




C++ZeroCost
Conf 



Оптимизируем на примере dot-product

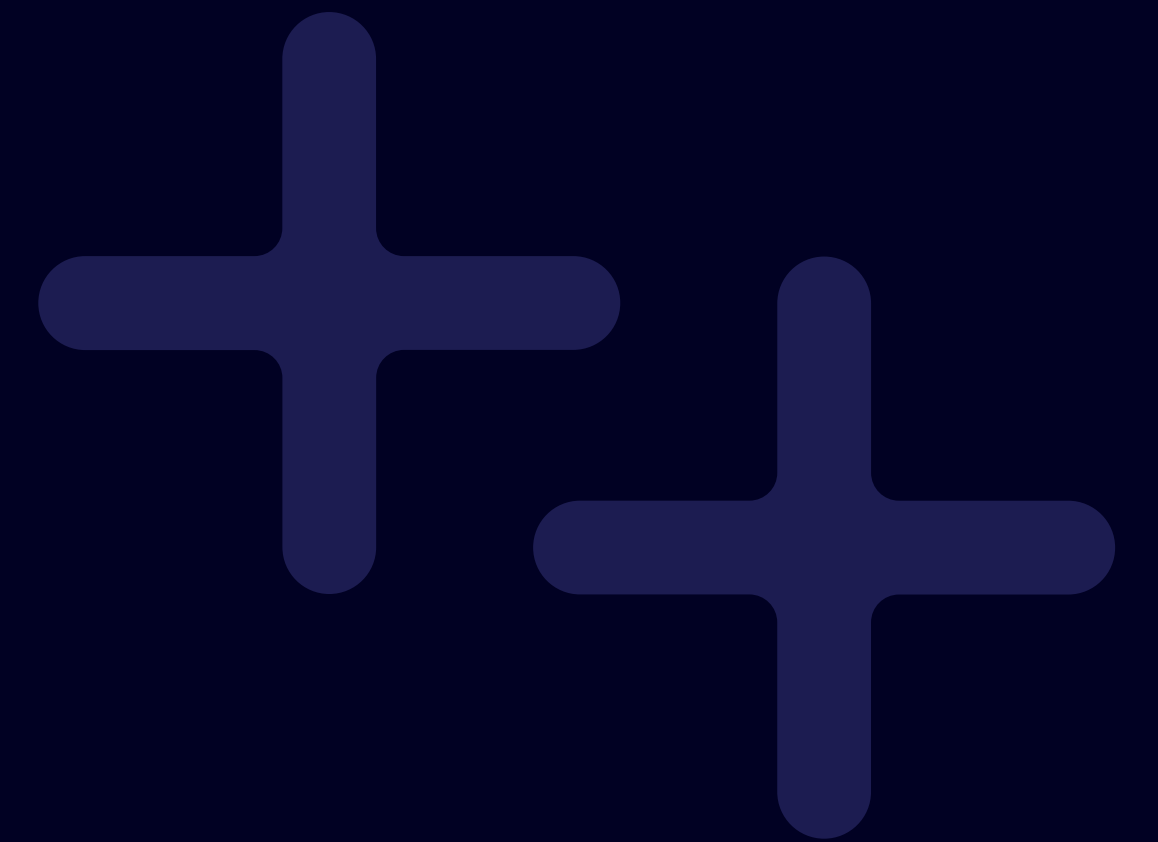


Хузиев Ильнур
Руководитель группы разработки в web-поиске

Yandex for `developers` `*//>`

Содержание

- 01 Постановка и первые оптимизации
- 02 Лирическое отступление
- 03 Оптимизации IV-2
- 04 Оптимизируем память



01

Постановка и первые оптимизации

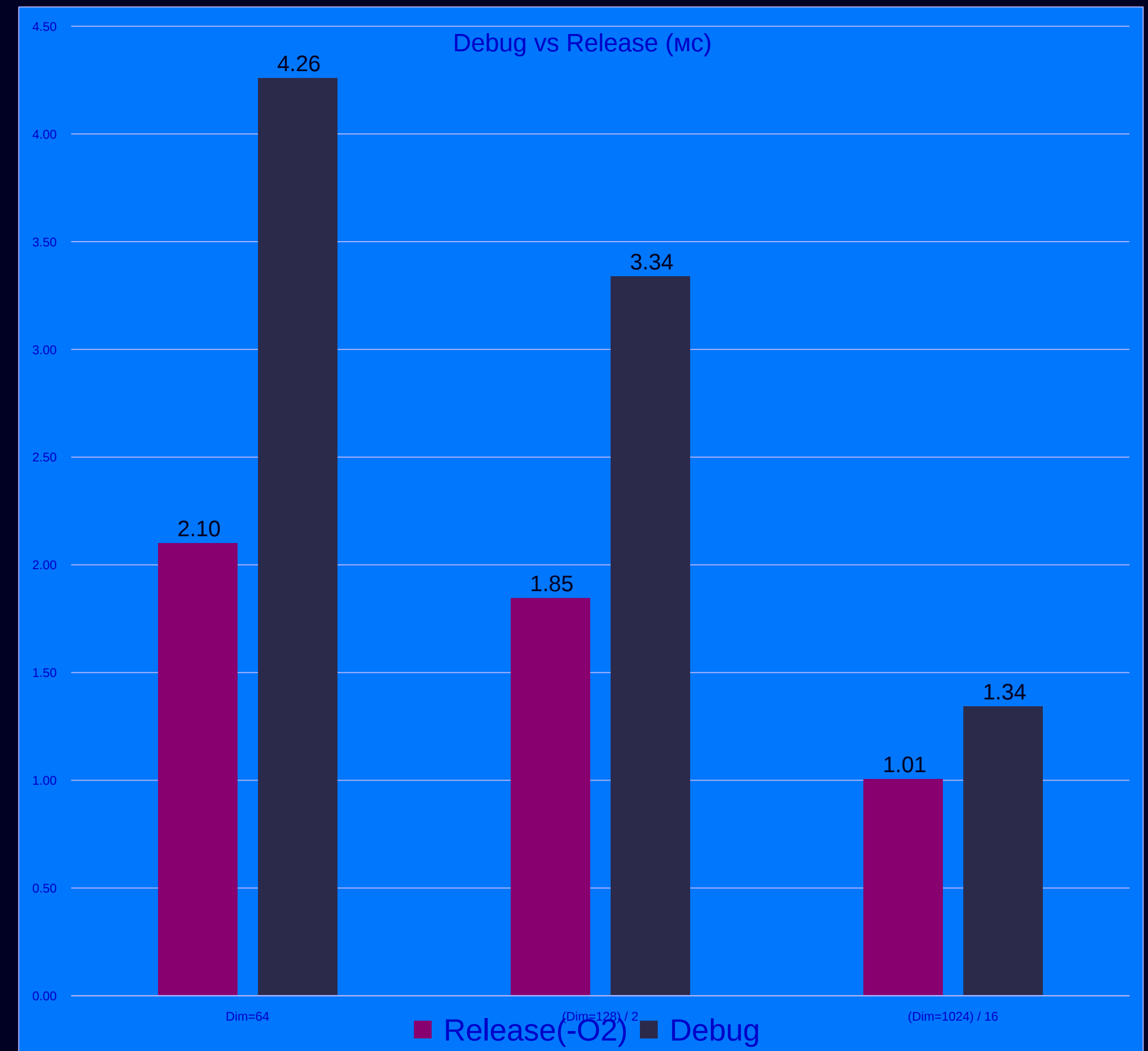
Постановка задачи

- Есть много элементов (100к - 100млн) - база документов
- Каждый элемент представлен вектором фиксированной размерности (эмбед)
 - Рассматривать будем размерности 64, 128, 1024
- Требуется вычислить DotProduct
 - Между эмбедом-запросом
 - И случайным подмножеством документов из базы (10к штук)

```
1 float DotProduct(const float* a, const float* b, size_t dim) {  
2     float res = 0;  
3     for(size_t i = 0; i < dim; ++i) {  
4         res += a[i] * b[i];  
5     }  
6     return res;  
7 }
```

Что измеряем и пример диаграммы

- Измеряем время умножения 10к случайных элементов из базы на эмбед-запрос
- На графиках время в мс (**меньше-лучше**)
- Основная размерность - 64
- Результаты для 128 и 1024 нормируем (делим на 2 и на 16)
- Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz



Есть ли потенциал для оптимизаций?

➤ Неужто компилятор не может идеально оптимизировать такой простой код?

➤ Что делает функция:

- читает из памяти в регистры
- что-то простое на регистрах

```
1 float DotProduct(const float* a, const float* b, size_t dim) {  
2     float res = 0;  
3     for(size_t i = 0; i < dim; ++i) {  
4         res += a[i] * b[i];  
5     }  
6     return res;  
7 }
```

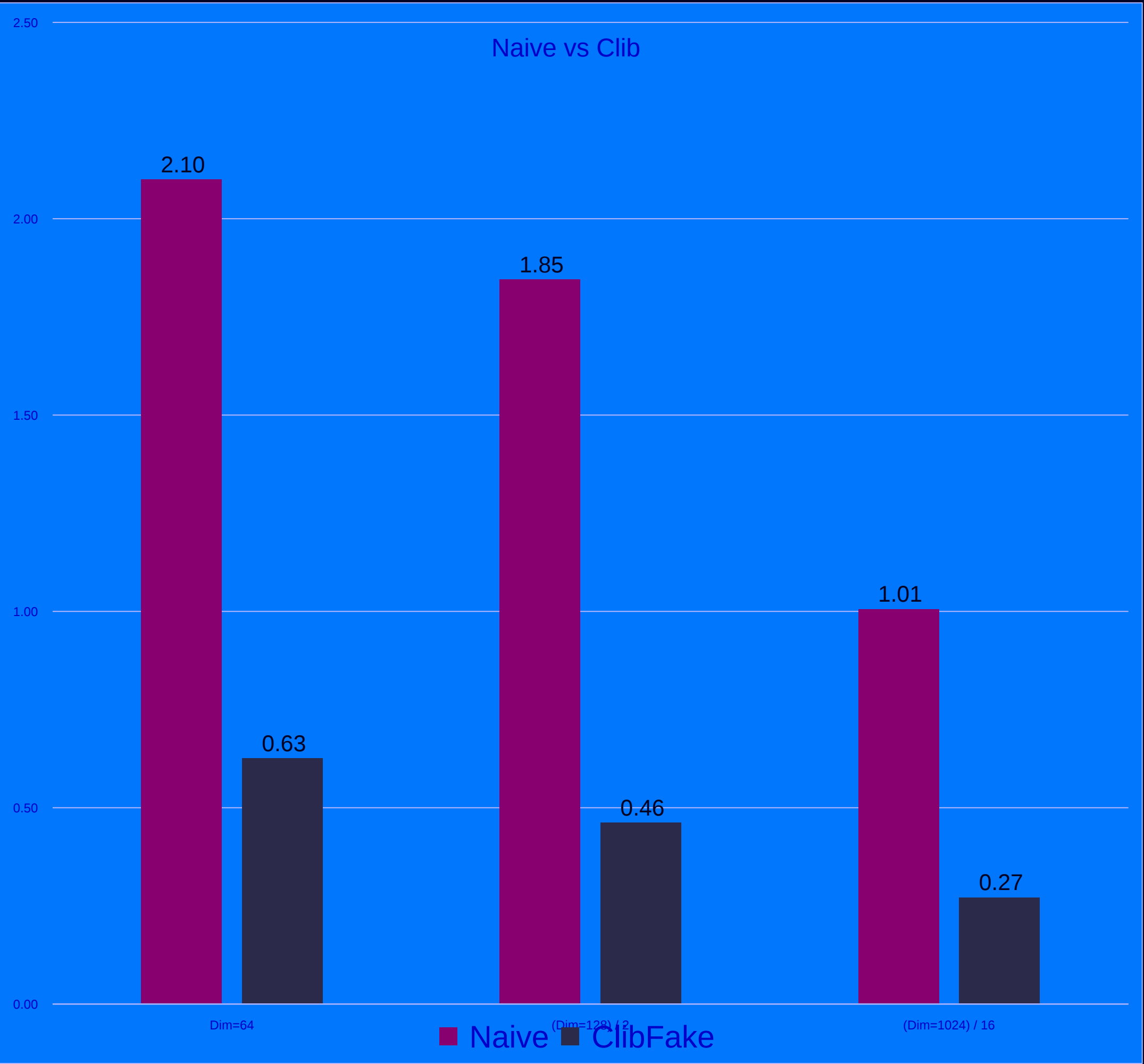
➤ Давайте сделаем аналог на заведомо хороших реализациях с похожим профилем нагрузки
(Из тестовых данных специально удалим нулевые байты)

```
1 float ClibFakeDotProduct(const float* a, const float* b, size_t dim) {  
2     return  
3     int((const char*) (memchr(a, 0, dim) ?: a) - (const char*)a)  
4     + int((const char*) (memchr(b, 0, dim) ?: b) - (const char*)b);  
5 }
```

Clib vs Naive: отставание в 3-4 раза

```
1 float DotProduct(const float* a, const float* b, size_t dim) {
2     float res = 0;
3     for(size_t i = 0; i < dim; ++i) {
4         res += a[i] * b[i];
5     }
6     return res;
7 }
```

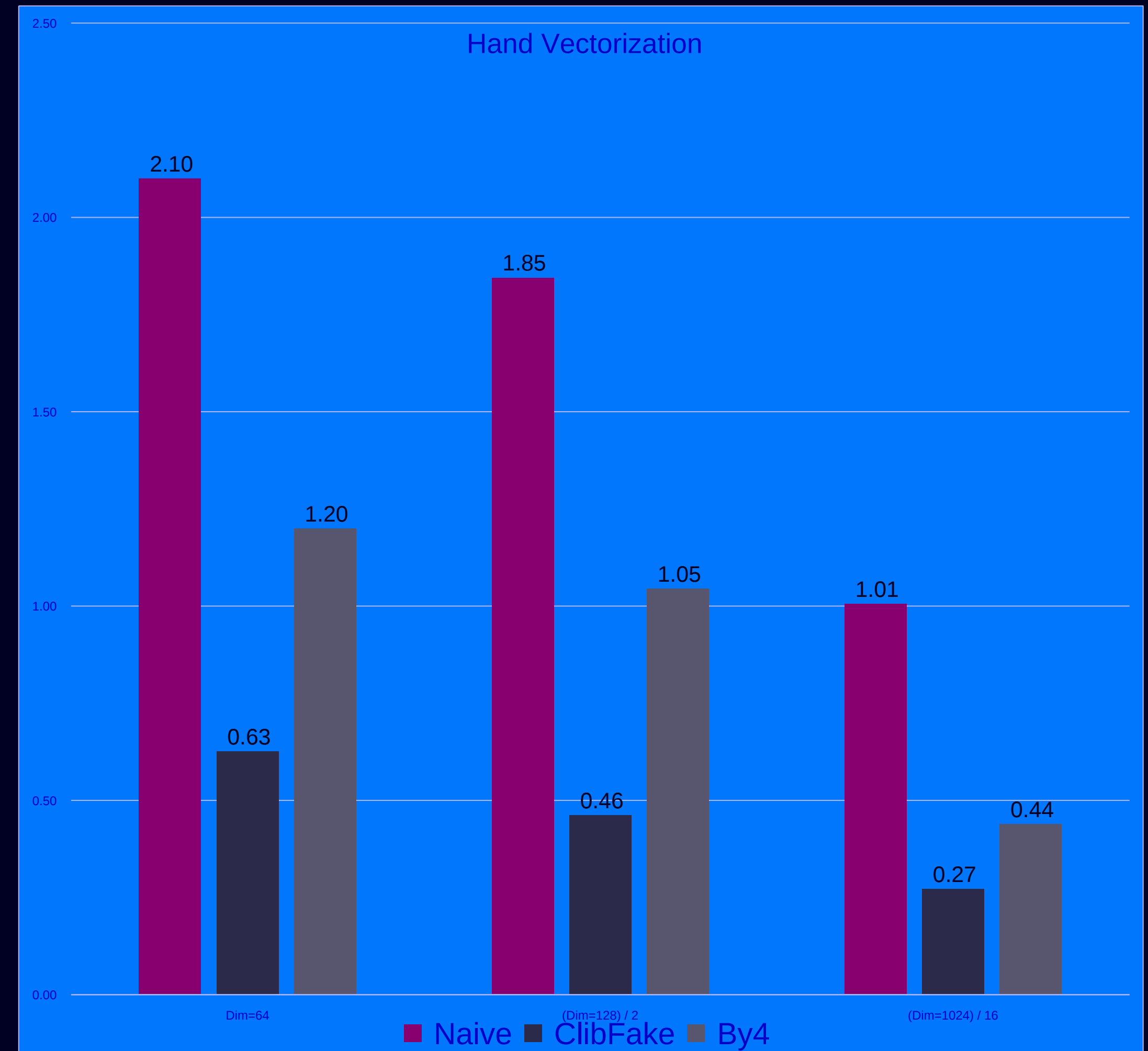
```
1 float ClibFakeDotProduct(const float* a, const float* b, size_t dim) {
2     return
3         int((const char*) (memchr(a, 0, dim) ?: a) - (const char*) a) *
4         int((const char*) (memchr(b, 0, dim) ?: b) - (const char*) b) *
5         1.0f;
6 }
```



Почему компилятор не смог?

- Версия: не разрешена векторизация (SIMD) (хотя сборка с -msse4.2)
- Поможет ли векторизовать в ручную?

```
1 float DotProductForceBy4(const float* a, const float* b, size_t dim) {  
2     float res1 = 0;  
3     float res2 = 0;  
4     float res3 = 0;  
5     float res4 = 0;  
6     for(size_t i = 0; i < dim; i += 4) {  
7         res1 += a[i] * b[i];  
8         res2 += a[i + 1] * b[i + 1];  
9         res3 += a[i + 2] * b[i + 2];  
10        res4 += a[i + 3] * b[i + 3];  
11    }  
12    return (res1 + res3) + (res2 + res4);  
13 }
```



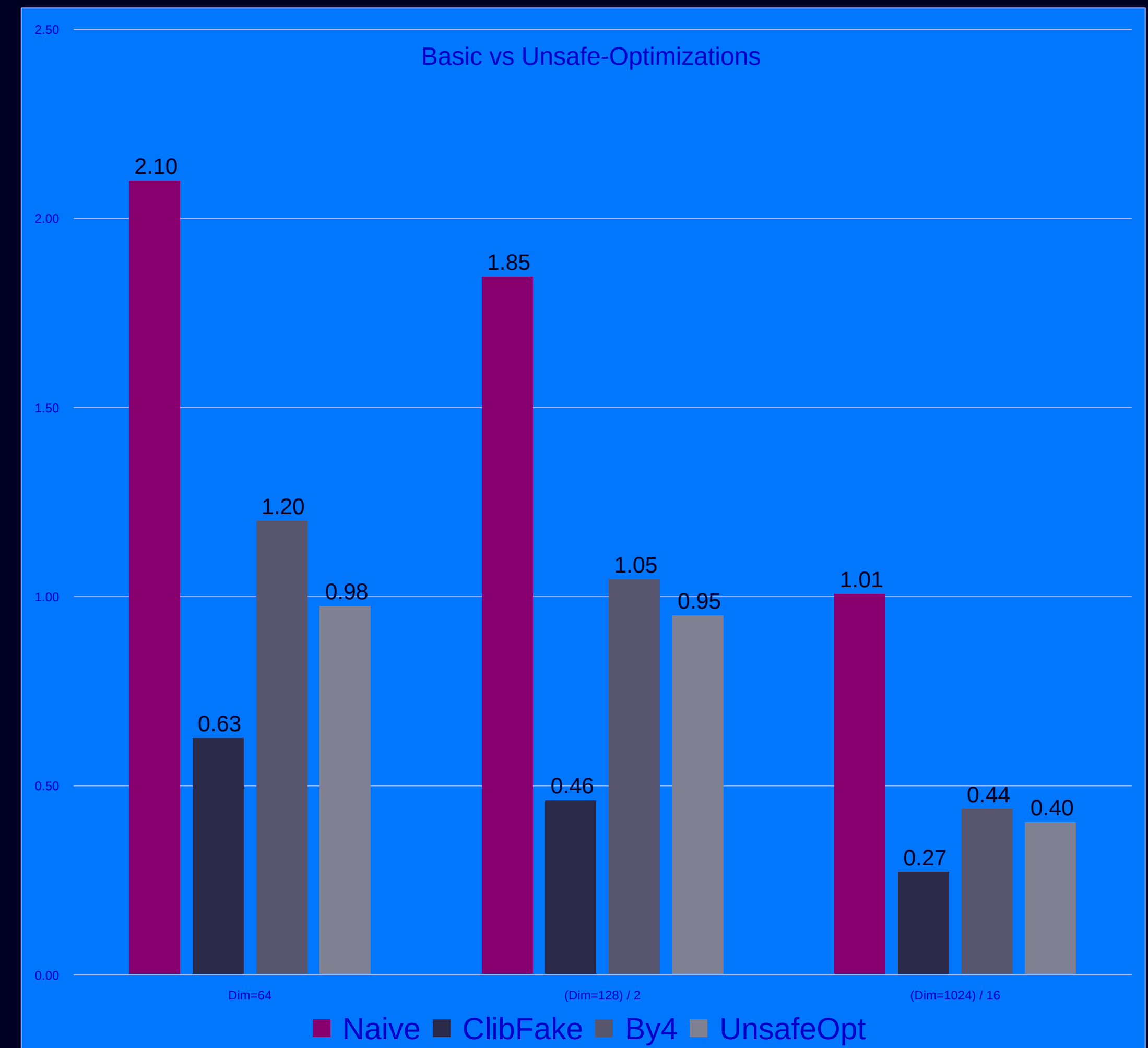
Нам не повезло с типом данных:

- Операции над float не ассоциативны (зависят от порядка выполнения)
- $(A + B) + C \neq A + (B + C)$

Есть выход: -funsafe-math-optimizations

UnsafeOpt на диаграмме = Naive + unsafe-math-optimizations

Получили примерно тоже самое что ручная векторизация, но лучше



Разрешим AVX, AVX2, AVX512

Флаги:

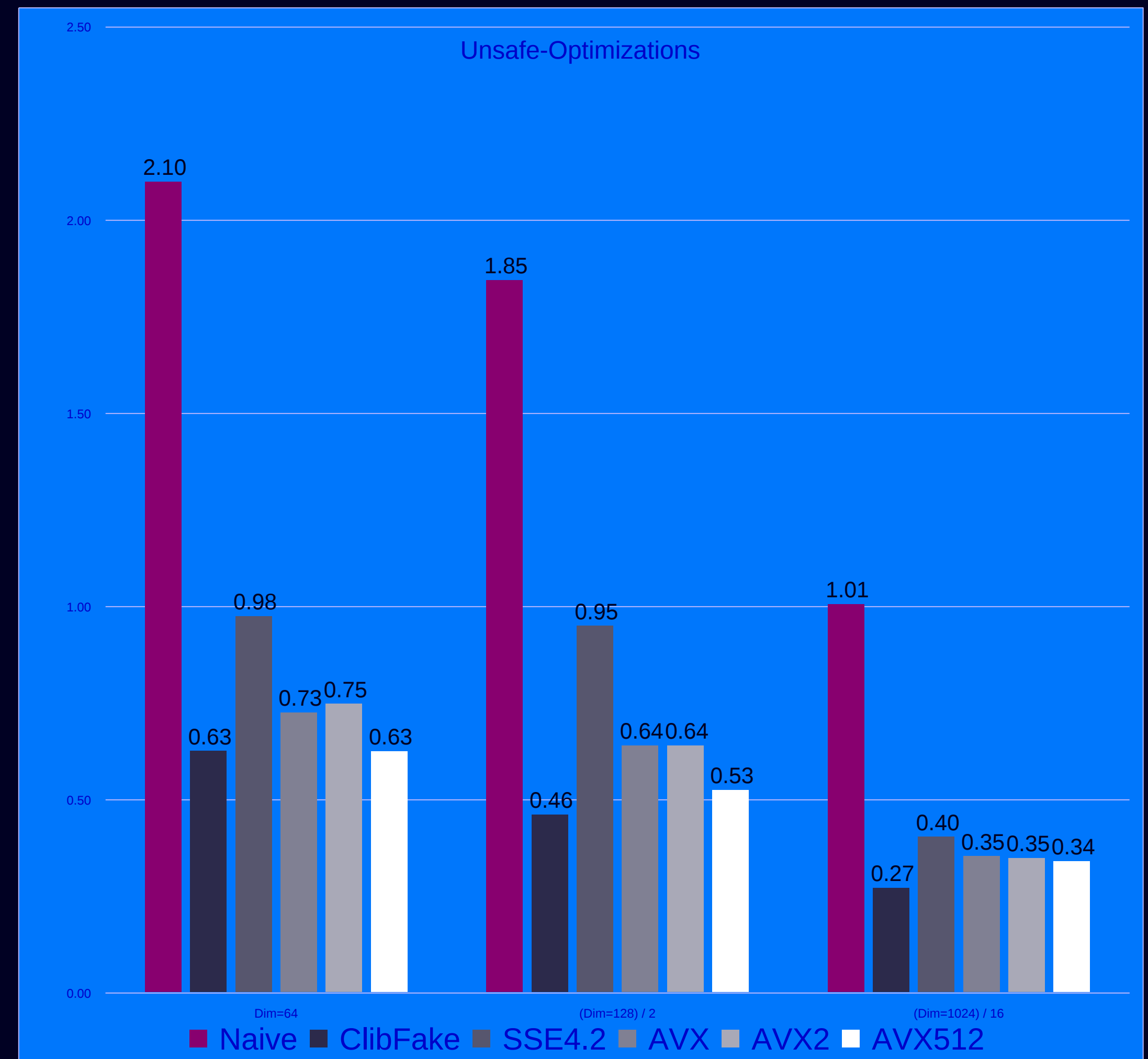
- AVX: -mavx
- AVX2: -mavx2
- AVX512: -mavx512f -mavx512bw -mavx512cd -mavx512dq -mavx512v

Достигли цели просто параметрами компиляции

Минусы unsafe-math-optimizations:

- Результат существенно зависит от флагов компиляции (и компилятора)
- Сложно тестировать

Мини-вывод: убедитесь что векторизация случилась



02

Лирическое отступление

Гетерогенный парк и runtime-cpu-dispatching

Что делать если целевых аппаратных платформ несколько?

Включать лучшие опции - это SIGILL

Включать только поддерживаемые везде - расточительно

Варианты 1: несколько сборок

- Уносит сложность на стадию деплоя
- Сложно тестировать

Вариант 2: runtime-cpu-dispatching

- накладные расходы в runtime

Runtime-cpu-dispatching

Нужно сделать несколько translation-unit

- с реализациями под каждый из требуемых наборов инструкций
- слинковать их

Реализовать логику выбора конечной реализации в текущем окружении

```
1 struct TRuntimeCpuInfoDispatch {
2     ....static const bool HaveAvx;
3     ....static const bool HaveAvx2;
4     ....static const bool HaveAvx512;
5 };
6
7 float TDetectOptimistic::DotProduct(const float* a, const float* b, size_t dim) {
8     ....if (TRuntimeCpuInfoDispatch::HaveAvx512) {
9         ....return TNaiveAvx512Auto::DotProduct(a, b, dim);
10    }
11    ....if (TRuntimeCpuInfoDispatch::HaveAvx2) {
12        ....return TNaiveAvx2Auto::DotProduct(a, b, dim);
13    }
14    ....if (TRuntimeCpuInfoDispatch::HaveAvx) {
15        ....return TNaiveAvxAuto::DotProduct(a, b, dim);
16    }
17    ....return TBy4SSE4UnsafeOpt::DotProduct(a, b, dim);
18 }
```

```
1 struct TRuntimeCpuInfoDispatch {
2     ....static const bool HaveSse40Only;
3     ....static const bool HaveAvxOnly;
4     ....static const bool HaveAvx20Only;
5 };
6
7 float TDetectPessimistic::DotProduct(const float* a, const float* b, size_t dim) {
8     ....if (TRuntimeCpuInfoDispatch::HaveSse40Only) {
9         ....return TBy4SSE4UnsafeOpt::DotProduct(a, b, dim);
10    }
11    ....if (TRuntimeCpuInfoDispatch::HaveAvxOnly) {
12        ....return TNaiveAvxAuto::DotProduct(a, b, dim);
13    }
14    ....if (TRuntimeCpuInfoDispatch::HaveAvx20Only) {
15        ....return TNaiveAvx2Auto::DotProduct(a, b, dim);
16    }
17    ....//if (TRuntimeCpuInfoDispatch::HaveAvx512) {
18    ....//    ....return TNaiveAvx512Auto::DotProduct(a, b, dim);
19    ....//}
20 }
21
```

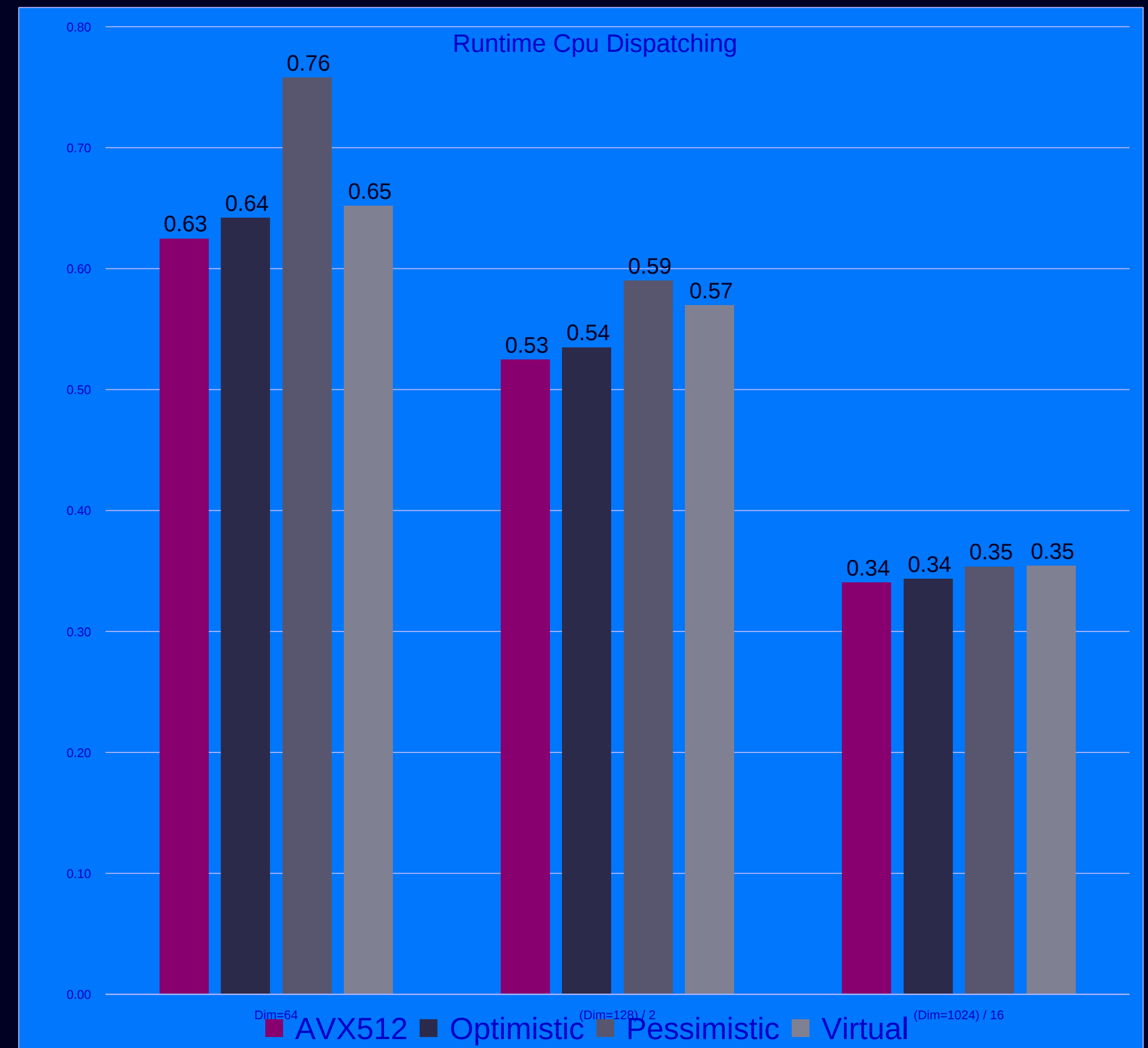

Runtime-cpu-dispatching

```
1 struct IDotProduct {
2     ...virtual float VDotProduct(const float* a, const float* b, size_t dim) const = 0;
3     ...virtual ~IDotProduct() {}
4 };
5
6 struct TRuntimeCpuInfoDispatch {
7     ...static const std::unique_ptr<const IDotProduct> Fabric;
8 };
9
10 float TVirtualJump::DotProduct(const float* a, const float* b, size_t dim) {
11     ...return TRuntimeCpuInfoDispatch::Fabric->VDotProduct(a, b, dim);
12 }
```

Не взятые if-ы - дорого

Виртуальный вызов - хороший дефолт

Чем меньше операция под rcd - тем выше накладные расходы



03

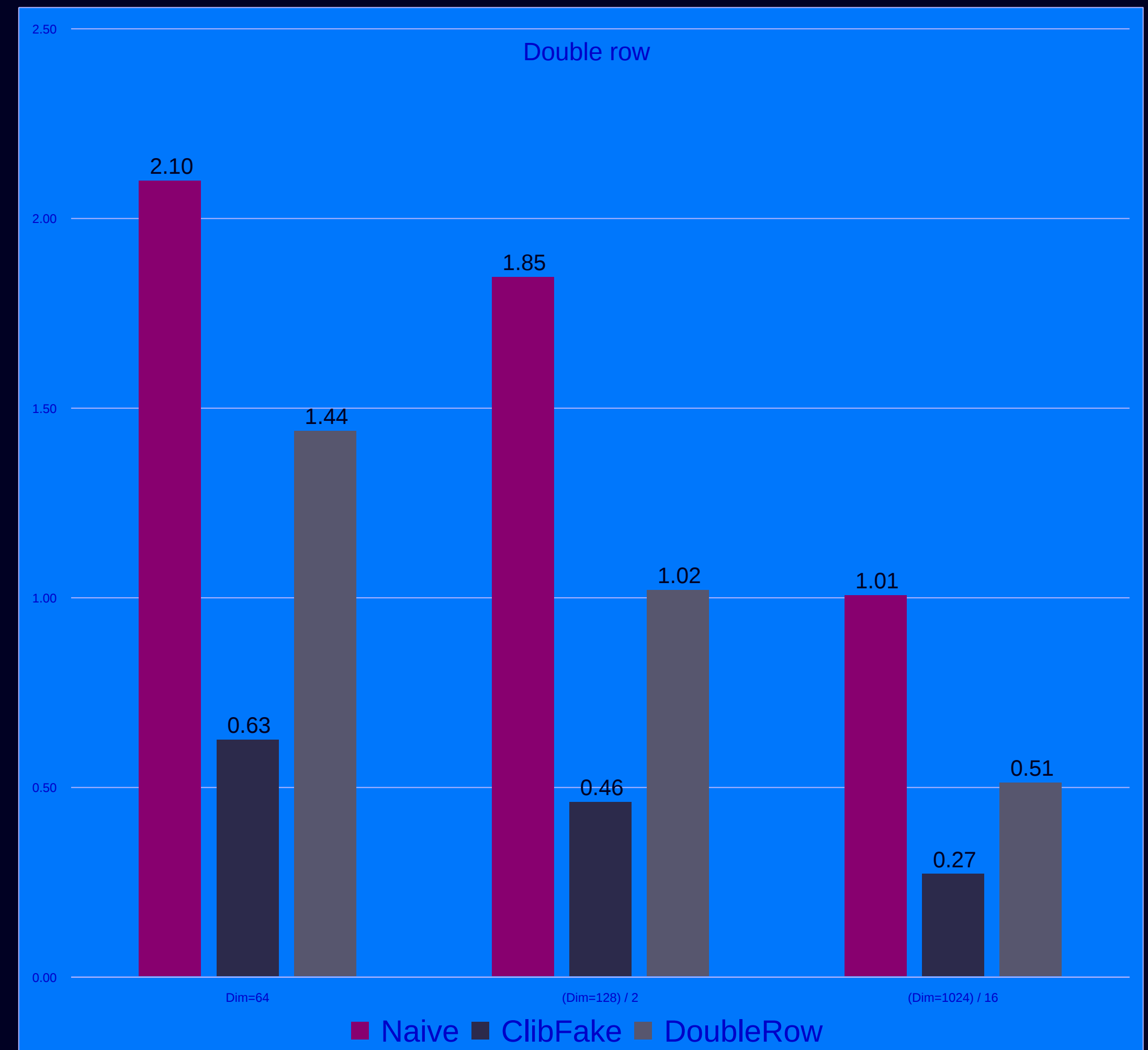
Оптимизации lvl2

Смотрим на более крупные блоки

Макро-оптимизации

- Вспомним что эмбед-запрос - всего один, а документов много.
- Важно: эта оптимизация ортогональна векторизации

```
1  std::pair<float, float> DotProductDouble(  
2  ...const float* a,  
3  ...const float* b, const float* c,  
4  ...size_t dim  
5  ) {  
6  ...float res1 = 0;  
7  ...float res2 = 0;  
8  ...for(size_t i = 0; i < dim; ++i) {  
9  ...    res1 += a[i] * b[i];  
10 ...    res2 += a[i] * c[i];  
11 ...}  
12 ...return {res1, res2};  
13 }
```

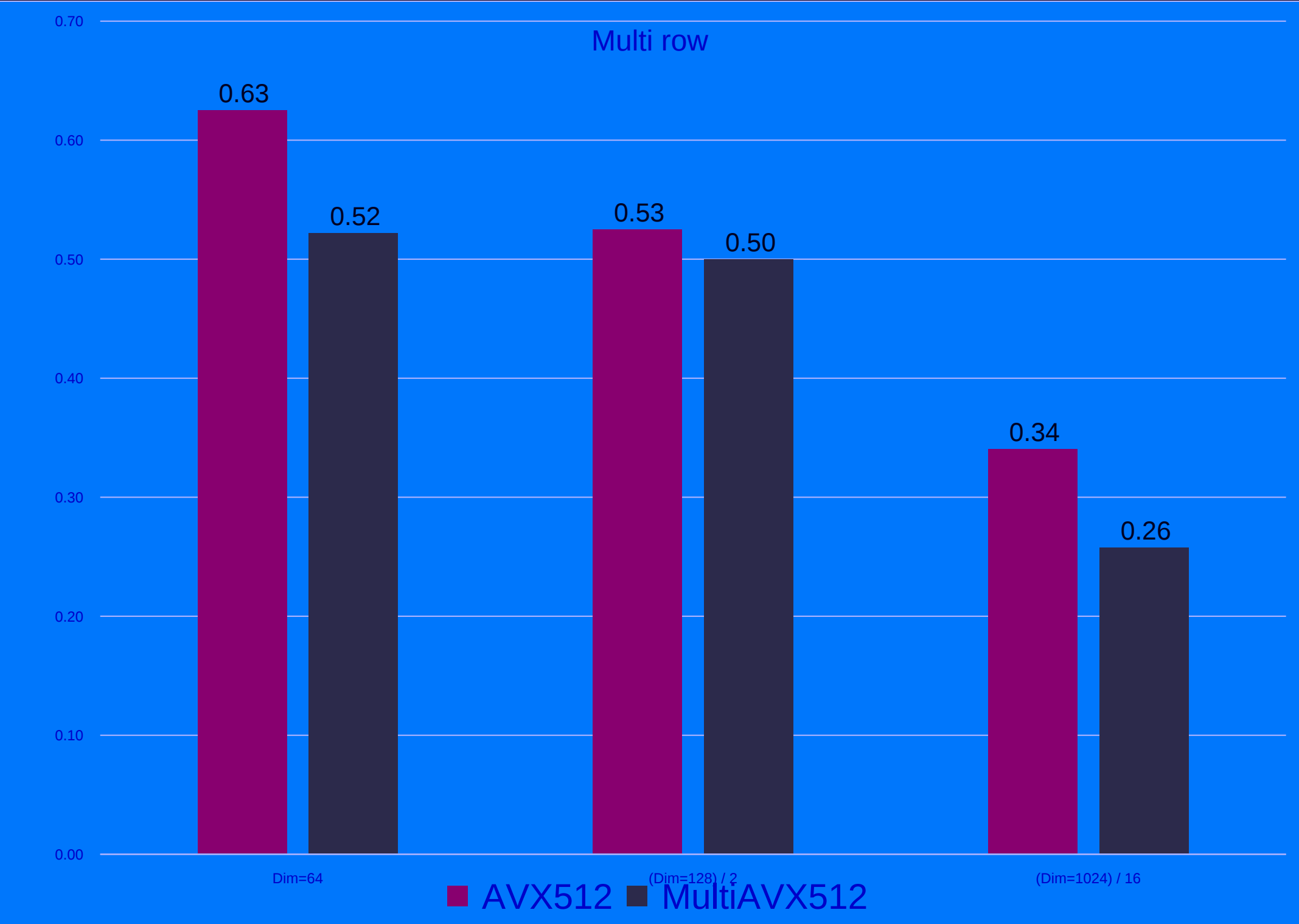


Совместим с векторизацией: 20% выигрыша

```
1 void MultiDotProduct(  
2     ....const float* a,  
3     ....const float* allB,  
4     ....size_t dim,  
5     ....const uint32_t* elemsIds,  
6     ....size_t elemsNum,  
7     ....float* results  
8 ) {  
9     ....size_t constexpr Step = 4;  
10    ....size_t e = 0;  
11    ....float tmp0 = 0;  
12    ....float tmp1 = 0;  
13    ....float tmp2 = 0;  
14    ....float tmp3 = 0;  
15    ....for(; e + Step <= elemsNum; e += Step) {  
16        ....const float* r0 = allB + dim * elemsIds[e + 0];  
17        ....const float* r1 = allB + dim * elemsIds[e + 1];  
18        ....const float* r2 = allB + dim * elemsIds[e + 2];  
19        ....const float* r3 = allB + dim * elemsIds[e + 3];  
20  
21        ....for(size_t i = 0; i < dim; i += 1) {  
22            ....tmp0 += a[i] * r0[i];  
23            ....tmp1 += a[i] * r1[i];  
24            ....tmp2 += a[i] * r2[i];  
25            ....tmp3 += a[i] * r3[i];  
26        }  
27  
28        ....results[e + 0] = tmp0;  
29        ....results[e + 1] = tmp1;  
30        ....results[e + 2] = tmp2;  
31        ....results[e + 3] = tmp3;  
32    }  
33    ....for(; e < elemsNum; e += 1) { //tail  
34        ....results[e] = DotProduct(a, allB + dim * elemsIds[e], dim);  
35    }  
36 }
```

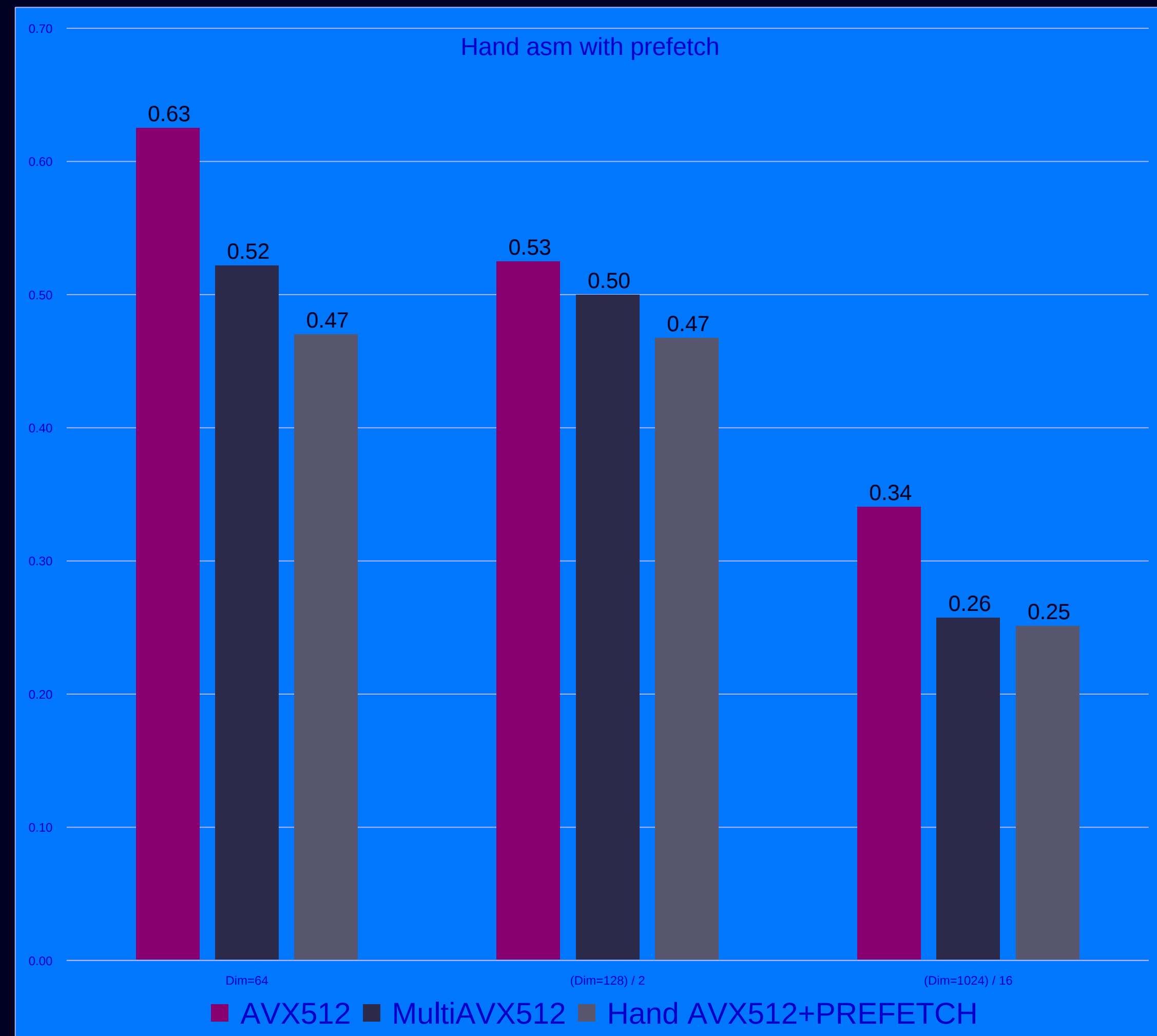
Идём по 4 документа за раз

Включаем опции оптимизации и векторизации



Ещё 10% за prefetch

```
1 MultiDotProduct(  
2     ....const float* a,  
3     ....const float* allB,  
4     ....size_t dim,  
5     ....const uint32_t* elemsIds,  
6     ....size_t elemsNum,  
7     ....float* results  
8 ) {  
9     ....size_t e = 0;  
10    ....constexpr size_t Step = 4;  
11    ....constexpr size_t ElemsInVec = (sizeof(_mm512) / sizeof(float));  
12    ....for(; e + Step <= elemsNum; e += Step) {  
13        ...._mm512 sum0 = _mm512_setzero_ps();  
14        ...._mm512 sum1 = _mm512_setzero_ps();  
15        ...._mm512 sum2 = _mm512_setzero_ps();  
16        ...._mm512 sum3 = _mm512_setzero_ps();  
17  
18        ....const float* e0 = allB + dim * elemsIds[e + 0];  
19        ....const float* e1 = allB + dim * elemsIds[e + 1];  
20        ....const float* e2 = allB + dim * elemsIds[e + 2];  
21        ....const float* e3 = allB + dim * elemsIds[e + 3];  
22  
23        ....for(size_t position = 0; position < dim; position += ElemsInVec) {  
24            ...._mm512 left = _mm512_load_ps(a + position);  
25            ....sum0 = _mm512_fmadd_ps(left, _mm512_load_ps(e0 + position), sum0);  
26            ....sum1 = _mm512_fmadd_ps(left, _mm512_load_ps(e1 + position), sum1);  
27            ....sum2 = _mm512_fmadd_ps(left, _mm512_load_ps(e2 + position), sum2);  
28            ....sum3 = _mm512_fmadd_ps(left, _mm512_load_ps(e3 + position), sum3);  
29        ....}  
30        ....__builtin_prefetch(a, 0);  
31        ....__builtin_prefetch(allB + dim * elemsIds[e + 4 + 0], 0);  
32        ....__builtin_prefetch(allB + dim * elemsIds[e + 4 + 1], 0);  
33        ....__builtin_prefetch(allB + dim * elemsIds[e + 4 + 2], 0);  
34        ....__builtin_prefetch(allB + dim * elemsIds[e + 4 + 3], 0);  
35  
36        ....results[e + 0] = _mm512_reduce_add_ps(sum0);  
37        ....results[e + 1] = _mm512_reduce_add_ps(sum1);  
38        ....results[e + 2] = _mm512_reduce_add_ps(sum2);  
39        ....results[e + 3] = _mm512_reduce_add_ps(sum3);  
40    ....}  
41    ....for(; e < elemsNum; e += 1) {  
42        ....results[e] = DotProduct(a, allB + dim * elemsIds[e], dim);  
43    ....}  
44 }
```



Зачем бывает нужно писать на asm

Оптимизатор не всегда справляется

Точно задать семантику (и порядок вычислений)

Что позволяет для разных платформ написать эквивалентные по результату реализации

- Возможно отказаться от тех вариантов которые сложно эмулировать на более старых архитектурах
- Использование более базовых версий для обработки хвостов

04

Оптимизируем память

Как быть - есть хочется растить число элементов в базе?

Рассмотрим работу с запакованным форматом базы

«О бедном RAM замолвите словечко»

Как быть - есть хочется растить число элементов в базе?

Вариант 1: больше железа и/или шардирование

- увеличивает стоимость эксплуатации
- шардирование - всегда сложно

Вариант 2: сжимать вектора

- дорого по сри (?)
- ухудшает точность

Какие есть подходы для сжатия?

Стандартный метод - равномерное в uint8

- Fixed-length преобразование
- Восстановление - линейная операция
- Полученной точности хватает для широкого круга задач

Есть варианты улучшений точности

- Отсекать хвосты
- Свои параметры для каждой координаты

Справа - стандартный способ «вылить воду из чайника» - разжатие с буфером

```
1 void MultiDotProduct(  
2     ....const float* a,  
3     ....const uint8_t* allB,  
4     ....size_t dim,  
5     ....const uint32_t* elemsIds,  
6     ....size_t elemsNum,  
7     ....float bias,  
8     ....float coeff,  
9     ....float* results  
10 ) {  
11     ....std::vector<float> buf(dim);  
12     ....for(size_t e = 0; e < elemsNum; e += 1) {  
13         ....const uint8_t* rowPtr = allB + dim * elemsIds[e];  
14         ....for(size_t i = 0; i < dim; i += 1) {  
15             ....buf[i] = coeff * rowPtr[i] + bias;  
16         ....}  
17         ....results[e] = DotProduct(a, buf.cbegin(), dim);  
18     ....}  
19 }
```

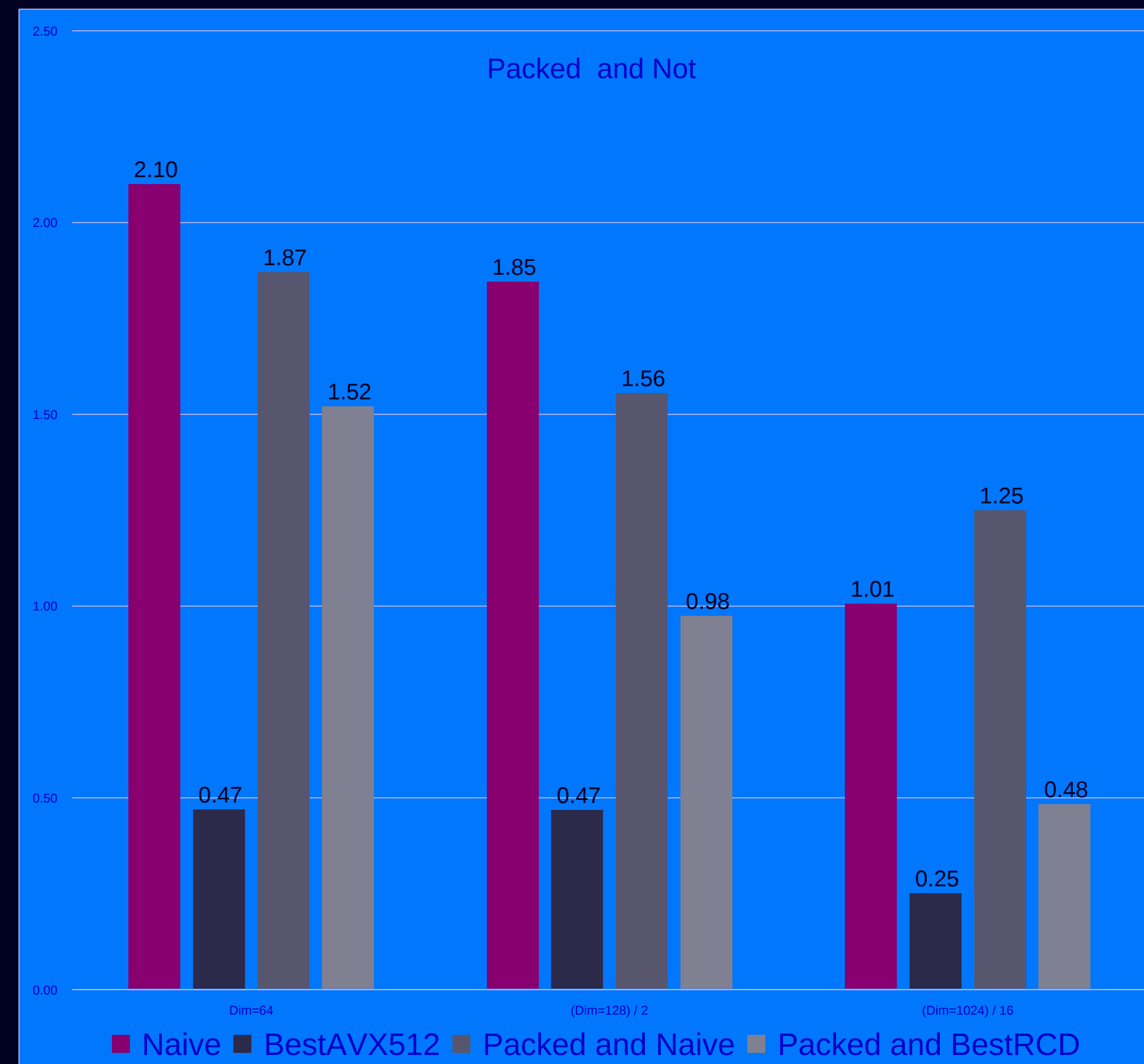
Разжатие с буфером: протеворечивые результаты

Без unsafe оптимизаций - заметное улучшение на малых размерностях

- Меньше memory-footprint
- Буфер всегда в кэше

При использовании оптимизированной версии **значительно ухудшение** (выигрыш теряется)

- Разжатие очень дорогое
- И требует доп записи в память

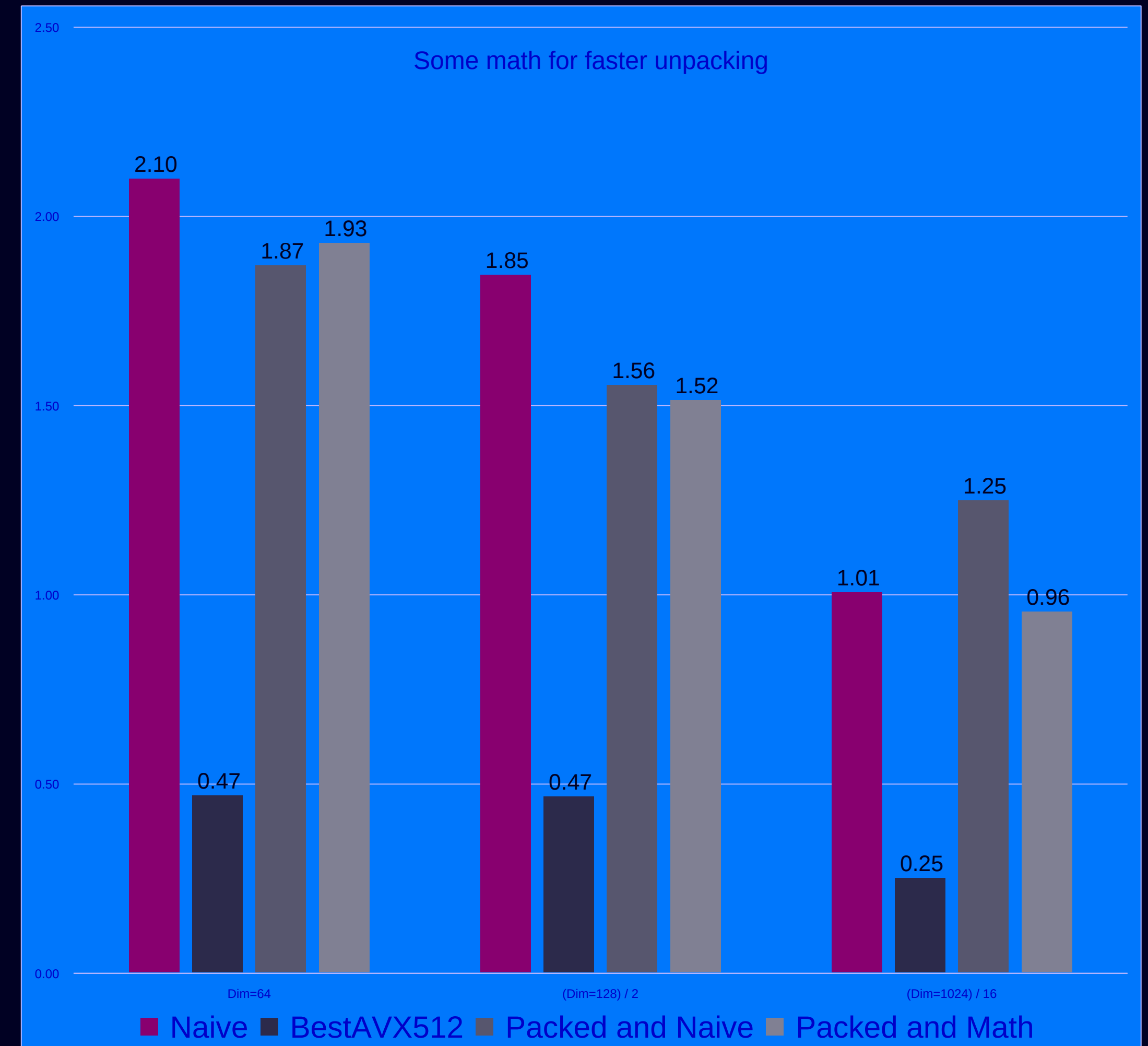


Теоретические оптимизации не всегда улучшают

$\text{sum} \{a * (c * b + d)\} = \{\text{sum} (a * b)\} * c + d'$

Вынесли часть работы за скобки - местами сделали хуже, местами - лучше

```
1 void MultiDotProduct(  
2     ...const float* a,  
3     ...const uint8_t* allB,  
4     ...size_t dim,  
5     ...const uint32_t* elemsIds,  
6     ...size_t elemsNum,  
7     ...float bias,  
8     ...float coeff,  
9     ...float* results  
10 ) {  
11     ...float bb = 0;  
12     ...for(size_t i = 0; i < dim; i += 1) {  
13         ...bb += a[i];  
14     ...}  
15     ...bb *= bias;  
16  
17     ...for(size_t e = 0; e < elemsNum; e += 1) {  
18         ...results[e] = 0;  
19         ...const uint8_t* rowPtr = allB + dim * elemsIds[e];  
20         ...for(size_t i = 0; i < dim; i += 1) {  
21             ...results[e] += a[i] * float(rowPtr[i]);  
22         ...}  
23         ...results[e] = results[e] * coeff + bb;  
24     ...}  
25 }
```

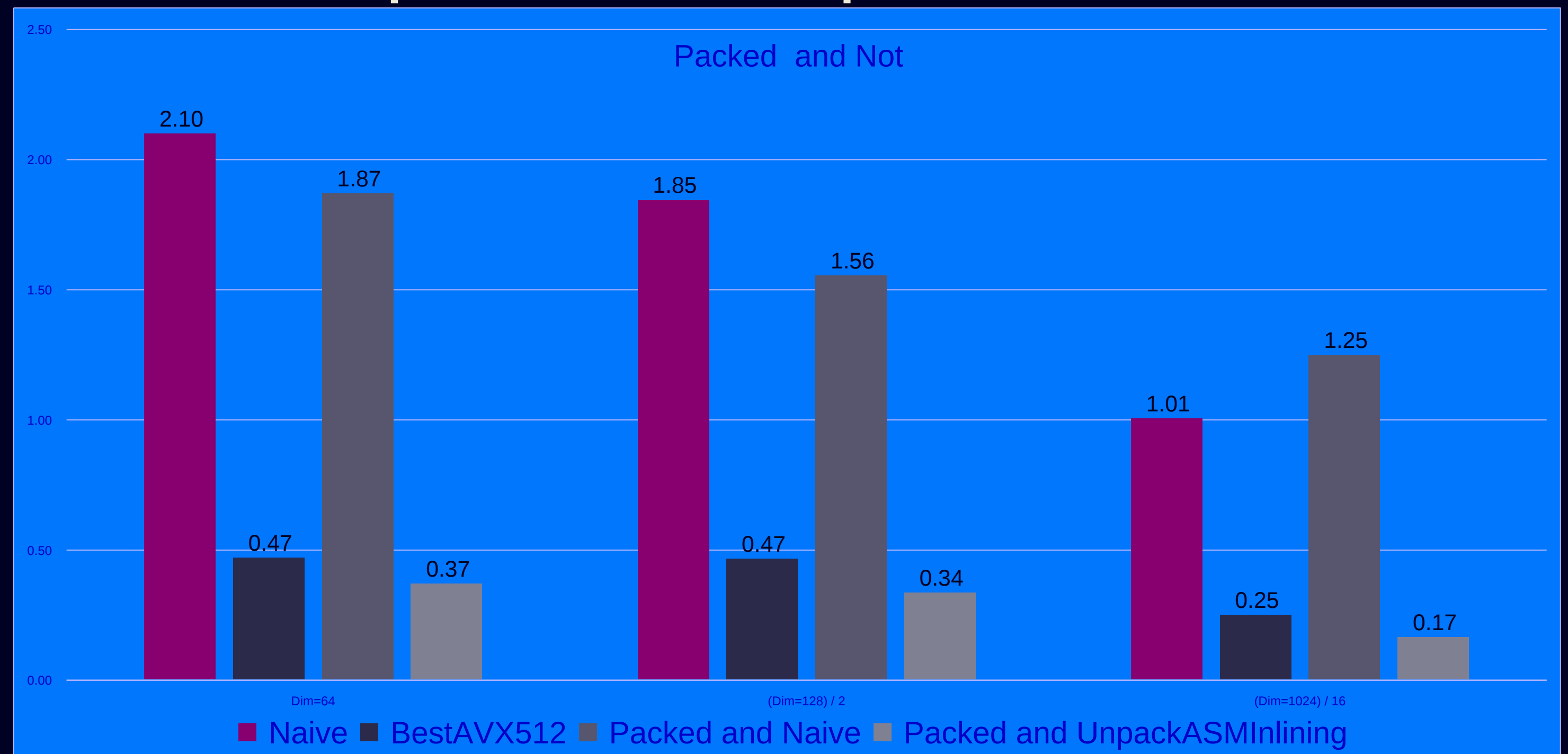


Инлайнинг разжатия в умножение + asm

```
1 void MultiDotProduct(  
2     const float* a,  
3     const uint8_t* allB,  
4     size_t dim,  
5     const uint32_t* elemsIds,  
6     size_t elemsNum,  
7     float bias,  
8     float coeff,  
9     float* results  
10 ) {  
11     float bb = 0;  
12     for(size_t i = 0; i < dim; i += 1) {  
13         bb += a[i];  
14     }  
15     bb *= bias;  
16     size_t e = 0;  
17     constexpr size_t Step = 4;  
18     constexpr size_t ElemsInVec = (sizeof(__m512) / sizeof(float));  
19     for(; e + Step <= elemsNum; e += Step) {  
20         __m512 sum0 = _mm512_setzero_ps();  
21         //repeat for sum1, sum2, sum3  
22         const uint8_t* e0 = allB + dim * elemsIds[e + 0];  
23         // repeat for e1, e2, e3  
24         for(size_t position = 0; position < dim; position += ElemsInVec) {  
25             __m512 left = _mm512_load_ps(a + position);  
26             __m512 right0 = _mm512_cvtepi32_ps(  
27                 _mm512_cvtepu16_epi32(  
28                     _mm256_cvtepu8_epi16(  
29                         _mm_loadu_epi8(e0 + position)  
30                     )  
31                 )  
32             );  
33             // repeat for right1, right2, right3  
34             sum0 = _mm512_fmadd_ps(left, right0, sum0);  
35             //repeat for sum1, sum2, sum3  
36         }  
37         results[e + 0] = _mm512_reduce_add_ps(sum0) * coeff + bb;  
38         //repeat for results[e + 1], results[e + 2], results[e + 3]  
39     }  
40     MultiDotProductTailElems(a, allB, dim, elemsIds + e, elemsNum - e, bias,  
41     results + e * 4);  
42 }  
43 }
```

- Написали явный inline разжатия, взяв все прошлые оптимизации
- Число обращений в память уменьшилось
- Сложность работы с регистрами - не ключевой вопрос

В итоге - выиграли и память и сри



Лирическое отступление 2: Float16

Отлично работающая в ml технология (нейронки + гри)

Неплохая точность для ml задач

Хороший дефолт для сжатия fp32 чисел из $[-1,1]$

Есть поддержка в железе (векторизованные конвертации в fp32)

Нет поддержки в языке

В большинстве случаев также можно ожидать выигрыша за счёт уменьшения memory-footprint

Примеры библиотек

- > <http://half.sourceforge.net/>
- > <https://github.com/catboost/catboost/tree/master/library/cpp/float16>

Выводы

Для микрооптимизаций:

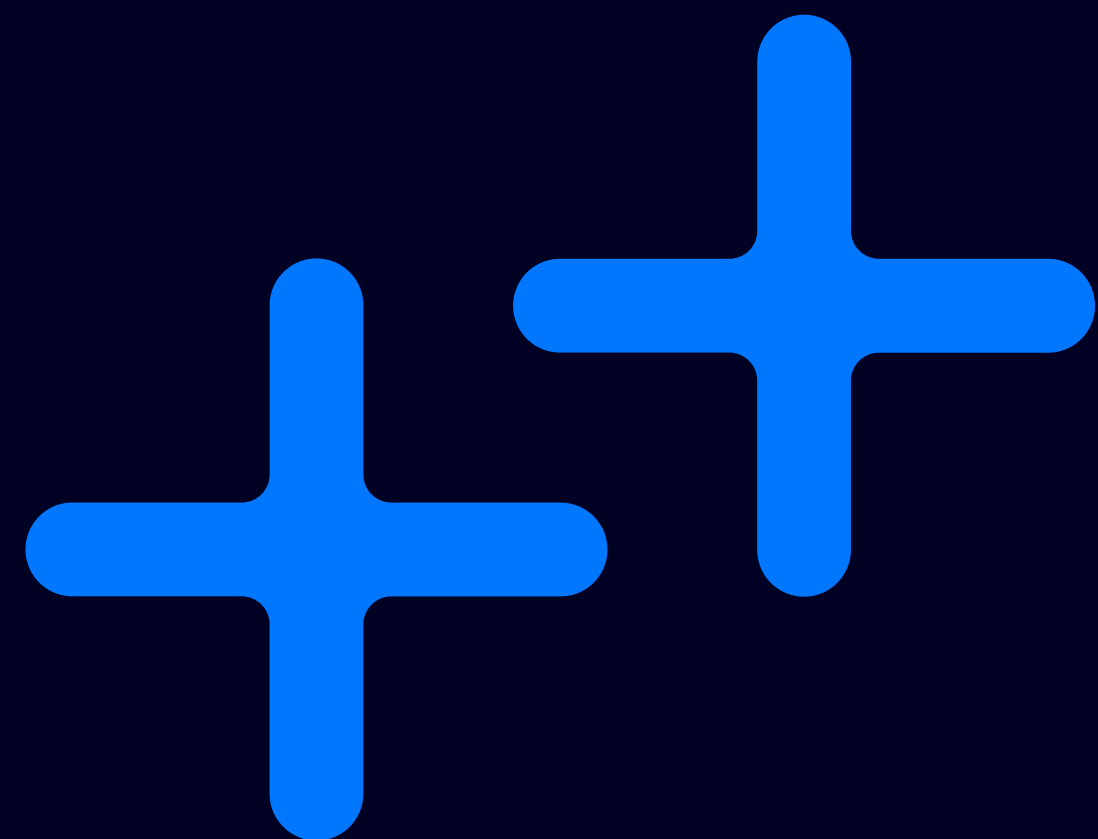
- Желательно иметь опорную точку
- Важно следить за числом походов в память
- И векторизацией

Оптимизации возможна на уровне архитектуры:

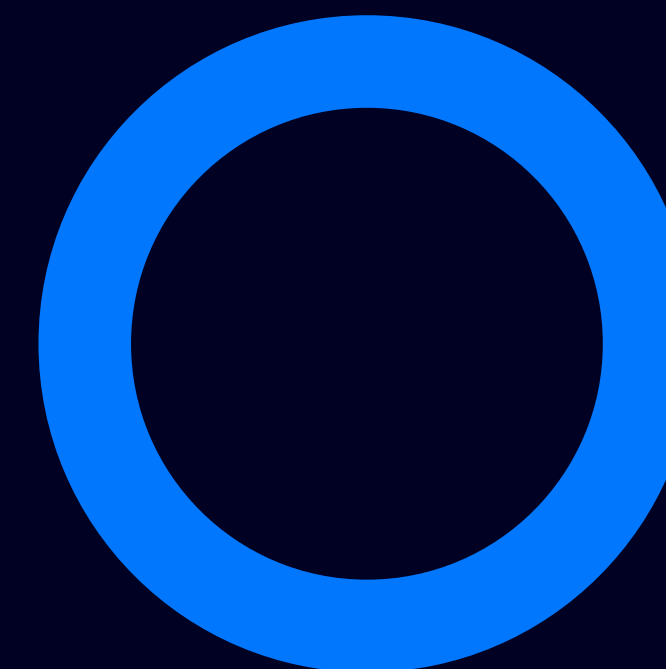
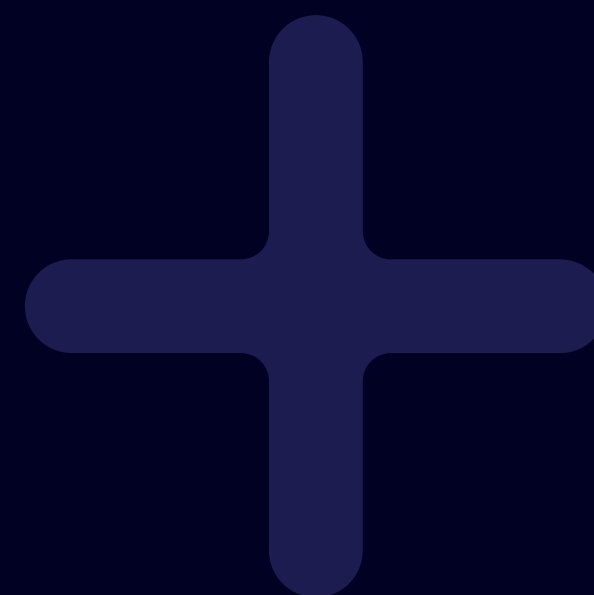
- Часть оптимизаций будет не возможна, если не правильно проведены уровни абстракций
- Может быть дополнительный налог при неправильном выборе (как в случае с rcd)

Семантические оптимизации

- Могут оказаться в разы эффективнее zero-diff оптимизаций



Оптимизируйте на всю глубину: от постановки до инструкций





C++ZeroCost
Conf 

Спасибо за внимание

Хузиев Ильнур

Руководитель группы разработки в web-поиске

https://github.com/ilnurkh/conf_zero_cost_2021



ilnurkh@yandex-team.ru

@ilnurkh



Yandex for **developers** *//>