OVERVIEW AND REMARKS

AUTOMATIC VECTORISATION IN GCC/CLANG (++) COMPILERS

AGENDA

- Overview
- Simple examples (Restrictions and comparisons)
- Advances examples (Restrictions and comparisons)
- Marked insights
- Summary

MECHANIZM RÓWNOLEGŁEGO PRZETWARZANIA STRUMIENI DANYCH W POJEDYNCZEJ INSTRUKCJI -> “SIMD” = SINGLE INSTRUCTION, MULTIPLE DATA

Wektoryzacja

ODMIANY REALIZACJI ROZSZERZEŃ ROZKAZÓW W PROCESORACH

- MMX (r.1996) “Matrix Math Extensions” (Intel pentium processors), rejestry: xmm
- SSE (r.1999) dodanie możliwość operacji na liczbach zmienne przeciekowych
- SSE2 (r. 2000) rozszerzenie puli rozkazów
- SSE3 (r.2004) rozszerzenie puli rozkazów
- SSE4 (r.2007) rozszerzenie puli rozkazów
- SSE5 (r.2009) rozszerzenie puli rozkazów
- AVX 1,2, 512 i nowsze (r.2008) (użycie rejestrów YMM0…15), rozszerzenie listy rozkazów z SSE

Data types - “packed”

packed type (treating 64-bit data as vector / matrix/ array of equal cell size)

Type	Instruction suffix	Form
Byte	“-b”	8x8 bits
Word	“-w”	4x16 bits
(double) dWord	“-d”	2x32 bits
(quad) qWord	“-q”	1x64 bits

AVX /AVX2 / AVX 512 rely on registers xmm0…15 (128 bit wide) / ymm0…15 (256 bit wide) / zmm0…15 (512 wide)

Possibility of data distribution is increasing with newer versions of each architecture and usage of wider registers.

Mnemonics

Characteristic from of mnemonics consists:

Prefix : v (vector) / p (parallel)

Suffix:

- PS, PD - floating point numbers vector
- SS, SD - scalar (1st vector elem.), also floating point number

Example (for AVX family): VMOVAPD, VSQRTPD, VSUBPD

V | classic mnemonic | extension data type

Example (for SSE family): MULPS, DIVSS, PAVGW

NONE or P | classic mnemonic | extension data type

C/C++ library for SMID functions:

https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#ig_expand=31,250,1430

DATA PREPARATION

Acceptable data feed for SIMD instructions :

- Basic / literal data types -> (int, float, double, bool, char …)
- Continuous in memory data structure (lists, arrays, heaps, vectors …)
- Custom data types compatible with SIMD instructions (Intel's SIMD-oriented FastMT library)
- Direct usage of intrinsics functions
- Predeclaration of data vectors np: “ double *x = __builtin_assume_aligned(a, 16); ” a: * double

*(intrinsic - a function which the compiler implements directly when possible, rather than linking to a library-provided implementation of the function)

OVERVIEW - VECTORISABLE FUNCTIONS

acos	ceil	fabs	round
acosh	Cos	floor	sin
asin	Cosh	fmax	sinh
asinh	erf	fmin	sqrt
atan	Erfc	log	tan
atan2	Erfinv	log10	tanh
atanh	Exp	log2	trunc
cbrt	exp2	pow	

SIMPLE EXAMPLES

Simple examples - basic loops

Input (C / C++)

```
#define size 400

unsigned int in[size];
unsigned int out[size];
int i;

void f_vect() {

  for ( i = 0; i < size; i++ ) out[i] = in[i]+10;

}
```

```
in.cpp:12:3: remark: vectorized loop (vectorization width: 8, interleaved count: 4) [-Rpass=loop-vectorize]
  for ( i = 0; i < size; i++ ) out[i] = in[i]+10;
```

Output (NASM assembler)

```
__Z6f_vectv:                            ## @_Z6f_vectv
    .cfi_startproc
## %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    movl    $24, %eax
    leaq    _in(%rip), %rcx
    vpbroadcastd    LCPI0_0(%rip), %ymm0    ## ymm0 = [10,10,10,10,10,10,10,10]
    leaq    _out(%rip), %rdx
    .p2align    4, 0x90
LBB0_1:                                 ## =>This Inner Loop Header: Depth=1
    vpaddd  -96(%rcx,%rax,4), %ymm0, %ymm1
    vpaddd  -64(%rcx,%rax,4), %ymm0, %ymm2
    vpaddd  -32(%rcx,%rax,4), %ymm0, %ymm3
    vpaddd  (%rcx,%rax,4), %ymm0, %ymm4
    vmovdqu %ymm1, -96(%rdx,%rax,4)
    vmovdqu %ymm2, -64(%rdx,%rax,4)
    vmovdqu %ymm3, -32(%rdx,%rax,4)
    vmovdqu %ymm4, (%rdx,%rax,4)
    addq    $32, %rax
    cmpq    $408, %rax                  ## imm = 0x198
    jne LBB0_1
```

Simple examples - basic loops

Input (C / C++)

```c
#define size 400

unsigned int in[size];
unsigned int out[size];
int i = 400, k = 0;

void f_vect() {

    while ( i-- > 100 ) {
      *(out+size-k) = *(in+k) | 123456;
      k++;
    }

}
```

Output (NASM assembler)

```
    vbroadcastss    LCPI0_0(%rip), %ymm0    ## ymm0 = [123456,123456,123456,123456,123456,123456,123456,123456]
    .p2align    4, 0x90
LBB0_4:                                 ## =>This Inner Loop Header: Depth=1
    vorps   -96(%rdi), %ymm0, %ymm1
    vorps   -64(%rdi), %ymm0, %ymm2
    vorps   -32(%rdi), %ymm0, %ymm3
    vorps   (%rdi), %ymm0, %ymm4
    vpermilps   $27, %ymm1, %ymm1       ## ymm1 = ymm1[3,2,1,0,7,6,5,4]
    vpermpd $78, %ymm1, %ymm1               ## ymm1 = ymm1[2,3,0,1]
    vmovups %ymm1, (%rsi,%rdx,4)
    vpermilps   $27, %ymm2, %ymm1       ## ymm1 = ymm2[3,2,1,0,7,6,5,4]
    vpermpd $78, %ymm1, %ymm1               ## ymm1 = ymm1[2,3,0,1]
    vmovups %ymm1, -32(%rsi,%rdx,4)
    vpermilps   $27, %ymm3, %ymm1       ## ymm1 = ymm3[3,2,1,0,7,6,5,4]
    vpermpd $78, %ymm1, %ymm1               ## ymm1 = ymm1[2,3,0,1]
    vmovups %ymm1, -64(%rsi,%rdx,4)
    vpermilps   $27, %ymm4, %ymm1       ## ymm1 = ymm4[3,2,1,0,7,6,5,4]
    vpermpd $78, %ymm1, %ymm1               ## ymm1 = ymm1[2,3,0,1]
    vmovups %ymm1, -96(%rsi,%rdx,4)
    subq    $-128, %rdi
    addq    $-32, %rdx
    cmpq    %rdx, %r10
    jne LBB0_4
## %bb.5:
    cmpq    %r8, %r9
```

Simple examples - (basic loops, usage of vectorisable functions)

Input (C / C++)

```
#define size 400

long int in[size];
long int out[size];
int i;

void f_vect() {

    do {
      out[i] = ceil(sqrt(in[5+i]));
      out[i] *= 5;
    } while ( i++ < size-5 );

}
```

```
in.cpp:12:5: remark: vectorized loop (vectorization width: 4, interleaved count: 1) [-Rpass=loop-vectorize]
```

Output (NASM assembler)

```
LBB0_3:                                 ## =>This Inner Loop Header: Depth=1
    vmovupd (%rcx,%rsi,8), %xmm0
    vmovdqu 16(%rcx,%rsi,8), %xmm1
    vpextrq $1, %xmm1, %rdx
    vcvtsi2sd   %rdx, %xmm3, %xmm2
    vmovq   %xmm1, %rdx
    vcvtsi2sd   %rdx, %xmm3, %xmm1
    vunpcklpd   %xmm2, %xmm1, %xmm1     ## xmm1 = xmm1[0],xmm2[0]
    vpextrq $1, %xmm0, %rdx
    vcvtsi2sd   %rdx, %xmm3, %xmm2
    vmovq   %xmm0, %rdx
    vcvtsi2sd   %rdx, %xmm3, %xmm0
    vunpcklpd   %xmm2, %xmm0, %xmm0     ## xmm0 = xmm0[0],xmm2[0]
    vinsertf128 $1, %xmm1, %ymm0, %ymm0
    vsqrtpd %ymm0, %ymm0
    vroundpd    $10, %ymm0, %ymm0
    vextractf128    $1, %ymm0, %xmm1
    vcvttsd2si  %xmm1, %rdx
    vmovq   %rdx, %xmm2
    vpermilpd   $1, %xmm1, %xmm1        ## xmm1 = xmm1[1,0]
    vcvttsd2si  %xmm1, %rdx
    vmovq   %rdx, %xmm1
    vpunpcklqdq %xmm1, %xmm2, %xmm1     ## xmm1 = xmm2[0],xmm1[0]
    vcvttsd2si  %xmm0, %rdx
    vmovq   %rdx, %xmm2
    vpermilpd   $1, %xmm0, %xmm0        ## xmm0 = xmm0[1,0]
    vcvttsd2si  %xmm0, %rdx
    vmovq   %rdx, %xmm0
    vpunpcklqdq %xmm0, %xmm2, %xmm0     ## xmm0 = xmm2[0],xmm0[0]
    vinserti128 $1, %xmm1, %ymm0, %ymm0
    vpsllq  $2, %ymm0, %ymm1
    vpaddq  %ymm0, %ymm1, %ymm0
    vmovdqu %ymm0, (%r11,%rsi,8)
    addq    $4, %rsi
    cmpq    %rsi, %rdi
    jne LBB0_3
```

Simple examples - basic loops (different data sizes)

Input (C / C++)

```
#define size 1000

short in[size];
long in2[size];
int in3[size];
long out[size];

int i;

void f_vect() {

    for ( i = 0 ; i < size; i++)
      out[i] = in[i] + in2[i] + in[3];

}
```

Output (NASM assembler)

```
LBB0_1:                                 ## =>This Inner Loop Header: Depth=1
    vpmovsxwq   -24(%rdx,%rcx,2), %ymm1
    vpmovsxwq   -16(%rdx,%rcx,2), %ymm2
    vpmovsxwq   -8(%rdx,%rcx,2), %ymm3
    vpmovsxwq   (%rdx,%rcx,2), %ymm4
    vpaddq  -96(%rsi,%rcx,8), %ymm1, %ymm1
    vpaddq  -64(%rsi,%rcx,8), %ymm2, %ymm2
    vpaddq  -32(%rsi,%rcx,8), %ymm3, %ymm3
    vpaddq  (%rsi,%rcx,8), %ymm4, %ymm4
    vpaddq  %ymm0, %ymm1, %ymm1
    vpaddq  %ymm0, %ymm2, %ymm2
    vpaddq  %ymm0, %ymm3, %ymm3
    vpaddq  %ymm0, %ymm4, %ymm4
    vmovdqu %ymm1, -96(%rdi,%rcx,8)
    vmovdqu %ymm2, -64(%rdi,%rcx,8)
    vmovdqu %ymm3, -32(%rdi,%rcx,8)
    vmovdqu %ymm4, (%rdi,%rcx,8)
    addq    $16, %rcx
    cmpq    $1004, %rcx                 ## imm = 0x3EC
    jne LBB0_1
```

Simple examples - basic loops (flow control)

Input (C / C++)

```
#define size 1000

short in[size];
short int out[size];
int i;

void f_vect() {

    for ( i = 0 ; i < size; i++)
      out[i] = in[i]*50>50 ? 0 : in[i]*50;

}
```

```
in.cpp:15:5: remark: vectorized loop (vectorization width: 8, interleaved count: 4) [-Rpass=loop-vectorize]
```

Output (NASM assembler)

```
LBB0_1:                                 ## =>This Inner Loop Header: Depth=1
    vpmovsxwd   -48(%rcx,%rax,2), %ymm2
    vpmovsxwd   -32(%rcx,%rax,2), %ymm3
    vpmovsxwd   -16(%rcx,%rax,2), %ymm4
    vpmovsxwd   (%rcx,%rax,2), %ymm5
    vpmulld %ymm0, %ymm2, %ymm2
    vpmulld %ymm0, %ymm3, %ymm3
    vpmulld %ymm0, %ymm4, %ymm4
    vpmulld %ymm0, %ymm5, %ymm5
    vpcmpgtd    %ymm0, %ymm2, %ymm6
    vextracti128    $1, %ymm6, %xmm7
    vpackssdw   %xmm7, %xmm6, %xmm9
    vpcmpgtd    %ymm0, %ymm3, %ymm7
    vextracti128    $1, %ymm7, %xmm1
    vpackssdw   %xmm1, %xmm7, %xmm10
    vpcmpgtd    %ymm0, %ymm4, %ymm7
    vextracti128    $1, %ymm7, %xmm6
    vpackssdw   %xmm6, %xmm7, %xmm6
    vpcmpgtd    %ymm0, %ymm5, %ymm7
    vextracti128    $1, %ymm7, %xmm1
    vpackssdw   %xmm1, %xmm7, %xmm1
    vpshufb %ymm8, %ymm2, %ymm2
    vpermq  $232, %ymm2, %ymm2              ## ymm2 = ymm2[0,2,2,3]
    vpshufb %ymm8, %ymm3, %ymm3
    vpermq  $232, %ymm3, %ymm3              ## ymm3 = ymm3[0,2,2,3]
    vpshufb %ymm8, %ymm4, %ymm4
    vpermq  $232, %ymm4, %ymm4              ## ymm4 = ymm4[0,2,2,3]
    vpshufb %ymm8, %ymm5, %ymm5
    vpermq  $232, %ymm5, %ymm5              ## ymm5 = ymm5[0,2,2,3]
    vpandn  %xmm2, %xmm9, %xmm2
    vpandn  %xmm3, %xmm10, %xmm3
    vpandn  %xmm4, %xmm6, %xmm4
    vpandn  %xmm5, %xmm1, %xmm1
    vmovdqa %xmm2, -48(%rdx,%rax,2)
    vmovdqa %xmm3, -32(%rdx,%rax,2)
    vmovdqa %xmm4, -16(%rdx,%rax,2)
    vmovdqa %xmm1, (%rdx,%rax,2)
    addq    $32, %rax
    cmpq    $1016, %rax                     ## imm = 0x3F8
    jne LBB0_1
```

Simple examples - basic loops (loop nesting)

Input (C / C++)

```
#define size 1000

double in[size];
double in2[size];
double* out[100];
int i, k;

void f_vect() {

    for ( i = 0 ; i < 100; i+=2)
      for (k = 0; k < size; k++) {
          out[i][k] = in[k];
          out[i+1][k] = in2[k];
      }

}
```

```
in.cpp:15:7: remark: vectorized loop (vectorization width: 4, interleaved count: 2) [-Rpass=loop-vectorize]
```

Output (NASM assembler)

```
LBB0_7:                                 ##   Parent Loop BB0_1 Depth=1
                                        ## =>  This Inner Loop Header: Depth=2
    vmovups -96(%rsi,%rax,8), %ymm0
    vmovups -64(%rsi,%rax,8), %ymm1
    vmovups %ymm0, -96(%rdx,%rax,8)
    vmovups %ymm1, -64(%rdx,%rax,8)
    vmovups -96(%rcx,%rax,8), %ymm0
    vmovups -64(%rcx,%rax,8), %ymm1
    vmovups %ymm0, -96(%rdi,%rax,8)
    vmovups %ymm1, -64(%rdi,%rax,8)
    cmpq    $1004, %rax                 ## imm = 0x3EC
    je  LBB0_11
## %bb.8:                               ##   in Loop: Header=BB0_7 Depth=2
    vmovups -32(%rsi,%rax,8), %ymm0
    vmovups (%rsi,%rax,8), %ymm1
    vmovups %ymm0, -32(%rdx,%rax,8)
    vmovups %ymm1, (%rdx,%rax,8)
    vmovups -32(%rcx,%rax,8), %ymm0
    vmovups (%rcx,%rax,8), %ymm1
    vmovups %ymm0, -32(%rdi,%rax,8)
    vmovups %ymm1, (%rdi,%rax,8)
    addq    $16, %rax
    jmp LBB0_7
    .p2align    4, 0x90
LBB0_9:                                 ##   in Loop: Header=BB0_1 Depth=1
    movl    $1, %eax
    .p2align    4, 0x90
LBB0_10:                                ##   Parent Loop BB0_1 Depth=1
                                        ## =>  This Inner Loop Header: Depth=2
    vmovsd  -8(%rsi,%rax,8), %xmm0      ## xmm0 = mem[0],zero
    vmovsd  %xmm0, -8(%rdx,%rax,8)
    vmovsd  -8(%rcx,%rax,8), %xmm0      ## xmm0 = mem[0],zero
    vmovsd  %xmm0, -8(%rdi,%rax,8)
    vmovsd  (%rsi,%rax,8), %xmm0        ## xmm0 = mem[0],zero
    vmovsd  %xmm0, (%rdx,%rax,8)
    vmovsd  (%rcx,%rax,8), %xmm0        ## xmm0 = mem[0],zero
    vmovsd  %xmm0, (%rdi,%rax,8)
    addq    $2, %rax
    cmpq    $1001, %rax                 ## imm = 0x3E9
    jne LBB0_10
    jmp LBB0_11
```

Arithmetic operations

Logical operations

Inline (short) if statements

Loop nesting

Pointers operations with inconsistent memory offsets (pointer chasing)

Usage of various loops with adjusted invariant

Operations over inconsistent data sizes

Usage of variables external to loops

Restrictions - loop conditions

Vectorisable code (C/ C++)

```
for (i = 0; i < size ; i++ ) { }
```

```
for (i = 0; i < size-10 ; i+=8 ) { }
```

```
for (i = size; i > 20 ; i-=5 ) {  }
```

```
int n = 100000;

void f_vect() {

    for (i = 0; i < n; i++ ) { out[i+1] = in[i] * 10 + 254 + sqrt(in[i]); }

}
```

Unvectorisable code (C/ C++)

```
for (i = 0; i < n - i; i++ ) { }
```

```
for (i = 0; i < n; i+=k ) {

   k *= 2;
   out[i] = sqrt(out[i]);

}
```

```
  i = 0;
  while ( true ) {
    in[i++]++;
    i %= 100;
  }
```

```
int n = 100000;

void f_vect() {

    for (i = 0; i < n; i++ ) {
      out[i+1] = in[i] * 10 + 254 + sqrt(in[i]);
      n--;
    }

}
```

```
for (i = 0; i < n; i++ ) {

  out[i] = sqrt(out[i]);
  if ( true && i < k ) break;

}
```

```
in.cpp:21:4: remark: loop not vectorized: could not determine number of loop iterations [-Rpass-analysis=loop-vectorize]
```

Restrictions - function call / control flow

Vectorisable code (C/ C++)

```
void f_vect() {
    i = 10;
      while (i++ < 300) {
          out[i] = in[i] > 10? floor(in[i]) : ceil(in[i]);
      }
}
```

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

__m256i _mm256_adds_epi8 (__m256i a, __m256i b)	vpaddsb
__m256i _mm256_adds_epu16 (__m256i a, __m256i b)	vpaddusw
__m256i _mm256_adds_epu8 (__m256i a, __m256i b)	vpaddusb
__m256d _mm256_addsub_pd (__m256d a, __m256d b)	vaddsubpd
__m256 _mm256_addsub_ps (__m256 a, __m256 b)	vaddsubps
__m256i _mm256_alignr_epi8 (__m256i a, __m256i b, const	vpalignr
__m256d _mm256_and_pd (__m256d a, __m256d b)	vandpd
__m256 _mm256_and_ps (__m256 a, __m256 b)	vandps
__m256i _mm256_and_si256 (__m256i a, __m256i b)	vpand
__m256d _mm256_andnot_pd (__m256d a, __m256d b)	vandnpd
__m256 _mm256_andnot_ps (__m256 a, __m256 b)	vandnps
__m256i _mm256_andnot_si256 (__m256i a, __m256i b)	vpandn
__m256i _mm256_avg_epu16 (__m256i a, __m256i b)	vpavgw
__m256i _mm256_avg_epu8 (__m256i a, __m256i b)	vpavgb
__m128 _mm_bcstnebf16_ps (const __bf16* __A)	vbcstnebf162ps
__m256 _mm256_bcstnebf16_ps (const __bf16* __A)	vbcstnebf162ps
__m128 _mm_bcstnesh_ps (const _Float16* __A)	vbcstnesh2ps
__m256 _mm256_bcstnesh_ps (const _Float16* __A)	vbcstnesh2ps

Unvectorisable code (C/ C++)

```
void f_vect() {
    i = 10;
    do
    {
      f1:
      out[i] = sqrt(out[i] * 2);

      if (out[i] > 50) {
        goto f1;
      }

    } while ( i++ < 1000 );
    return;
}
```

```
void f_vect() {
    for ( i = 3; i <= 333; i++ ) {
        switch (i%3) {
          case 0:
              out[i] = 1000;
              break;

          case 1:
              out[i] *= 2;
              break;

          case 2:
              out[i] -= out[i-1];
              break;
        }
    }
}
```

```
remark: loop not vectorized: unsafe dependent memory operations in loop.
```

Vectorisable code (C/ C++)

```
void quad(int length, float *a, float *b, float *c, float *x1, float *x2) {
    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.;
      }
    }
}
```

```
double foo(double a, double b, double c) {  return (a+b)*sin(c); }

void f_vect() {

    for ( i = 3; i <= 333; i++ ) {

        out[i] = foo(in[i], in[i-1], in[i-2]);
    }

}
```

ADVANCED EXAMPLES

Advanced examples (function unrolling)

Input (C / C++)

```
void function1() {
  int i;
  for (i=0; i<120; i++)
    a[i] = b[i] + c[i];
}
```

Output (NASM assembler)

```
__Z9function1v:                         ## @_Z9function1v
    .cfi_startproc
## %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    movdqa  _c(%rip), %xmm0
    paddd   _b(%rip), %xmm0
    movdqa  %xmm0, _a(%rip)
    movdqa  _c+16(%rip), %xmm0
    paddd   _b+16(%rip), %xmm0
    movdqa  %xmm0, _a+16(%rip)
    movdqa  _c+32(%rip), %xmm0
    paddd   _b+32(%rip), %xmm0
    movdqa  %xmm0, _a+32(%rip)
    movdqa  _c+48(%rip), %xmm0
    paddd   _b+48(%rip), %xmm0
    movdqa  %xmm0, _a+48(%rip)
    movdqa  _c+64(%rip), %xmm0
    paddd   _b+64(%rip), %xmm0
    movdqa  %xmm0, _a+64(%rip)
    movdqa  _c+80(%rip), %xmm0
    paddd   _b+80(%rip), %xmm0
    movdqa  %xmm0, _a+80(%rip)
    movdqa  _c+96(%rip), %xmm0
    paddd   _b+96(%rip), %xmm0
    movdqa  %xmm0, _a+96(%rip)
    movdqa  _c+112(%rip), %xmm0
    paddd   _b+112(%rip), %xmm0
    movdqa  %xmm0, _a+112(%rip)
    movdqa  _c+128(%rip), %xmm0
    paddd   _b+128(%rip), %xmm0
    movdqa  %xmm0, _a+128(%rip)
    movdqa  _c+144(%rip), %xmm0
    paddd   _b+144(%rip), %xmm0
    movdqa  %xmm0, _a+144(%rip)
    movdqa  _c+160(%rip), %xmm0
    paddd   _b+160(%rip), %xmm0
    movdqa  %xmm0, _a+160(%rip)
    movdqa  _c+176(%rip), %xmm0
    paddd   _b+176(%rip), %xmm0
    movdqa  %xmm0, _a+176(%rip)
    movdqa  _c+192(%rip), %xmm0
    paddd   _b+192(%rip), %xmm0
    movdqa  %xmm0, _a+192(%rip)
    movdqa  _c+208(%rip), %xmm0
    paddd   _b+208(%rip), %xmm0
    movdqa  %xmm0, _a+208(%rip)
    movdqa  _c+224(%rip), %xmm0
    paddd   _b+224(%rip), %xmm0
    movdqa  %xmm0, _a+224(%rip)
    movdqa  _c+240(%rip), %xmm0
    paddd   _b+240(%rip), %xmm0
    movdqa  %xmm0, _a+240(%rip)
    movdqa  _c+256(%rip), %xmm0
    paddd   _b+256(%rip), %xmm0
    movdqa  %xmm0, _a+256(%rip)
    movdqa  _c+272(%rip), %xmm0
    paddd   _b+272(%rip), %xmm0
    movdqa  %xmm0, _a+272(%rip)
    movdqa  _c+288(%rip), %xmm0
    paddd   _b+288(%rip), %xmm0
    movdqa  %xmm0, _a+288(%rip)
    movdqa  _c+304(%rip), %xmm0
    paddd   _b+304(%rip), %xmm0
    movdqa  %xmm0, _a+304(%rip)
    movdqa  _c+320(%rip), %xmm0
    paddd   _b+320(%rip), %xmm0
    movdqa  %xmm0, _a+320(%rip)
    movdqa  _c+336(%rip), %xmm0
    paddd   _b+336(%rip), %xmm0
    movdqa  %xmm0, _a+336(%rip)
    movdqa  _c+352(%rip), %xmm0
    paddd   _b+352(%rip), %xmm0
    movdqa  %xmm0, _a+352(%rip)
    movdqa  _c+368(%rip), %xmm0
    paddd   _b+368(%rip), %xmm0
    movdqa  %xmm0, _a+368(%rip)
    movdqa  _c+384(%rip), %xmm0
    paddd   _b+384(%rip), %xmm0
    movdqa  %xmm0, _a+384(%rip)
    movdqa  _c+400(%rip), %xmm0
    paddd   _b+400(%rip), %xmm0
    movdqa  %xmm0, _a+400(%rip)
    movdqa  _c+416(%rip), %xmm0
    paddd   _b+416(%rip), %xmm0
    movdqa  %xmm0, _a+416(%rip)
    movdqa  _c+432(%rip), %xmm0
    paddd   _b+432(%rip), %xmm0
    movdqa  %xmm0, _a+432(%rip)
    movdqa  _c+448(%rip), %xmm0
    paddd   _b+448(%rip), %xmm0
    movdqa  %xmm0, _a+448(%rip)
    movdqa  _c+464(%rip), %xmm0
    paddd   _b+464(%rip), %xmm0
    movdqa  %xmm0, _a+464(%rip)
    popq    %rbp
    retq
```

Advanced examples (functions chaining)

Input (C / C++)

```
#define size 10000

double in[size];
double out[size];
int i;

void f_vect() {

    for (i = 0; i < size ; i++ )
      in2[i] = random()*5;
    for ( i = 0 ; i < size; i++)
      out[i] = floor(exp(pow(sin(ceil(sin(in2[i]))), 4)));

}
```

```
in.cpp:16:5: remark: the cost-model indicates that vectorization is not beneficial [-Rpass-missed=loop-vectorize]
```

Output (NASM assembler)

```
LBB0_1:                                 ## =>This Inner Loop Header: Depth=1
        callq   _random
        leaq    (%rax,%rax,4), %rax
        vcvtsi2sd   %rax, %xmm1, %xmm0
        movslq  _i(%rip), %rax
        leal    1(%rax), %ecx
        vmovsd  %xmm0, (%r14,%rax,8)
        movl    %ecx, _i(%rip)
        cmpq    $9999, %rax                     ## imm = 0x270F
        jl  LBB0_1
## %bb.2:
        movl    $1, %ebx
        leaq    _in(%rip), %r15
        .p2align    4, 0x90
LBB0_3:                                 ## =>This Inner Loop Header: Depth=1
        vmovsd  -8(%r14,%rbx,8), %xmm0          ## xmm0 = mem[0],zero
        vmovsd  (%r14,%rbx,8), %xmm1            ## xmm1 = mem[0],zero
        vmovsd  %xmm1, -32(%rbp)                ## 8-byte Spill
        callq   _sin
        vroundsd    $10, %xmm0, %xmm0, %xmm0
        callq   _sin
        vmovsd  LCPI0_0(%rip), %xmm1            ## xmm1 = mem[0],zero
        callq   _pow
        callq   _exp
        vroundsd    $9, %xmm0, %xmm0, %xmm0
        vmovsd  %xmm0, -8(%r15,%rbx,8)
        vmovsd  -32(%rbp), %xmm0                ## 8-byte Reload
                                                ## xmm0 = mem[0],zero
        callq   _sin
        vroundsd    $10, %xmm0, %xmm0, %xmm0
        callq   _sin
        vmovsd  LCPI0_0(%rip), %xmm1            ## xmm1 = mem[0],zero
        callq   _pow
        callq   _exp
        vroundsd    $9, %xmm0, %xmm0, %xmm0
        vmovsd  %xmm0, (%r15,%rbx,8)
        addq    $2, %rbx
        cmpq    $10001, %rbx                    ## imm = 0x2711
        jne LBB0_3
```

Advanced examples (SLP - “Superword-level parallelism”)

Input (C / C++)

```
void func(int a1, int a2, int b1, int b2, int *A) {
  A[0] = a1*(a1 + b1);
  A[1] = a2*(a2 + b2);
  A[2] = a1*(a1 + b1);
  A[3] = a2*(a2 + b2);
  A[4] = a1*(a1 + b1);
  A[5] = a2*(a2 + b2);
  A[6] = a1*(a1 + b1);
  A[7] = a2*(a2 + b2);
  A[8] = a1*(a1 + b1);
  A[9] = a2*(a2 + b2);
  A[10] = a1*(a1 + b1);
  A[11] = a2*(a2 + b2);
  A[12] = a1*(a1 + b1);
  A[13] = a2*(a2 + b2);
  A[14] = a1*(a1 + b1);
  A[15] = a2*(a2 + b2);
}
```

Output (NASM assembler)

```
__Z4funciiiiPi:                         ## @_Z4funciiiiPi
    .cfi_startproc
## %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    vmovd   %edx, %xmm0
    vpinsrd $1, %ecx, %xmm0, %xmm0
    vmovd   %edi, %xmm1
    vpinsrd $1, %esi, %xmm1, %xmm1
    vpaddd  %xmm1, %xmm0, %xmm0
    vpmulld %xmm1, %xmm0, %xmm0
    vpbroadcastq    %xmm0, %ymm0
    vmovdqu %ymm0, (%r8)
    vmovdqu %ymm0, 32(%r8)
    popq    %rbp
    vzeroupper
    retq
    .cfi_endproc
                                        ## -- End function
```

Advanced examples (SLP - “Superword-level parallelism”) / without vectorisation

Input (C / C++)

```
void func(int a1, int a2, int b1, int b2, int *A) {
  A[0] = a1*(a1 + b1);
  A[1] = a2*(a2 + b2);
  A[2] = a1*(a1 + b1);
  A[3] = a2*(a2 + b2);
  A[4] = a1*(a1 + b1);
  A[5] = a2*(a2 + b2);
  A[6] = a1*(a1 + b1);
  A[7] = a2*(a2 + b2);
  A[8] = a1*(a1 + b1);
  A[9] = a2*(a2 + b2);
  A[10] = a1*(a1 + b1);
  A[11] = a2*(a2 + b2);
  A[12] = a1*(a1 + b1);
  A[13] = a2*(a2 + b2);
  A[14] = a1*(a1 + b1);
  A[15] = a2*(a2 + b2);
}
```

Output (NASM assembler)

```
__Z4funciiiiPi:                         ## @_Z4funciiiiPi
    .cfi_startproc
## %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    addl    %edi, %edx
    imull   %edi, %edx
    movl    %edx, (%r8)
    addl    %esi, %ecx
    imull   %esi, %ecx
    movl    %ecx, 4(%r8)
    movl    %edx, 8(%r8)
    movl    %ecx, 12(%r8)
    movl    %edx, 16(%r8)
    movl    %ecx, 20(%r8)
    movl    %edx, 24(%r8)
    movl    %ecx, 28(%r8)
    movl    %edx, 32(%r8)
    movl    %ecx, 36(%r8)
    movl    %edx, 40(%r8)
    movl    %ecx, 44(%r8)
    movl    %edx, 48(%r8)
    movl    %ecx, 52(%r8)
    movl    %edx, 56(%r8)
    movl    %ecx, 60(%r8)
    popq    %rbp
    retq
    .cfi_endproc
                                        ## -- End function
```

Advanced examples

Vectorisable code (C/ C++)

Unvectorisable code (C/ C++)

```c
/*Read-after-write dependency*/
void vec_test3() {
    for (int i = 1; i < DATA_SIZE; i++) {
    a[i] = a[i-1] + b[i];
    c[i] = a[i] + d[i];
    }
}
```

```c
/*Read-after-write dependencies:*/
void vec_test1() {
    for (int i = 1; i < DATA_SIZE; i++) {
    a[i] = a[i-1] + b[i];
    }
}
```

```c
/*Write-after-reading dependency*/
void vec_test4() {
    for (int i = 1; i < DATA_SIZE; i++) {
    a[i] = b[i-1] + c[i-1];
    b[i] = a[i] * d[i];
    }
}
```

```c
/*Write-after-reading dependencies:*/
void vec_test2() {
    for (int i = 1; i < DATA_SIZE; i++) {
    a[i-1] = a[i] + b[i];
    }
}
```

```c
/*Loop-carried dependency*/
void vec_test5() {
    for (int i = 0; i < DATA_SIZE; i++) {
    a[i] = b[i] + c[i];
    }
    for (int i = 1; i < DATA_SIZE; i++) {
    a[i] = a[i-1] + d[i];
    }
```

Advanced examples

Vectorisable code (C/ C++)

```c
#include <stdio.h>

int main() {
    int arr[1000];
    int sum = 0;

    // Initialize array
    for (int i = 0; i < 1000; i++) {
        arr[i] = i;
    }

    // Sum array elements using a loop
    for (int i = 0; i < 1000; i++) {
        sum = sum + arr[i];
    }

    return 0;
}
```

Unvectorisable code (C/ C++)

```c
#include <stdio.h>

int main(int argc, char **argv) {
    int sum = 0;
    for (int i = 1; i < 10; i++) {
        sum += i;
    }
    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    int n = 100000;
    int *a = malloc(sizeof(int) * n);
    int i, sum = 0;

    // Initialize array with random values
    for (i = 0; i < n; i++) {
        a[i] = rand() % 10;
    }

    // Compute sum of array elements
    for (i = 0; i < n; i++) {
        sum += a[i];
    }
    return 0;
}
```

SLP vectorizer / Loop vectorizer

Function unrolling / collapsing

Functions chaining

Write after read data dependency (a[i-1] = a[1]) [safe]

Read after write data dependency (a[i] = a[i -1]) [not safe]

Thanks for your attention

- http://locklessinc.com/articles/vectorize/
- https://www.llvm.org/docs/Vectorizers.html
- https://en.wikipedia.org/wiki/MMX_(instruction_set)
- https://en.wikipedia.org/wiki/Advanced_Vector_Extensions
- https://gcc.gnu.org/projects/tree-ssa/vectorization.html
- https://github.com/neurolabusc/simd
- https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#ig_expand=31,250,1430,2,161,162,163,172
- https://www.intel.pl/content/www/pl/pl/support/articles/000005779/processors.html
- https://www.intel.com/content/www/us/en/developer/tools/isa-extensions/overview.html
- https://www.intel.com/content/dam/develop/external/us/en/documents/31848-compilerautovectorizationguide.pdf
- https://yunmingzhang.wordpress.com/2016/12/02/vectorization-with-clang/