Векторизация вычислений. Векторные регистры

Векторизация

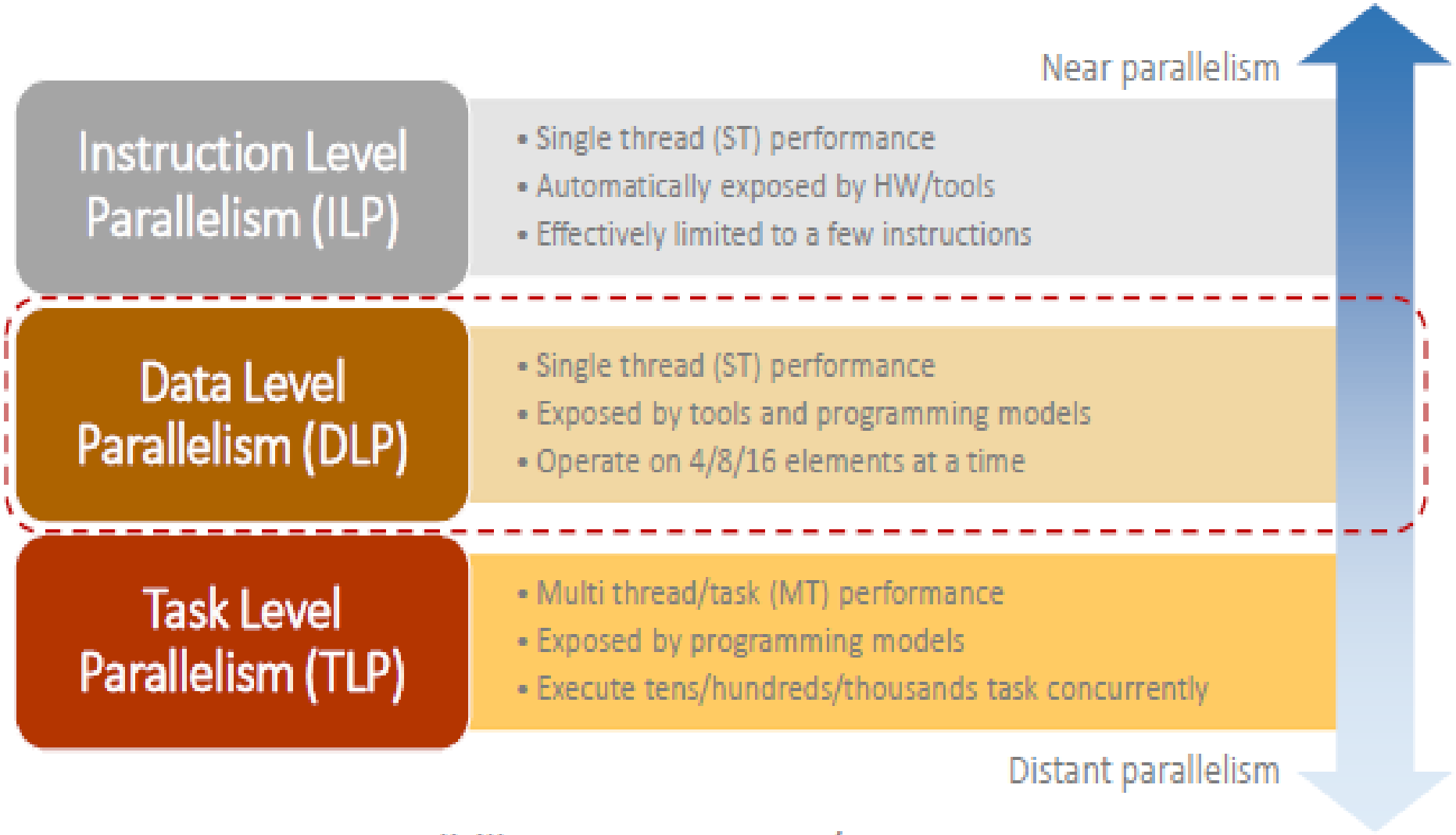
SIMD – одна инструкция для вектора данных

- Скалярное выполнение
  - x86/x87/SSE
  - одна инструкция дает один результат
- Векторное выполнение
  - SSE или AVX инструкции
  - одна инструкция дает вектор результатов

```
for (i=0;i<=MAX;i++)
    c[i]=a[i]+b[i];
```

Exploiting the parallel universe

Three levels of parallelism supported by Intel hardware



Programmers responsibility to expose DLP/TLP

Больше ядер, шире вектор

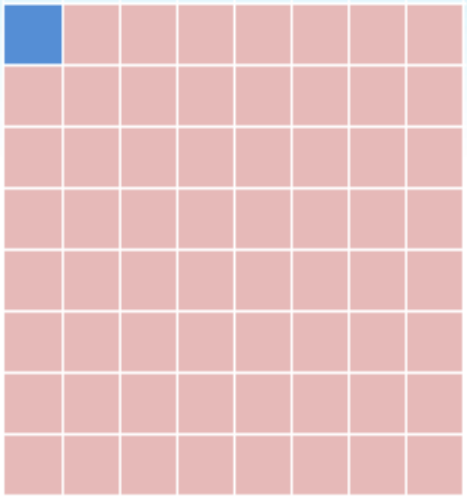
Images do not reflect actual die sizes

	Intel® Xeon® processor 64-bit	Intel® Xeon® processor 5100 series	Intel® Xeon® processor 5500 series	Intel® Xeon® processor 5600 series	Intel® Xeon® processor code-named Sandy Bridge	Intel® Xeon® processor code-named Ivy Bridge	Intel® Xeon® processor code-named Haswell	Intel® future processor	Intel® Xeon Phi co-processor
Ядра	1	2	4	6	8	10	12		>60
Потоки	2	2	8	12	16	20	24		>240
Ширина SIMD	128	128	128	128	256	256	256	512	512
	SSE2	SSSE3	SSE4.2	SSE4.2	AVX	AVX	AVX2 FMA3 TSX	AVX512	

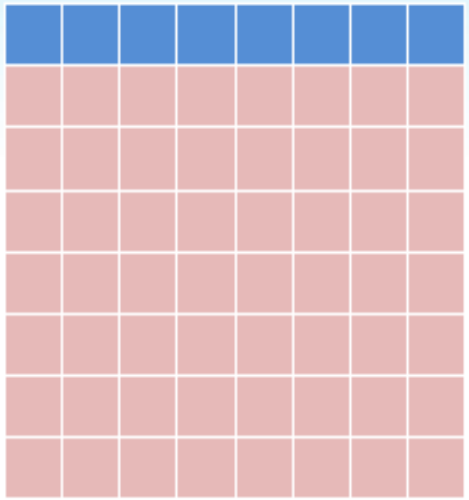
Максимум производительности

Где он?

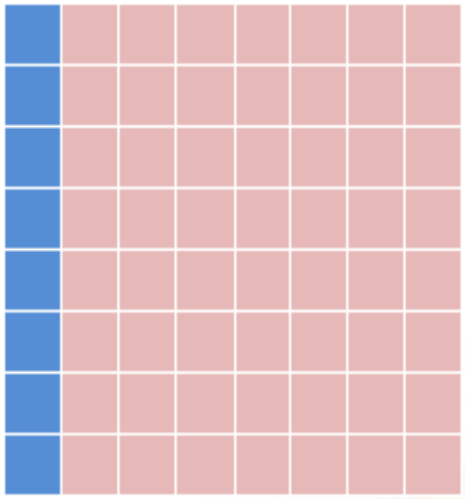
Скалярный код



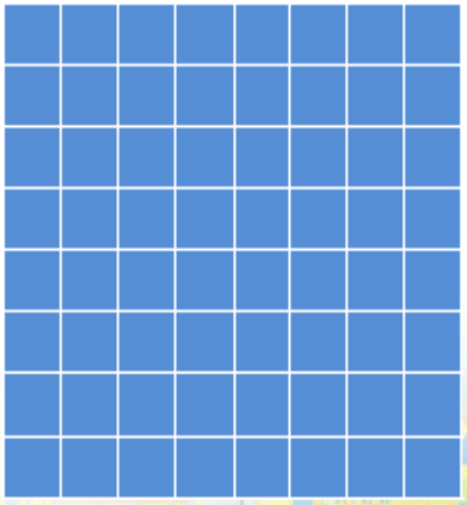
Векторный код



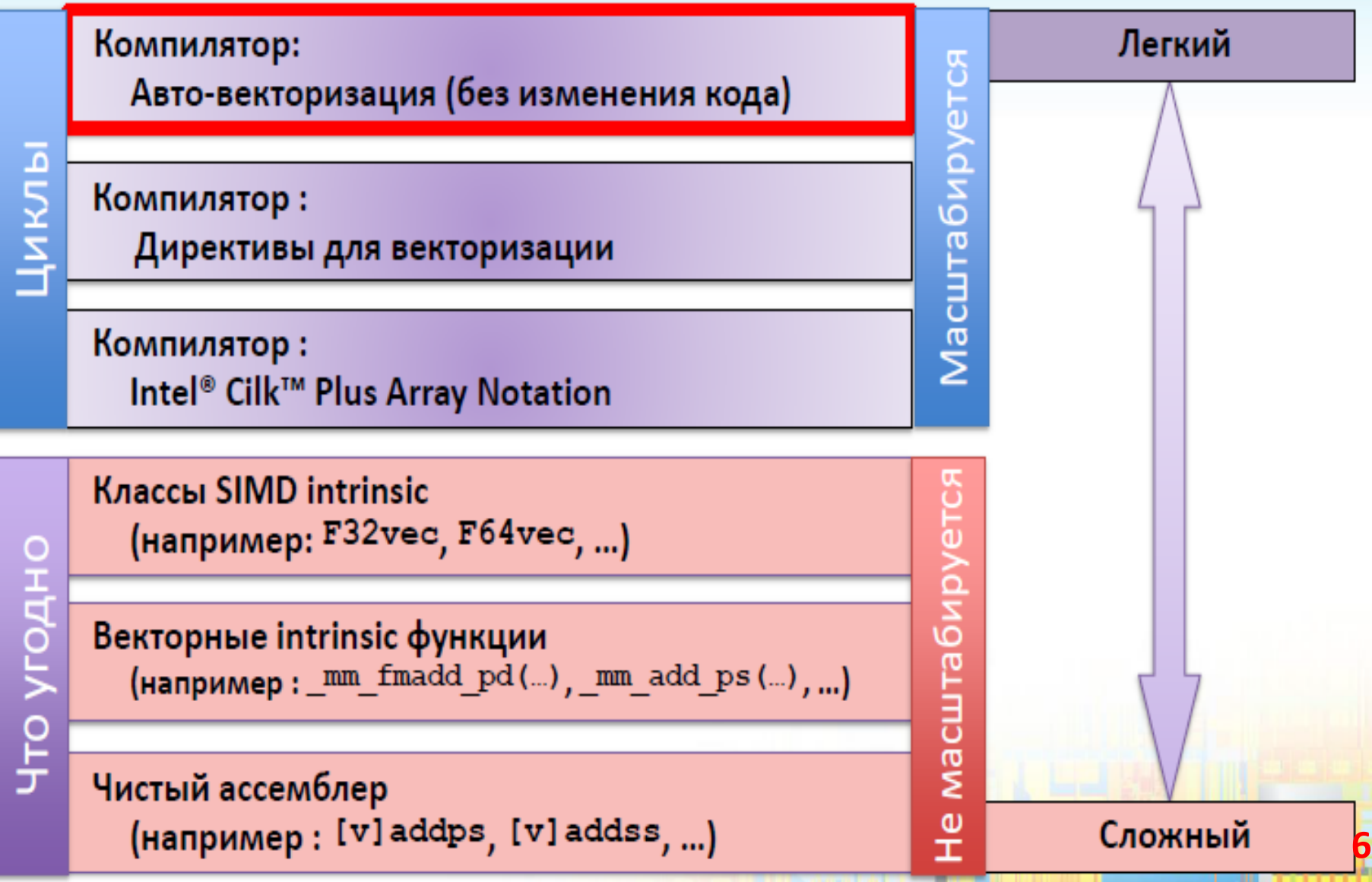
Параллельный код



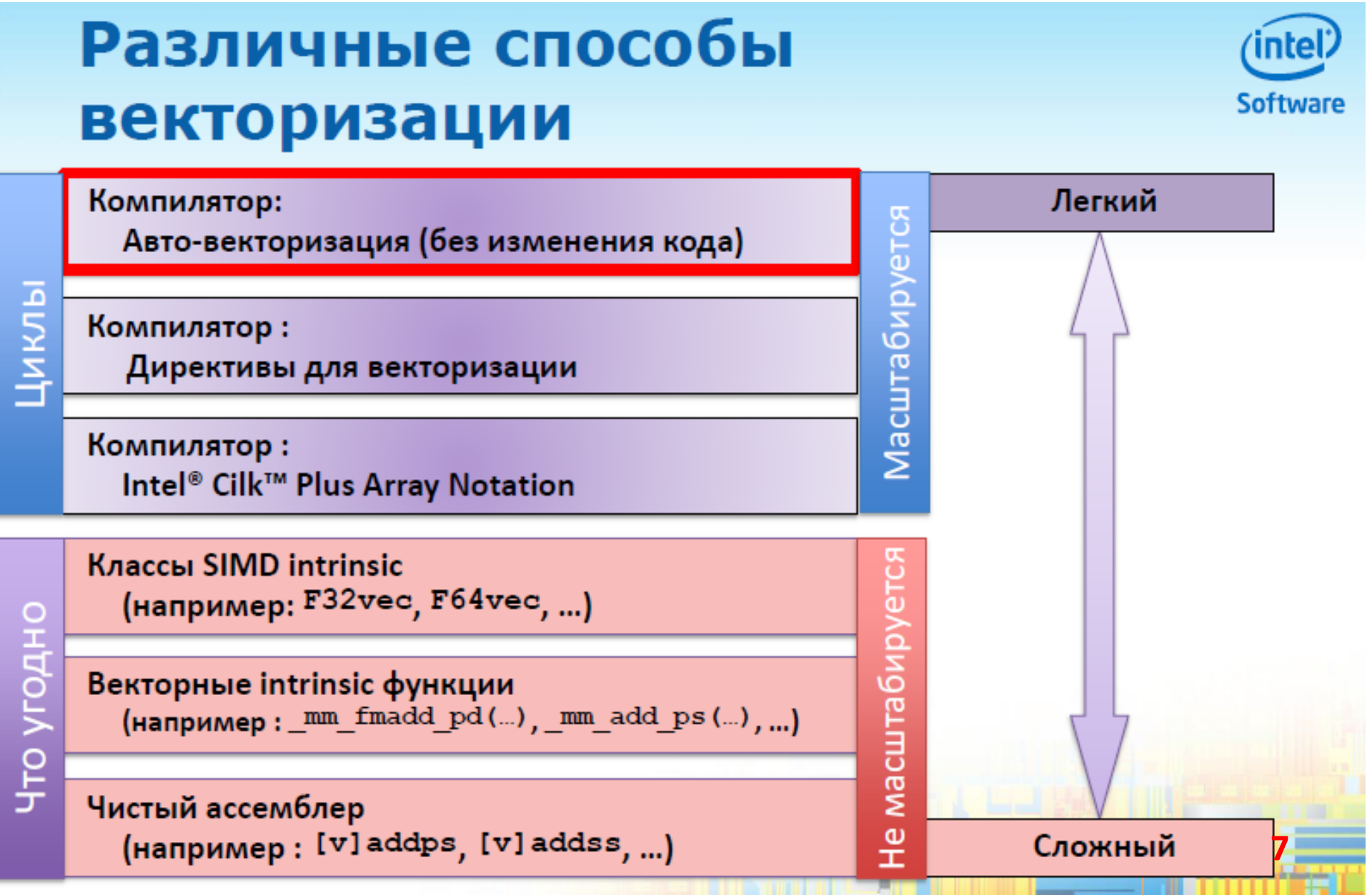
Вместе - лучше



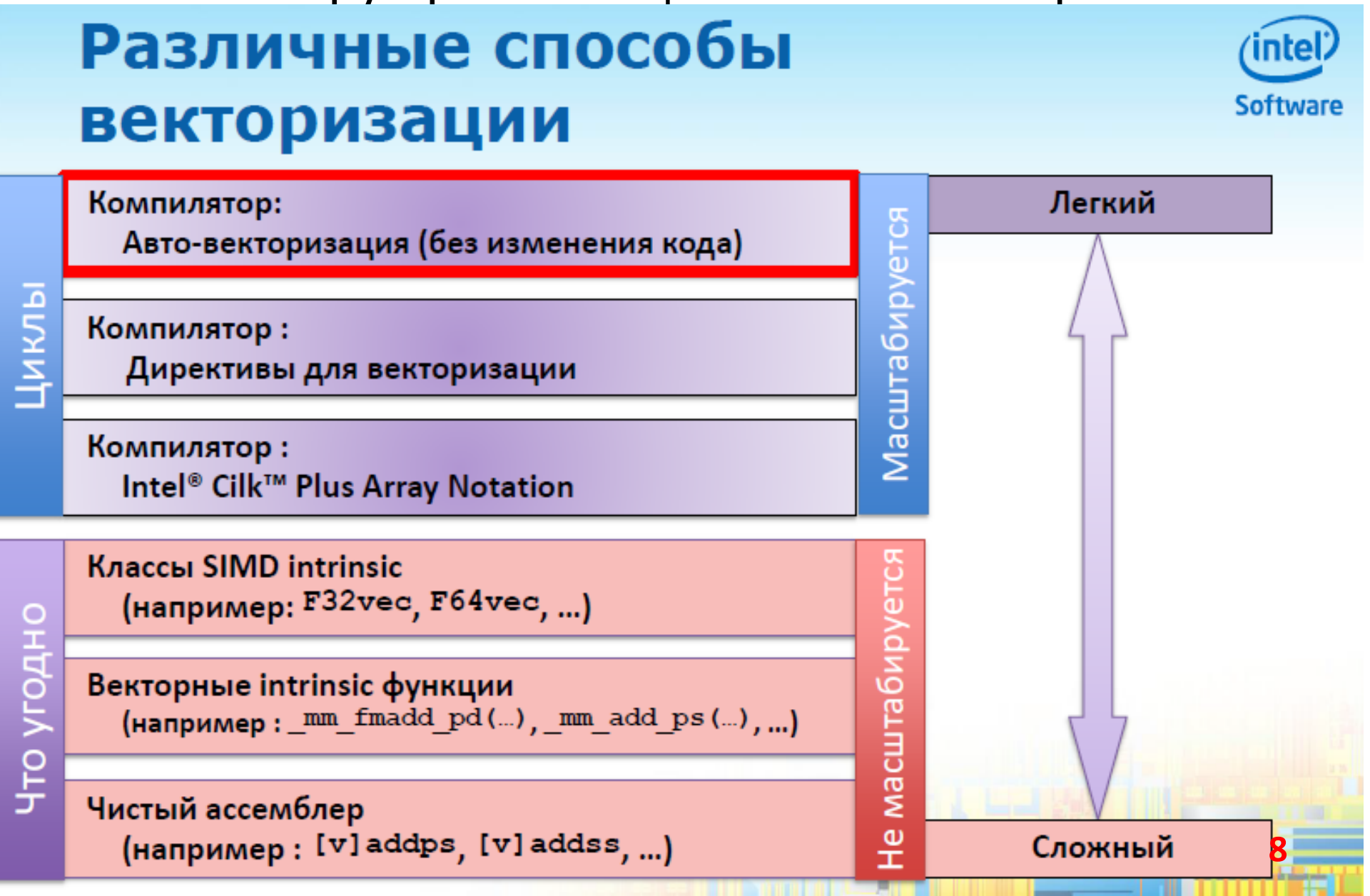
Различные способы векторизации



- **Автовекторизация** – компилятор генерирует векторизованный код сам



- **Директивы для векторизации** – «подсказки» компилятору при помощи библиотеки OpenMP



- **Директивы для векторизации** – «подсказки» компилятору при помощи библиотеки OpenMP

```
void addfl(float *a, float *b, float *c, float *d, float *e, int n)
{
#pragma omp simd
for(int i = 0; i < n; i++)
    a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
}
```

Intel® Cilk™ Plus – Array notation

Software

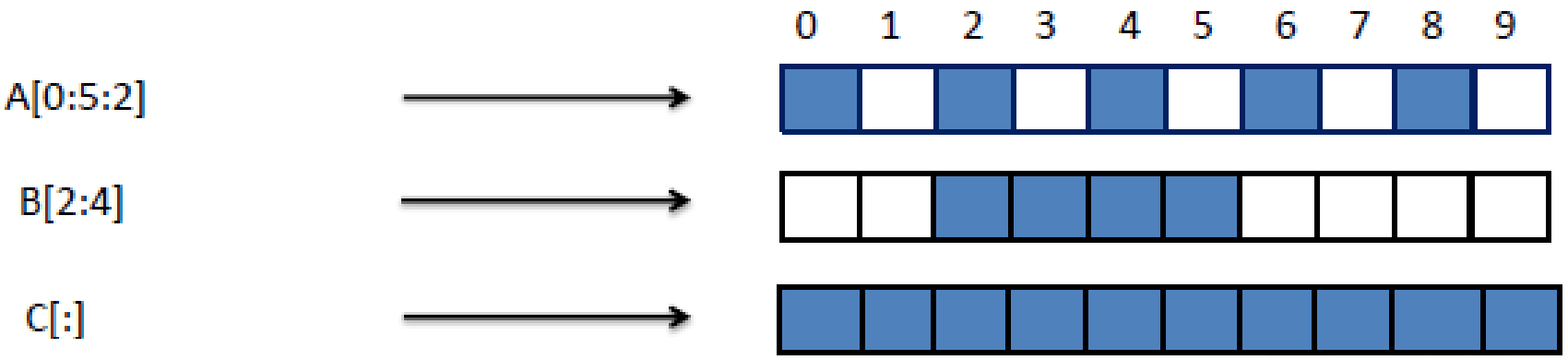
Указатель или массив

```
base[first:length:stride]
```

Начальный индекс

Число элементов

Шаг (опционален)




```
int A[10], *B;
A[1:5:2] = B[1:5];
```

```
for (i = 0; i < 5; i++)
  A[(i*2) + 1] = B[(i*1) + 1];
```

Ясный синтаксис для простой векторизации

Assembler-ные вставки в C-код

test.c:

```
int main() {
    asm (“abc de f");  // вставим такую инструкцию
    return 0;
}
```

```
$gcc -S -o test.S test.c
$cat test.S
```

Вывод

GCC никак не интерпретирует содержимое ассемблерной вставки!

Общая структура ассемблерной вставки:

asm [volatile] («команды и директивы ассемблера»

: выходные параметры

: входные параметры

: изменяемые параметры/регистры);

Существует и более короткая форма:

asm [volatile] («команды ассемблера»);

Пример1:

```
asm ( "movl $10, %eax;”
      "movl $20, %ebx;”
      "addl %ebx, %eax;" );
```

Пример2:

```
asm ("movl %0,%%eax"::"i"(1));
```

превратится в

```
movl $1,%eax
```

"i" – ограничитель, указывающий, что в качестве параметра выступает константа

Пример 3:

```
int no = 100, val ;
asm ("movl %1, %%ebx;"
     "movl %%ebx, %0;"
     : "=r" ( val ) /* output */
     : "r" ( no ) /* input */
     : "%ebx" /* clobbered register */ );
```

“r” – ограничитель, указывающий, что переменные *val* и *no* требуется сохранить в каких-либо регистрах общего назначения

“=r” – используется в списке выходных параметров (регистр должен использоваться «write only»)

Пример 4:

```
asm ("leal (%1,%1,4), %0"
        : "=r" (five_times_x)
        : "r" (x) );
```

Вопрос: *что сделает данная вставка?*

Ограничители для конкретных регистров

r	Register(s)
a	%eax, %ax, %al
b	%ebx, %bx, %bl
c	%ecx, %cx, %cl
d	%edx, %dx, %dl
S	%esi, %si
D	%edi, %di

Пример 5:

```
asm volatile ("mov %0, %%eax"::"m"(a));
```

“m” – ограничитель, указывающий, что переменная a находится в памяти

Ключевое слово «volatile»

Слово **volatile** служит для того чтобы указать компилятору, что вставляемый ассемблерный код может давать побочные эффекты, поэтому попытки оптимизации могут привести к логическим ошибкам.

СОВЕТ: Всегда указывайте **asm volatile** в тех случаях, когда ваша ассемблерная вставка должна «стоять там, где стоит».

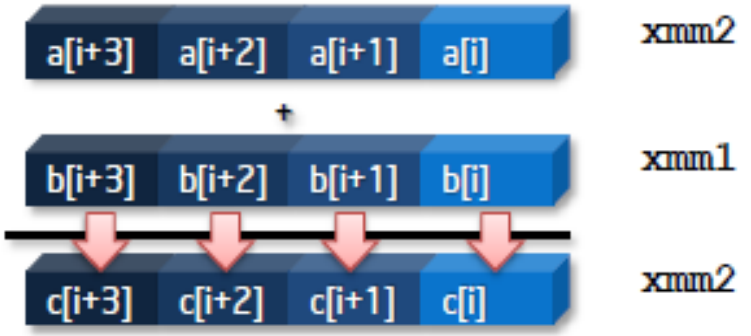
SSE: Packed vs. Scalar

- **"Упакованные" (packed)** инструкции оперируют со всеми элементами вектора
- Большинство таких инструкций имеют "**скалярную" (scalar)** версию, работающую только с одним элементом
- Для использования SIMD возможностей необходимо избегать использования скалярных версий инструкций

Packed single-precision FP Addition:

```
addps xmm2, xmm1
```

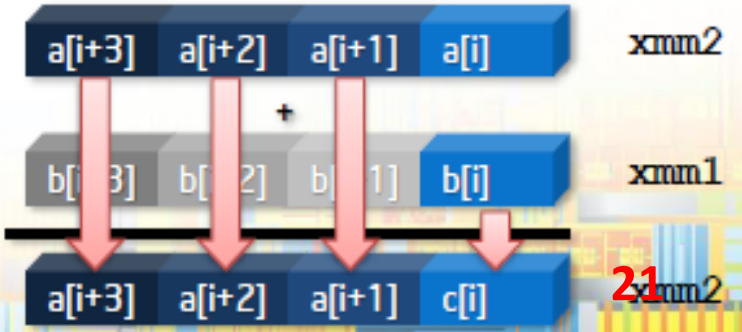
single-precision FP data type
packed execution mode



Scalar single-precision FP Addition:

```
addss xmm2, xmm1
```

single-precision FP data type
scalar execution mode



SSE-инструкции сложения

- **addss xmm0, xmm1** - scalar single precision floating-point value
- **addps xmm0, xmm1** - packed single precision floating-point values
- **addsd xmm0, xmm1** - scalar double precision floating-point value
- **addpd xmm0, xmm1** - packed double precision floating-point values
- **paddd xmm0, xmm1** - packed double word (32-битных целочисленных) values

SSE-инструкции копирования

- **movaps**—Move Aligned Packed Single-Precision Floating-Point
- **movapd**—Move Aligned Packed Double-Precision Floating-Point
- **movups**—Move Unaligned Packed Single-Precision Floating-Point
- **movupd**—Move Unaligned Packed Double-Precision Floating-Point

Примечание: intrinsics и assembler

- **_mm_load_ps** соответствует **movaps** – *из памяти в регистр*

- **_mm_store_ps** соответствует **movaps** – *из регистра в память*

Переход от SSE к AVX

- 2 режима:
  - **VEX.256**:
    Операции с 256-битными векторам
  - **VEX.128**:
    Операции с младшими 128 битами вектора
    Старшие 128 бит обнуляются

VEX.256:

```
vaddps ymm1, ymm2, ymm3
```

VEX.128:

```
vaddps xmm1, xmm2, xmm3
vaddss xmm1, xmm2, xmm3
```

v = VEX encoding (AVX)
3 операнда в иснтрукциях!

- Переход от SSE:
  - Практически все SSE инструкции доступны как VEX.128 версии, например:
    addps/addss: vaddps/vaddss xmm1, xmm2, xmm3
  - Некторые из них работают в режиме VEX.256:
    vaddps ymm1, ymm2, ymm3
  - Скалярных версий SSE инструкций в VEX.256 режиме **нет**:
    ~~**vaddss ymm1, ymm2, ymm3**~~
  - Совместное использование SSE & AVX инструкция возможно, но влечёт потерю производительности (**state change penalty**); избегайте такое использование!
- AVX использует трёх- и четырёх-операндные инструкции!

```
__asm__ volatile
(
 "movups %[a], %%xmm0\n\t"        // поместить 4 переменные с
   плавающей точкой из a в регистр xmm0
 "movups %[b], %%xmm1\n\t"        // поместить 4 переменные с
   плавающей точкой из b в регистр xmm1
 "mulps %%xmm1, %%xmm0\n\t"  // перемножить пакеты
   плавающих точек: xmm0 = xmm0 * xmm1
                                  // xmm00 = xmm00 * xmm10
                                  // xmm01 = xmm01 * xmm11
 "movups %%xmm0, %[a]\n\t"        // выгрузить результаты из
   регистра xmm0 по адресам a
 :  : [a]"m"(*a), [b]"m"(*b)
 : "%xmm0", "%xmm1" );
```

Intel Intrinsics Guide

- Лабораторная работа 7
  - Справочники по intrinsics: SSE intrinsics, Intel Intrinsics Guide, MSDN: Compiler intrinsics
  - Пример использования процедуры SGEMM библиотеки BLAS в программе на Си

Можем смотреть соответствие команд assembler-а из расширений SSE и AVX в сопоставлении с Intrinsic-функциями.

Лучший учебник по инструкциям assembler?

AMD64 Architecture

Programmer's Manual

Volume 4:

128-Bit and 256-Bit

Media Instructions

https://www.amd.com/system/files/TechDocs/26568.pdf