Архитектура памяти

Февраль, 2022

План

- Проблема роста производительности ЭВМ
- Иерархия памяти, что такое память
- Метрики кэш-памяти
- Устройство кэш-памяти
- Оптимизация под кэш
- Пример кода обработки изображений
- Аппаратная оптимизация подсистемы памяти

ПРОБЛЕМА РОСТА ПРОИЗВОДИТЕЛЬНОСТИ ЭВМ

	ЭВМ золотой эры 1980 г. (Herb Sutter. **Machine Architecture**)	Современные ЭВМ (DDR4) 2022 г.	Улучшение
Частота CPU (MHz)	6	>6000	**x 1000**
Размер (Мб)	2	$>3*10^6$	**x 1.5 млн.**
Пропускная способность (Мб/сек)	13	>26000	**+2000x**
Латентность (наносекунды)	225	<20	**+10x**
Латентность (такты)	1.4	36	**-25x**

Главная проблема – **“стена памяти”**

При увеличении объемов и характеристик, память неизбежно отстает от процессора по производительности.

ТИПИЧНЫЕ ХАРАКТЕРИСТИКИ И УРОВНИ ИЕРАРХИИ ПАМЯТИ

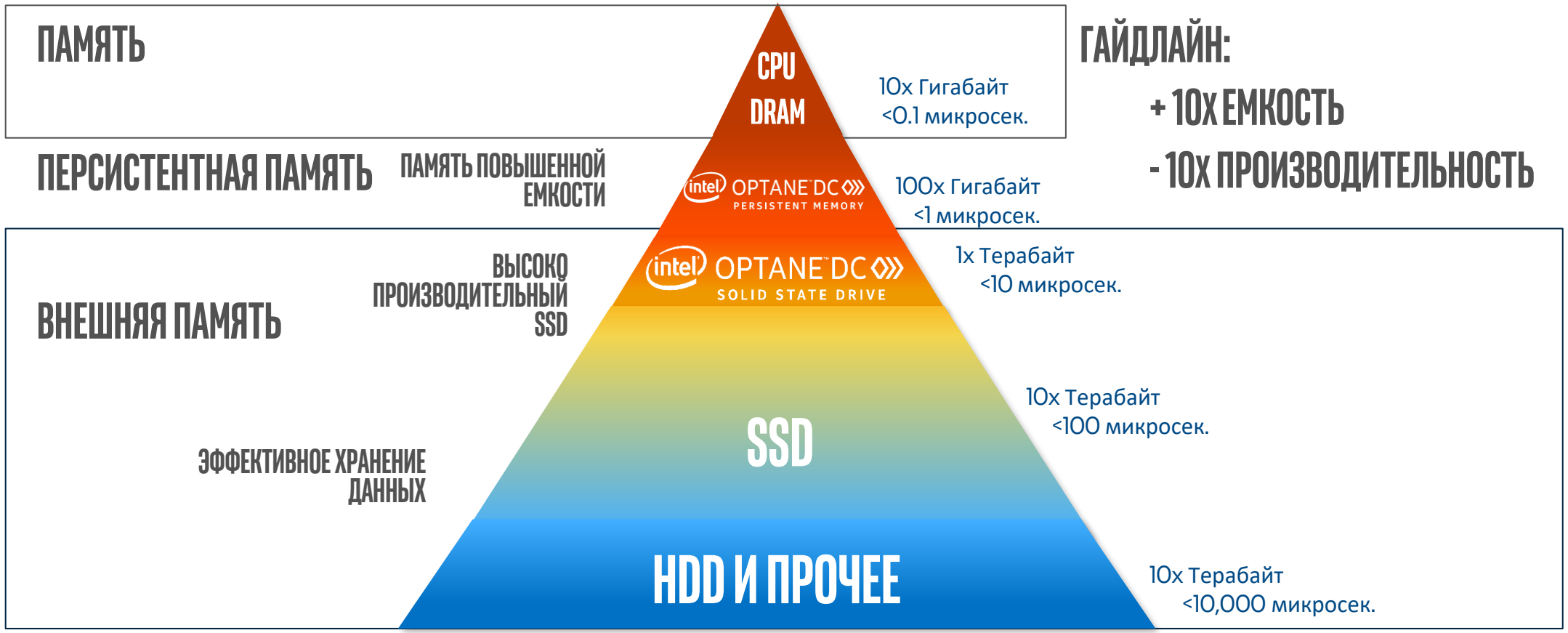
Регистры
- **Латентность**: 0 тактов
- **Память**: 32 x 512 бит SIMD (до AVX512)

L1/L2/L3 Кэш
- **Латентность** : 1-40 тактов
- **Память**: 32Кб - 32Мб

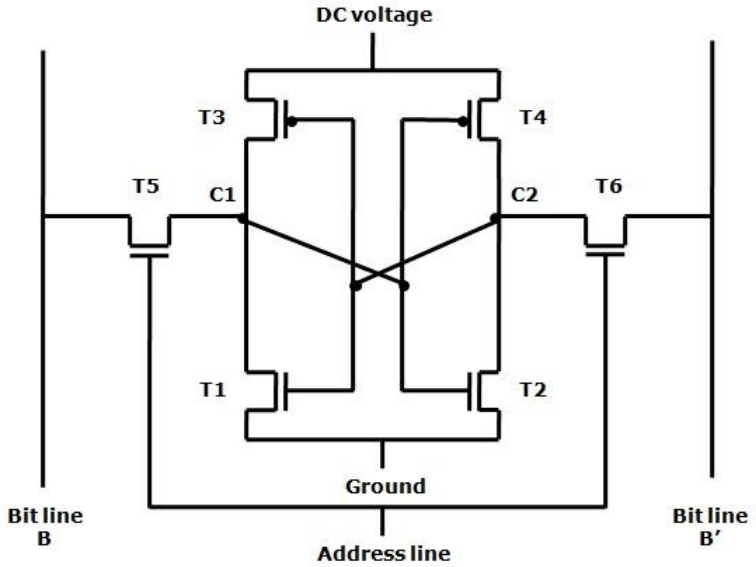
Оперативная память (RAM)
- **Латентность** : ~ 30-100 тактов
- **Память**: GB-TB

Жесткие диски
- Solid-State Disk (SSD):
  - **Латентность** : 0.1 ms (~ 300k тактов)
  - **Память** : 128 GB – 2 TB
- Hard Disk (HDD):
  - **Латентность** : 10 ms (~ 30M тактов)
  - **Память** : 1 – 10 TB

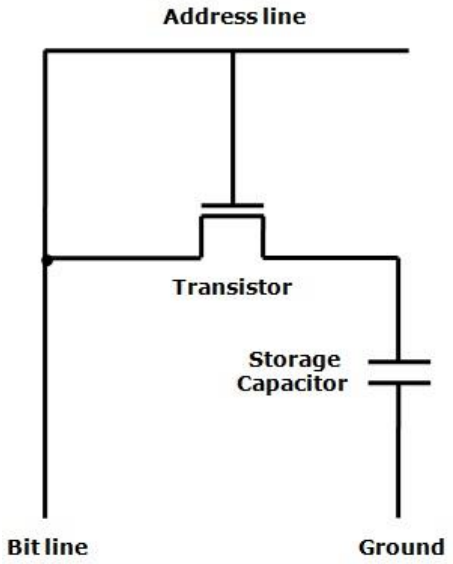
ТИПИЧНЫЕ ХАРАКТЕРИСТИКИ И УРОВНИ ИЕРАРХИИ ПАМЯТИ



ЗАЧЕМ НУЖНА ИЕРАРХИЯ



Static RAM (SRAM) Cell



Dynamic RAM (DRAM) Cell

КЭШ - Статическая память

+ Быстрая
- Большой физический размер
- Дорогая
- Сложная для проектирования

DRAM – Днамическая память

- Медленная
+ Малый размер
+ Дешевая
+ Простая при проектировании

ЗАЧЕМ НУЖНА ИЕРАРХИЯ

ЛОКАЛЬНОСТЬ ССЫЛОК

- **Локальность ссылок (locality of reference)** –
  свойство программ повторно (часто) обращаться к одним и тем же адресам в памяти (данным, инструкциям)
- **Типы локальности ссылок:**

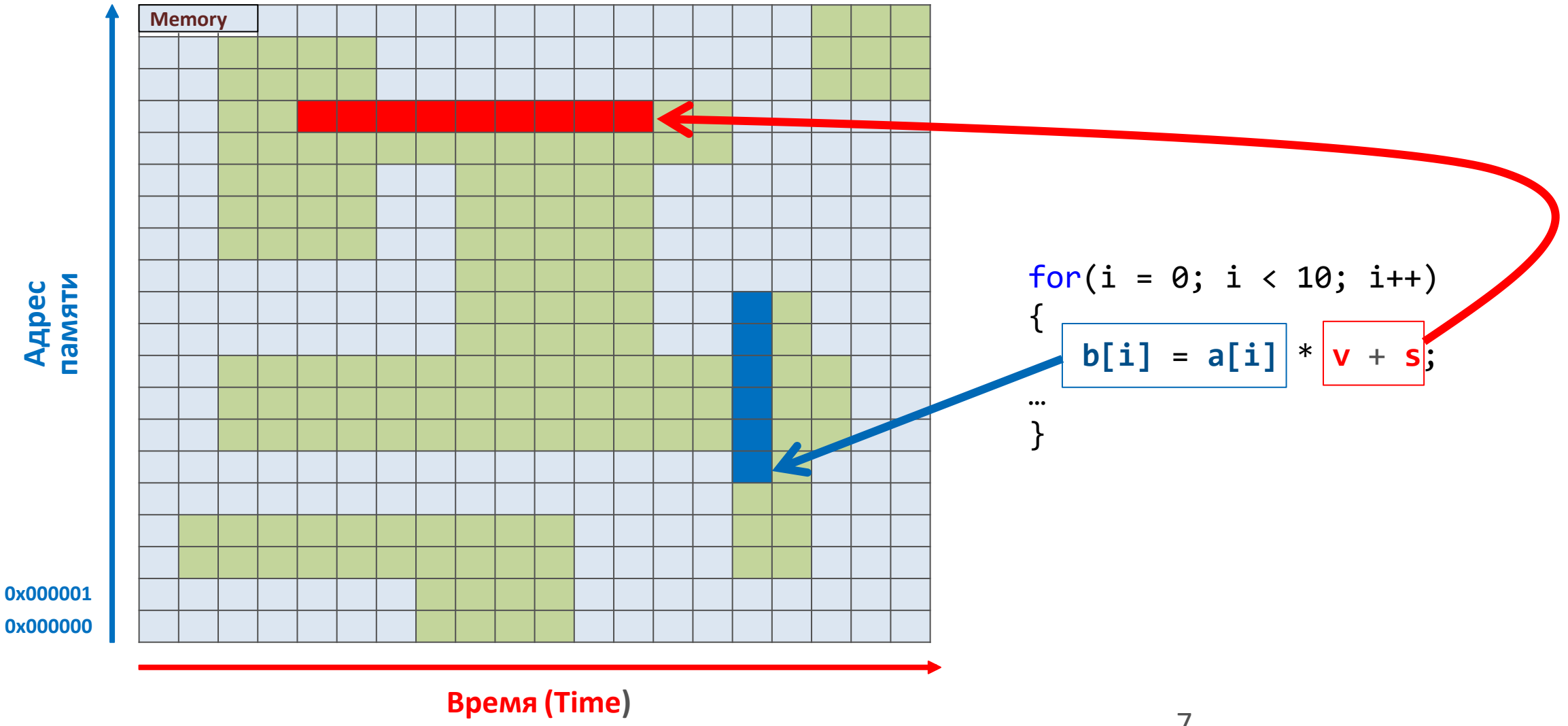
  1. Временная локальность (temporal locality) –
  повторное обращение к одному и тому же адресу через короткий промежуток времени (например, в цикле)

  2. Пространственная локальность (spatial locality) –
  свойство программ повторно обращаться через короткий промежуток времени к адресам близко расположенным в памяти друг к другу

Какие технологии используют локальность ссылок:

1. Кэш-память
2. SIMD-регистры
3. Страничная орагнизация памяти в ОС

ЛОКАЛЬНОСТЬ ССЫЛОК



МЕТРИКИ КЭШ ПАМЯТИ

Попадание – CPU находит запрошенный адрес в кэше

Промах – CPU не находит данные в кэше, поиск в следующем уровне памяти

Важная метрика - вероятность промаха (типовое соотношение промахов ко всем обращениям к кешу).

Штрафное время – накладные расходы при кэш-промахе

Время попадания – время доставки данных процессору в случае опадания, включая поиск в кэше

Поиск в кэшах идет последовательно
– L1 промах
– L2 промах
– L3 промах
– ОЗУ

Вероятность промаха

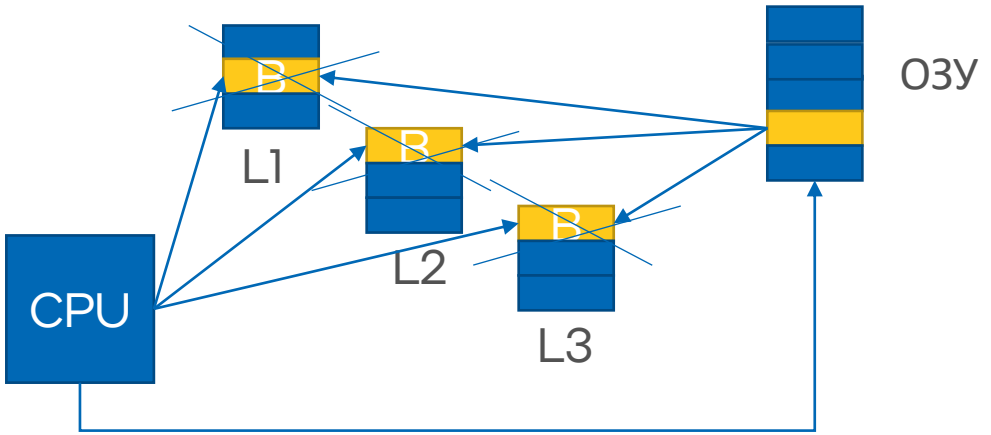
- 3 - 10% для L1
- Для L2 может быть очень мала < 1% (зависит от задачи)

Время попадания

- 1 - 2 такта для L1
- 5 - 20 тактов для L2
- 50 -70 тактов для L3

Штрафное время

- 50-200 тактов для оперативной памяти (тенденция к цвеличению)



Почему вероятность прома важнее попадания?

- Большая разница между числом промахов и попаданий
  - В 100 раз только для L1
  - 99% попаданий в 2 раза лучше, чем 97%?

    Время попадания – 1 такт,

    Штрафное время - 100 тактов

  Среднее время доступа:

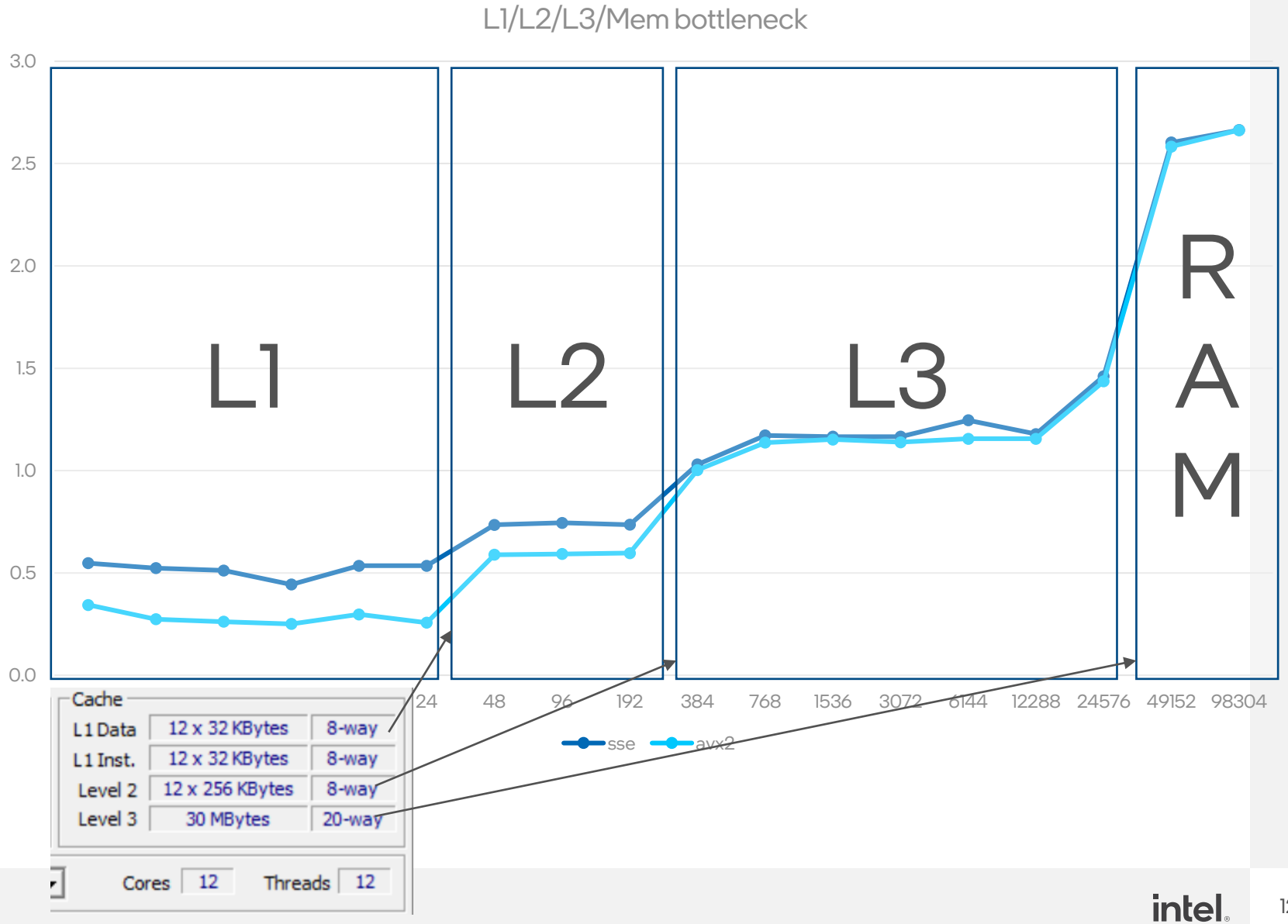
  97% попаданий: 0.97 * 1 такт + 0.03 * 100 тактов= 4 такта

  99% попаданий: 0.99 * 1 такт + 0.01 * 100 тактов = 2 такта

КОГДА КЭШ ЯВЛЯЕТСЯ УЗКИМ МЕСТОМ

```
for (i = 0; i < len; i+=4) {
    __m128 x0, x1, x2;
    x0 = _mm_loadu_ps(pSrc0 + i);
    x1 = _mm_loadu_ps(pSrc1 + i);
    x2 = _mm_add_ps(x0, x1);
    _mm_storeu_ps(pDst+i, x2);
}
```

len	size(Kb)	cpe		
		sse	avx2	ratio
64	0.75	0.5	0.3	1.6
128	1.5	0.5	0.3	1.9
256	3	0.5	0.3	2.0
512	6	0.4	0.3	1.8
1024	12	0.5	0.3	1.8
2048	24	0.5	0.3	2.1
4096	48	0.7	0.6	1.2
8192	96	0.7	0.6	1.3
16384	192	0.7	0.6	1.2
32768	384	1.0	1.0	1.0
65536	768	1.2	1.1	1.0
131072	1536	1.2	1.2	1.0
262144	3072	1.2	1.1	1.0
524288	6144	1.2	1.2	1.1
1048576	12288	1.2	1.2	1.0
2097152	24576	1.5	1.4	1.0
4194304	49152	2.6	2.6	1.0
8388608	98304	2.7	2.7	1.0
16777216	196608	2.7	2.7	1.0



КОГДА КЭШ НЕ ЯВЛЯЕТСЯ УЗКИМ МЕСТОМ

```
for (i = 0; i < len; i+=4) {
    __m128 x0, x1, x2;
    x0 = _mm_loadu_ps(pSrc0 + i);
    x1 = _mm_loadu_ps(pSrc1 + i);
    x0 = _mm_add_ps(x0, x1);
    x0 = _mm_add_ps(x0, x1);
    x0 = _mm_add_ps(x0, x1);
    x0 = _mm_add_ps(x0, x1);
    x0 = _mm_add_ps(x0, x1);
    x0 = _mm_add_ps(x0, x1);
    x0 = _mm_add_ps(x0, x1);
    x0 = _mm_add_ps(x0, x1);
    x0 = _mm_add_ps(x0, x1);
     .....
    x0 = _mm_add_ps(x0, x1);
    x0 = _mm_add_ps(x0, x1);
    x0 = _mm_add_ps(x0, x1);
    x0 = _mm_add_ps(x0, x1);
    x0 = _mm_add_ps(x0, x1);
    x0 = _mm_add_ps(x0, x1);
    x0 = _mm_add_ps(x0, x1);
    x0 = _mm_add_ps(x0, x1);
    x0 = _mm_add_ps(x0, x1);
    x0 = _mm_add_ps(x0, x1);
    x0 = _mm_add_ps(x0, x1);
    x0 = _mm_add_ps(x0, x1);
    x0 = _mm_add_ps(x0, x1);
    _mm_storeu_ps(pDst+i, x0);
}
```

```
for (i = 0; i < len; i+=8) {
    __m256 x0, x1, x2;
    x0 = _mm256_loadu_ps(pSrc0 + i);
    x1 = _mm256_loadu_ps(pSrc1 + i);
    x0 = _mm256_add_ps(x0, x1);
    x0 = _mm256_add_ps(x0, x1);
    x0 = _mm256_add_ps(x0, x1);
    x0 = _mm256_add_ps(x0, x1);
    x0 = _mm256_add_ps(x0, x1);
    x0 = _mm256_add_ps(x0, x1);
    x0 = _mm256_add_ps(x0, x1);
    x0 = _mm256_add_ps(x0, x1);
    x0 = _mm256_add_ps(x0, x1);
     .....
    x0 = _mm256_add_ps(x0, x1);
    x0 = _mm256_add_ps(x0, x1);
    x0 = _mm256_add_ps(x0, x1);
    x0 = _mm256_add_ps(x0, x1);
    x0 = _mm256_add_ps(x0, x1);
    x0 = _mm256_add_ps(x0, x1);
    x0 = _mm256_add_ps(x0, x1);
    x0 = _mm256_add_ps(x0, x1);
    x0 = _mm256_add_ps(x0, x1);
    x0 = _mm256_add_ps(x0, x1);
    x0 = _mm256_add_ps(x0, x1);
    x0 = _mm256_add_ps(x0, x1);
    x0 = _mm256_add_ps(x0, x1);
    _mm256_storeu_ps(pDst+i, x0);
}
```

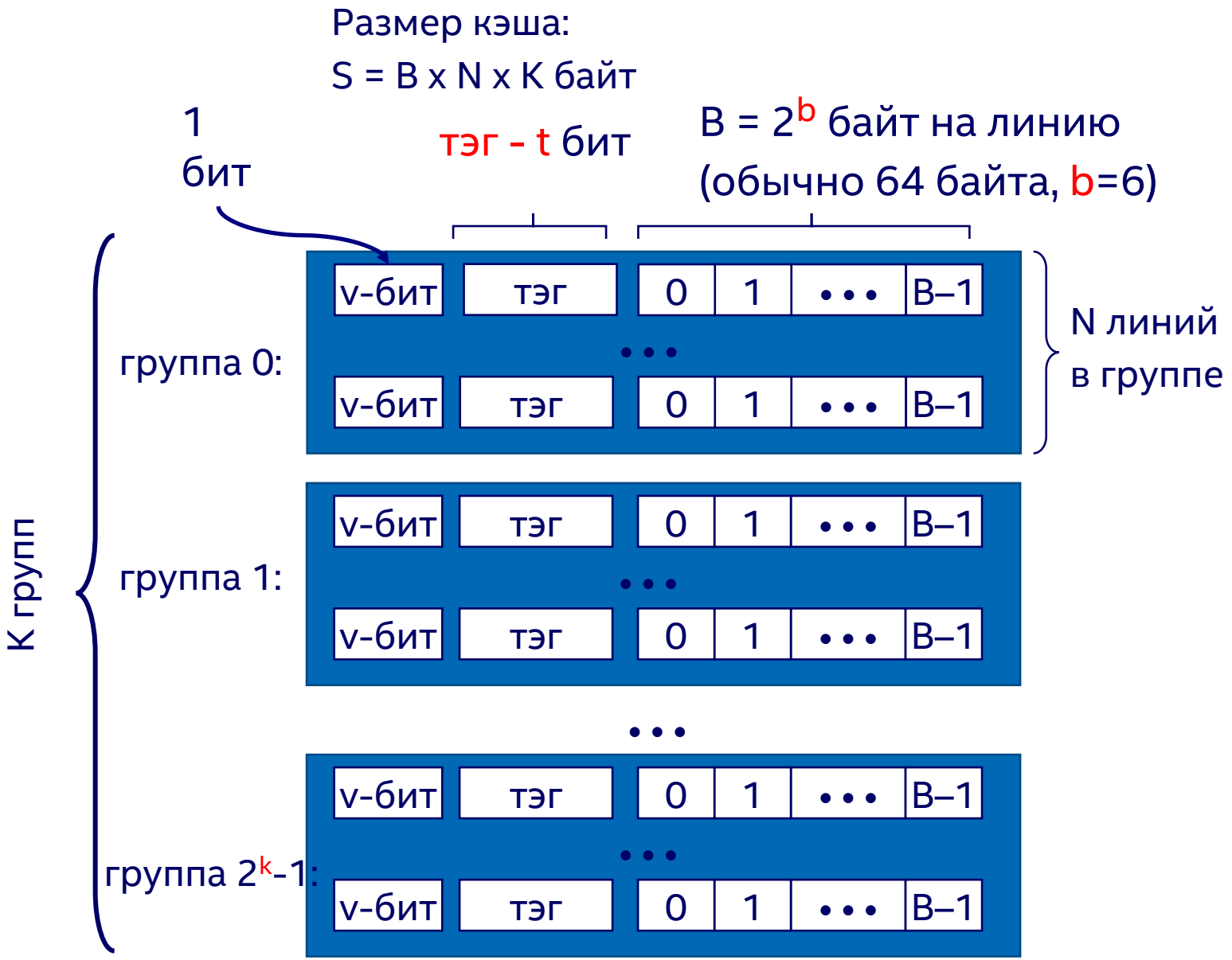
		sse	avx2	
len	size(Kb)	cpe	cpe	ratio
64	0.75	7.508	3.881	1.93
128	1.5	7.54	3.813	1.98
256	3	7.525	3.832	1.96
512	6	7.515	3.797	1.98
1024	12	7.576	3.806	1.99
2048	24	7.526	3.765	2.00
4096	48	7.567	3.785	2.00
8192	96	7.558	3.77	2.00
16384	192	7.534	3.77	2.00
32768	384	7.536	3.773	2.00
65536	768	7.534	3.768	2.00
131072	1536	7.534	3.768	2.00
262144	3072	7.566	3.774	2.00
524288	6144	7.58	3.823	1.98
1048576	12288	7.58	3.822	1.98
2097152	24576	7.596	3.862	1.97
4194304	49152	7.813	4.11	1.90
8388608	98304	7.825	4.127	1.90
16777216	196608	7.821	4.131	1.89

УСТРОЙСТВО КЭША

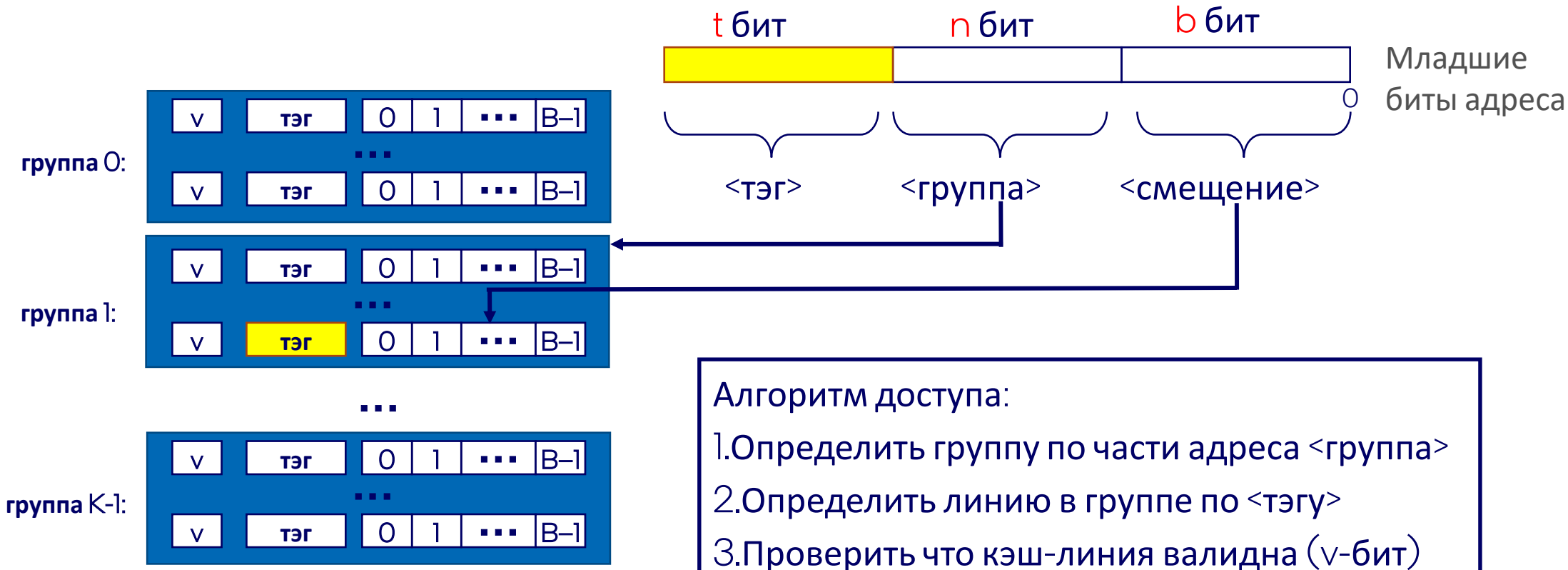
- Кэш – это массив памяти, разбитый на группы
- Группы состоят из линий
- Линия содержит блок данных (обычно 64 байта)

Типы кэшей:

- Кэш прямого отображения: N=1
- Полностью-ассоциативный кэш: K=1, N = S/B
- Множественно-ассоциативный многоканальный кэш: N = S/(B*K)



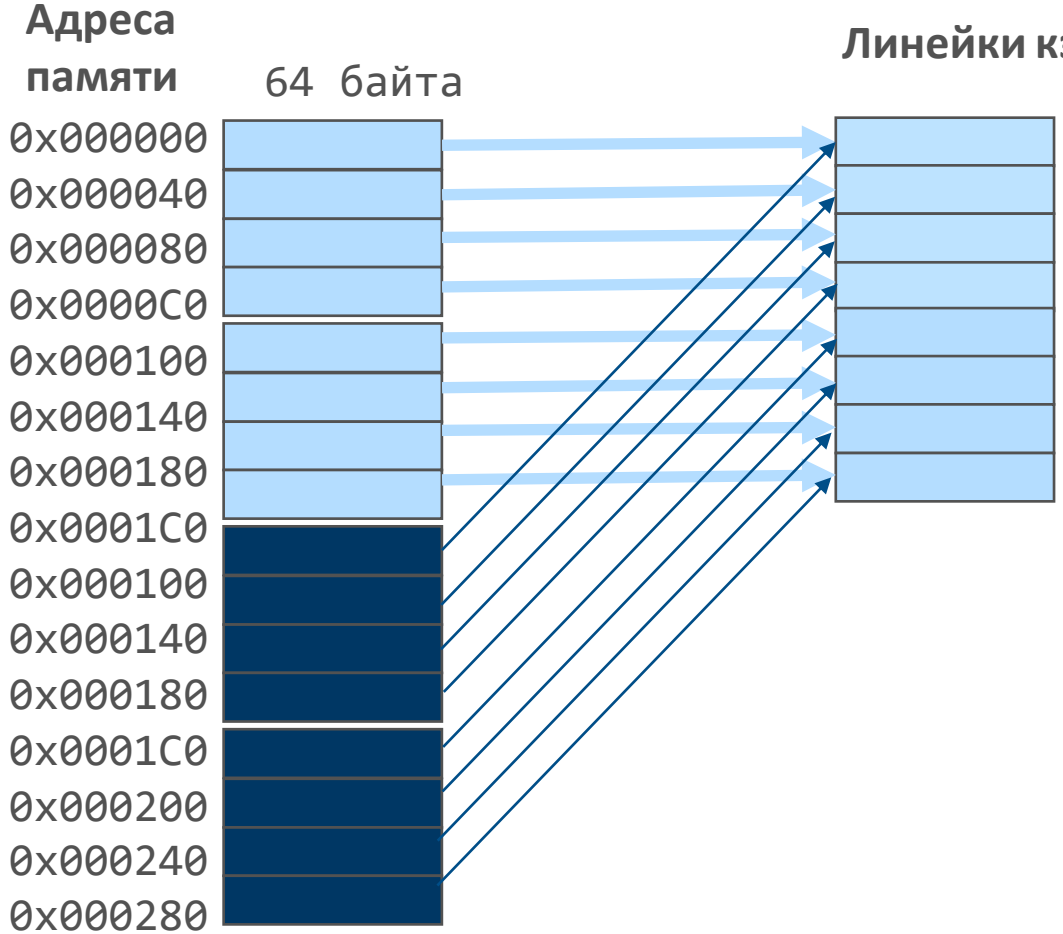
СТРУКТУРА АДРЕСА



Алгоритм доступа:

1. Определить группу по части адреса <группа>
2. Определить линию в группе по <тэгу>
3. Проверить что кэш-линия валидна (v-бит)
4. Использовать данные по <смещению>

КЭШ ПРЯМОГО ОТОБРАЖЕНИЯ



Кэш

Линия: 64 байта

Размер: 512 **байт**

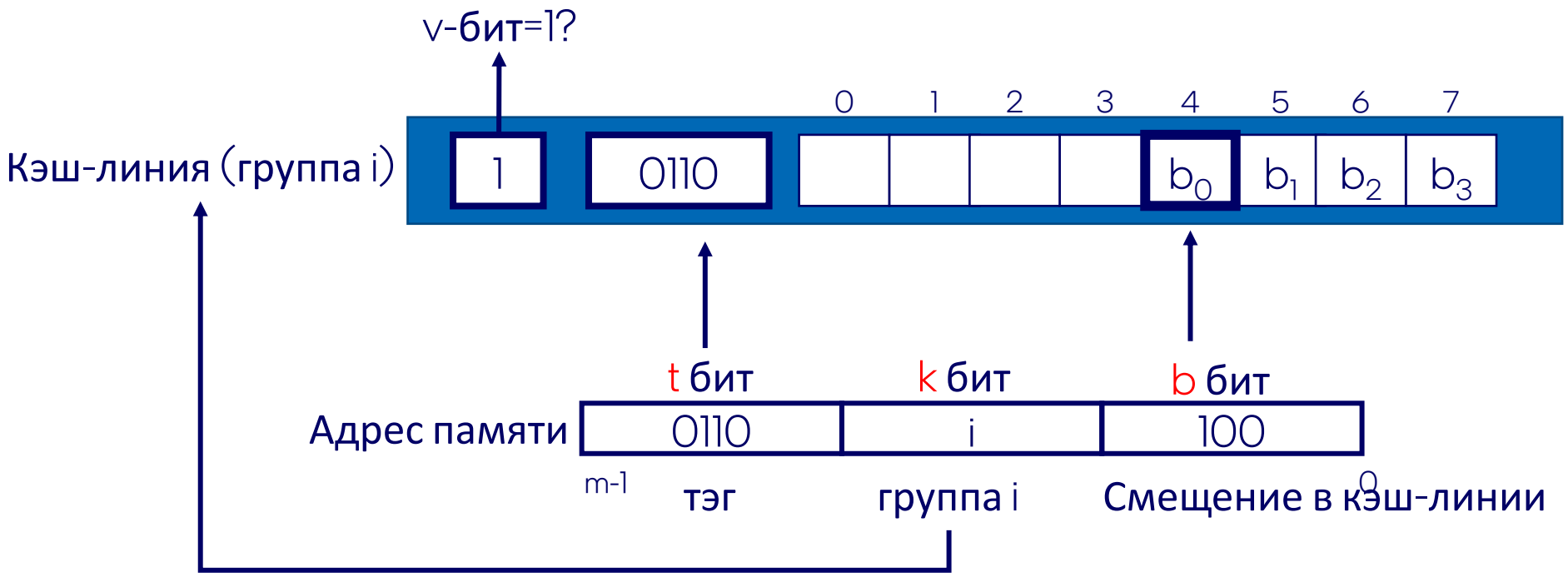
Память разбита на блоки, кратные размеру кэша, любой блок памяти оказаться в кэше.

\+ Простая и дешевая реализация

\- Медленный поиск из-за большой вероятности промаха

КЭШ ПРЯМОГО ОТОБРАЖЕНИЯ

- Как работает поиск
  - Группа состоит из одной кэш-линии, поэтому поиск по всем группам, в которой v-бит = 1 и тэг совпадает со старшей частью адреса.



КАК РАБОТАЕТ КЭШ ПРЯМОГО ТОБРАЖЕНИЯ

t=1	k=2	b=1
x	xx	x

Адрес кодируется 4 битами (16-байтная память M[16]):

Кэш:

$K=2^k=4$ – число групп по одной кэш линии

b=1 бит, адресует смещение до 2 байт в одной кэш-линии

Адреса памяти в порядке обращения:

0 **[0000$_2$],** miss

1 **[0001$_2$],** hit

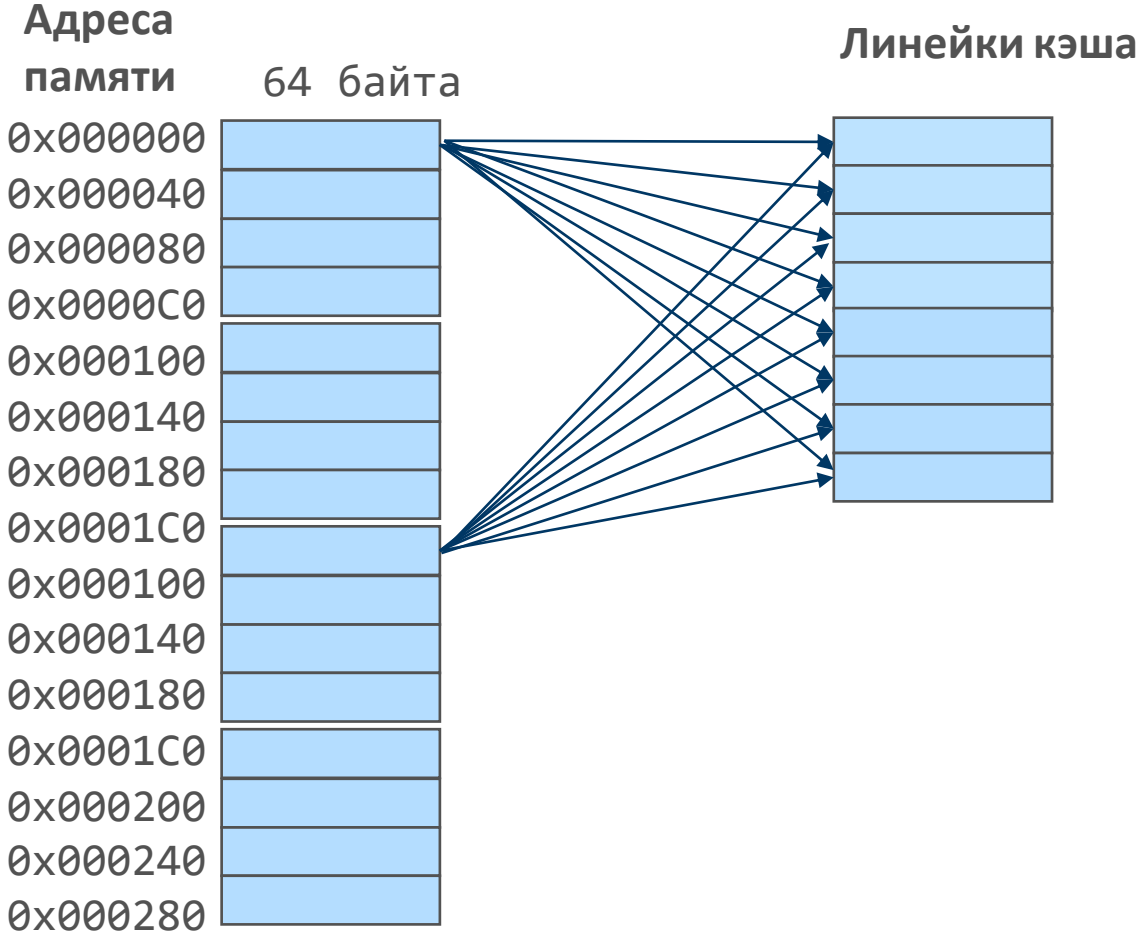
7 **[0111$_2$],** miss

8 **[1000$_2$],** miss

0 **[0000$_2$]** miss

группа	V	тэг	data
00	1	0	M[0-1]
01			
10			
11	1	0	M[6-7]

ПОЛНОСТЬЮ-АССОЦИАТИВНЫЙ КЭШ

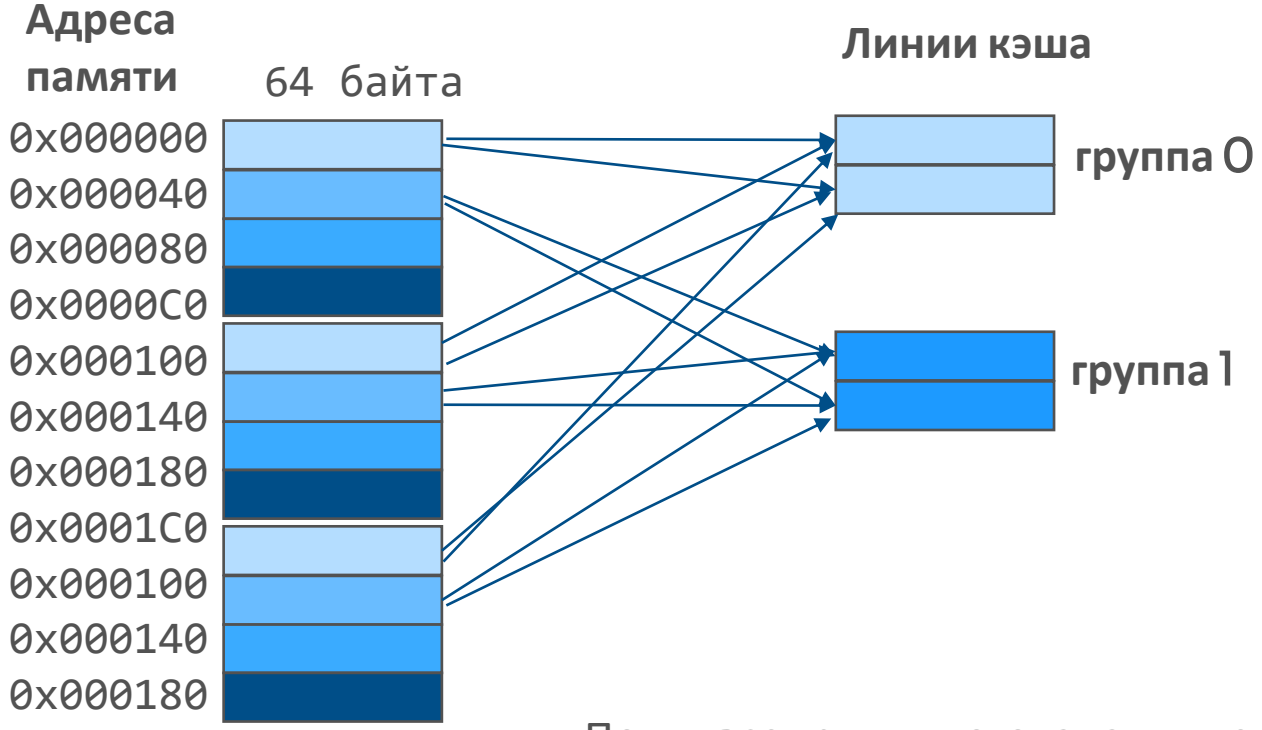


Кэш
Линия: 64 байта
Размер: 512 **байт**

Любой блок памяти в 64 байта может оказаться в любой кэш линии.

- Слишком дорогая реализация, быстрый поиск

МНОЖЕСТВЕННО-АССОЦИАТИВНЫЙ МНОГОКАНАЛЬНЫЙ КЭШ



Кэш

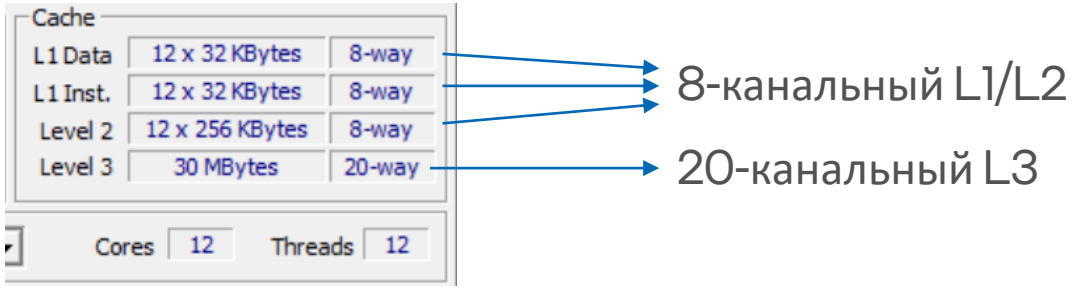
Линия: 64 байта

Каналов: 2

Размер: 512 байт

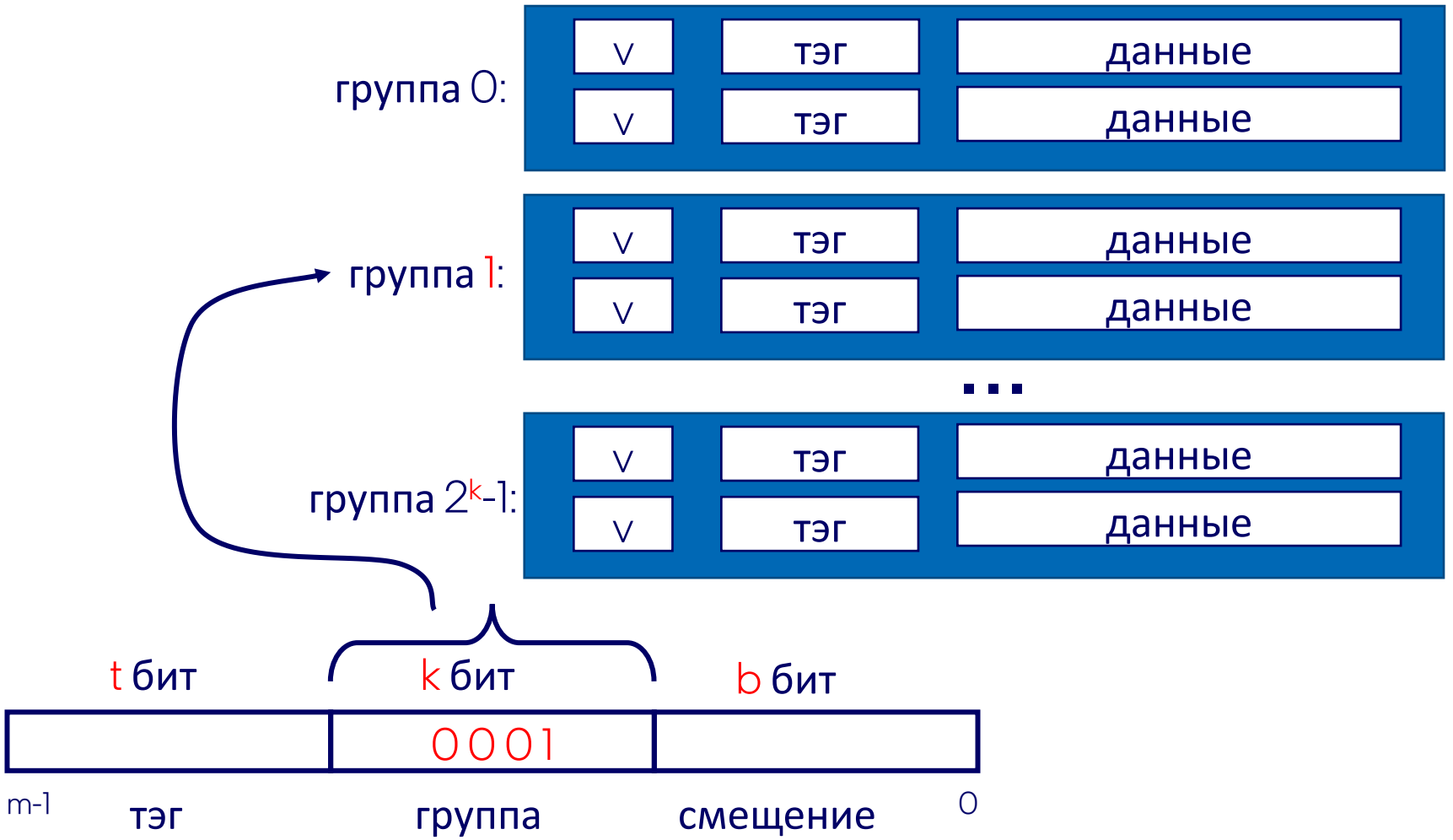
- Кэш равномерно разделен на группы кэш-линий, количество линий в группе – это количество каналов обращения
- 64 байта отображаются на одну линию по любому каналу,
- Одновременный поиск по всем каналам

Почти все кэши многоканальные, т.к. это оптимальный вариант по стоимости и производительности



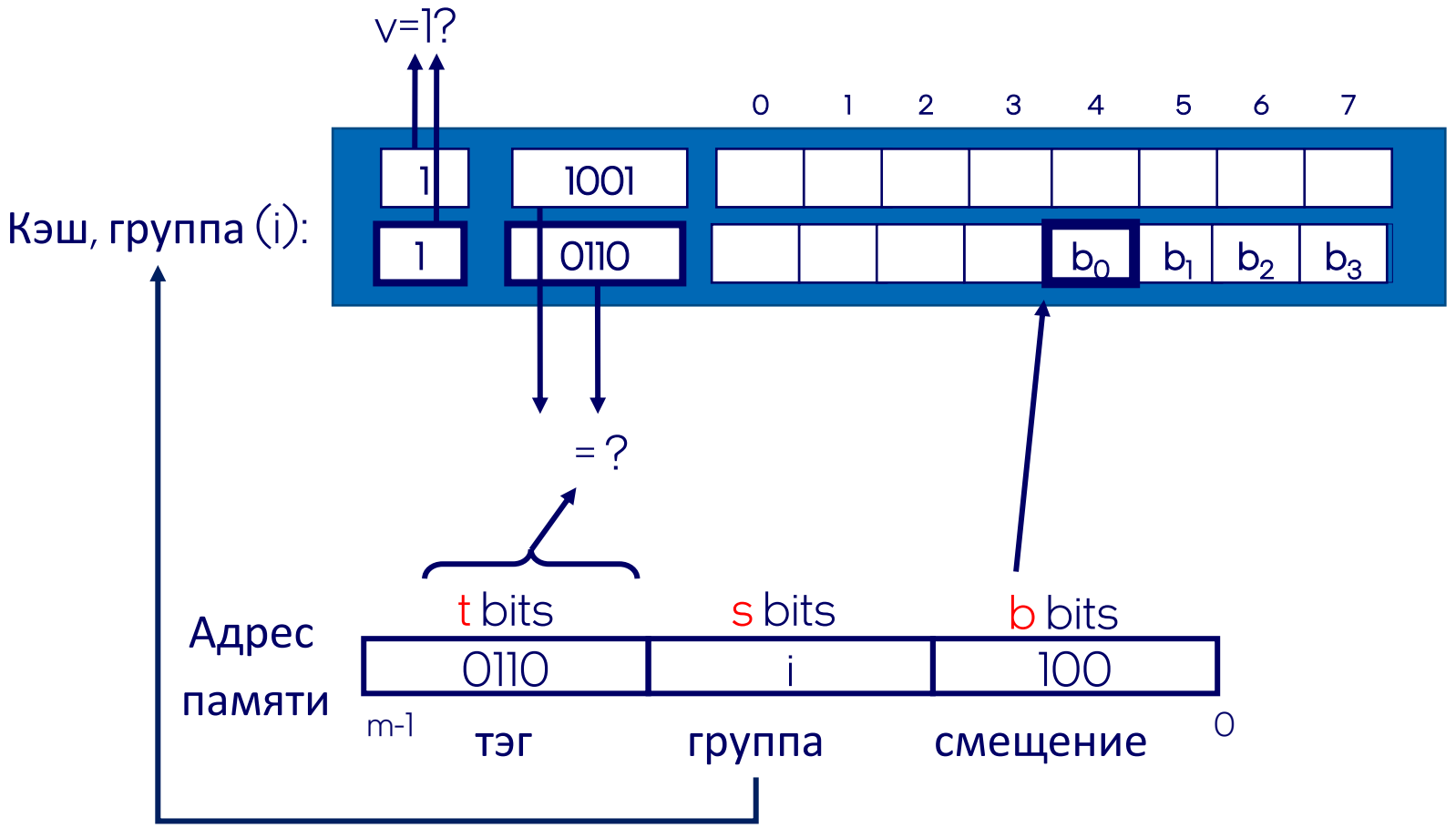
МНОЖЕСТВЕННО-АССОЦИАТИВНЫЙ МНОГОКАНАЛЬНЫЙ КЭШ

- группа выбирается также п очасти адреса памяти

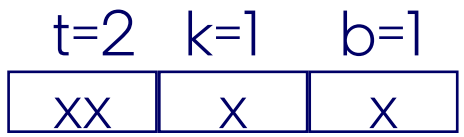


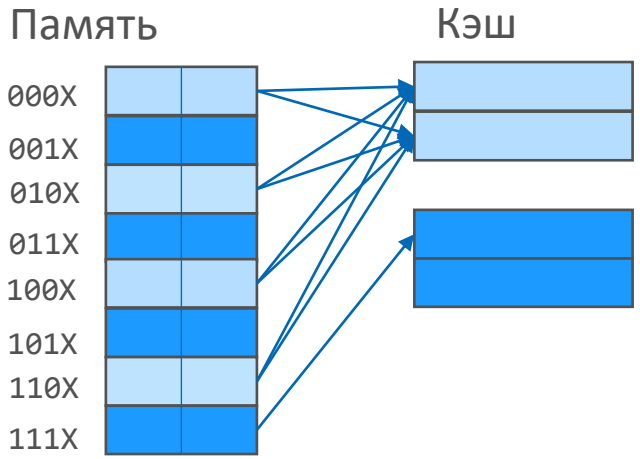
МНОЖЕСТВЕННО-АССОЦИАТИВНЫЙ МНОГОКАНАЛЬНЫЙ КЭШ

- Группа состоит из нескольких кэш-линий, поэтому поиск происходит внутри группы



КАК РАБОТАЕТ 2-КАНАЛЬНЫЙ МНОЖЕСТВЕННО-АССОЦИАТИВНЫЙ КЭШ





Адрес кодируется 4 битами (16-байтная память M[16]):

Кэш:

$K=2^k=2$ – число групп по 2 кэш-линии

b=1 бит, адресует смещение до 2 байт в одной кэш-линии

Адреса памяти в порядке обращения:

0 **[0000$_2$],** miss

1 **[0001$_2$],** hit

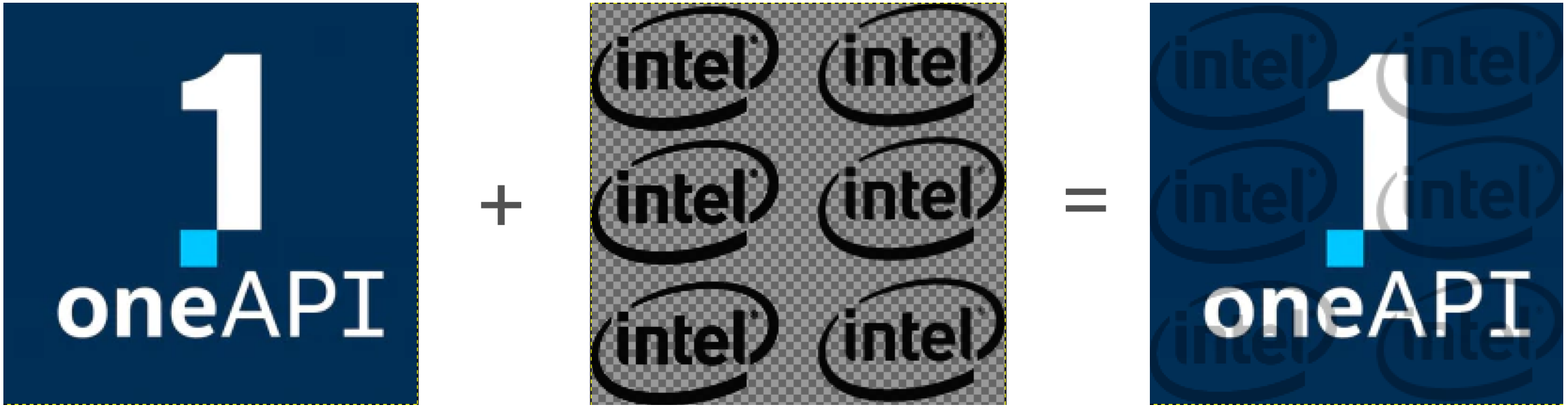
7 **[0111$_2$],** miss

8 **[1000$_2$],** miss

0 **[0000$_2$]** hit

группа	v	тэг	data
0	1	00	M[0-1]
0	1	10	M[8-9]
1	1	01	M[6-7]
1	0		

ПРИМЕР: СУММА ИЗОБРАЖЕНИЙ



Пример использования - Альфа-блендинг для водного знака с прозрачным фоном:

$A = A * (1-a_B) + 0.5 *(A * a_B + B * a_B)$

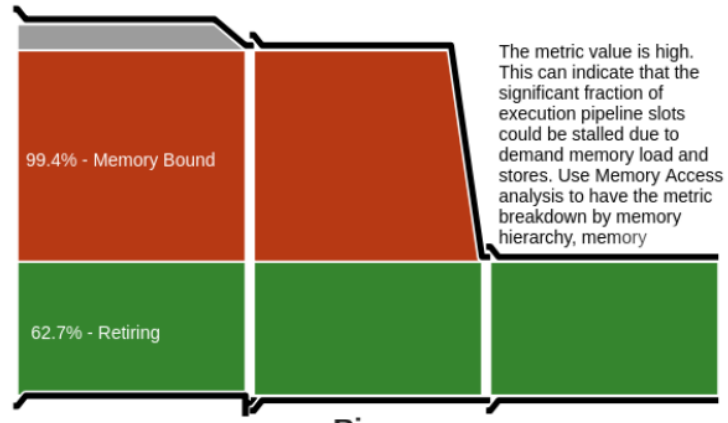
ПРИМЕР: СУММА ИЗОБРАЖЕНИЙ

```
void summ_image(unsigned char* image, unsigned char* resImage, const std::size_t width, const std::size_t height, const
std::size_t bpp)
{
    const std::size_t size{ width * height * bpp };

    const std::size_t iterCount{ ITERATION_COUNT };

    for (std::size_t iter{}; iter < iterCount; ++iter)
    {
        for (std::size_t x{}; x < size; ++x)
        {
            resImage[x] += image[x];
        }
    }
}
```

Intel VTune “Pipe”

Elapsed Time: 3.803s

Clockticks:	12,356,800,000	
Instructions Retired:	21,550,400,000	
CPI Rate:	0.573	
MUX Reliability:	0.943	
Retiring:	62.7%	of Pipeline Slots
Front-End Bound:	12.3%	of Pipeline Slots
Bad Speculation:	0.2%	of Pipeline Slots
Back-End Bound:	24.9%	of Pipeline Slots
Memory Bound:	99.4%	of Pipeline Slots
Core Bound:	0.0%	of Pipeline Slots
Average CPU Frequency:	3.6 GHz	
Total Thread Count:	1	
Paused Time:	0.210s	



μPipe

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

ОПТИМИЗАЦИЯ ПОД КЭШ

В рамках одного потока

- Выравнивание памяти

загрузка в кэш-линии по 64 байта, по выровненным адресам происходит быстрее, но в современных CPU несуществеено

- Разбиение на блоки

разбиение данных и операций над ними на блоки, размеры которых не выходят за пределы КЭШ-памяти.

В рамках многопоточности

По возможности увеличивать нагрузку на потоки, и не выходить за рамки кэш-памяти. В современных CPU это возможно за счет увеличения размеров MLC (L2).

ОПТИМИЗАЦИЯ ПОД КЭШ – КЭШ-БЛОКИНГ

Обработка 2D данных блоками, которые помещаются в кэш, в идеале в L1.

Наиболее оптимальные блоки, хранящие последовательные данные в памяти.

Для суммы изображений достаточно 1D блоков

Идеальный порядок расположения данных в блоке , на практике не применимо, но реализовано аапаратно в GPU. Для 1D блоков непрерывность данных в памяти соблюдена!

ПРИМЕР: СУММА ИЗОБРАЖЕНИЙ – КЭШ БЛОКИНГ

```
void summ_image_block_simd(unsigned char* image, unsigned char* resImage, const std::size_t width, const std::size_t
height, const std::size_t bpp)
{
    const std::size_t size{ width * height * bpp };

    const std::size_t xBlockSize{ L1_CACHE_SIZE / 2u };
    const std::size_t xBlocksCount{ size / xBlockSize };
    const std::size_t xBlocksRemainder{ size % xBlockSize };
    const std::size_t iterCount{ ITERATION_COUNT };

    auto limit = (xBlocksRemainder == 0u) ? size : size - xBlockSize;

    // block sum
    for (std::size_t iter{}; iter < iterCount; ++iter)
    {
        for (std::size_t xx{}; xx < limit; xx += xBlockSize)
        {
            for (std::size_t x{ xx }; x < xx + xBlockSize; x+=32u)
            {
                __m256i ymm = _mm256_add_epi8(_mm256_load_si256((__m256i*)(resImage + x)),
                                              _mm256_load_si256((__m256i*)(image + x)));
                _mm256_store_si256((__m256i*)(resImage + x), ymm);
            }
        }
    }
}
```

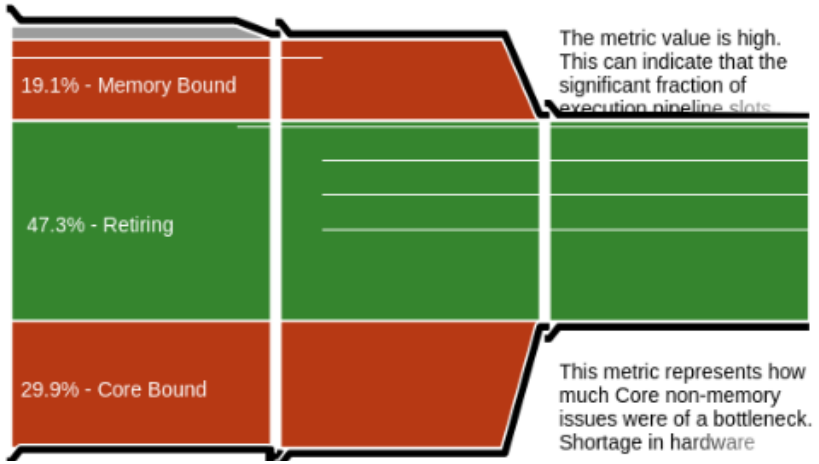
ПРИМЕР: СУММА ИЗОБРАЖЕНИЙ – КЭШ БЛОКИНГ

```
void summ_image_block_simd(unsigned char* image, unsigned char* resImage, const std::size_t width, const std::size_t
height, const std::size_t bpp)
{
    const std::size_t size{ width * height * bpp };

    const std::size_t xBlockSize{ L1_CACHE_SIZE / 2u };
    const std::size_t xBlocksCount{ size / xBlockSize };
    const std::size_t xBlocksRemainder{ size % xBlockSize };
    const std::size_t iterCount{ ITERATION_COUNT };

    auto limit = (xBlocksRemainder == 0u) ? size : size - xBlockSize;

    // block sum
    for (std::size_t xx{}; xx < limit; xx += xBlockSize)
    {
        for (std::size_t iter{}; iter < iterCount; ++iter)
        {
            for (std::size_t x{ xx }; x < xx + xBlockSize; x+=32u)
            {
                __m256i ymm = _mm256_add_epi8(_mm256_load_si256((__m256i*)(resImage + x)),
                                              _mm256_load_si256((__m256i*)(image + x)));
                _mm256_store_si256((__m256i*)(resImage + x), ymm);
            }
        }
    }
}
```

ЗАВИСИМОСТЬ ОТ РАЗМЕРА БЛОКА: L3<100*L1/L2<10*L1

Elapsed Time: 0.540s

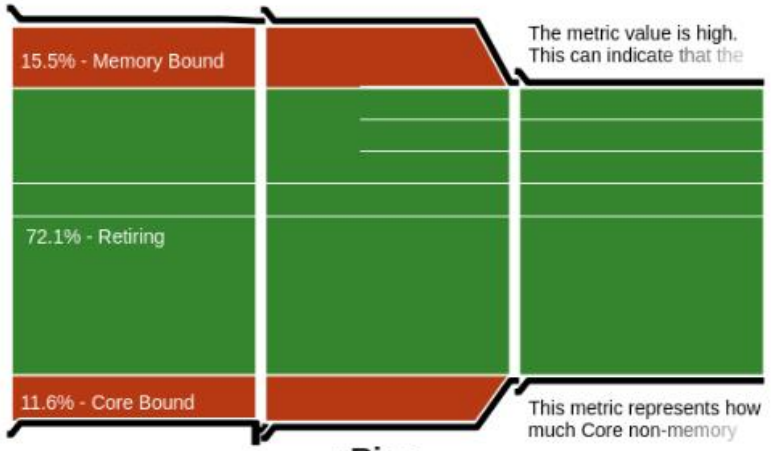
Clockticks:	1,089,600,000	
Instructions Retired:	2,699,200,000	
CPI Rate:	0.404	
MUX Reliability:	0.817	
Retiring:	47.3%	of Pipeline Slots
Front-End Bound:	3.2%	of Pipeline Slots
Bad Speculation:	0.5%	of Pipeline Slots
Back-End Bound:	49.1%	of Pipeline Slots
Memory Bound:	19.1%	of Pipeline Slots
L1 Bound:	0.0%	of Clockticks
L2 Bound:	0.0%	of Clockticks
L3 Bound:	10.8%	of Clockticks
Contested Accesses:	0.0%	of Clockticks
Data Sharing:	0.0%	of Clockticks
L3 Latency:	100.0%	of Clockticks



BlockSize = 100 * L1
(L3 bound,
2*BlockSize>6Mb)

Elapsed Time: 0.414s

Clockticks:	648,000,000	
Instructions Retired:	1,451,200,000	
CPI Rate:	0.447	
MUX Reliability:	0.745	
Retiring:	72.1%	of Pipeline Slots
Front-End Bound:	0.4%	of Pipeline Slots
Bad Speculation:	0.4%	of Pipeline Slots
Back-End Bound:	27.1%	of Pipeline Slots
Memory Bound:	15.5%	of Pipeline Slots
L1 Bound:	1.5%	of Clockticks
L2 Bound:	7.7%	of Clockticks
L3 Bound:	0.0%	of Clockticks
DRAM Bound:	0.0%	of Clockticks
Store Bound:	58.6%	of Clockticks
Core Bound:	11.6%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	50.9%	of Clockticks
Average CPU Frequency:	3.5 GHz	
Total Thread Count:	1	
Paused Time:	0.207s	



µPipe

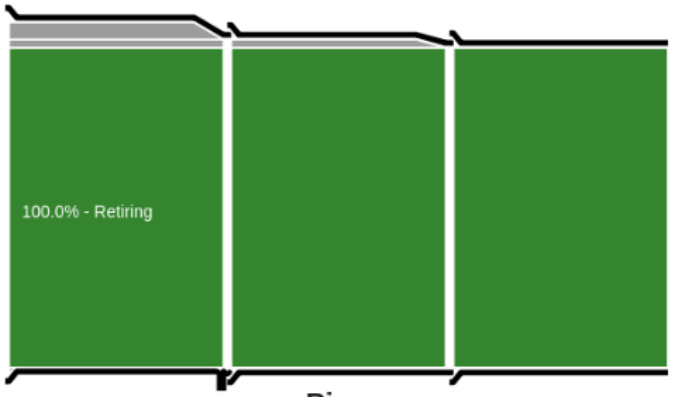
This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

BlockSize = 10 * L1
(L2 bound,
2*BlockSize > 256Kb)

ЗАВИСИМОСТЬ ОТ РАЗМЕРА БЛОКА: L3<100*L1/L2<10*L1

Elapsed Time: 0.347s

Clockticks:	417,600,000	
Instructions Retired:	1,457,600,000	
CPI Rate:	0.286	
MUX Reliability:	N/A*	
Retiring:	100.0%	of Pipeline Slots
General Retirement:	100.0%	of Pipeline Slots
Microcode Sequencer:	0.0%	of Pipeline Slots
Front-End Bound:	5.4%	of Pipeline Slots
Bad Speculation:	1.2%	of Pipeline Slots
Back-End Bound:	0.0%	of Pipeline Slots
Average CPU Frequency:	3.5 GHz	
Total Thread Count:	1	
Paused Time:	0.210s	



µPipe

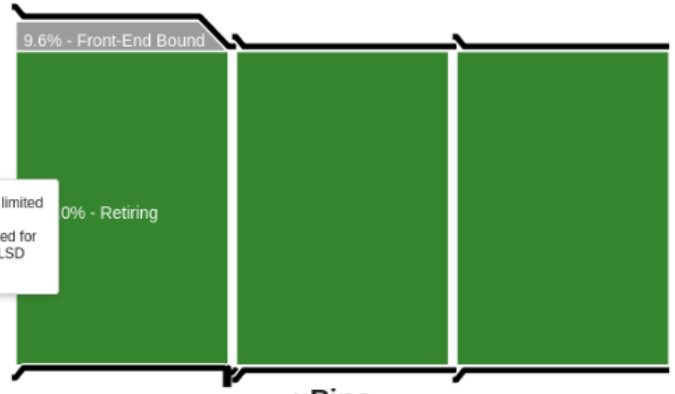
This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

BlockSize = L1/2

Elapsed Time: 0.839s

Clockticks:	2,118,400,000	
Instructions Retired:	8,483,200,000	
CPI Rate:	0.250	
MUX Reliability:	0.985	
Retiring:	100.0%	of Pipeline Slots
General Retirement:	100.0%	of Pipeline Slots
Microcode Sequencer:	0.0%	of Pipeline Slots
Front-E		
Fro		
Fro		
Front-End Bandwidth LSD:	9.9%	of Clockticks
(Info) DSB Coverage:	4.6%	
(Info) LSD Coverage:	95.4%	
Bad Speculation:	0.8%	of Pipeline Slots
Back-End Bound:	0.0%	of Pipeline Slots
Average CPU Frequency:	3.6 GHz	
Total Thread Count:	1	
Paused Time:	0.206s	

This metric represents a fraction of cycles during which CPU operation was limited by the LSD (Loop Stream Detector) unit. Typically, LSD provides good uOp supply. However, in some rare cases, optimal uOp delivery cannot be reached for small loops whose size (in terms of number of uOps) does not suit well the LSD structure.

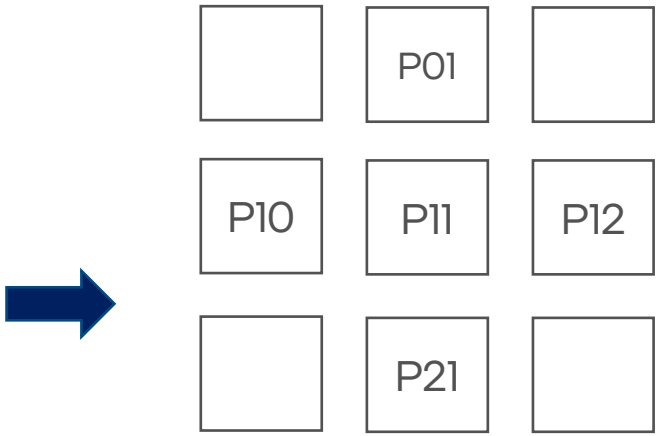


µPipe

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

BlockSize = L1/1024

ПРИМЕР: СВЕРТОЧНЫЙ ФИЛЬТР

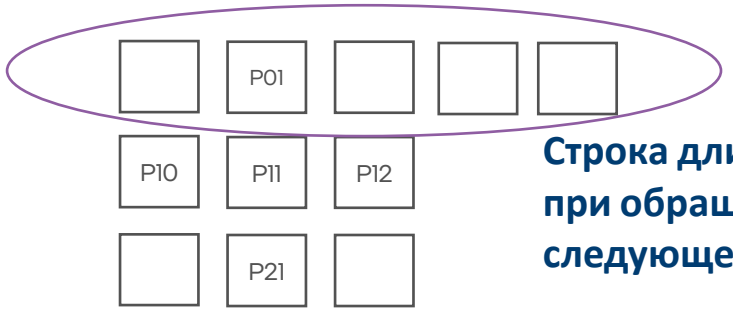


Ядро фильтра (smoothing)

P11 = 0.5 * P11 + 0.125 * (P01+P10+P12+P21)

=

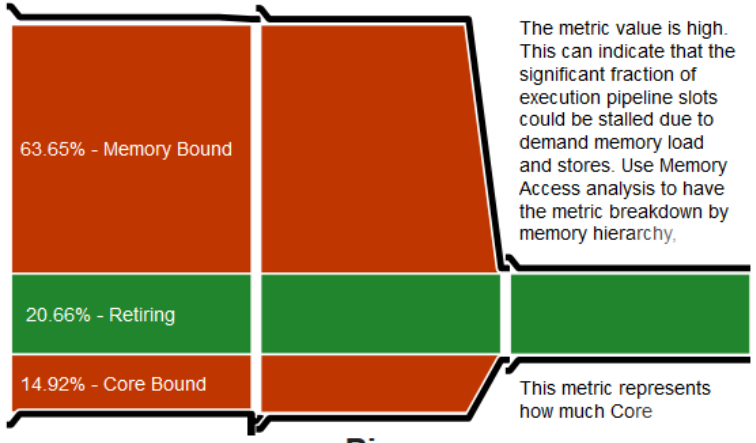
ПРИМЕР: СВЕРТОЧНЫЙ ФИЛЬТР



Строка длиннее L1, кеш-мисс при обращение к элементу следующей строки

```
const std::size_t unit{ 1u };
    const std::size_t size{ width * height * bpp };
    std::copy(image, image + size, resImage);
    const std::size_t iterCount{ ITERATION_COUNT2 };
    // iterative smooth
    for (std::size_t iter{}; iter < iterCount; ++iter)
    {
        for (std::size_t x{ 1u }; x < width - 1u; ++x)
        {
            for (std::size_t y{ 1u }; y < height - 1u; ++y)
            {
                process_pixel(x, y, bpp, width, height, image, resImage);
            }
        }
        std::swap(resImage, image);
    }
```

Clockticks:	13,168,470,000	
Instructions Retired:	10,824,870,000	
CPI Rate:	1.217	
MUX Reliability:	0.992	
Retiring:	20.7%	of Pipeline Slots
Front-End Bound:	0.5%	of Pipeline Slots
Bad Speculation:	0.3%	of Pipeline Slots
Back-End Bound:	78.6%	of Pipeline Slots
Memory Bound:	63.7%	of Pipeline Slots
L1 Bound:	5.0%	of Clockticks
DTLB Overhead:	57.4%	of Clockticks
Loads Blocked by Store Forwarding:	0.0%	of Clockticks
Lock Latency:	0.1%	of Clockticks
Split Loads:	0.0%	of Clockticks
4K Aliasing:	76.8%	of Clockticks
FB Full:	2.2%	of Clockticks
L2 Bound:	6.7%	of Clockticks
L3 Bound:	32.4%	of Clockticks
Contested Accesses:	0.1%	of Clockticks
Data Sharing:	0.0%	of Clockticks



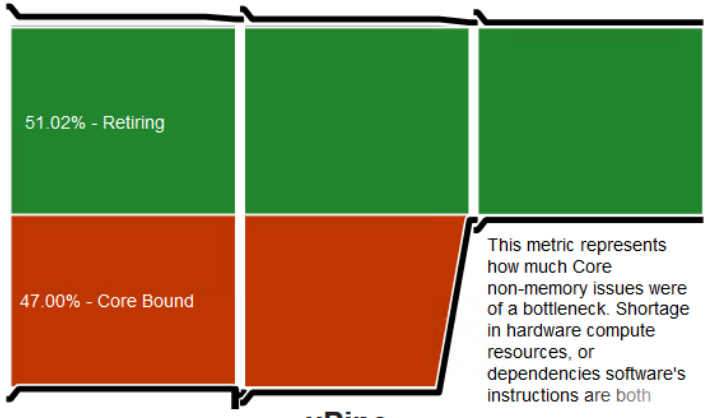
µPipe

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

ПРИМЕР: СВЕРТОЧНЫЙ ФИЛЬТР – КЕШ БЛОКИНГ

Block size is L1/2

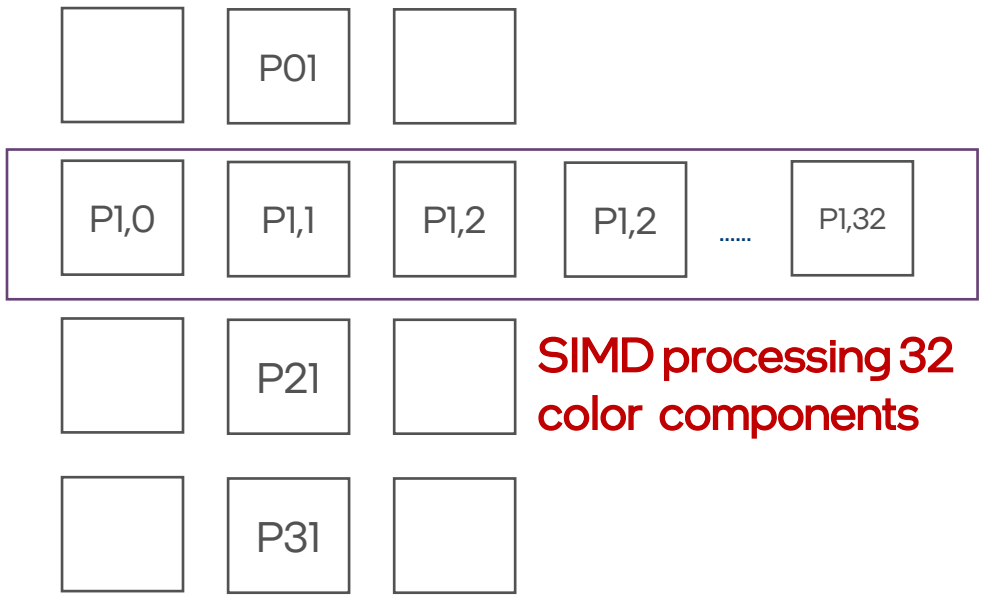
Clockticks:	5,008,500,000	
Instructions Retired:	10,749,480,000	
CPI Rate:	0.466	
MUX Reliability:	0.970	
Retiring:	51.0%	of Pipeline Slots
Front-End Bound:	0.7%	of Pipeline Slots
Bad Speculation:	0.4%	of Pipeline Slots
Back-End Bound:	47.9%	of Pipeline Slots
Memory Bound:	0.9%	of Pipeline Slots
L1 Bound:	0.5%	of Clockticks
L2 Bound:	0.0%	of Clockticks
L3 Bound:	0.1%	of Clockticks
DRAM Bound:	0.0%	of Clockticks
Store Bound:	0.0%	of Clockticks
Core Bound:	47.0%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	34.0%	of Clockticks
Total Thread Count:	4	
Paused Time:	12.367s	



This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

```
const std::size_t unit{ 1u };
const std::size_t size{ width * height * bpp };
std::copy(image, image + size, resImage);
const std::size_t xBlockSize{ L1_CACHE_SIZE / 2u / 50u / bpp - 2u * bpp};
const std::size_t yBlockSize{ 50u };
const std::size_t xBlocksCount = (width - 2u) / xBlockSize;
const std::size_t yBlocksCount = (height - 2u) / yBlockSize;
const std::size_t iterCount{ ITERATION_COUNT2 };
// iterative smooth
for (std::size_t iter{}; iter < iterCount; ++iter)
{
        for (std::size_t yy{1u}; yy < height - 1u; yy += yBlockSize)
        {
           for (std::size_t xx{1u}; xx < width - 1u; xx += xBlockSize)
           {
               for (std::size_t y = yy; y <
                        std::min(yy + yBlockSize, height - 1u); ++y)
              {
                 for (std::size_t x = xx; x <
                        std::min(xx + xBlockSize, width - 1u); ++x)
                 {
                    process_pixel(x, y, bpp, width, height, image, resImage);
                 }
              }
           }
    }
    std::swap(resImage, image);
} // iteratios loop
```

ПРИМЕР: СВЕРТОЧНЫЙ ФИЛЬТР



SIMD processing 32 color components

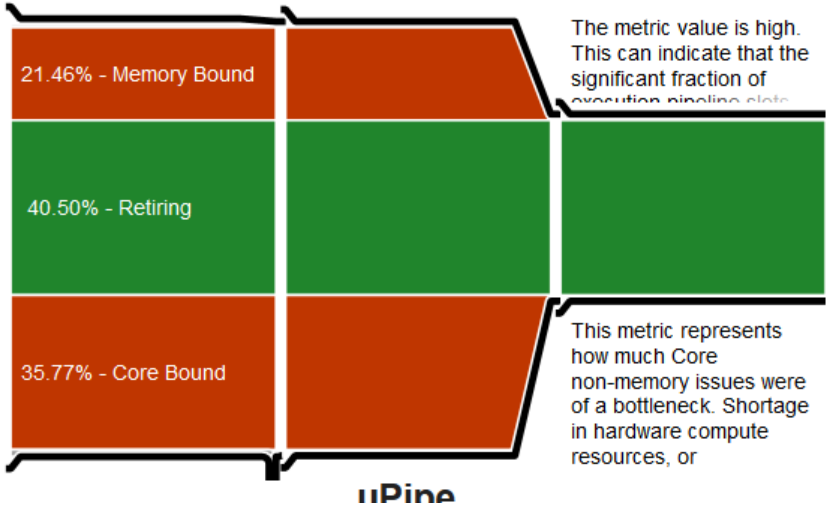
```
for (std::size_t color{}; color < bpp; ++color)
   {
       resImage[offset + color] = static_cast<unsigned char>(
           (image[offsetU + color] >> 3u) +
           (image[offsetD + color] >> 3u) +
           (image[offsetL + color] >> 3u) +
           (image[offsetR + color] >> 3u) +
           (image[offset + color] >> 1u));
   }
```

```
__m256i row = shiftByN(_mm256_loadu_si256((__m256i*)(image + offset)), 1);
   __m256i rowU = shiftByN(_mm256_loadu_si256((__m256i*)(image + offsetU)), 3);
   __m256i rowD = shiftByN(_mm256_loadu_si256((__m256i*)(image + offsetD)), 3);
   __m256i rowL = shiftByN(_mm256_loadu_si256((__m256i*)(image + offsetL)), 3);
   __m256i rowR = shiftByN(_mm256_loadu_si256((__m256i*)(image + offsetR)), 3);
   row = _mm256_add_epi8(row, rowU);
   row = _mm256_add_epi8(row, rowD);
   row = _mm256_add_epi8(row, rowL);
   row = _mm256_add_epi8(row, rowR);
```

```
 static inline __m256i shiftByN(__m256i src, int n)
 {
     __m256i line1a = _mm256_srli_epi16(src, n + 8);
     __m256i line1b = _mm256_slli_epi16(src, 8);
     line1a = _mm256_slli_epi16(line1a, 8);
     line1b = _mm256_srli_epi16(line1b, n + 8);
     return _mm256_or_si256(line1a, line1b);
 }
```

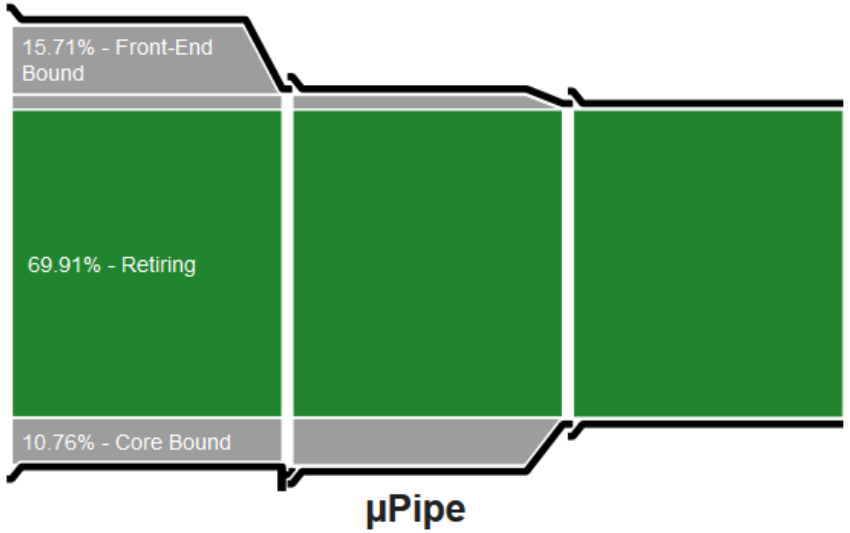
ПРИМЕР: СВЕРТОЧНЫЙ ФИЛЬТР

Clockticks:	416,640,000	
Instructions Retired:	836,850,000	
CPI Rate:	0.498	
MUX Reliability:	0.765	
Retiring:	40.5%	of Pipeline Slots
Front-End Bound:	0.6%	of Pipeline Slots
Bad Speculation:	1.6%	of Pipeline Slots
Back-End Bound:	57.2%	of Pipeline Slots
Memory Bound:	21.5%	of Pipeline Slots
L1 Bound:	3.9%	of Clockticks
L2 Bound:	3.9%	of Clockticks
L3 Bound:	5.2%	of Clockticks
Contested Accesses:	0.0%	of Clockticks
Data Sharing:	0.0%	of Clockticks
L3 Latency:	38.5%	of Clockticks
SQ Full:	1.3%	of Clockticks



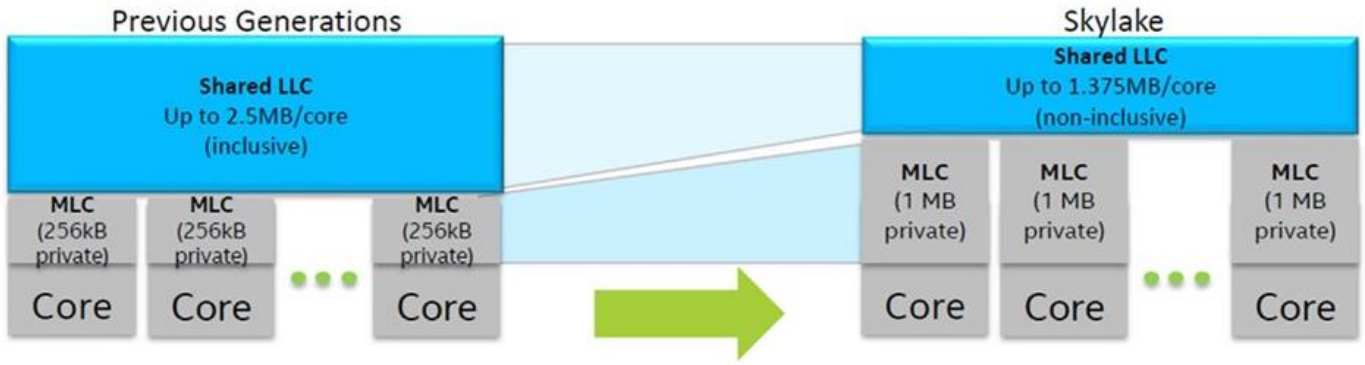
SIMD

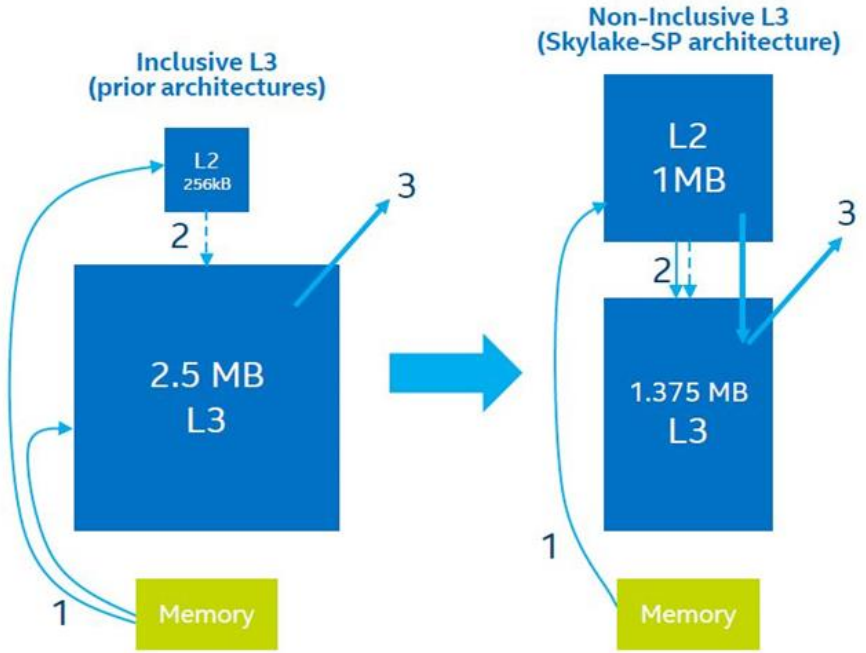
Clockticks:	457,380,000	
Instructions Retired:	838,530,000	
CPI Rate:	0.545	
MUX Reliability:	0.697	
Retiring:	69.9%	of Pipeline Slots
Front-End Bound:	15.7%	of Pipeline Slots
Bad Speculation:	0.3%	of Pipeline Slots
Back-End Bound:	14.1%	of Pipeline Slots
Total Thread Count:	4	
Paused Time:	16.574s	



SIMD + кэш-блокинг

АППАРАТНАЯ ОПТИМИЗАЦИЯ КЭШЕЙ



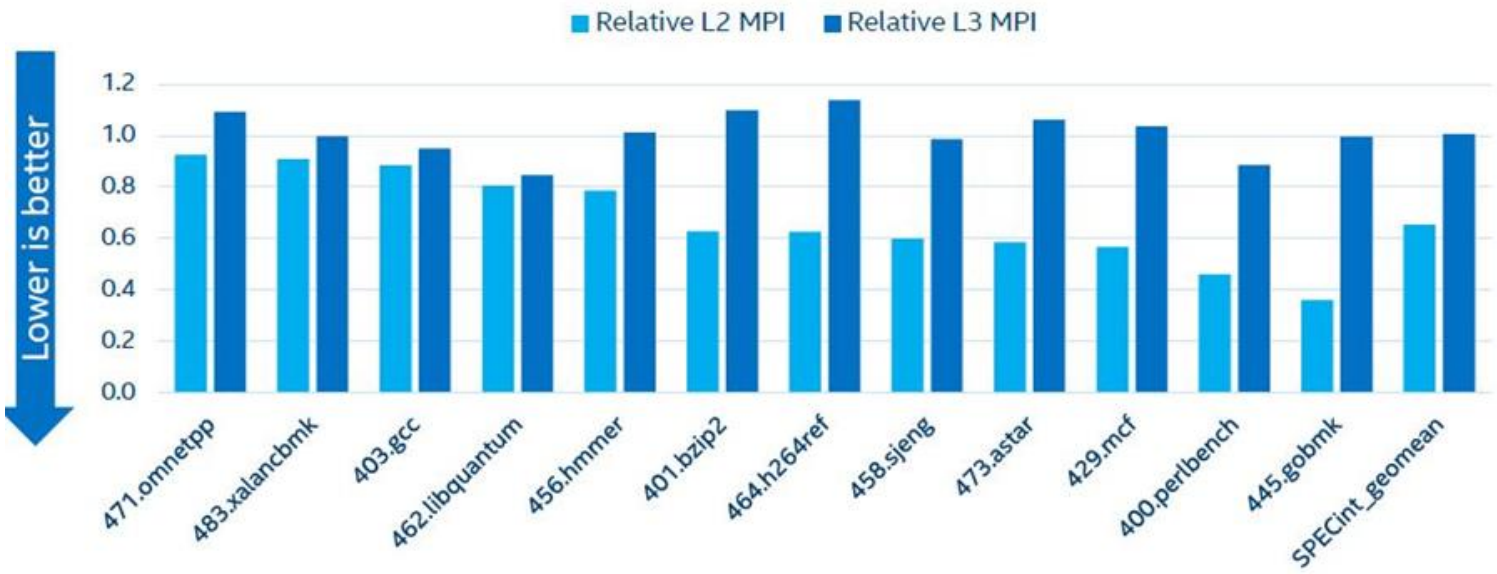


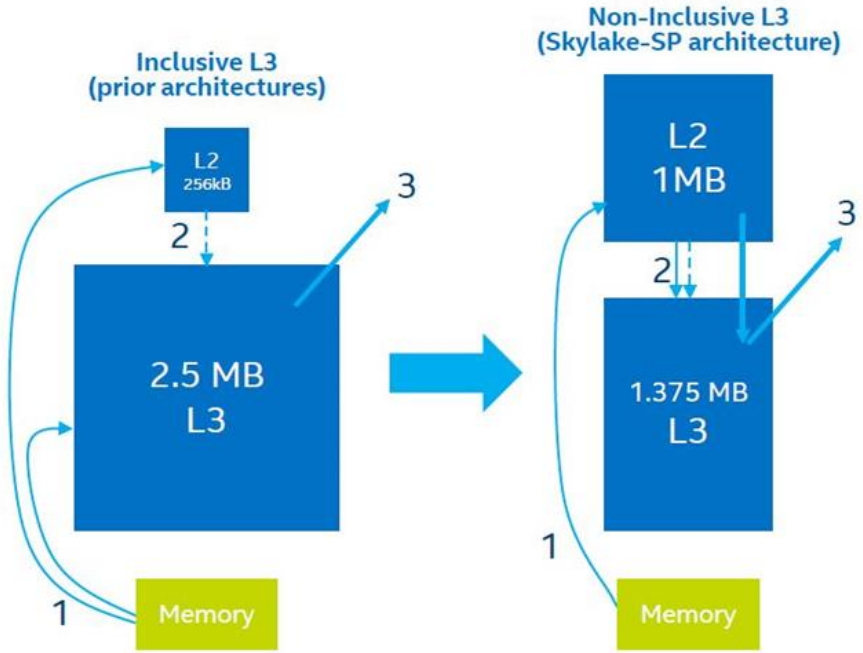
Увеличение MLC L2/LLC L3 - смещение баланса в сторону локальной памяти ядер от шареной памяти.

Отказ от инклюзивного L3 кэша:

1. Загрузка из памяти делается только в L2
2. При заполненном L2, наименее востребованные по числу обращений данные данные скидываются в L3.
3. Расшаренные L2 данные между ядрами дублируются в L3, L2 miss + L3 hit инициирует обмен линиями L2-L3, а не дублирование в L2.

АППАРАТНАЯ ОПТИМИЗАЦИЯ КЭШЕЙ

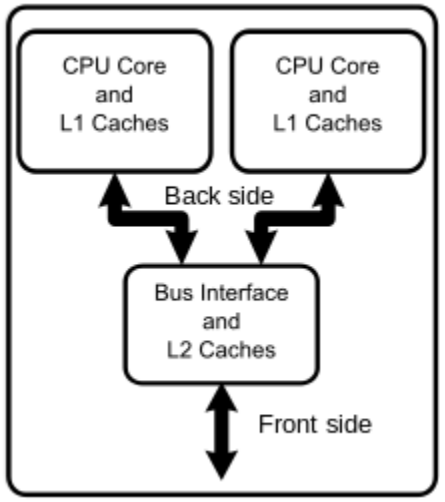


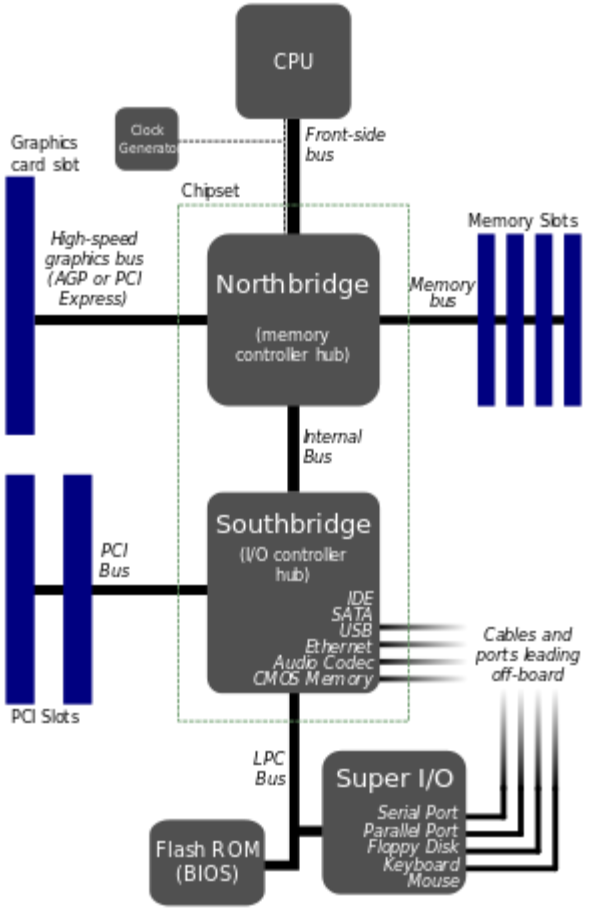


Соотношение MLC (L2) промахов в SKYLAKE, по сравнению с предыдущими архитектурами, меньше единицы, при несущественном росте промахов для LLC (L3) (соотношение >1).

АППАРАТНАЯ ОПТИМИЗАЦИЯ ИЕРАРХИИ

- Архитектуры CPU 1990x-2010x
  FSB (**front-side bus**) – коммуникационная шина обеспечивающая доступ CPU к интерфейсу памяти (через северный мост - northbridge)





С появлением **многоядерных** архитектур проблема неоднородного доступа к памяти (NUMA – non-uniform memory access) для кэшей.

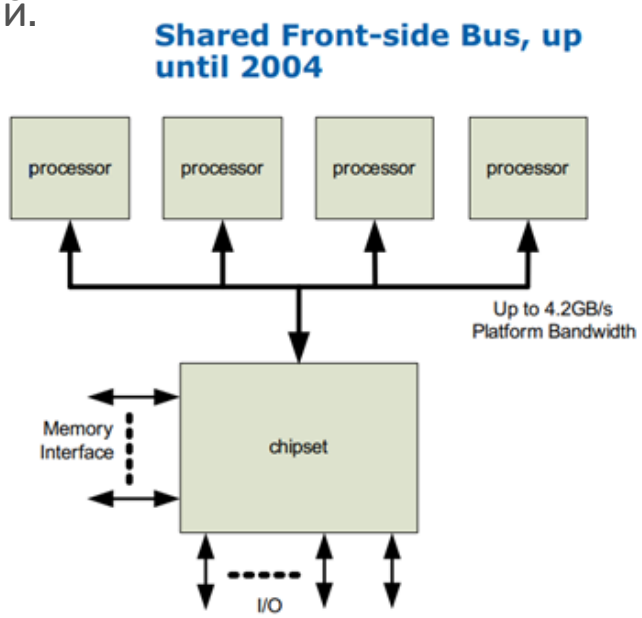
Для многоядерных процессоров:
L2 кэш был общий,
L3 кэша не было,
либо он располагался за пределами процессора.
FSB – узкое место, повышение производительности было ограничено

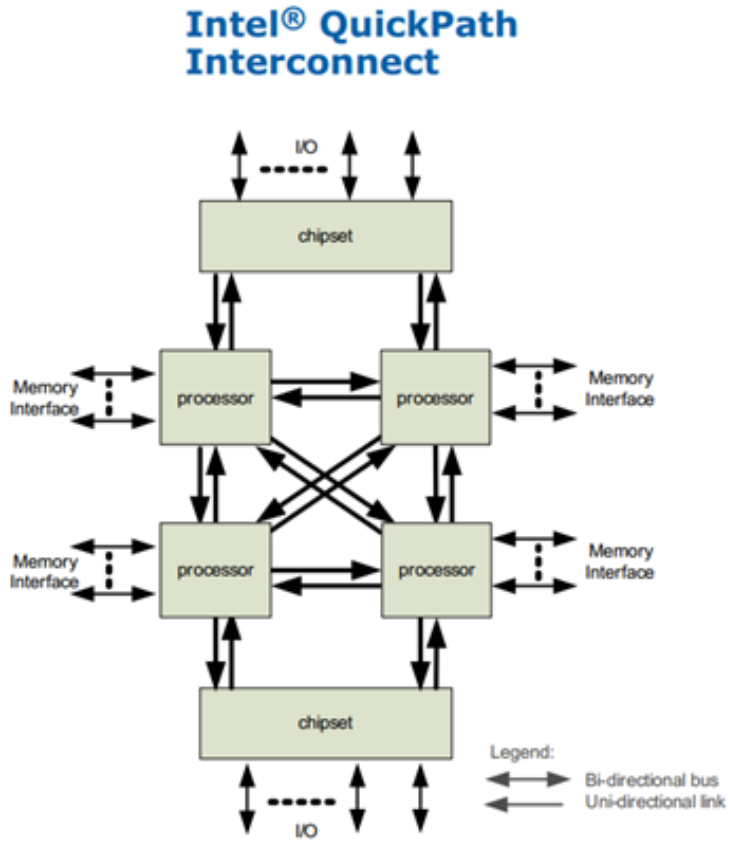
Тенденция – перенос коммуникационных интерфейсов и кэш-памяти ближе к ядрам CPU.

ДЕЦЕНТРАЛИЗАЦИЯ КЭШЕЙ И ИНТЕРФЕЙСОВ ДОСТУПА

При смене архитектуры Core2Duo/Quad- >Core I – децентрализация L2 кэша

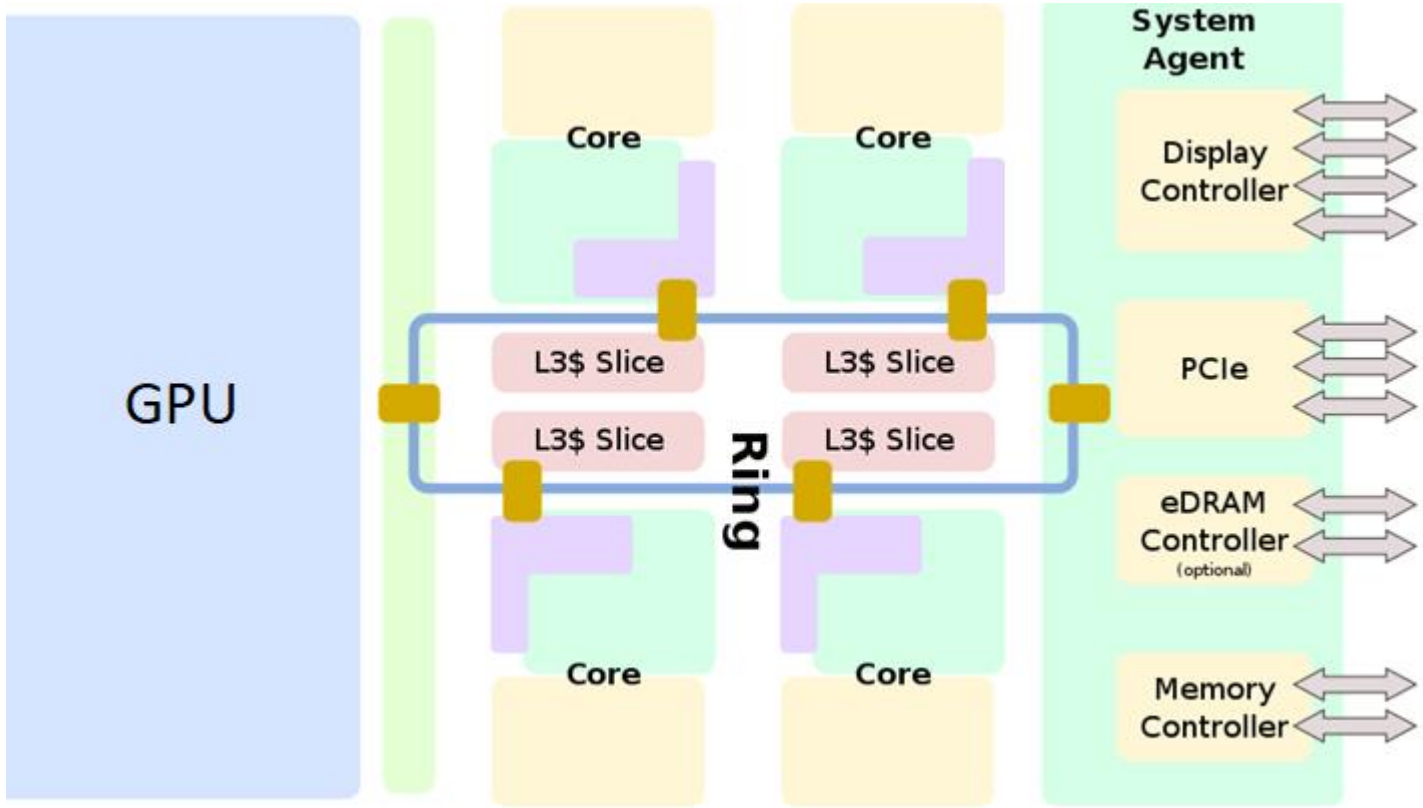
Intel® QuickPath Interconnect (QPI) высокопроизводительный коммуникационный интерфейс точка-к-точке (впервые реализован с 2008 года – Sandy Bridge).
- Решил проблему узкого места FSB, децентрализовав интерфейсы памяти
- Обеспечивал протокол когерентности кэшей (MESI) – решение проблемы NUMA для кэшей.



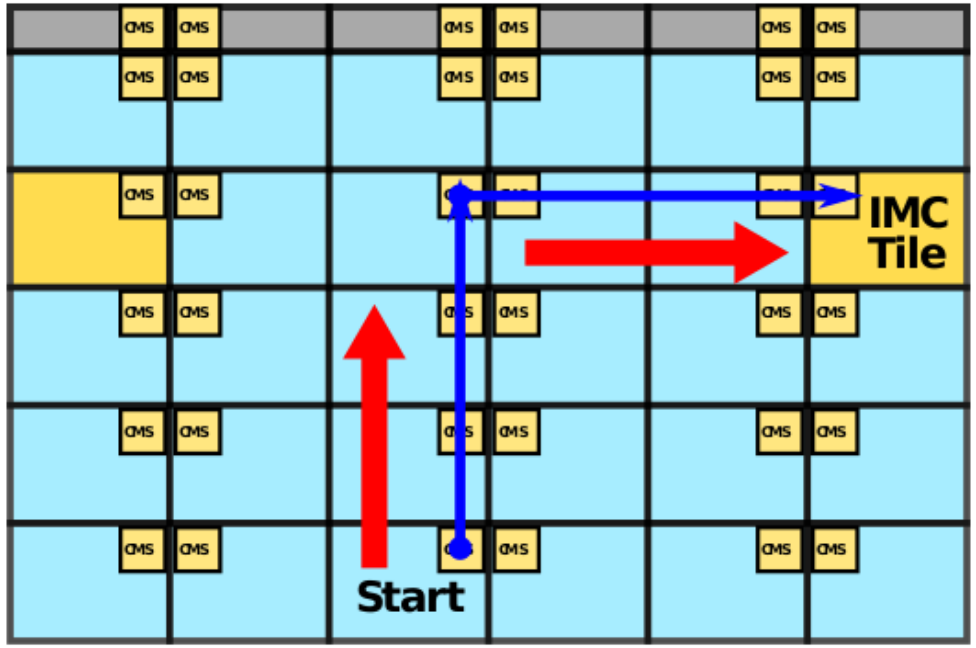


КОЛЬЦЕВЫЕ ИНТЕРФЕЙСЫ ДОСТУПА К L3 КЭШУ

- Skylake LLC ring(client)

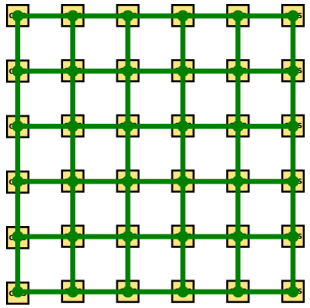


СЕТОЧНЫЙ КОММУНИКАЦИОННЫЙ ИНТЕРФЕЙС



- Mesh Interconnect Architecture – Intel Skylake

- **Сеточный интерфейс коммуникации (MESH) – сетка из кольцевых интерфейсов коммуникации между вычислительными устройствами**
- **Сетка работает на тайловом расположении компонент, тайлы заполняют CPU и могут быть 2 типов:**
- **ядро процессора**
- **контроллер памяти IMC (integrated memory controller)**



Коммникационный интерфейс между тайлами имеет вид сетки (MESH)


intel