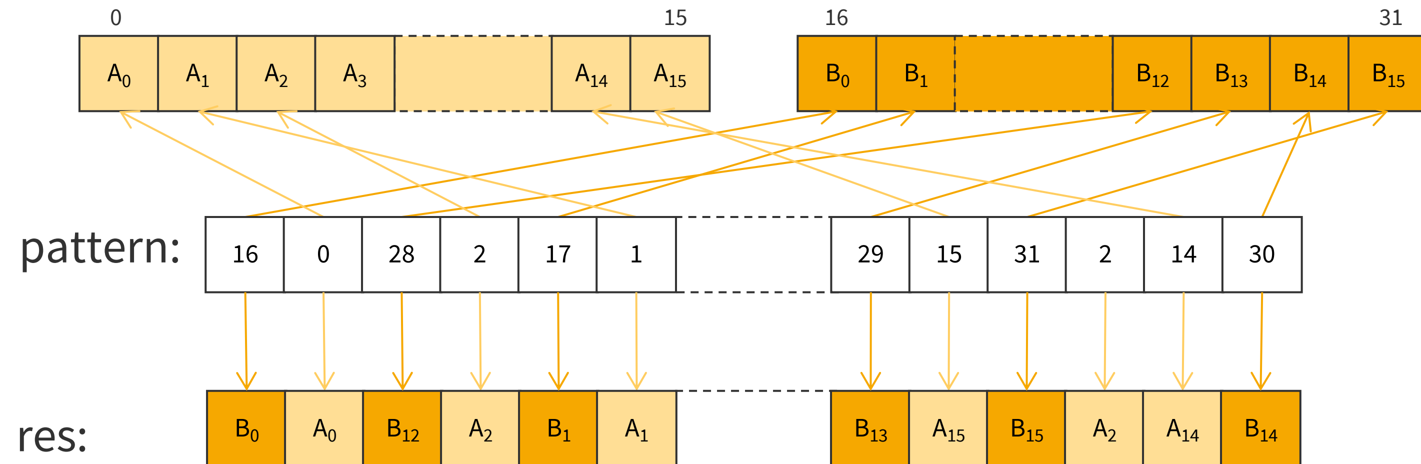


Vector Data Realignment and Permutation (2)

```
res = vec_perm(a, b, pattern)
```

Bytewise rearrange two vectors according to provided pattern.

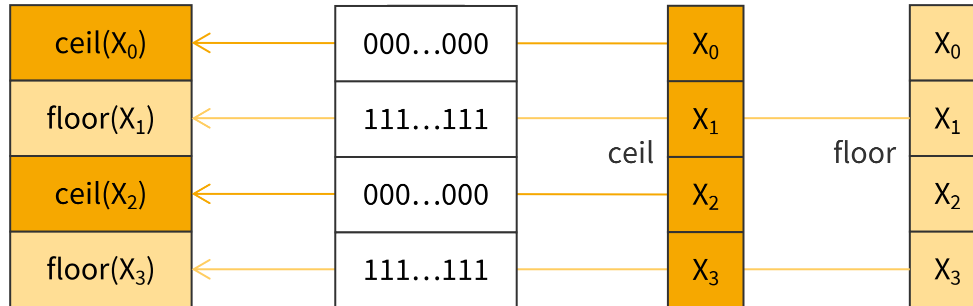
`pattern` denotes indices in assumed 32 byte array of concatenated `a` and `b`.



Vector Bit Selection (1)

Sometimes two vectors should be combined, but their bytes not moved.

Example: Every even element of a vector should be rounded up, and every odd one rounded down.

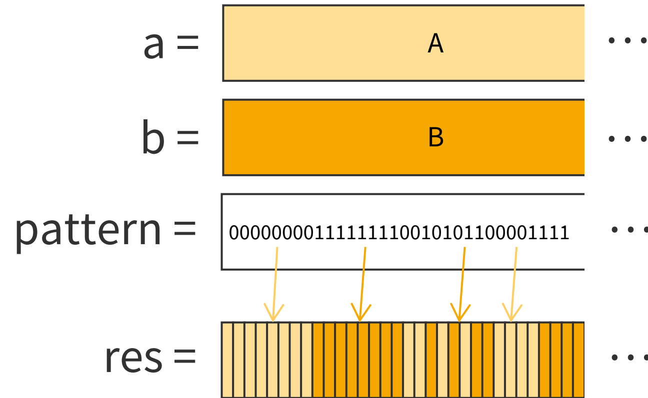



```
vector float a = vec_ceil(X);
vector float b = vec_floor(X);
vector unsigned int pattern = {0, 0xffffffff, 0, 0xffffffff};
vector float res = vec_sel(a, b, pattern);
```

Vector Bit Selection (2)

```
res = vec_sel(a, b, pattern)
```

Bit-wise pick contents from `a` or b, depending if corresponding bit in `pattern` is 0 or 1.

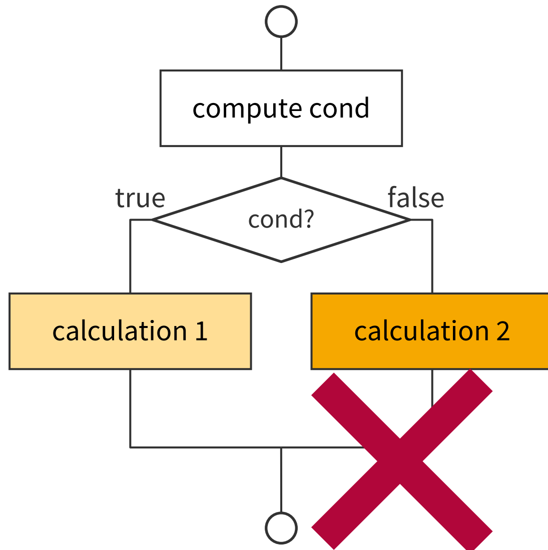


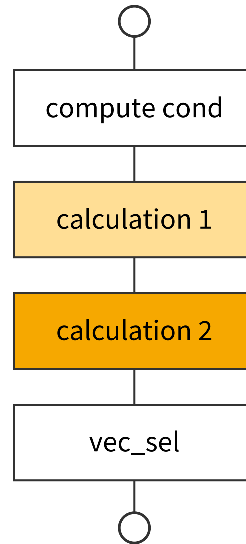
res[bit i] = a[bit i] if pattern[bit i] == 0 else b[bit i]

Conditional Programming (1)

There are **no branches** for element computation in AltiVec.

Instead compute both variants and then use **bit-wise select**.





Conditional Programming (2)

Remember the vector types?

```
vector unsigned char
vector signed char
vector bool char
```

16x false (= 0x0) or true (0xff)

```
vector unsigned short
vector signed short
vector bool short
vector pixel
```

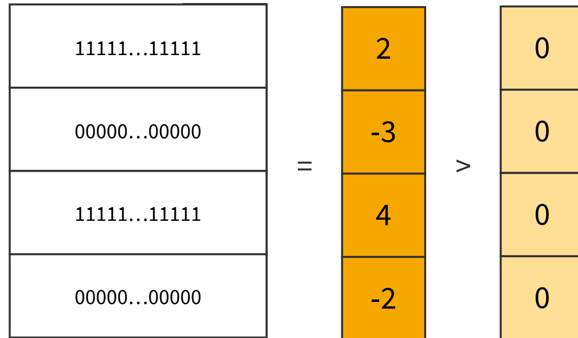
8x false (= 0x0) or true (0xffff)

```
vector unsigned int
vector signed int
vector bool int
vector float
```

4x false (= 0x0) or true (0xffffffff)

Conditional Programming (3)


```
vector bool int res = vec_cmpgt(a, b);
```



vec_cmpgt >
vec_cmpge >= (for gcc on **floats only**)
vec_cmpeq ==
vec_cmple <= (for gcc on **floats only**)
vec_cmplt <

```
vec_and    (a & b)
vec_or     (a | b)
vec_nand  ~(a & b)
vec_orc    (a | ~b)
...
```

Conditional Programming (4)

```
vector signed int calc_abs(vector signed int a)
{
    vector signed int vzero = {0, 0, 0, 0};
    vector signed int neg_a = vec_sub(vzero, a);
    vector bool int vpat = vec_cmpgt(vzero, a);

    return vec_sel(a, neg_a, vpat);
}
```

0 < a: false

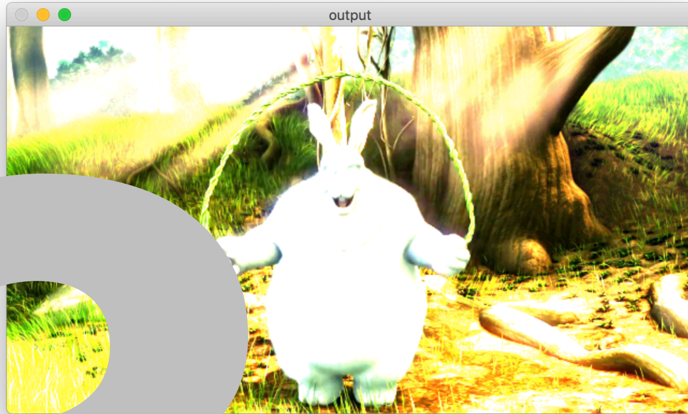
0 < a: true

Y U NO `vec_abs(a)`

ParProg20 C1
Integrated Accelerators

Sven Köhler

HPI Hasso Plattner Institut




```
void scale(float *input, int num,
           float scale)
{
    int i;
    for (i = 0; i < num; i++) {
        input[i] *= scale;
    }
}
```

3 Learning by example

Scale an Array by Factor (Vector)

```
void scale(float *input, int num, float scale)
{
    int i;
    vector float vscale = {scale, scale, scale, scale};
    for (i = 0; i < num; i += 4) {
        vector float *current = ((vector float *)&input[i]);
        *current = vec_mul(vscale, *current);
    }
}
```

<Do you see a problem?>

HPI Hasso Plattner Institut

Scale an Array by Factor (Vector, Safe)

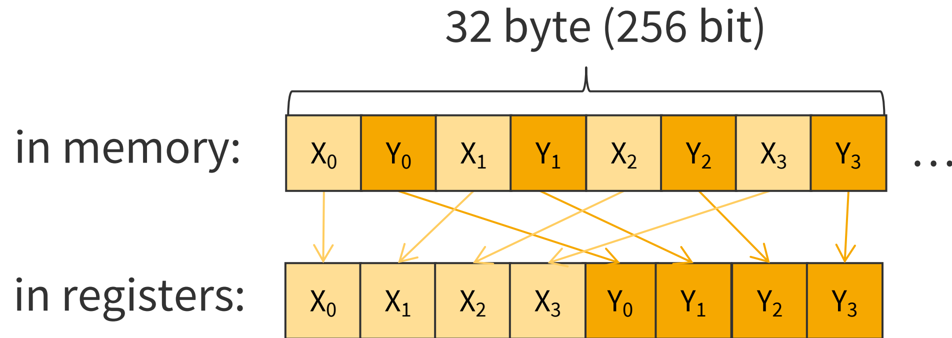
```
void scale(float *input, int num, float scale)
{
    int i;
    vector float vscale = {scale, scale, scale, scale};
    for (i = 0; i < num - 4; i += 4) {
        vector float *current = ((vector float *)&input[i]);
        *current = vec_mul(vscale, *current);
    }
    for (; i < num; i++) {
        input[i] = scale * input[i];
    }
}
```

Scale an Array by Factor (Vector, Safe, Alternative)

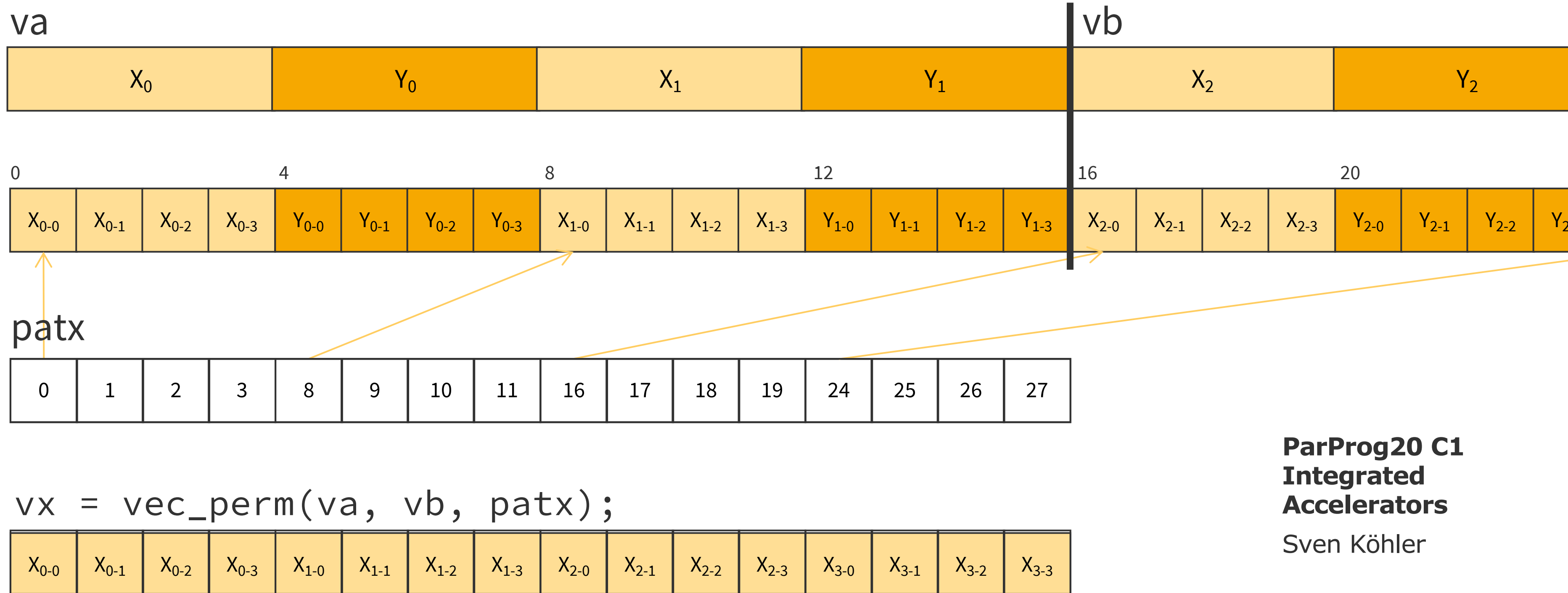
```
void scale(float *input, int num, float scale)
{
    int i;
    vector float vscale = {scale, scale, scale, scale};
    vector float *vinput = (vector float *)input;
    for (i = 0; i < num / 4; i++) {
        vinput[i] = vec_mul(vscale, vinput[i]);
    }
    for (i = (num / 4) * 4; i < num; i++) {
        input[i] = scale * input[i];
    }
}
```

Squared Absolute of Points (1)

```
struct point2d {
    float x, y;
};

void squared_2d_abs(struct point2d *input,
                             float *output, int num);
```

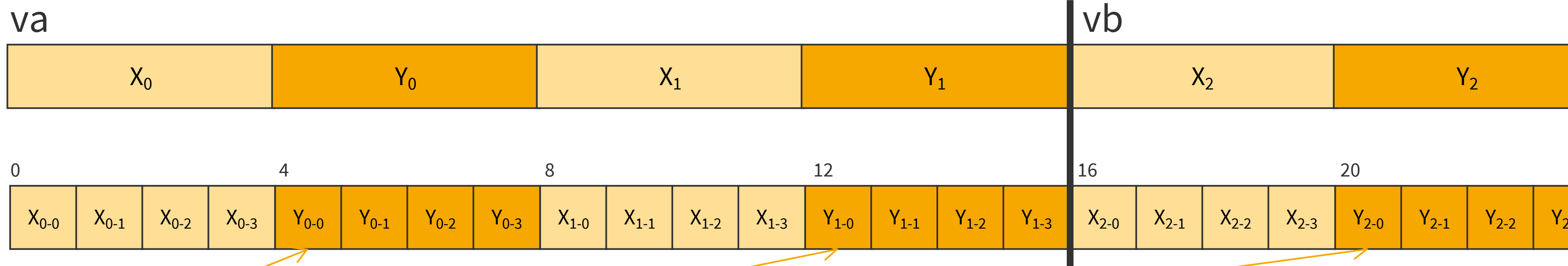


Squared Absolute of Points (2) – Permute Bytes to Get X



HPI Hasso Plattner Institut

Squared Absolute of Points (2) – Permute Bytes to Get Y



4	5	6	7	12	13	14	15	20	21	22	23	28	29	30	31

```
vy = vec_perm(va, vb, paty);
```

Y_{0-0}	Y_{0-1}	Y_{0-2}	Y_{0-3}	Y_{1-0}	Y_{1-1}	Y_{1-2}	Y_{1-3}	Y_{2-0}	Y_{2-1}	Y_{2-2}	Y_{2-3}	Y_{3-0}	Y_{3-1}	Y_{3-2}	Y_{3-3}

Squared Absolute of Points (4) – Patterns in C

```
vector unsigned char patx = {0x00, 0x01, 0x02, 0x03,
                             0x08, 0x09, 0x0a, 0x0b,
                             0x10, 0x11, 0x12, 0x13,
                             0x18, 0x19, 0x1a, 0x1b};

vector unsigned char paty = {0x04, 0x05, 0x06, 0x07,
                             0x0c, 0x0d, 0x0e, 0x0f,
                             0x14, 0x15, 0x16, 0x17,
                             0x1c, 0x1d, 0x1e, 0x1f};
```

<Any endianness issues here?>

Rule of thumb: No element size or storage for platform change

=> No endianness issues!

Squared Absolute of Points (5) – The Loop

```
int i;
vector float *vinput  = (vector float *)input;
vector float *voutput = (vector float *)output;

for (i = 0; i < num / 4; i++) {
    vector float va = vinput[2 * i];
    vector float vb = vinput[2 * i + 1];

    vector float vx = vec_perm(va, vb, patx);
    vector float vy = vec_perm(va, vb, paty);
    voutput[i] = vec_add(vec_mul(vx, vx), vec_mul(vy, vy));
}

for (i = 4 * (num / 4); i < num; i++) {
    output[i] =   input[i].x * input[i].x
                + input[i].y * input[i].y;
}
```

4

Short overview of SS[S]E[2,3,4]/AVX[-2,-512]

Hasso Plattner Institut

Vector registers on Intel architectures

AVX-512 register scheme as extension from the AVX (YMM0-YMM15) and SSE (XMM0-XMM15) registers

511 … 256	255 … 128	127 … 0
ZMM0	YMM0	XMM0
ZMM1	YMM1	XMM1
ZMM2	YMM2	XMM2
ZMM3	YMM3	XMM3
ZMM4	YMM4	XMM4
ZMM5	YMM5	XMM5
ZMM6	YMM6	XMM6
ZMM7	YMM7	XMM7
ZMM8	YMM8	XMM8
ZMM9	YMM9	XMM9
ZMM10	YMM10	XMM10
ZMM11	YMM11	XMM11
ZMM12	YMM12	XMM12
ZMM13	YMM13	XMM13
ZMM14	YMM14	XMM14
ZMM15	YMM15	XMM15
ZMM16	YMM16	XMM16
ZMM17	YMM17	XMM17
ZMM18	YMM18	XMM18
ZMM19	YMM19	XMM19
ZMM20	YMM20	XMM20
ZMM21	YMM21	XMM21
ZMM22	YMM22	XMM22
ZMM23	YMM23	XMM23

Overlapping register files for each ISA extension.
With AVX-512 extended to 32 registers.

New C data types:

`__m128`	4 floats
`__m128d`	2 doubles
`__m128i`	multiple (un)signed integers (8-128bit)
`__m256`	8 floats
`__m256d`	4 doubles
`__m256i`	multiple (un)signed integers (8-128bit)
`__m512`	…

Instructions typically use input registers as output:
mulps r0, r1 ::= r0 *= r1

Intrinsic function name patterns (ICC/GCC/MSVC)

Hasso Plattner Institut

```
#include <x86intrin.h> or #include <[version]mmintrin.h>
```

Dedicated intrinsic names for data types (mirrors instructions):

`ps` vectors contain floats (packed single-precision)

`pd` vectors contain doubles (packed double-precision)

`epi8/epi16/epi32/epi64`
vectors contain 8-bit/16-bit/32-bit/64-bit signed integers

`epu8/epu16/epu32/epu64`
vectors contain 8-bit/16-bit/32-bit/64-bit unsigned integers

`si128/si256`
unspecified 128-bit vector or 256-bit vector [e.g. loads]

`m128/m128i/m128d/m256/m256i/m256d`
identifies input vector types, when different from the type of the returned vector

Loading and Storing Memory

Memory loads require vector aligned addresses:

throws GP exception if unaligned

```
__m256 vec = _mm256_load_ps(data);
```

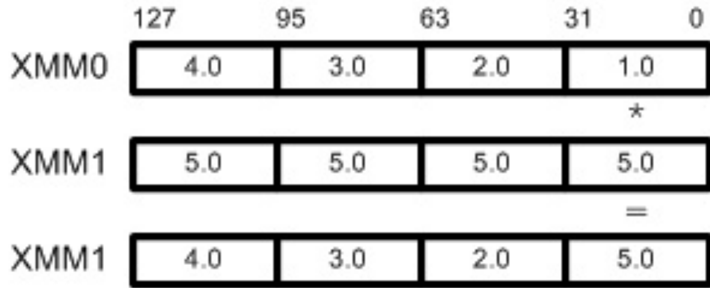
slower, but handles unaligned data

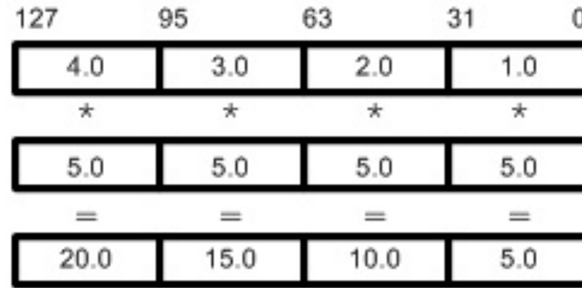
```
__m256 vec = _mm256_loadu_ps(data);
```

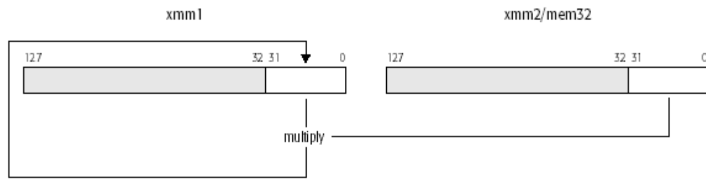
Values, again, can be cast too native pointers to be used for storing:

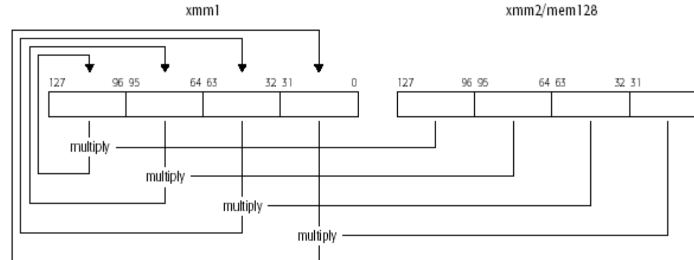
```
int *output = (int *)&vec;

__m256 *dst = (__m256 *)aligned_buffer;
dst[0] = vec;

_mm256_store[u]_ps(dst, vec);
```

Scalar operations in vector registers



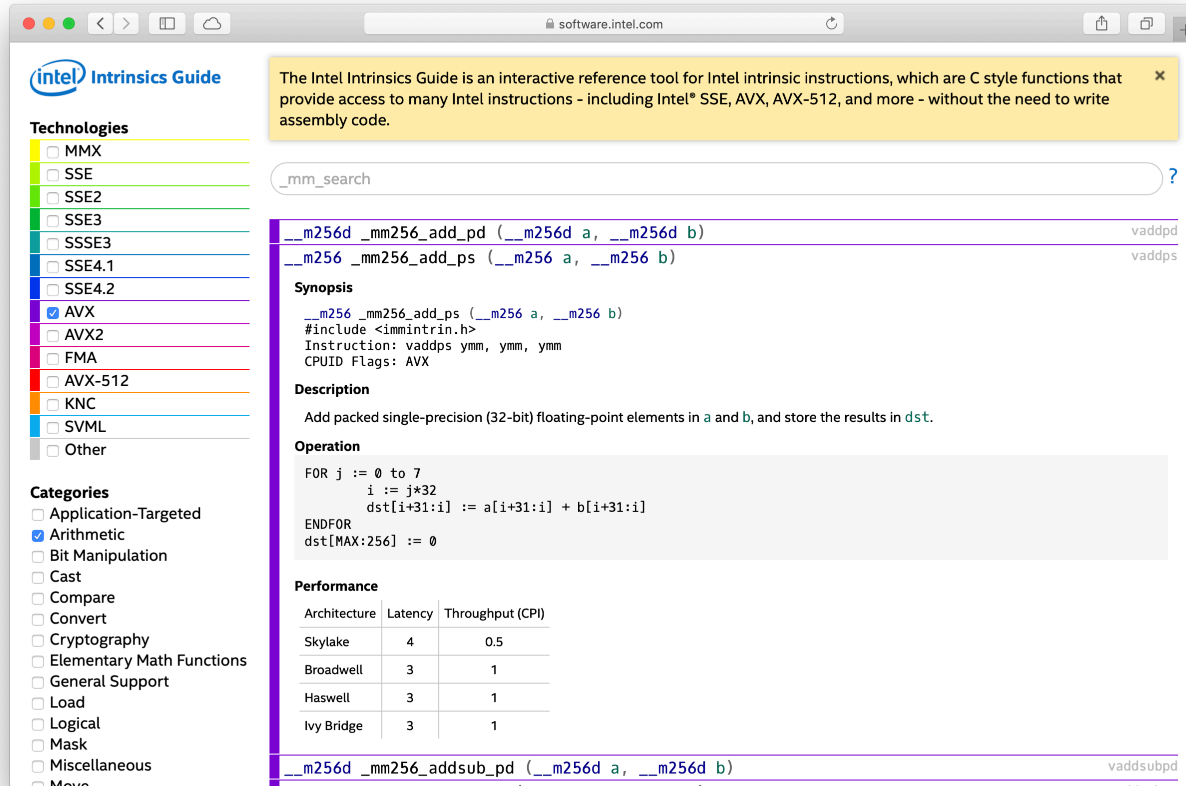






HPI Hasso Plattner Institut

Intel Intrinsics Guide



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

5

Autovectorization

Enable Autovectorization and Logging (GCC)

`-ftree-vectorize -m<arch>`	enable automatic code vectorization (part of `-O3`)
`-fopt-info-vec[-optimized]`	log loops optimized.
`-fopt-info-vec-missed`	log loops failed to optimized detailed information.
`-fopt-info-vec-note`	verbose info on loops and optimizations done
`-fopt-info-vec-all`	enable all above

```
example4.c:14:10: optimized: loop vectorized using 16 byte vectors
example4.c:9:6: note: vectorized 1 loops in function.

autovector.cpp:22:22: missed: couldn't vectorize loop
autovector.cpp:25:14: missed: not vectorized: complicated access
pattern.
```

What loops can be vectorized

- Countable loops
- Static counts (length does not change)
- Single entry and single exit (read: no data-depended `break`)
- All function calls can be in-lined, or are math intrinsics (`sin`, `floor`, …)
- Straight-line code (no `switch`-statements), mask-able `if`/`continue`

```
for (int i=0; i<length; i++) {
    float s = b[i]*b[i] - 4*a[i]*c[i];
    if ( s >= 0 ) {
        s = sqrt(s) ;
        x2[i] = (-b[i]+s)/(2.*a[i]);
        x1[i] = (-b[i]-s)/(2.*a[i]);
    } else {
        x2[i] = 0.;
        x1[i] = 0.;
    }
}
```

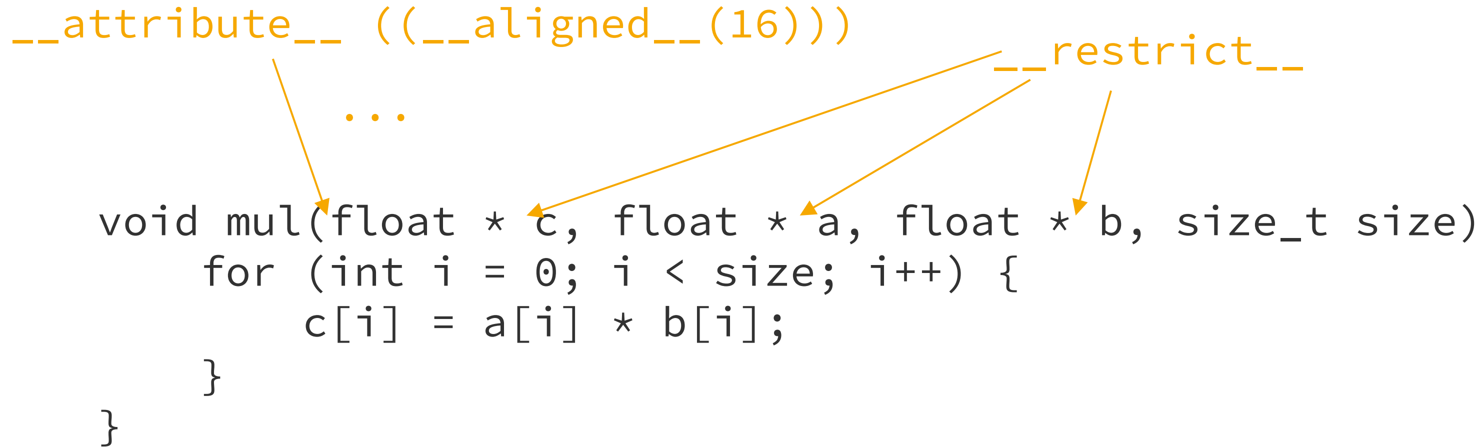
What cannot be vectorized

- Non-contiguous Memory Accesses (often in nested loops)
  - `for (int i=0; i<SIZE; i+=2) b[i] += a[i] * x[i];`
  - `for (int i=0; i<SIZE; i+=2) b[i] += a[i] * x[index[i]];`

- Data dependencies within vector length
  - `x[i]   = x[i-1]*2;` (read-after-write)
  - `x[i-1] = x[i]  *2;` (write-after-read)
  - Except: `sum = sum + x[j] * y[j]` (reduction)

https://software.intel.com/sites/default/files/m/4/8/8/2/a/31848-CompilerAutovectorizationGuide.pdf
https://software.intel.com/en-us/articles/common-vectorization-tips

Helping your compiler to vectorize



<Do you see a problem?>

What happens if a, b, or c overlap?

What if any of them is not aligned?

Hasso Plattner Institut
Digital Engineering · Universität Potsdam

And now for a break and
a cup of Ceylon with milk*.

*or beverage of your choice