

# Лекция 4

## Векторизация циклов с ветвлениями

**Курносов Михаил Георгиевич**

E-mail: [mkurnosov@gmail.com](mailto:mkurnosov@gmail.com)

WWW: [www.mkurnosov.net](http://www.mkurnosov.net)

Курс «Распределенная обработка информации»

Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)

Осенний семестр, 2019

# Поиск максимума в массиве: скалярная версия

```
float find_max(float *v, int n)
{
    float max = -FLT_MAX;
    for (int i = 0; i < n; i++)
        if (v[i] > max)
            max = v[i];
    return max;
}
```

```
double run_scalar()
{
    float *v = xmalloc(sizeof(*v) * n);
    for (int i = 0; i < n; i++)
        v[i] = i + 1.0;

    double t = wtime();
    float res = find_max(v, n);
    t = wtime() - t;

    float valid_result = (float)n;
    printf("Result (scalar): %.6f err = %f\n", res, fabsf(valid_result - res));
    printf("Elapsed time (scalar): %.6f sec.\n", t);
    free(v);
    return t;
}
```

# Поиск максимума в массиве: SSE, float

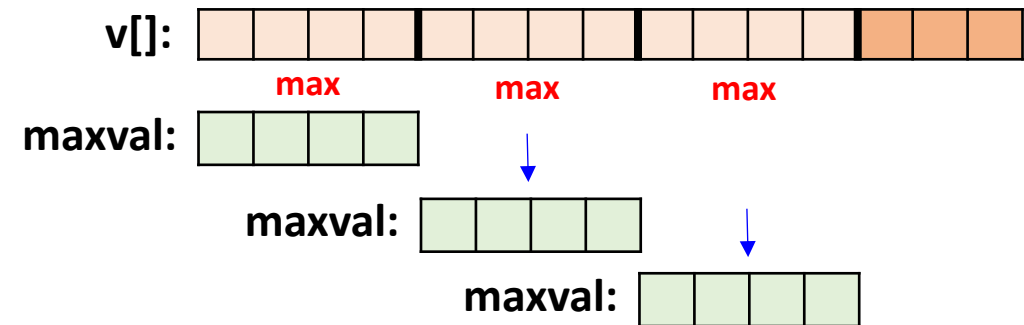
```
float find_max_sse(float * restrict v, int n)
{
    __m128 *vv = (__m128 *)v;
    int k = n / 4;

    __m128 maxval = _mm_set1_ps(-FLT_MAX);
    for (int i = 0; i < k; i++)
        maxval = _mm_max_ps(maxval, vv[i]);

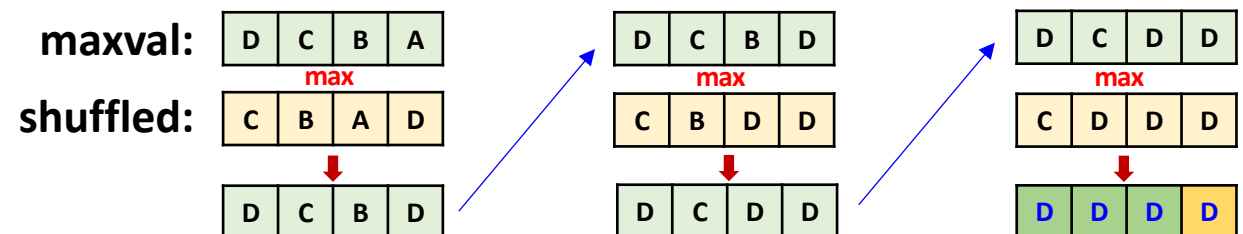
    // Horizontal max
    // a = [a3, a2, a1, a0]
    // shuffle(a, a, _MM_SHUFFLE(2, 1, 0, 3)) ==> [a2, a1, a0, a3]
    maxval = _mm_max_ps(maxval, _mm_shuffle_ps(maxval, maxval, _MM_SHUFFLE(2, 1, 0, 3)));
    maxval = _mm_max_ps(maxval, _mm_shuffle_ps(maxval, maxval, _MM_SHUFFLE(2, 1, 0, 3)));
    maxval = _mm_max_ps(maxval, _mm_shuffle_ps(maxval, maxval, _MM_SHUFFLE(2, 1, 0, 3)));
    float max;
    _mm_store_ss(&max, maxval);

    for (int i = k * 4; i < n; i++)
        if (v[i] > max)
            max = v[i];
    return max;
}
```

## 1) Векторный поиск максимума (вертикальная операция)



## 2) Горизонтальный поиск максимума в векторе



## 3) Скалярный поиск максимума в «хвосте» массива

# Поиск максимума в массиве: SSE, float

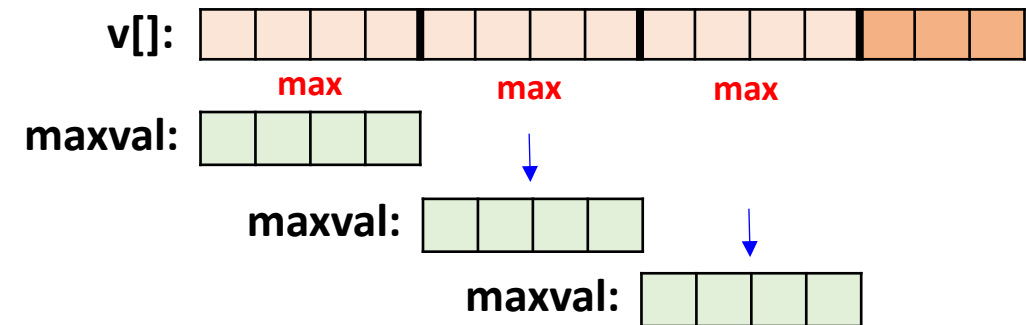
```
float find_max_sse(float * restrict v, int n)
{
    __m128 *vv = (__m128 *)v;
    int k = n / 4;

    __m128 maxval = _mm_set1_ps(-FLT_MAX);
    for (int i = 0; i < k; i++)
        maxval = _mm_max_ps(maxval, vv[i]);

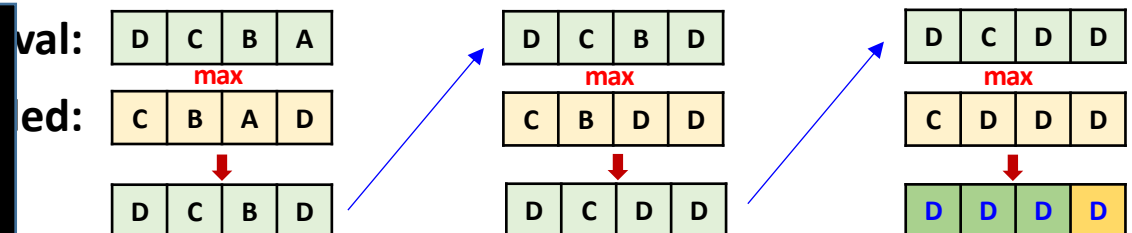
    // Horizontal max
    // a = [a3, a2, a1, a0]
    // shuffle(a, a, _MM_SHUFFLE(2, 1, 0, 3)) ==> [a2, a1, a0, a3]
    maxval = _mm_max_ps(maxval, _mm_shuffle_ps(maxval, maxval, _MM_SHUFFLE(2, 1, 0, 3)));
    maxval = _mm_max_ps(maxval, _mm_shuffle_ps(maxval, maxval, _MM_SHUFFLE(2, 1, 0, 3)));
    maxval = _mm_max_ps(maxval, _mm_shuffle_ps(maxval, maxval, _MM_SHUFFLE(2, 1, 0, 3)));
    float max;
    _mm_store_ss(&max, maxval);
}
```

```
# Intel Core i5-3320M - Ivy Bridge (Sandy Bridge shrink)
Reduction: n = 1000003
Result (scalar): 1000003.000000 err = 0.000000
Elapsed time (scalar): 0.002047 sec.
Result (vectorized): 1000003.000000 err = 0.000000
Elapsed time (vectorized): 0.000529 sec.
Speedup: 3.87
```

## 1) Векторный поиск максимума (вертикальная операция)



## 2) Горизонтальный поиск максимума в векторе



## 3) Скалярный поиск максимума в «хвосте» массива

# Поиск максимума в массиве: **AVX**, float

```
float find_max_avx(float * restrict v, int n)
{
    __m256 *vv = (__m256 *)v;
    int k = n / 8;

    __m256 maxval = _mm256_set1_ps(-FLT_MAX);
    for (int i = 0; i < k; i++)
        maxval = _mm256_max_ps(maxval, vv[i]);

    maxval = _mm256_max_ps(maxval, _mm256_shuffle_ps(maxval, maxval, _MM_SHUFFLE(2, 1, 0, 3)));
    maxval = _mm256_max_ps(maxval, _mm256_shuffle_ps(maxval, maxval, _MM_SHUFFLE(2, 1, 0, 3)));
    maxval = _mm256_max_ps(maxval, _mm256_shuffle_ps(maxval, maxval, _MM_SHUFFLE(2, 1, 0, 3)));
    float t[8];
    _mm256_store_ps(t, maxval);
    float max = t[0] > t[5] ? t[0] : t[5];

    for (int i = k * 8; i < n; i++)
        if (v[i] > max)
            max = v[i];
    return max;
}
```

**1) Горизонтальная операция над двумя частями по 128 бит**  
 $[a7, a6, a5, a4 \mid a3, a2, a1, a0] \Rightarrow [a6, a5, a4, a7 \mid a2, a1, a0, a3]$   
 $[a6, a5, a4, a7 \mid a2, a1, a0, a3] \Rightarrow [a5, a4, a7, a6 \mid a1, a0, a3, a2]$   
 $[a5, a4, a7, a6 \mid a1, a0, a3, a2] \Rightarrow [a4, a7, a6, a5 \mid a0, a3, a2, a1]$   
 $[a4, a7, a6, a5 \mid a0, a3, a2, a1]$

**2) Выбор максимального из t[5] и t[0]**

**3) Обработка «хвоста»**

# Поиск максимума в массиве: AVX, float

```
float find_max_avx(float * restrict v, int n)
```

```
{
```

```
    __m256 *vv = (__m256 *)v;
```

```
    int k = n / 8;
```

```
    __m256 maxval = _mm256_set1_ps(-FLT_MAX);
```

```
    for (int i = 0; i < k; i++)
```

```
        maxval = _mm256_max_ps(maxval, vv[i]);
```

```
    maxval = _mm256_max_ps(maxval, _mm256_shuffle_ps(maxval, maxval, _MM_SHUFFLE(2, 1, 0, 3)));
```

```
    maxval = _mm256_max_ps(maxval, _mm256_shuffle_ps(maxval, maxval, _MM_SHUFFLE(2, 1, 0, 3)));
```

```
    maxval = _mm256_max_ps(maxval, _mm256_shuffle_ps(maxval, maxval, _MM_SHUFFLE(2, 1, 0, 3)));
```

```
    float t[8];
```

```
    _mm256_store_ps(t, maxval);
```

```
    float max = t[0] > t[5] ? t[0] : t[5];
```

```
    for (int i = k * 8; i < n; i++)
```

```
        if (v[i] > max)
```

```
            max = v[i];
```

```
# Intel Core i5-3320M - Ivy Bridge (Sandy Bridge shrink)
```

```
Reduction: n = 1000003
```

```
Result (scalar): 1000003.000000 err = 0.000000
```

```
Elapsed time (scalar): 0.002053 sec.
```

```
Result (vectorized): 1000003.000000 err = 0.000000
```

```
Elapsed time (vectorized): 0.000428 sec.
```

```
Speedup: 4.80
```

**1) Горизонтальная операция над двумя частями по 128 бит**

$[a7, a6, a5, a4 \mid a3, a2, a1, a0] \Rightarrow [a6, a5, a4, a7 \mid a2, a1, a0, a3]$

$[a6, a5, a4, a7 \mid a2, a1, a0, a3] \Rightarrow [a5, a4, a7, a6 \mid a1, a0, a3, a2]$

$[a5, a4, a7, a6 \mid a1, a0, a3, a2] \Rightarrow [a4, a7, a6, a5 \mid a0, a3, a2, a1]$

$[a4, a7, a6, a5 \mid a0, a3, a2, a1]$

максимального из t[5] и t[0]

отка «хвоста»

# Поиск максимума в массиве: AVX, float, **permute**

```
float find_max_avx(float * restrict v, int n)
{
    __m256 *vv = (__m256 *)v;
    int k = n / 8;

    __m256 maxval = _mm256_set1_ps(-FLT_MAX);
    for (int i = 0; i < k; i++)
        maxval = _mm256_max_ps(maxval, vv[i]);

    // Horizontal max
    maxval = _mm256_max_ps(maxval, _mm256_permute_ps(maxval, _MM_SHUFFLE(2, 1, 0, 3)));
    maxval = _mm256_max_ps(maxval, _mm256_permute_ps(maxval, _MM_SHUFFLE(2, 1, 0, 3)));
    maxval = _mm256_max_ps(maxval, _mm256_permute_ps(maxval, _MM_SHUFFLE(2, 1, 0, 3)));

    float t[8];
    _mm256_store_ps(t, maxval);
    float max = t[0] > t[5] ? t[0] : t[5];

    for (int i = k * 8; i < n; i++)
        if (v[i] > max)
            max = v[i];
    return max;
}
```

```
# Intel Core i5-3320M - Ivy Bridge (Sandy Bridge shrink)
Reduction: n = 1000003
Result (scalar): 1000003.000000 err = 0.000000
Elapsed time (scalar): 0.002201 sec.
Result (vectorized): 1000003.000000 err = 0.000000
Elapsed time (vectorized): 0.000433 sec.
Speedup: 5.08
```

# Вычисление квадратного корня: скалярная версия

```
void compute_sqrt(float *in, float *out, int n)
{
    for (int i = 0; i < n; i++) {
        if (in[i] > 0)
            out[i] = sqrtf(in[i]);
        else
            out[i] = 0.0;
    }
}
```

Направление ветвления в цикле  
зависит от входных данных

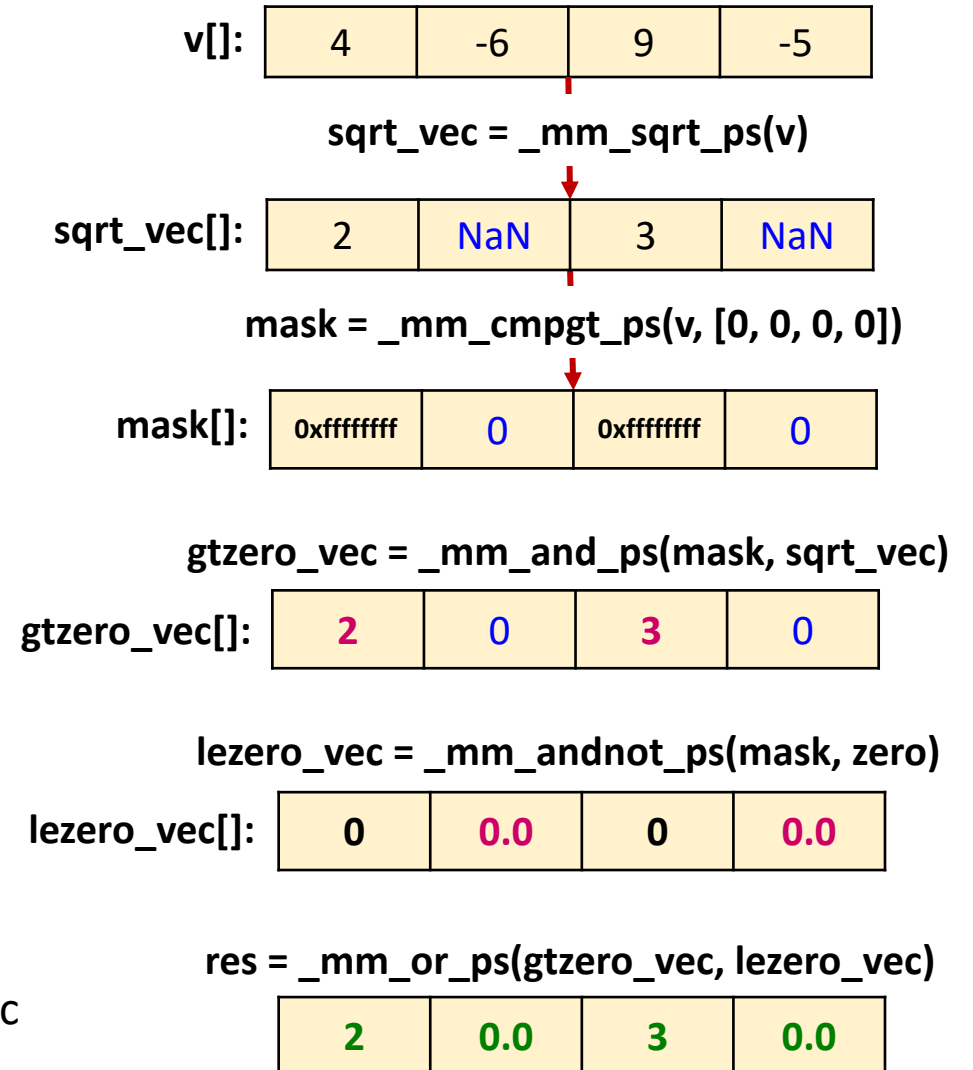
```
double run_scalar()
{
    float *in = xmalloc(sizeof(*in) * n);
    float *out = xmalloc(sizeof(*out) * n);
    srand(0);
    for (int i = 0; i < n; i++) {
        in[i] = rand() > RAND_MAX / 2 ? 0 : rand() / (float)RAND_MAX * 1000.0;
    }
    double t = wtime();
    compute_sqrt(in, out, n);
    t = wtime() - t;

    printf("Elapsed time (scalar): %.6f sec.\n", t);
    free(in);
    free(out);
    return t;
}
```



# Вычисление квадратного корня: SSE, float

1. **Вычисляем корень квадратный для вектора  $v[0:3]$**  – если значение  $v[i]$  меньше или равно нулю, результат NaN  
`sqrt_vec = _mm_sqrt_ps(v)`
2. **Выполняем векторное сравнение:**  $v[0:3] > [0, 0, 0, 0]$   
`mask = _mm_cmpgt_ps(v, zero)`  
результат сравнения – вектор `mask[0:3]`, в котором `mask[i] = v[i] > 0 ? 0xffffffff : 0`
3. **Извлекаем из `sqrt_vec[0:3]` элементы**, для которых выполнено условие  $v[i] > 0$   
`gtzero_vec = _mm_and_ps(mask, sqrt_vec)`  
В `gtzero_vec` элементы NaN заменены на 0
4. **Извлекаем из `zero[0:3]` элементы**, для которых условие не выполнено:  $v[i] \leq 0$   
`lezero_vec = _mm_andnot_ps(mask, zero)`  
В `lezero_vec` элементы 0 заменены на 0.0 (значение в ветви else).
5. **Объединяем результаты ветвей** – векторы `gtzero_vec` и `lezero_vec`  
`res = _mm_or_ps(gtzero_vec, lezero_vec)`

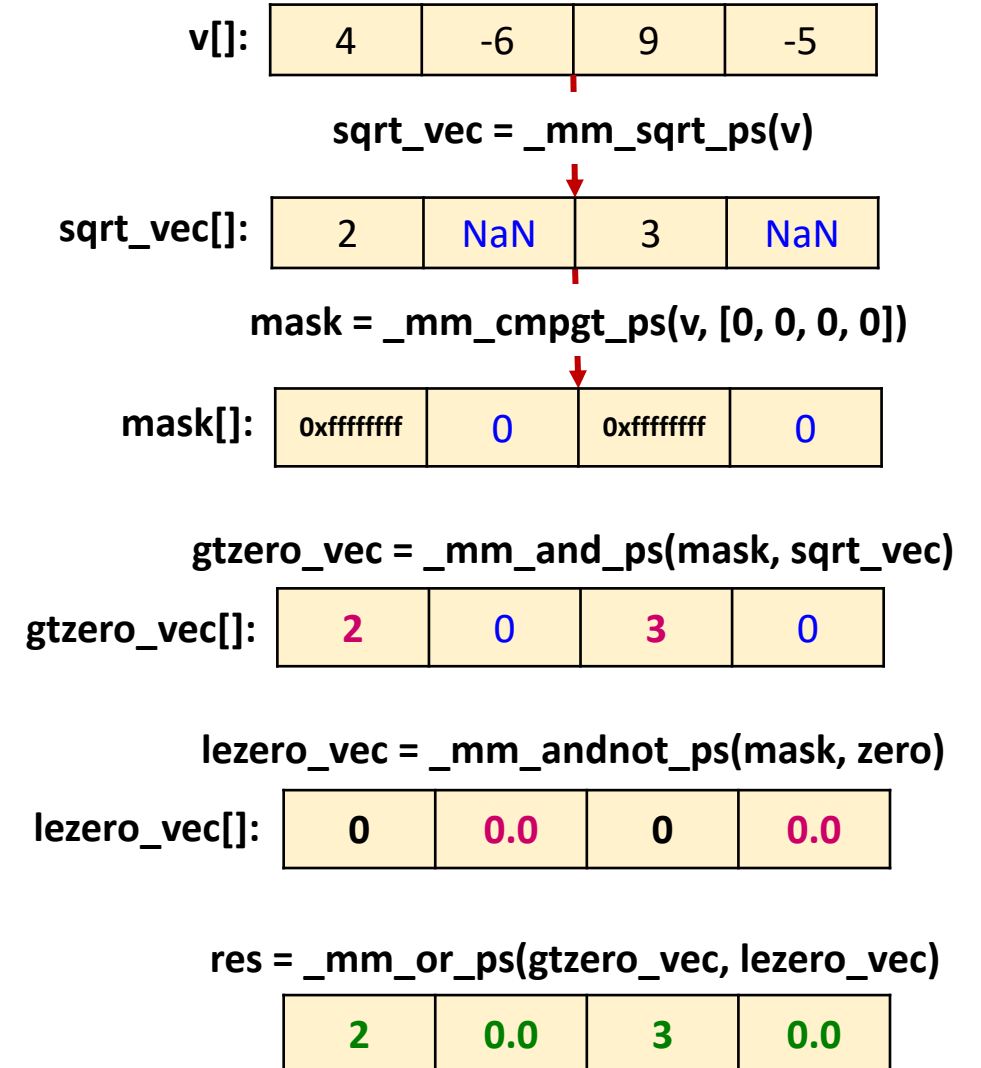


# Вычисление квадратного корня: SSE, float

```
void compute_sqrt_sse(float *in, float *out, int n)
{
    __m128 *in_vec = (__m128 *)in;
    __m128 *out_vec = (__m128 *)out;
    int k = n / 4;

    __m128 zero = _mm_setzero_ps();
    for (int i = 0; i < k; i++) {
        __m128 v = _mm_load_ps((float *)&in_vec[i]);
        __m128 sqrt_vec = _mm_sqrt_ps(v);
        __m128 mask = _mm_cmpgt_ps(v, zero);
        __m128 gtzero_vec = _mm_and_ps(mask, sqrt_vec);
        __m128 lezero_vec = _mm_andnot_ps(mask, zero);
        out_vec[i] = _mm_or_ps(gtzero_vec, lezero_vec);
    }

    for (int i = k * 4; i < n; i++)
        out[i] = in[i] > 0 ? sqrtf(in[i]) : 0.0;
}
```



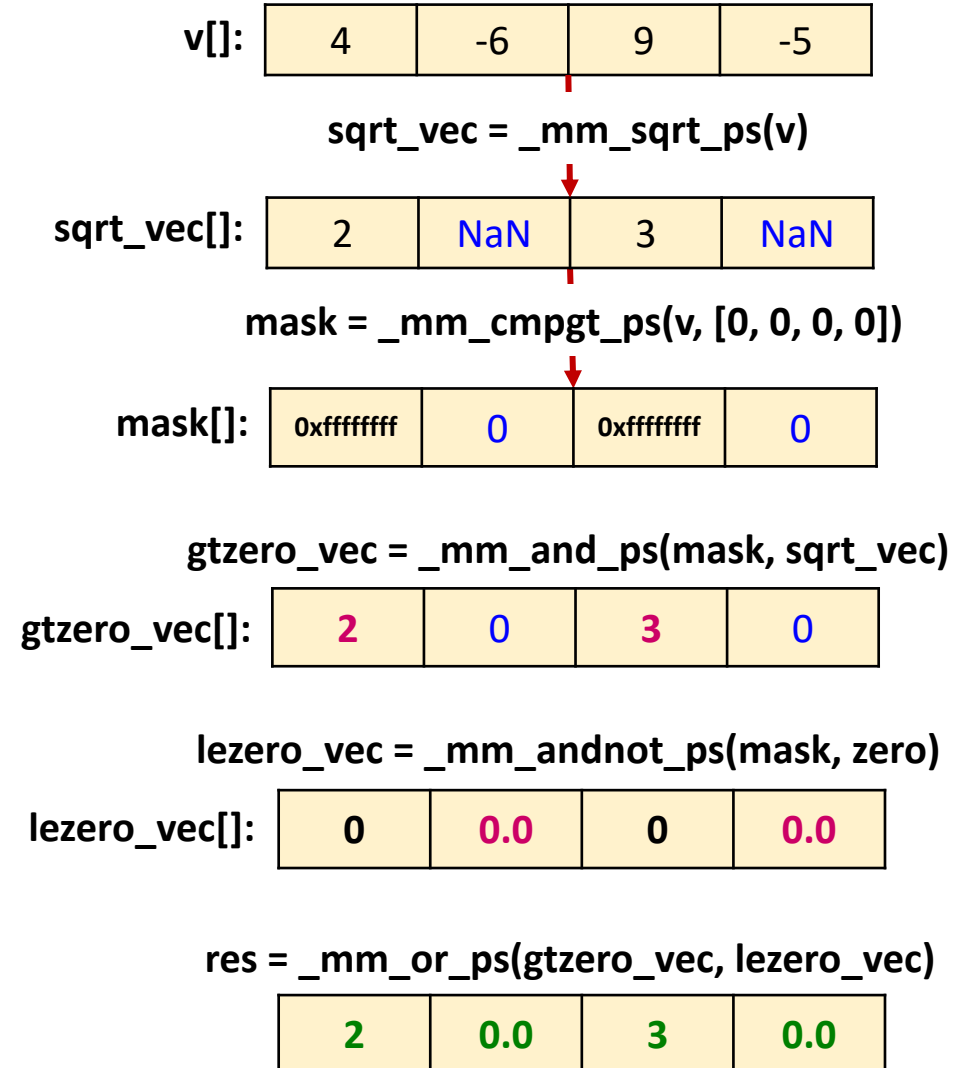
# Вычисление квадратного корня: SSE, float

```
void compute_sqrt_sse(float *in, float *out, int n)
{
    __m128 *in_vec = (__m128 *)in;
    __m128 *out_vec = (__m128 *)out;
    int k = n / 4;

    __m128 zero = _mm_setzero_ps();
    for (int i = 0; i < k; i++) {
        __m128 v = _mm_load_ps((float *)&in_vec[i]);
        __m128 sqrt_vec = _mm_sqrt_ps(v);
        __m128 mask = _mm_cmpgt_ps(v, zero);
        __m128 gtzero_vec = _mm_and_ps(mask, sqrt_vec);
        __m128 lezero_vec = _mm_andnot_ps(mask, zero);
        out_vec[i] = _mm_or_ps(gtzero_vec, lezero_vec);
    }

    for (int i = k * 4; i < n; i++)
```

```
# Intel Core i5-3320M - Ivy Bridge (Sandy Bridge shrink)
Tabulate sqrt: n = 1000003
Elapsed time (scalar): 0.009815 sec.
Elapsed time (vectorized): 0.002176 sec.
Speedup: 4.51
```



# Вычисление квадратного корня: AVX, float

```
void compute_sqrt_avx(float *in, float *out, int n)
{
    __m256 *in_vec = (__m256 *)in;
    __m256 *out_vec = (__m256 *)out;
    int k = n / 8;

    __m256 zero = _mm256_setzero_ps();
    for (int i = 0; i < k; i++) {
        __m256 v = _mm256_load_ps((float *)&in_vec[i]);
        __m256 sqrt_vec = _mm256_sqrt_ps(v);
        __m256 mask = _mm256_cmp_ps(v, zero, _CMP_GT_OQ);
        __m256 gtzero_vec = _mm256_and_ps(mask, sqrt_vec);
        __m256 lezero_vec = _mm256_andnot_ps(mask, zero);
        out_vec[i] = _mm256_or_ps(gtzero_vec, lezero_vec);
    }

    for (int i = k * 8; i < n; i++)
        out[i] = in[i] > 0 ? sqrtf(in[i]) : 0.0;
}
```

# Вычисление квадратного корня: AVX, float, **blend**

```
void compute_sqrt_avx(float *in, float *out, int n)
{
    __m256 *in_vec = (__m256 *)in;
    __m256 *out_vec = (__m256 *)out;
    int k = n / 8;

    __m256 zero = _mm256_setzero_ps();
    for (int i = 0; i < k; i++) {
        // 1. Compute sqrt: all elements <= 0 will be filled with NaNs
        // 2. Vector compare (greater-than): in[i] > 0
        //    mask = cmpgt([7, 1, 0, 2, 0, 4, 4, 9], zero) ==>
        //    mask = [0xffffffff, 0xffffffff, 0, 0xffffffff, 0, 0xffffffff, ..., 0xffffffff]
        //
        // 3. Blend (merge results from two vectors by mask)
        //    blend(zero, [7, 1, 0, 2, 0, 4, 4, 9], mask) = [7, 1, 0f, 2, 0f, 4, 4, 9]
        //
        __m256 v = _mm256_load_ps((float *)&in_vec[i]);
        __m256 sqrt_vec = _mm256_sqrt_ps(v);
        __m256 mask = _mm256_cmp_ps(v, zero, _CMP_GT_OQ);
        out_vec[i] = _mm256_blendv_ps(zero, sqrt_vec, mask);
    }
    for (int i = k * 8; i < n; i++)
        out[i] = in[i] > 0 ? sqrtf(in[i]) : 0.0;
}
```

# **Автоматическая векторизация кода компилятором**

# Автоматическая векторизация: GCC 5.3.1

```
enum { n = 1000003 };

int main(int argc, char **argv)
{
    float *a = malloc(sizeof(*a) * n);
    float *b = malloc(sizeof(*b) * n);
    float *c = malloc(sizeof(*c) * n);

    for (int i = 0; i < n; i++) {
        a[i] = 1.0;
        b[i] = 2.0;
    }

    for (int i = 0; i < n; i++) {
        c[i] = a[i] + b[i];
    }

    free(c); free(b); free(a);
    return 0;
}
```

```
$ gcc -Wall -std=c99 -O2 -march=native -ftree-vectorize -fopt-info-vec -c vec.c -o vec.o
vec.c:19:5: note: loop vectorized
vec.c:14:5: note: loop vectorized
gcc -o vec vec.o -lm
```

# Автоматическая векторизация: clang 3.7

```
enum { n = 1000003 };

int main(int argc, char **argv)
{
    float *a = malloc(sizeof(*a) * n); float *b = malloc(sizeof(*b) * n); float *c = malloc(sizeof(*c) * n);

    for (int i = 0; i < n; i++) {
        a[i] = 1.0; b[i] = 2.0;
    }

    for (int i = 0; i < n; i++) {
        c[i] = a[i] + b[i];
    }

    free(c); free(b); free(a);
    return 0;
}
```

```
$ clang -Wall -O2 -Rpass=loop-vectorize -Rpass-missed=loop-vectorize \
    -Rpass-analysis=loop-vectorize -c vec.c -o vec.o
vec.c:14:5: remark: vectorized loop (vectorization width: 4, interleaved count: 2) [-Rpass=loop-vectorize]
    for (int i = 0; i < n; i++) {
    ^
vec.c:19:5: remark: vectorized loop (vectorization width: 4, interleaved count: 2) [-Rpass=loop-vectorize]
    for (int i = 0; i < n; i++) {
    ^
$ clang -o vec vec.o
```



# Требования к циклам

- **Требования к циклам**

- ☐ Отсутствие зависимости по данным между итерациями цикла
- ☐ Отсутствие вызовов функций в цикле
- ☐ Число итераций цикла должно быть вычислимым
- ☐ Обращение к последовательным смежным элементам массива
- ☐ Отсутствие сложных ветвлений

- **Проблемные ситуации**

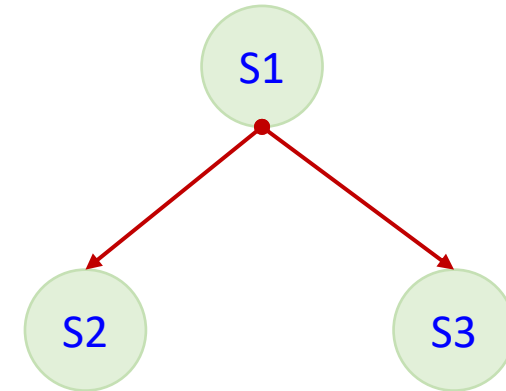
- ☐ Сложный цикл – может не хватить векторных регистров
- ☐ Смешанные типы данных (int, float, char)

# Зависимости по данным между инструкциями

S1:  $A = B + C$

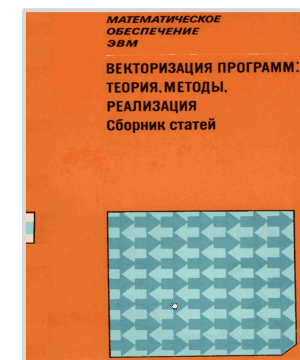
S2:  $D = A + 2$

S3:  $E = A + 3$



Граф зависимостей по данным  
(data-dependence graph)

- S2 зависит от S1 – S1 и S2 нельзя выполнять параллельно
- S3 зависит от S1 – S1 и S3 нельзя выполнять параллельно
- S2 и S3 можно выполнять параллельно



**Векторизация программ.  
Теория, методы, реализация**  
(сборник статей). – М.: Мир, 1991. – 275 с.

# Виды зависимости по данным между инструкциями

## 1. Потокковая зависимость, истинная зависимость

(Read After Write – RAW, true dependence, flow/data dependency):  $S1 \delta S2$

S1:  $a = \dots$

S2:  $b = a$

## 2. Антязависимость (Write After Read – WAR, anti-dependence): $S1 \bar{\delta} S2$

S1:  $b = a$

S2:  $a = \dots$

## 3. Выходная зависимость (Write After Write – WAW, output dependence): $S1 \delta^o S2$

S1:  $a = \dots$

S2:  $a = \dots$

# Виды зависимости по данным между итерациями циклов

- Имеется две строки программы S1 и S2
- Обозначим:
  - $\text{Write}(S)$  – множество ячеек памяти, в которые S осуществляет запись
  - $\text{Read}(S)$  – множество ячеек памяти, которые S читает
- **Условия Бернштейна.** Строка S2 зависит от строки S1 тогда и только тогда, когда
$$(\text{Read}(S1) \cap \text{Write}(S2)) \cup (\text{Write}(S1) \cap \text{Read}(S2)) \cup (\text{Write}(S1) \cap \text{Write}(S2)) \neq \emptyset$$

[\*] A. J. Bernstein. *Program Analysis for Parallel Processing* // IEEE Trans. on Electronic Computers, 1966.

## ■ Развертка цикла по итерациям:

S1:  $a[1] = a[0] + b[0]$

S2:  $a[2] = a[1] + b[1]$

- $\text{Read}(S1) = \{a[0], b[0]\}, \text{Write}(S1) = \{a[1]\}$
- $\text{Read}(S2) = \{a[1], b[1]\}, \text{Write}(S2) = \{a[2]\}$

$$(\text{Read}(S1) \cap \text{Write}(S2)) \cup (\text{Write}(S1) \cap \text{Read}(S2)) \cup (\text{Write}(S1) \cap \text{Write}(S2)) = \emptyset \cup \{a[1]\} \cup \emptyset = \{a[1]\}$$

# Зависимости по данным между итерациями циклов

```
for (int i = 0; i < n; i++) {  
    a[i] = 1.0;  
    b[i] = 2.0;  
}
```

```
S1: for (int i = 0; i < n - 1; i++) {  
    a[i + 1] = a[i] + b[i];  
}
```

- Развертка цикла по итерациям (строка S1):

S1: a[1] = a[0] + b[0]

S1: a[2] = a[1] + b[1] // Read After Write dep.

S1: a[3] = a[2] + b[2] // Read After Write dep.

S1: a[4] = a[3] + b[3] // Read After Write dep.

```
$ gcc -ftree-vectorize -fopt-info-vec -fopt-info-vec-missed ./vec.c
```

```
vec.c:18:5: note: not vectorized, possible dependence between data-refs vec.c:18:5: note: bad data dependence.
```

```
vec.c:18:5: note: not vectorized, possible dependence between data-refs vec.c:18:5: note: bad data dependence.
```

```
...
```

```
vec.c:13:5: note: loop vectorized
```

```
...
```

```
$ clang -Rpass=loop-vectorize -Rpass-missed=loop-vectorize -Rpass-analysis=loop-vectorize ./vec.c -ovec
```

```
./vec.c:13:5: remark: vectorized loop (vectorization width: 4, interleaved count: 2) [-Rpass=loop-vectorize]
```

```
./vec.c:19:20: remark: loop not vectorized: value that could not be identified as reduction is used outside the loop  
[-Rpass-analysis=loop-vectorize]
```

```
./vec.c:18:5: remark: loop not vectorized: use -Rpass-analysis=loop-vectorize for more info  
[-Rpass-missed=loop-vectorize]
```

# Виды зависимости по данным между итерациями циклов

## 1. Read After Write (RAW, true dependence, flow/data dependency)

S1: **a** = ...

S2: b = **a**

## 2. Write After Read (WAR, anti-dependence)

S1: b = **a**

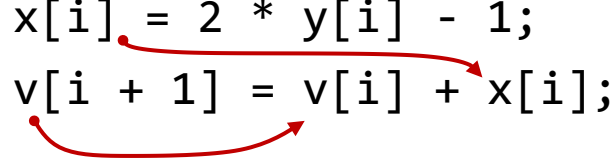
S2: **a** = ...

## 3. Write After Write (WAW, output dependence)

S1: **a** = ...

S2: **a** = ...

```
S1:   for (int i = 0; i < n - 1; i++) {  
      x[i] = 2 * y[i] - 1;  
S2:   v[i + 1] = v[i] + x[i];  
      }  
}
```



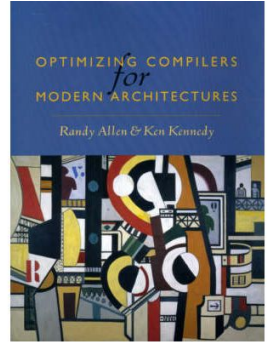
S1 --> S2 (RAW- flow dependence) –  
*цикло-независимая зависимость*

S2 --> S2 (RAW) – *циклическая зависимость*  
(требуется выполнить не менее одной итерации  
для ее возникновения)

# Виды зависимости по данным между итерациями циклов

**Утверждение.** Цикл может быть векторизован тогда и только тогда, когда в нем отсутствуют циклические зависимости между операциями [\*].

[\*] Randy Allen, Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*.  
- Morgan Kaufmann Publishers, 2001.



- Предполагается наличие векторных регистров бесконечной длины (VL)
- Циклические зависимости, для возникновения которых требуется не менее  $VL + 1$  итераций, могут быть проигнорированы (например, 5 итераций для SSE-инструкций типа float)

S2:

```
for (int i = 0; i < n - 1; i++) {  
    v[i] = v[i + 3] + 4;  
}
```

Iter 1:  $v[0] = v[3] + 4$   
Iter 2:  $v[1] = v[4] + 4$   
Iter 3:  $v[2] = v[5] + 4$   
Iter 4:  $v[3] = v[6] + 4$   
Iter 5:  $v[4] = v[7] + 4$   
Iter 6:  $v[5] = v[8] + 4$   
...

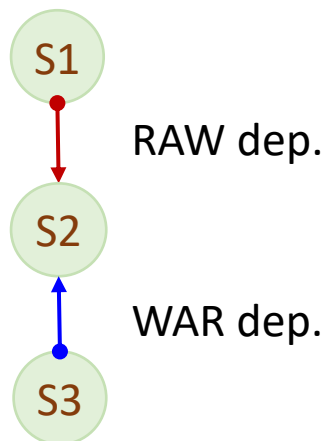
Можно векторизовать  
для (SSE, double),  
но не для (SSE, float)!

Зависимость через 3  
итерации

# Автоматическая векторизация циклов

- Компилятор анализирует граф зависимостей по данным для самых внутренних циклов
- Если в графе зависимостей по данным отсутствуют контуры (замкнутые пути), то его можно векторизовать

```
for (int i = 0; i < n; i++) {  
S1:  a[i] = b[i];  
S2:  c[i] = a[i] + b[i];  
S3:  e[i] = c[i + 1];  
}
```



a[0] = b[0]  
c[0] = a[0] + b[0]  
e[0] = c[1]

a[1] = b[1]  
c[1] = a[1] + b[1]  
e[1] = c[2]

a[2] = b[2]  
c[2] = a[2] + b[2]  
e[2] = c[3]

a[3] = b[3]  
c[3] = a[3] + b[3]  
e[3] = c[4]

Контуры  
отсутствуют



// S3 должна предшествовать S2  
a[0:n-1] = b[0:n-1]  
e[0:n-1] = c[1:n]  
c[0:n-1] = a[0:n-1] + b[0:n-1]



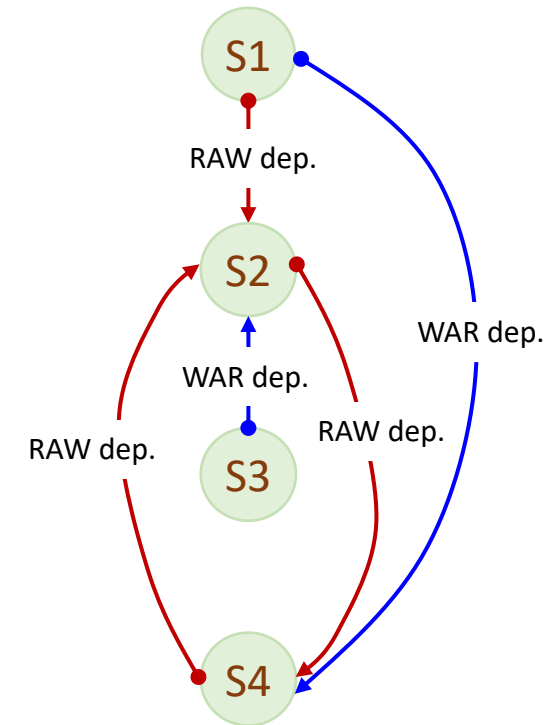
# Автоматическая векторизация циклов

```
for (int i = 2; i <= n; i++) {  
S1:   a[i] = b[i];  
S2:   c[i] = a[i] + b[i - 1];  
S3:   e[i] = c[i + 1];  
S4:   b[i] = c[i] + 2;  
}
```

- **Инструкции S2 и S4 образуют контур**
- Все операторы контура исполняются последовательно (...)
- Остальные инструкции векторизуемы



```
a[2:n] = b[2:n]  
e[2:n] = c[3:n + 1]  
for (int i = 2; i <= n; i++) {  
S2:   c[i] = a[i] + b[i - 1];  
S4:   b[i] = c[i] + 2;  
}
```



Инструкции S2 и S4 образуют контур

# Автоматическая векторизация циклов

```
void mul_alpha(int *x, int *y, int a, int n)
{
    for (int i = 0; i < n; i++)
        y[i] = a * x[i];
}
```

- Что известно об указателях  $x$  и  $y$ ?
- Возможно указывают на один массив (пересекаются)
- Компилятору необходимо проводить межпроцедурный анализ или использовать «подсказки»

```
void mul_alpha(int * restrict x, int * restrict y, int a, int n)
{
    for (int i = 0; i < n; i++)
        y[i] = a * x[i];
}
```

- **restrict** – для доступа к объекту используется данный указатель  $p$  или значение, основанное на указателе  $p$  (например,  $p + 1$ )

# Литература

- Randy Allen, Ken Kennedy. ***Optimizing Compilers for Modern Architectures: A Dependence-Based Approach***. - Morgan Kaufmann Publishers, 2001.
- Steven Muchnick. ***Advanced Compiler Design and Implementation***, 1997
- Aart J.C. Bik. ***Software Vectorization Handbook, The: Applying Intel Multimedia Extensions for Maximum Performance***, 2004.
- Keith Cooper, Linda Torczon. ***Engineering a Compiler***, 2011
- ***Векторизация программ. Теория, методы, реализация*** (сборник статей). – М.: Мир, 1991. – 275 с.
- Auto-vectorization in GCC // <https://gcc.gnu.org/projects/tree-ssa/vectorization.html>
- Auto-Vectorization in LLVM // <http://llvm.org/docs/Vectorizers.html>