Семинар 2 (21)
Векторизация кода

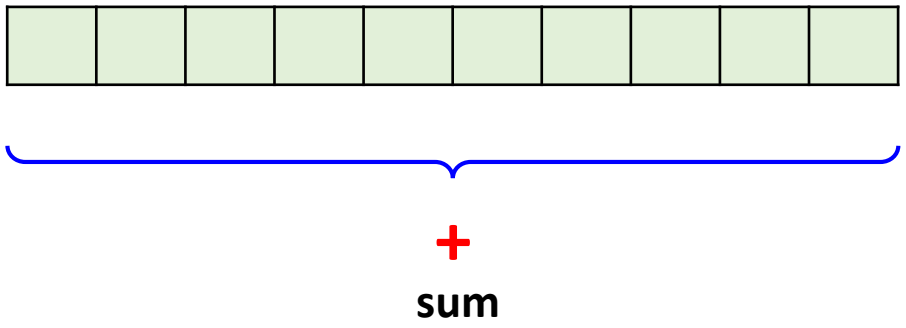
Михаил Курносов

E-mail: mkurnosov@gmail.com
WWW: www.mkurnosov.net

Цикл семинаров «Основы параллельного программирования»
Институт физики полупроводников им. А. В. Ржанова СО РАН
Новосибирск, 2016

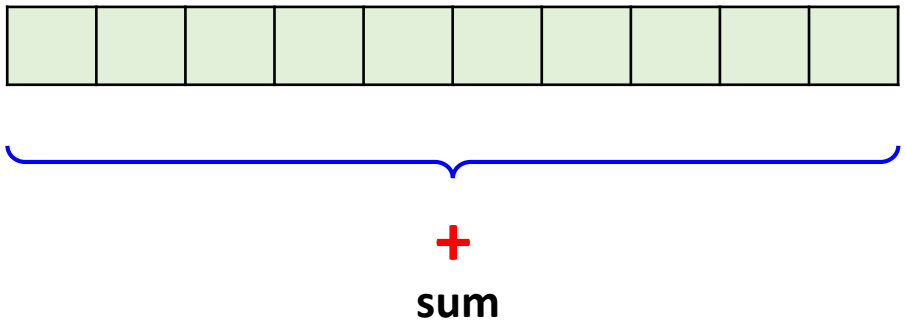
Редукция (reduction, reduce)

```
enum { n = 1000003 };

float sum(float *v, int n)
{
    float s = 0;
    for (int i = 0; i < n; i++)
        s += v[i];
    return s;
}

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 = sum(v, n);
    t = wtime() - t;

    float valid_result = (1.0 + (float)n) * 0.5 * 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;
}
```



Редукция (reduction, reduce)

```
enum { n = 1000003 };

float sum(float *v, int n)
{
    float s = 0;
    for (int i = 0; i < n; i++)
        s += v[i];
    return s;
}

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 = sum(v, n);
    t = wtime() - t;

    float valid_result = (1.0 + (float)n) * 0.5 * 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;
}
```

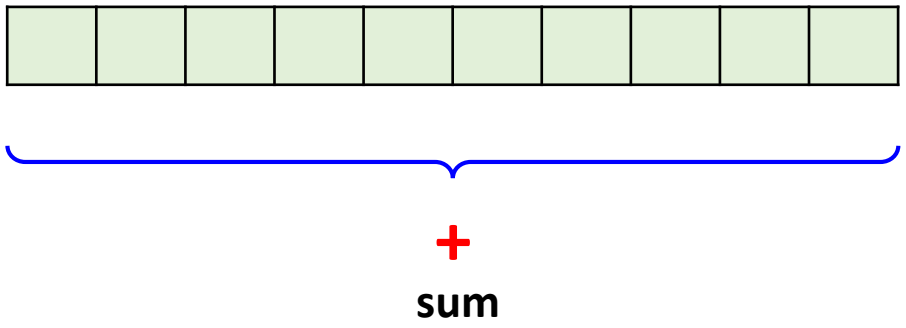



```
$ ./reduction
Reduction: n = 1000003
Result (scalar): 499944423424.000000 err = 59080704.0
Elapsed time (scalar): 0.001011 sec.
```

?

Редукция (reduction, reduce)

```
enum { n = 1000003 };

float sum(float *v, int n)
{
    float s = 0;
    for (int i = 0; i < n; i++)
        s += v[i];
    return s;
}

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 = sum(v, n);
    t = wtime() - t;

    float valid_result = (1.0 + (float)n) * 0.5 * 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;
}
```



- **float** (IEEE 754, single-precision) имеет ограниченную точность
- погрешность результата суммирование *n* чисел в худшем случае растет пропорционально *n*

Вариант 1: переход от float к double

```
double sum(double *v, int n)
{
    double s = 0;
    for (int i = 0; i < n; i++)
        s += v[i];
    return s;
}

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

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

    double valid_result = (1.0 + (double)n) * 0.5 * 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;
}
```

```
$ ./reduction
Reduction: n = 1000003
Result (scalar): 500003500006.000000 err = 0.000000
Elapsed time (scalar): 0.001031 sec.
```

Вариант 2: компенсационное суммирование Кэхэна

```
float sum(float *v, int n)
{
    float s = 0;
    for (int i = 0; i < n; i++)
        s += v[i];
    return s;
}
```

```
/*
 * Алгоритм Кэхэна (Kahan's summation) -- компенсационное
 * суммирование чисел с плавающей запятой в формате IEEE 754 [*]
 *
 * [*] Kahan W. Further remarks on reducing truncation errors //
 * Communications of the ACM - 1964 - Vol. 8(1). - P. 40.
 */
float sum_kahan(float *v, int n)
{
    float s = v[0];
    float c = (float)0.0;

    for (int i = 1; i < n; i++) {
        float y = v[i] - c;
        float t = s + y;
        c = (t - s) - y;
        s = t;
    }
    return s;
}
```

погрешность
не зависит от *n*
(только от точности float)

- **W. M. Kahan** – один из основных разработчиков IEEE 754 (Turing Award-1989, ACM Fellow) http://www.cs.berkeley.edu/~wkahan

Вариант 2: компенсационное суммирование Кэхэна

```
float sum(float *v, int n)
{
    float s = 0;
    for (int i = 0; i < n; i++)
        s += v[i];
    return s;
}
```

```
/*
 * Алгоритм Кэхэна (Kahan's summation) -- компенсационное
 * суммирование чисел с плавающей запятой в формате IEEE 754 [*]
 *
 * [*] Kahan W. Further remarks on reducing truncation errors //
 * Communications of the ACM - 1964 - Vol. 8(1). - P. 40.
 */
float sum_kahan(float *v, int n)
{
    float s = v[0];
    float c = (float)0.0;

    for (int i = 1; i < n; i++) {
        float y = v[i] - c;
        float t = s + y;
        c = (t - s) - y;
        s = t;
    }
    return s;
```

погрешность
не зависит от *n*
(только от точности float)

```
$ ./reduction
Reduction: n = 1000003
Result (scalar): 500003504128.000000 err = 0.000000
Elapsed time (scalar): 0.004312 sec.
```

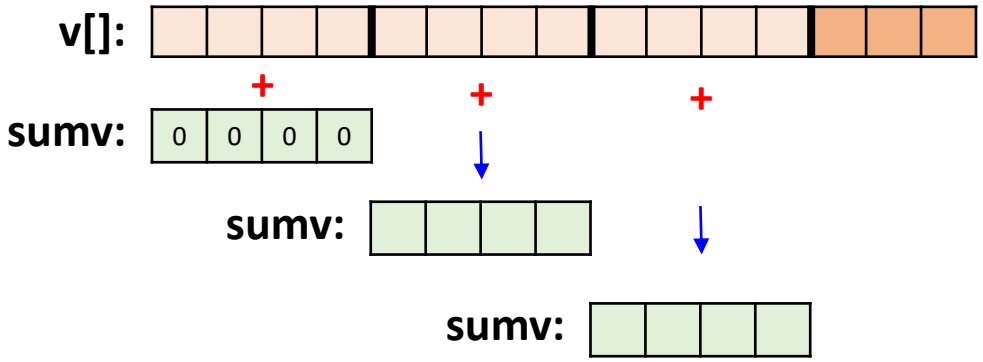
Векторная версия редукции: SSE, float

```
double run_vectorized()
{
    float *v = _mm_malloc(sizeof(*v) * n, 16);
    for (int i = 0; i < n; i++)
        v[i] = 2.0;

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

    float valid_result = 2.0 * (float)n;
    printf("Result (vectorized): %.6f err = %f\n", res, fabsf(valid_result - res));
    printf("Elapsed time (vectorized): %.6f sec.\n", t);
    free(v);
    return t;
}
```

Векторная версия редукции: SSE, float

```
float sum_sse(float * restrict v, int n)
{
    __m128 *vv = (__m128 *)v;
    int k = n / 4;
    __m128 sumv = _mm_setzero_ps();
    for (int i = 0; i < k; i++) {
        sumv = _mm_add_ps(sumv, vv[i]);
    }
```

1) Векторное суммирование (вертикальное)




```
    // s = sumv[0] + sumv[1] + sumv[2] + sumv[3]
    float t[4] __attribute__ ((aligned (16)));
    _mm_store_ps(t, sumv);
    float s = t[0] + t[1] + t[2] + t[3];

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

2) Горизонтальное суммирование

```
s = sumv[0] + sumv[1] + sumv[2] + sumv[3]
```

3) Скалярное суммирование элементов в «хвосте» массива

Векторная версия редукции: SSE, float

```
float sum_sse(float * restrict v, int n)
{
    __m128 *vv = (__m128 *)v;
    int k = n / 4;
    __m128 sumv = _mm_setzero_ps();
    for (int i = 0; i < k; i++) {
        sumv = _mm_add_ps(sumv, vv[i]);
    }


    // s = sumv[0] + sumv[1] + sumv[2] + sumv[3]
    float t[4] __attribute__ ((aligned (16)));
    _mm_store_ps(t, sumv);
    float s = t[0] + t[1] + t[2] + t[3];
```

```
# Intel Core i5-3320M – Ivy Bridge (Sandy Bridge shrink)
Reduction: n = 1000003
Result (scalar): 2000006.000000 err = 0.000000
Elapsed time (scalar): 0.001074 sec.
Result (vectorized): 2000006.000000 err = 0.000000
Elapsed time (vectorized): 0.000760 sec.
Speedup: 1.41
```

```
# Oak: Intel Xeon E5620 – Westmere (Nehalem shrink)
Reduction: n = 1000003
Result (scalar): 2000006.000000 err = 0.000000
Elapsed time (scalar): 0.001259 sec.
Result (vectorized): 2000006.000000 err = 0.000000
Elapsed time (vectorized): 0.000315 sec.
Speedup: 3.99
```

```
# cngpu1: Intel Core i5 4690 – Haswell
Reduction: n = 1000003
Result (scalar): 2000006.000000 err = 0.000000
Elapsed time (scalar): 0.003862 sec.
Result (vectorized): 2000006.000000 err = 0.000000
Elapsed time (vectorized): 0.002523 sec.
Speedup: 1.53
```

```
# cnmic: Intel Xeon E5-2620 v3 – Haswell
Reduction: n = 1000003
Result (scalar): 2000006.000000 err = 0.000000
Elapsed time (scalar): 0.001256 sec.
Result (vectorized): 2000006.000000 err = 0.000000
Elapsed time (vectorized): 0.000319 sec.
Speedup: 3.94
```

Векторная версия редукции: погрешность вычислений

```
double run_vectorized()
{
    float *v = _mm_malloc(sizeof(*v) * n, 16);
    for (int i = 0; i < n; i++)
        v[i] = i + 1.0;   // Изменили инициализацию с “2.0” на “i + 1.0”

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

    float valid_result = (1.0 + (float)n) * 0.5 * n;
    printf("Result (vectorized): %.6f err = %f\n", res, fabsf(valid_result - res));
    printf("Elapsed time (vectorized): %.6f sec.\n", t);
    free(v);
    return t;
}
```

Результаты скалярной и векторной версий не совпадают!

```
$ ./reduction
Reduction: n = 1000003
Result (scalar): 499944423424.000000 err = 59080704.000000
Elapsed time (scalar): 0.001007 sec.
Result (vectorized): 500010975232.000000 err = 7471104.000000
Elapsed time (vectorized): 0.000770 sec.
Speedup: 1.31
```

Векторная версия редукции: погрешность вычислений

- **Скалярная версия:**

```
s = v[0] + v[1] + v[2] + … + v[n - 1]
```

- **Векторная SSE-версия (float):**

```
s = (v[0] + v[4] + v[8])  +  // sumv[0]
    (v[1] + v[5] + v[9])  +  // sumv[1]
    (v[2] + v[6] + v[10]) +  // sumv[2]
    (v[3] + v[7] + v[11]) +  // sumv[3]
    v[12] + v[13] + v[14]    // «хвост»
```

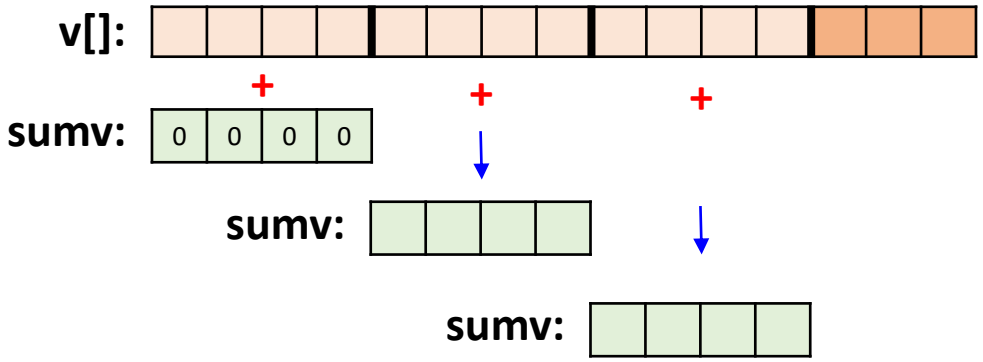
- В SSE-версии порядок выполнения операций отличается от скалярной версии

- Операция сложения чисел с плавающей запятой в формате IEEE 754 не ассоциативна и не коммутативна

$$a + b \mathrel{!}= b + a \qquad a + (b + c) \mathrel{!}= (a + b) + c$$

- David Goldberg. ***What Every Computer Scientist Should Know About Floating-Point Arithmetic*** // http://www.validlab.com/goldberg/paper.pdf

1) Векторное суммирование (вертикальное)



2) Горизонтальное суммирование

```
s = sumv[0] + sumv[1] + sumv[2] + sumv[3]
```

3) Скалярное суммирование элементов в «хвосте» массива

Векторная версия редукции: SSE, double

```
double sum_sse(double * restrict v, int n)
{
    __m128d *vv = (__m128d *)v;
    int k = n / 2;
    __m128d sumv = _mm_setzero_pd();
    for (int i = 0; i < k; i++) {
        sumv = _mm_add_pd(sumv, vv[i]);
    }
    // Compute s = sumv[0] + sumv[1] + sumv[2] + sumv[3]
    double t[2] __attribute__ ((aligned (16)));
    _mm_store_pd(t, sumv);
    double s = t[0] + t[1];

    for (int i = k * 2; i < n; i++)
        s += v[i];
    return s;
}
```

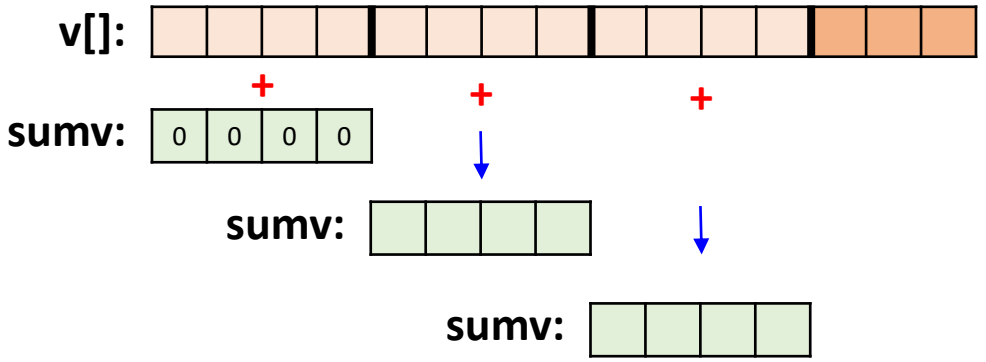
```
$ ./reduction
Reduction: n = 1000003
Result (scalar): 500003500006.000000 err = 0.000000
Elapsed time (scalar): 0.001134 sec.
Result (vectorized): 500003500006.000000 err = 0.000000
Elapsed time (vectorized): 0.001525 sec.
Speedup: 0.74
```

Векторная версия: горизонтальное суммирование SSE3

```
#include <pmmintrin.h>

float sum_sse(float * restrict v, int n)
{
    __m128 *vv = (__m128 *)v;
    int k = n / 4;
    __m128 sumv = _mm_setzero_ps();
    for (int i = 0; i < k; i++) {
        sumv = _mm_add_ps(sumv, vv[i]);
    }
```

```
    // s = sumv[0] + sumv[1] + sumv[2] + sumv[3]
    sumv = _mm_hadd_ps(sumv, sumv);
    sumv = _mm_hadd_ps(sumv, sumv);
    float s __attribute__ ((aligned (16))) = 0;
    _mm_store_ss(&s, sumv);

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

1) Векторное суммирование (вертикальное)



2) Горизонтальное суммирование SSE3

a = hadd(a, a) => a = [a3 + a2 | a1 + a0 | a3 + a2 | a1 + a0]

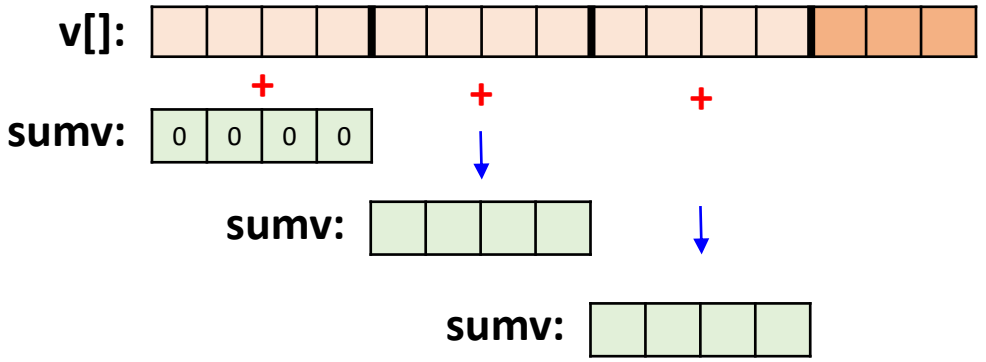
a = hadd(a, a) => a = [a3 + a2 + a1 + a0 | --/-- | --/-- | --/--]

3) Скалярное суммирование элементов в «хвосте» массива

Векторная версия: горизонтальное суммирование SSE3

```
#include <pmmintrin.h>

float sum_sse(float * restrict v, int n)
{
    __m128 *vv = (__m128 *)v;
    int k = n / 4;
    __m128 sumv = _mm_setzero_ps();
    for (int i = 0; i < k; i++) {
        sumv = _mm_add_ps(sumv, vv[i]);
    }

    // s = sumv[0] + sumv[1] + sumv[2] + sumv[3]
    sumv = _mm_hadd_ps(sumv, sumv);
    sumv = _mm_hadd_ps(sumv, sumv);
    float s __attribute__ ((aligned (16))) = 0;
    _mm_store_ss(&s, sumv);

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

1) Векторное суммирование (вертикальное)



2) Горизонтальное суммирование SSE3

a = hadd(a, a) => a = [a3 + a2 | a1 + a0 | a3 + a2 | a1 + a0]
a = hadd(a, a) => a = [a3 + a2 + a1 + a0 | --/-- | --/-- | --/--]

```
# Intel Core i5-3320M – Ivy Bridge (Sandy Bridge shrink)
Reduction: n = 1000003
Result (scalar): 499944423424.000000 err = 59080704.000000
Elapsed time (scalar): 0.001071 sec.
Result (vectorized): 500010975232.000000 err = 7471104.000000
Elapsed time (vectorized): 0.000342 sec.
Speedup: 3.13
```

Векторная версия: горизонтальное суммирование (double)

```
double sum_sse(double * restrict v, int n)
{
    __m128d *vv = (__m128d *)v;
    int k = n / 2;
    __m128d sumv = _mm_setzero_pd();
    for (int i = 0; i < k; i++) {
        sumv = _mm_add_pd(sumv, vv[i]);
    }

    // Compute s = sumv[0] + sumv[1] + sumv[2] + sumv[3]
    // SSE3 horizontal operation:
    //   hadd(a, a) => a = [a1 + a0 | a1 + a0]
    sumv = _mm_hadd_pd(sumv, sumv);
    double s __attribute__ ((aligned (16))) = 0;
    _mm_store_sd(&s, sumv);

    for (int i = k * 2; i < n; i++)
        s += v[i];
    return s;
}
```

```
# Intel Core i5-3320M – Ivy Bridge (Sandy Bridge shrink)
Reduction: n = 1000003
Result (scalar): 500003500006.000000 err = 0.000000
Elapsed time (scalar): 0.001047 sec.
Result (vectorized): 500003500006.000000 err = 0.000000
Elapsed time (vectorized): 0.000636 sec.
Speedup: 1.65
```

Векторная версия: AVX (double)

```
double run_vectorized()
{
    double *v = _mm_malloc(sizeof(*v) * n, 32);
    for (int i = 0; i < n; i++)
        v[i] = i + 1.0;

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

    double valid_result = (1.0 + (double)n) * 0.5 * n;
    printf("Result (vectorized): %.6f err = %f\n", res, fabsf(valid_result - res));
    printf("Elapsed time (vectorized): %.6f sec.\n", t);
    free(v);
    return t;
}
```

Векторная версия: AVX (double)

```
#include <immintrin.h>
double sum_avx(double * restrict v, int n)
{
    __m256d *vv = (__m256d *)v;
    int k = n / 4;
    __m256d sumv = _mm256_setzero_pd();
    for (int i = 0; i < k; i++) {
        sumv = _mm256_add_pd(sumv, vv[i]);
    }

    // Compute s = sumv[0] + sumv[1] + sumv[2] + sumv[3]
    // AVX _mm256_hadd_pd:
    //   _mm256_hadd_pd(a, a) => a = [a3 + a2 | a3 + a2 | a1 + a0 | a1 + a0]
    sumv = _mm256_hadd_pd(sumv, sumv);
    // Permute high and low 128 bits of sumv: [a1 + a0 | a1 + a0 | a3 + a2 | a3 + a2]
    __m256d sumv_permuted = _mm256_permute2f128_pd(sumv, sumv, 1);
    // sumv = [a1 + a0 + a3 + a2 | --//-- | ...]
    sumv = _mm256_add_pd(sumv_permuted, sumv);

    double t[4] __attribute__ ((aligned (16)));
    _mm256_store_pd(t, sumv);
    double s = t[0]; //double s = t[0] + t[1] + t[2] + t[3];
    for (int i = k * 4; i < n; i++)
        s += v[i];
    return s;
}
```

Векторная версия: AVX (double)

```
#include <immintrin.h>
double sum_avx(double * restrict v, int n)
{
    __m256d *vv = (__m256d *)v;
    int k = n / 4;
    __m256d sumv = _mm256_setzero_pd();
    for (int i = 0; i < k; i++) {
        sumv = _mm256_add_pd(sumv, vv[i]);
    }

    // Compute s = sumv[0] + sumv[1] + sumv[2] + sumv[3]
    // AVX _mm256_hadd_pd:
    //   _mm256_hadd_pd(a, a) => a = [a3 + a2 | a3 + a2 | a1 + a0 | a1 + a0]
    sumv = _mm256_hadd_pd(sumv, sumv);
    // Permute high and low 128 bits of sumv: [a1 + a0 | a1 + a0 | a3 + a2 | a3 + a2]
    __m256d sumv_permuted = _mm256_permute2f128_pd(sumv, sumv, 1);
    // sumv = [a1 + a0 + a3 + a2 | --//-- | ...]
    sumv = _mm256_add_pd(sumv_permuted, sumv);

    double t[4] __attribute__ ((aligned (16)));
    _mm256_store_pd(t, sumv);
    double s = t[0]; //double s = t[0] + t[1] + t[2]
    for (int i = k * 4; i < n; i++)
        s += v[i];
    return s;
}
```

```
# Intel Core i5-3320M – Ivy Bridge (Sandy Bridge shrink)
Reduction: n = 1000003
Result (scalar): 500003500006.000000 err = 0.000000
Elapsed time (scalar): 0.001061 sec.
Result (vectorized): 500003500006.000000 err = 0.000000
Elapsed time (vectorized): 0.000519 sec.
Speedup: 2.04
```

Задание

- Векторизовать вычисление скалярного произведения векторов (dot product)
- Шаблон находится в каталоге _dot_product_task

```
float sdot(float *x, float *y, int n)
{
    float s = 0;
    for (int i = 0; i < n; i++)
        s += x[i] * y[i];
    return s;
}
```

- Варианты решений:
  1) Loop: _mm_mul_ps + 2 x _mm_hadd_ps
  2) Loop: _mm_mul_ps + _mm_had_ps + _mm_add_ps
  3) Loop: _mm_dp_ps(x, y, 0xff)