



НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМ. Н.И. ЛОБАЧЕВСКОГО

ЦЕНТР КОМПЕТЕНЦИЙ ONEAPI В ННГУ

ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МАТЕМАТИКИ И МЕХАНИКИ



НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМ. Н.И. ЛОБАЧЕВСКОГО
ЦЕНТР КОМПЕТЕНЦИЙ oneAPI в ННГУ
ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МАТЕМАТИКИ И МЕХАНИКИ

***ВВЕДЕНИЕ В АНАЛИЗ ПРОИЗВОДИТЕЛЬНОСТИ
И ОПТИМИЗАЦИЮ ПРОГРАММ***

**Векторизация циклов. Общие
принципы и использование
компилятора**

Содержание

- ❑ Введение
- ❑ Векторные расширения: основные принципы
- ❑ Примеры: обработка изображений
- ❑ Заключение
- ❑ Литература

Введение

- Какой бывает параллелизм? Параллелизм на уровне *команд* и *данных*.
- Классификация по Флинну:

	Одиночный поток данных (single data stream)	Множественный поток данных (multiple data stream)
Одиночный поток команд (single instruction stream)	<i>SISD (single instruction, single data)</i> Последовательное выполнение, параллелизма нет	<u><i>SIMD (single instruction, multiple data)</i></u> Одна инструкция выполняется для вектора данных
Множественный поток команд (multiple instruction stream)	<i>MISD (multiple instruction, single data)</i> Редко встречающаяся архитектура; используется, например, для защиты от сбоев	<i>MIMD (multiple instruction, multiple data)</i> Традиционные многопроцессорные системы (многоядерные процессоры, кластера)

Введение

- ❑ Многие процессорные архитектуры поддерживают парадигму SIMD.
- ❑ Рассмотрим, как это можно эффективно задействовать в современных центральных процессорах.
- ❑ **План:**
 - Ознакомиться с SIMD-вычислениями в современных CPU.
 - Рассмотреть случаи, в которых современные оптимизирующие компиляторы порождают эффективный векторный код. Сформулировать общие правила и рекомендации.
 - На примере из области обработки изображений проиллюстрировать некоторые проблемы, возникающие при векторизации вычислений, и способы их решения, если это возможно.

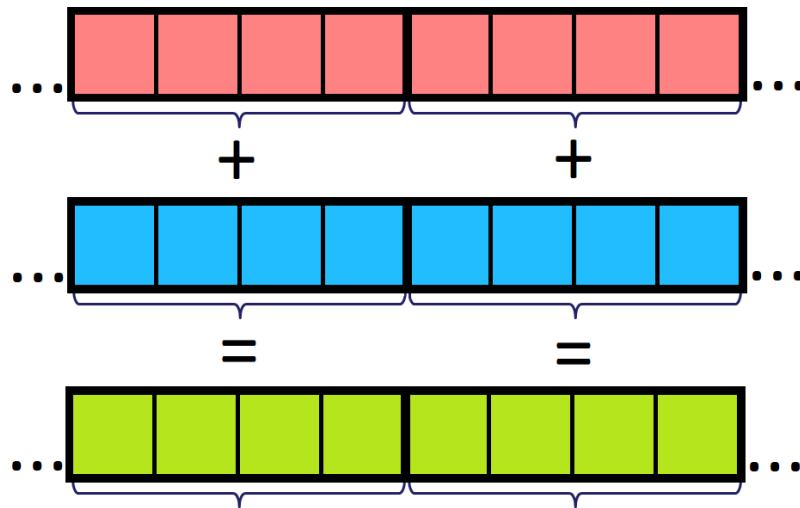
ВЕКТОРНЫЕ РАСШИРЕНИЯ: ОСНОВНЫЕ ПРИНЦИПЫ

SIMD

- ❑ *SIMD (single instruction, multiple data)* – принцип в современных компьютерных системах, подразумевающий обработку множества данных одной инструкцией (*параллелизм на уровне данных*).
- ❑ Выполнение одной операции одновременно над несколькими элементами.

Пример:
сложение векторов

Несколько одинаковых операций над отдельными однотипными элементами можно заменить одной операцией над вектором данных



SIMD

- ❑ Многие современные архитектуры поддерживают векторные вычисления (Intel, AMD, ARM, ...).
- ❑ Векторные вычисления – основа для различных ускорителей, в том числе GPU.
- ❑ Рассмотрим принцип векторных вычислений на примере SIMD-расширений для центральных процессоров Intel.

Набор инструкций	Длина регистра	Комментарий
MMX	64 бит	Первый набор SIMD-команд для архитектуры x86, обрабатывает только целые числа
SSE, SSE2, ..., SSE4	128 бит	Добавлены операции над числами с плавающей точкой
AVX, AVX2	256 бит	Добавлены инструкции с 3 операндами, FMA
AVX512	512 бит	Добавлено множество разнообразных полезных инструкций

Типы машинных инструкций (x86-64)

➤ ALU-операции

- Арифметические: add, sub, mul, ...
- Побитовые: and, or, xor, not, shl, shr, ...
- FMA ($d = a + b \cdot c$): FMA3 (d совпадает с a , b или c), FMA4

➤ Операции с памятью

- «Классические» операции загрузки/выгрузки из памяти в регистр: mov
- Специальные SIMD-инструкции: gather/scatter, broadcast, insert/extract, shuffles, ...
- Могут использоваться маски

➤ Control Flow инструкции

- Передача потока выполнения инструкции в определенной ячейке памяти: jmp
- Операции сравнения: cmp, conditional jmp, ...
- Вызов подпрограмм: call, ret

Способы векторизации

- ❑ Задействовать SIMD-инструкции можно несколькими способами, например:
 - Программировать на ассемблере.
 - Использовать intrinsics (специальные функции, вызов которых при компиляции заменяется на одну или несколько ассемблерных инструкций).
 - Не кроссплатформенные и сложные в реализации решения.
 - Надеяться на оптимизирующий компилятор.
 - Некоторые современные оптимизирующие компиляторы хорошо справляются с векторизацией циклов.
 - Чем проще и понятнее реализован цикл, тем проще компилятору.
 - Использовать готовые библиотеки с векторизованными функциями.
 - В библиотеках есть не все, что может быть нужно.
- ❑ Далее представлены некоторые принципы и приемы, полезные при автовекторизации.

Векторизация циклов

- ❑ Как компилятор преобразует цикл перед тем, как векторизовать?

```
for (int i = 0; i < N; i++)  
    d[i] = a[i] * b[i] + c[i];
```

Простой цикл
с операцией FMA

Пусть длина
векторного регистра
4 элемента

- ❑ Цикл разбивается на 2-3 цикла:
основной, обрабатывающий почти
все итерации, и *пролог* и/или
эпилог, обрабатывающие остаток.
- ❑ Тело основного цикла заменяется
одной SIMD инструкцией.
- ❑ Пролог/эпилог могут быть
скалярными или векторными.
- ❑ Пролог может обеспечить
выравнивание адресов в основном
цикле.

Пролог
(peel loop)

Основной
цикл

Эпилог
(remainder
loop)

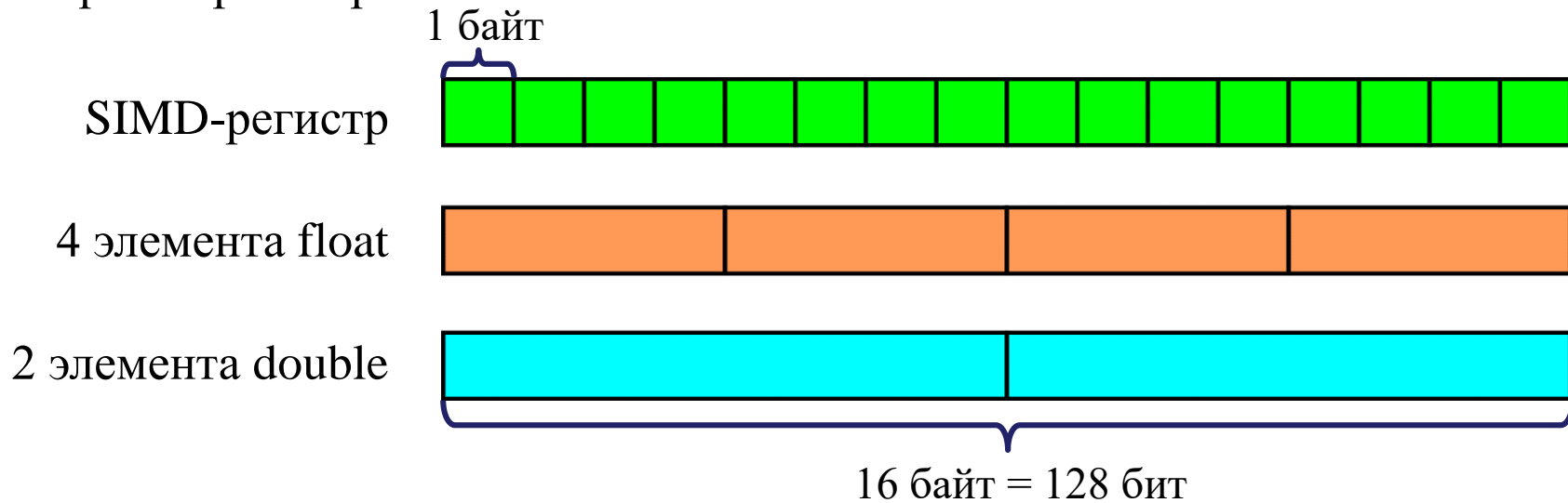
```
for (int i = 0; i < N1; i++) // N1<4  
    d[i] = a[i] * b[i] + c[i];
```

```
for (int i = N1; i < N2; i+=4) {  
    d[i] = a[i] * b[i] + c[i];  
    d[i+1] = a[i+1] * b[i+1] + c[i+1];  
    d[i+2] = a[i+2] * b[i+2] + c[i+2];  
    d[i+3] = a[i+3] * b[i+3] + c[i+3];  
}
```

```
for (int i = N2; i < N; i++) // N-N2<4  
    d[i] = a[i] * b[i] + c[i];
```

Длина векторного регистра

- Эффективность векторизации зависит от типа данных элементов и длины векторного регистра.



- В x86 (x64) системах широко распространены регистры xmm (128 бит), ymm (256 бит), zmm (512 бит).

Опции C++ компилятора Intel

- ❑ Чтобы задействовать векторные вычисления для Intel C++ Compiler Classic, достаточно использовать **-O2** (включает оптимизацию кода).
- ❑ Чтобы задействовать определенные SIMD-инструкции, можно использовать ключи компилятора: **-xSSE2**, **-xAVX**, **-xCORE-AVX2**, **-xCORE-AVX512**, **-xHost** (*рекомендуется*, задействует оптимальный набор инструкций для текущей архитектуры).
- ❑ Для Windows: **/QxAVX**, **/QxHost**, ...
- ❑ Чтобы задействовать регистры длины 512 бит для архитектур, поддерживающих AVX512, нужно использовать ключ **-qopt-zmm-usage=high**

Общие рекомендации

- ❑ Число итераций во время выполнения программы должно быть заранее известно.

Простой цикл с фиксированным числом итераций, может быть векторизован.

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

Известно число итераций, иногда может быть векторизован.

```
while (true) {  
    if (i >= N) break;  
    c[i] = a[i] + b[i];  
    i++;  
}
```

Известно число итераций, может быть векторизован.

```
while (i < N) {  
    c[i] = a[i] + b[i];  
    i++;  
}
```

Число итераций неизвестно. Может быть векторизован?

```
for (int i = 0; i < N; i++) {  
    if (c[i] > value) break;  
    c[i] = a[i] + b[i];  
}
```

Общие рекомендации

- ❑ Тело цикла должно быть достаточно простое. *Чем проще, тем лучше* 😊
- ❑ Для успешной векторизации в цикле не должно быть:
 - Других циклов
 - В случае вложенных циклов обычно векторизуется самый внутренний.
 - Возможное исключение – если компилятору удалось полностью развернуть внутренний цикл.
 - Вызовов функций
 - Встроенные (inline) функции не препятствуют векторизации.
 - Иногда слишком большое тело цикла выносится компилятором в отдельную функцию, что препятствует векторизации.
 - Можно вызывать некоторые математические функции из библиотеки компилятора.

Общие рекомендации. Математические функции

- ❑ Если в векторных циклах не должно быть вызовов функций, то как быть с математическими функциями?
- ❑ Можно использовать математическую функцию, вычисляющую значение для векторного аргумента.
- ❑ LibM – модуль компилятора, реализующий классические скалярные математические функции. В некоторых компиляторах LibM оптимизирован под современные архитектуры.
- ❑ Intel SVML – модуль ICC, реализующий векторные математические функции. Функции из SVML вычисляют результат для одного векторного регистра (xmm, ymm или zmm) с использованием SIMD.
- ❑ При вызове математической функции компилятор понимает, какой модуль использовать. Код выглядит одинаково.

Общие рекомендации. Зависимости по данным

- ❑ Стоит при векторизации избегать зависимостей по данным между итерациями.

- Read-after-write

```
for (int i = 0; i < N-1; i++)  
    a[i+1] = a[i] + 1;
```

Считывание на следующей итерации после записи.
Небезопасно для векторизации, ответ будет некорректный.

- Write-after-read

```
for (int i = 0; i < N-1; i++)  
    a[i] = a[i+1] + 1;
```

Запись на следующей итерации после чтения.
В целом для параллельного выполнения небезопасно, но в случае векторизации ответ будет корректный.

- Write-after-write

```
for (int i = 0; i < N; i++)  
    var = a[i];
```

Запись на следующей итерации после записи.
В общем случае небезопасно.

```
for (int i = 0; i < N; i++)  
    sum = sum + a[i]*b[j];
```

Некоторые типичные циклы компилятор распознает и безопасно векторизует.
Пример: векторное произведение (присутствуют все представленные зависимости).

Общие рекомендации

- ❑ Ветвления кода препятствуют векторизации.
 - Однако в большинстве случаев компилятору удастся заменять условный оператор на операции с масками.
 - Для некоторых типичных случаев, например, для поиска максимума и минимума, есть отдельные инструкции.
- ❑ Стоит избегать преобразований типов.
- ❑ Следует использовать эффективные паттерны доступа к памяти.
 - Наилучший вариант – доступ с единичным шагом.
 - Выравнивание массивов улучшает эффективность.
- ❑ Если компилятор посчитает, что векторизация неэффективна, то он либо будет использовать регистры меньшей длины, либо вообще не векторизует цикл.

Некоторые полезные прагмы (Intel Compiler)

❑ **#pragma vector <keyword>**

- **always** – векторизовать цикл, даже если компилятор полагает, что векторизация будет неэффективна.
- **aligned** – есть выравнивание по данным.
- ...

❑ **#pragma ivdep** – в цикле нет зависимостей по данным.

❑ **#pragma unroll** (с параметрами) – полностью или частично развернуть цикл, иногда это способствует лучшей векторизации.

❑ **#pragma omp simd** (с параметрами) – поддерживается стандартом OpenMP 4.0 и выше. Сообщает компилятору, что нужно векторизовать цикл и что зависимостей по данным нет (!). Эквивалентна *#pragma ivdep* и *#pragma vector always* вместе.

ПРИМЕРЫ: ОБРАБОТКА ИЗОБРАЖЕНИЙ

Структура хранения изображения

- ❑ Рассмотрим некоторые аспекты векторизации на примере задач из области обработки изображений.



Изображение – набор пикселей. Каждый пиксель имеет свой цвет. Цвет задается одним или несколькими значениями интенсивности (целое число от 0 до 255, 1 байт).

В черно-белых изображениях цвет представлен одним значением интенсивности. В цветных – несколькими.



Модель RGB (red, green, blue) – цвет представлен тремя значениями (каналами).



Структура хранения изображения:
последовательный набор
интенсивностей для каждого
пикселя.

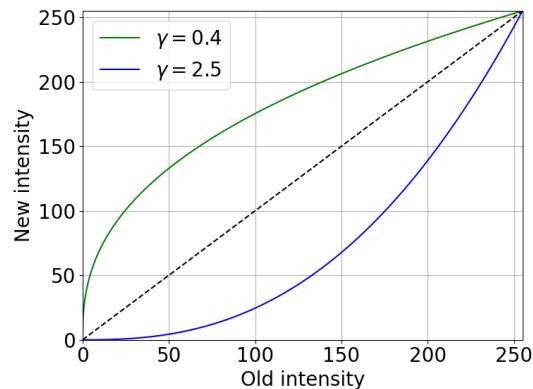
Гамма-коррекция

- ❑ Фильтрация – способ коррекции цвета пикселей изображения.
- ❑ Простейшие фильтры вычисляют новое значение интенсивности пикселя на основе старого значения по некоторой формуле.
- ❑ Гамма-коррекция: $I_{new} = c \cdot I_{old}^\gamma$



Пример гамма-коррекции освещения ($\gamma = 0,45$)

Коррекция яркости изображения:
 $\gamma < 1$ – преобразование темных тонов в светлые; $\gamma > 1$ – наоборот



Гамма-коррекция. Базовая версия

```
using IntensityType = uint8_t; // 1 byte, [0, 255]
const IntensityType MIN_INTENSITY = 0, MAX_INTENSITY = 255;
const float GAMMA = 0.45f;
const float FACTOR = std::pow(MAX_INTENSITY, 1.0f - GAMMA);

float correctPixelIntensity(float intensity) {
    return FACTOR * std::pow(intensity, GAMMA); // функция коррекции
}

void nonlinearCorrection(int height, int width, // размеры изображения
    int nChannels, // число цветовых каналов (RGB -> 3 канала)
    IntensityType* pixels, IntensityType* result // входное и выходное изображения
){
    for (int i = 0; i < height; i++)
        for (int j = 0; j < width; j++)
            for (int color = 0; color < nChannels; color++) {
                int index = nChannels * (i * width + j) + color;
                result[index] = correctPixelIntensity(pixels[index]);
            }
}
```

Тестовая инфраструктура

Процессор	Intel Core i7-11800H (Tiger Lake), 2,3 ГГц, Turbo 4,6 ГГц
Операционная система	Windows 10 version 21H2
Компилятор, профилировщик	<u>Intel oneAPI 2022.2:</u> Intel C++ Compiler Classic Intel Advisor

Гамма-коррекция. Базовая версия

- ☐ Размер изображения 3024×4032 пикселей, RGB (3 канала).
- ☐ Время измеряется в секундах.

Версия	Время, сек
Базовая	0,245

- ☐ Векторизовал ли компилятор цикл?
- ☐ Влияет ли на векторизацию вызов функций **correctPixelIntensity** и **std::pow**?
- ☐ Для ответа на эти вопросы соберем отчет компилятора об оптимизации.
- ☐ Ключ компиляции **/Qopt-report=5**, цифра означает степень подробности, **5** — максимальная.

Гамма-коррекция. Базовая версия

```
INLINE REPORT: (nonlinearCorrection(int, int, int, IntensityType *, IntensityType *))  
C:\gamma_rgb_v0_base_novec\gamma_rgb_v0_base_novec.cpp(26,1)  
-> INLINE: (31,5) correctPixelIntensity(float) (isz = 9) (sz = 16)  
-> INLINE (MANUAL): (18,12) pow(float, float) (isz = 1) (sz = 10)  
-> EXTERN: C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Tools\MSVC\14.29.30133\bin\amd64\x-  
powf(float, float)
```

Функция встроена
автоматически

```
LOOP BEGIN at C:\gamma_rgb_v0_base_novec\gamma_rgb_v0_base_novec.cpp(27,2)  
remark #25101: Loop Interchange not done due to: Original Order seems proper  
remark #25452: Original Order found to be proper, but by a close margin  
remark #15541: outer loop was not auto-vectorized: consider using SIMD dire
```

Внешний вызов
векторной математической
функции

```
LOOP BEGIN at C:\gamma_rgb_v0_base_novec\gamma_rgb_v0_base_novec.cpp(28,3)  
remark #15541: outer loop was not auto-vectorized: consider using SIMD dire
```

```
LOOP BEGIN at C:\gamma_rgb_v0_base_novec\gamma_rgb_v0_base_novec.cpp(29,4)
```

```
remark #15344: loop was not vectorized: vector dependence prevents vectorization  
remark #15346: vector dependence: assumed ANTI dependence between pixels[index] (223:5) and result[index] (31:5)  
remark #15346: vector dependence: assumed FLOW dependence between result[index] (31:5) and pixels[index] (223:5)
```

```
LOOP END
```

```
LOOP END
```

```
LOOP END
```

**ВЕКТОРНЫЕ ВЫЧИСЛЕНИЯ НЕ ЗАДЕЙСТВОВАНЫ,
КОМПИЛЯТОР ПОДОЗРЕВАЕТ ЗАВИСИМОСТИ ПО ДАННЫМ
МЕЖДУ ВХОДНЫМ И ВЫХОДНЫМ МАССИВАМИ**

Гамма-коррекция. Базовая версия

Function Call Sites and Loops	<input checked="" type="checkbox"/> Performance Issues	CPU Time		Type	Why No Vectorization?
		Total Time	Self Time		
loop in nonlinearCorrection at gamma_rgb_v0	2 Assumed dependency present	1,198s	0,584s	Scalar	vector dependence prevents vectorization
f_libm_powf_f9	1 Data type conversions present	0,489s	0,489s	Function	

Мат. функция из LibM

Source Top Down Code Analytics Assembly Recommendations

All Advisor-detectable issues: [C++](#) | [Fortran](#)

Assumed dependency present

The compiler assumed there is an anti-dependency (Write after read - WAR) or a true dependency (Read after write - RAW) in the loop. Improve performance by investigating the assumption and handling accordingly.

Confirm dependency is real

Run the [Dependencies analysis](#) to identify real data dependencies.

There is no confirmation that a real (proven) dependency is present in the loop.

Можно сделать
анализ зависимостей, если
непонятно, есть ли они

МЫ ТОЧНО
ЗНАЕМ, ЧТО
ЗАВИСИМОСТЕЙ
НЕТ

Advisor
показывает,
по какой причине
не векторизован
цикл

Гамма-коррекция. Зависимости по данным

- ❑ **#pragma ivdep** указывает компилятору, что нужно игнорировать предполагаемые зависимости по данным
- ❑ В данном примере, для этой цели можно использовать ключевое слово **restrict**. С помощью него можно сообщить компилятору, что на некоторую область памяти ссылается всего один указатель.
- ❑ Современные компиляторы иногда в подобных случаях создают две версии цикла: скалярную и векторную. В отчете об оптимизации это указывается. Решение о том, какую выбрать, принимается во время выполнения программы.
- ❑ Использовать **#pragma ivdep** и **restrict** стоит только в том случае, если зависимостей по данным действительно нет.

Гамма-коррекция. Зависимости по данным

- ❑ Укажем с помощью `#pragma ivdep`, что в цикле нет зависимостей по данным.

```
for (int i = 0; i < height; i++)  
    for (int j = 0; j < width; j++)  
        #pragma ivdep // перед внутренним циклом  
        for (int color = 0; color < nChannels; color++) { /* код */ }  
}
```

Версия	Время, сек
Базовая	0,245
Зависимости по данным	0,334

- ❑ Время увеличилось. Почему?

Гамма-коррекция. Зависимости по данным

```
LOOP BEGIN at C:\gamma_rgb_vl_ivdep\gamma_rgb_vl_ivdep.cpp(27,2)
remark #25101: Loop Interchange not done due to: Original Order seems proper
remark #25452: Original Order found to be proper, but by a close margin
remark #15542: loop was not vectorized: inner loop was already vectorized

LOOP BEGIN at C:\gamma_rgb_vl_ivdep\gamma_rgb_vl_ivdep.cpp(28,3)
remark #15542: loop was not vectorized: inner loop was already vectorized

LOOP BEGIN at C:\gamma_rgb_vl_ivdep\gamma_rgb_vl_ivdep.cpp(30,4)
remark #15389: vectorization support: reference pixels[index] has unaligned access.
remark #15389: vectorization support: reference result[index] has unaligned access
remark #15381: vectorization support: unaligned access used inside loop body
remark #15305: vectorization support: vector length 4
remark #15309: vectorization support: normalized vectorization overhead 0.124
remark #15300: LOOP WAS VECTORIZED
remark #15450: unmasked unaligned unit stride loads: 1
remark #15451: unmasked unaligned unit stride stores: 1
remark #15475: --- begin vector cost summary ---
remark #15476: scalar cost: 130
remark #15477: vector cost: 28.250
remark #15478: estimated potential speedup: 4.590
remark #15482: vectorized math library calls: 1
remark #15487: type converts: 2
remark #15488: --- end vector cost summary ---
LOOP END

LOOP BEGIN at C:\gamma_rgb_vl_ivdep\gamma_rgb_vl_ivdep.cpp(30,4)
<Remainder loop for vectorization>
remark #25456: Number of Array Refs Scalar Replaced In Loop: 1
LOOP END

LOOP END
LOOP END
```

**ДЛИНА ВЕКТОРНОГО
РЕГИСТРА ВСЕГО 4,
АРХИТЕКТУРА ПОЗВОЛЯЕТ
БОЛЬШЕ**

*Внутренний цикл
векторизован*

*«Остаток»
обрабатывается
отдельно*

Гамма-коррекция. Использование инструкций AVX512

- ❑ Код векторизован, но для инструкций SSE (длина регистра 128 бит).
Данным процессором поддерживаются инструкции AVX512 (512 бит).
- ❑ AVX512 – это не просто более длинный регистр (что в данном случае не поможет), но и более современные машинные инструкции. Компилятор может сгенерировать другой, более оптимальный код.
- ❑ Скомпилируем программу с ключом **/QxHost**, чтобы задействовать самые «современные» инструкции, поддерживаемые процессором.

Версия	Время, сек
Базовая	0,245
Зависимости по данным	0,334
Ключ -xHost	0,164

Гамма-коррекция. Использование инструкций AVX512

ROOFLINE

Function Call Sites and Loops	Performance Issues	CPU Time		Type
		Total Time	Self Time	
[loop in nonlinearCorrection]	2 Ineffective peeled/remainder loop(s) present	0,315s	0,158s	Peeled/Remainder
_svml_powf4_I9	1 Data type conversions present	0,157s	0,157s	Vector Function

Source Top Down Code Analytics Assembly Recommendations Why No Vectorization?

All Advisor-detectable issues: [C++](#) | [Fortran](#)

! Ineffective peeled/remainder loop(s) present

All or some source loop iterations are not executing in the loop body. Improve performance by moving source loop iterations from peeled/remainder loops to the loop body.

Force vectorized remainder

The compiler did not vectorize the remainder loop, even though doing so could improve performance. To fix: Force vectorization using a directive: `#pragma vector vecremainder`.

Example

```
...  
// Force the compiler to vectorize the remainder loop  
#pragma vector vecremainder
```

Вызов
векторной
мат. функции из
библиотеки
SVML

**ОСНОВНОЕ ВРЕМЯ РАБОТАЕТ
СКАЛЯРНЫЙ ЦИКЛ,
ОБРАБАТЫВАЮЩИЙ ОСТАТОК.
ПОЧЕМУ?**

Гамма-коррекция. Разворачивание цикла

- ❑ Во внутреннем цикле всего 3 итерации. Развернем цикл.
- ❑ Теперь внутренний цикл – цикл по переменной j .

```
const int N_CHANNELS = 3; // RGB

void nonlinearCorrection(int height, int width,
    IntensityType* pixels, IntensityType* result)
{
    for (int i = 0; i < height; i++)
        #pragma ivdep
        for (int j = 0; j < width; j++) {
            int index = N_CHANNELS * (i * width + j);
            result[index] = correctPixelIntensity(pixels[index]);
            result[index + 1] = correctPixelIntensity(pixels[index + 1]);
            result[index + 2] = correctPixelIntensity(pixels[index + 2]);
        }
}
```

Гамма-коррекция. Разворачивание цикла

- ❑ Иногда разворачивание цикла на 4, 8, 16, ... итераций способствует эффективности векторизации. Например, это дает возможность компилятору переставлять итерации местами.
- ❑ Можно использовать **#pragma unroll** или **#pragma unroll(·)**. Аргумент в скобках указывает, на сколько итераций нужно развернуть цикл.
- ❑ Если бы число итераций было известно на этапе компиляции, то весьма вероятно, что разворачивание произошло бы автоматически.

Версия	Время, сек
Базовая	0,245
Зависимости по данным	0,334
Ключ -xHost	0,164
Разворачивание цикла	0,066

Ускорение
3,7 раз
относительно
базовой
версии

Гамма-коррекция. Разворачивание цикла

ROOFLINE

Function Call Sites and Loops		Performance Issues	CPU Time		Ty.	Wh. No.	Vectorized Loops			Instruction Set Analysis			Data Types
			Total Ti...	Self Time			Vector I...	Efficiency	Gain...	VL ..	Traits		
[🔍] [loop in nonlinearCorrection]		2 Possible in...	0,078s	0,047s	Vect...		AVX512	100%	7,99x	8	Mask Manipulations; Shuffles; Type Conversions	Float32; Int32; Int8; UInt32; UByte	
[f] _svml_powf8_I9		1 Data type c...	0,032s	0,032s	Vect...		AVX2			8	Appr. Reciprocals(AVX); FMA; Permuters; Shifts; T...	Float32; Int32; UInt32	

Source Top Down Code Analytics Assembly Recommendations Why No Vectorization?

All Advisor-detectable issues: [C++](#) | [Fortran](#)

! Possible inefficient memory access patterns present
Inefficient memory access patterns may result in significant vector code execution slowdown or block automatic vectorization by the compiler. Improve performance by investigating.
☐ Confirm inefficient memory access patterns
There is no confirmation inefficient memory access patterns are present. To fix: Run a [Memory Access Patterns analysis](#).

! Data type conversions present
There are multiple data types within loops. Utilize hardware vectorization support more effectively by avoiding data type conversion.
☐ Use the smallest data type
The source loop contains data types of different widths. To fix: Use the smallest data type that gives the needed precision to use the entire vector register width.

Possible inefficient memory access patterns present
Confirm inefficient memory access

Хорошая эффективность векторизации

2 ПРОБЛЕМЫ ПРОИЗВОДИТЕЛЬНОСТИ

Преобразования типов

Компилятор подозревает, что данные лежат в памяти не подряд

Гамма-коррекция. Последовательный доступ к памяти

- ❑ Исправим код так, чтобы компилятору было понятно, что данные лежат в памяти последовательно.

```
void nonlinearCorrection(int height, int width,
    IntensityType* pixels, IntensityType* result)
{
    #pragma ivdep
    for (int i = 0; i < height * width * N_CHANNELS; i++) {
        result[i] = correctPixelIntensity(pixels[i]);
    }
}
```

- ❑ Это поможет исправить одну из проблем, найденных профилировщиком.

Гамма-коррекция. Последовательный доступ к памяти

Версия	Время, сек
Базовая	0,245
Зависимости по данным	0,334
Ключ -xHost	0,164
Разворачивание цикла	0,066
Последовательный доступ	0,046

- ☐ Компилятор сгенерировал более простые инструкции.

Гамма-коррекция. Преобразования типов

- ❑ Массив пикселей имеет тип `uint8_t` (1 байт). Математическая функция вычисляется с типом `float`. Возникают преобразования типов.
- ❑ Будем хранить пиксели изначально в массиве типа `float`.

```
using IntensityType = float;
```

Версия	Время, сек
Базовая	0,245
Зависимости по данным	0,334
Ключ -xHost	0,164
Разворачивание цикла	0,066
Последовательный доступ	0,046
Преобразования типов	0,044

Здесь
компилятор сам
хорошо справился
с преобразованиями
типов. Это
происходит
не всегда

Гамма-коррекция. Преобразования типов

[illegible]

Для AVX512 максимальная
длина векторного регистра
16 элементов Float32, но
используются регистры длины 8

Возможно,
стоит попробовать
выравнивание
данных (адрес
элемента кратен
16, 32 или 64)

Гамма-коррекция. Выравнивание массивов

- ❑ Выделим память так, чтобы начало массива находилось по адресу, кратному 64 (это связано с размером кэш-линии).
- ❑ Для подобного выделения памяти есть специальные функции, например, **memalign** (Си) или **std::aligned_alloc** (C++17).
- ❑ Перед циклом укажем, что данные выровнены (**#pragma vector aligned**).

Версия	Время, сек
Преобразования типов	0,044
Выравнивание	0,044

- ❑ Предупреждение в отчете об оптимизации пропало, но время не изменилось.
- ❑ В современных оптимизирующих компиляторах оператор **new** уже учитывает выравнивание.
- ❑ На некоторых инфраструктурах явное выравнивание данных ускоряет код.

Гамма-коррекция. Использование 512-битных (zmm) регистров

- ❑ Скомпилируем программу с ключом **/Qopt-zmm-usage=high**, тем самым задействуем регистры длиной 512 бит (16 элементов float).

Версия	Время, сек
Базовая	0,245
Зависимости по данным	0,334
Ключ -xHost	0,164
Разворачивание цикла	0,066
Последовательный доступ	0,046
Преобразования типов	0,044
zmm регистры	0,032

Ускорение
7,65 раз относительно
базовой версии. Время
работы определяется
реализацией `std::pow`
и пропускной
способностью
памяти

Гамма-коррекция. Использование 512-битных (ZMM) регистров

ROOF LINE

Function Call Sites and Loops	Performance Issues	CPU Time		Type	Wh. No.	Vectorized Loops				Instruction Set Analysis	
		Total Time	Self Time			Vector ISA	Efficiency	Gain...	VL (...)	Traits	Data Types
_svml_powf16_z0	<input type="checkbox"/>	0,078s	0,047s	Vector Function		AVX512			16	2-Source Permutes; A...	Float32; UInt32
[loop in nonlinearCorrection]	<input type="checkbox"/>	0,157s	0,080s	Vectorized (Body; Peeled)		AVX512	~100%	19,75x	16	Mask Manipulations	Float32; Int32; UInt32

Advisor больше не «видит» проблем в основном цикле

Эффективность векторизации, рассчитанная профилировщиком. Отношение оценки ускорения к длине векторного регистра

Оценка ускорения от векторизации больше 16х. Реальное может быть меньше. Например, может снижаться частота процессора при использовании SIMD инструкций

Длина векторного регистра 16 элементов Float32

Гамма-коррекция. Доступ к памяти с шагом

- ❑ Часто RGB изображения хранят так, что каждый пиксель занимает 4 байта. Последний байт не хранит информацию (адрес пикселя кратен степени 2).

```
const int N_CHANNELS = 4; // RGB + reserved

void nonlinearCorrection(int height, int width,
    IntensityType * pixels, IntensityType * result)
{
    #pragma ivdep
    for (int i = 0; i < height * width * N_CHANNELS; i+=N_CHANNELS) {
        result[i] = correctPixelIntensity(pixels[i]);
        result[i + 1] = correctPixelIntensity(pixels[i + 1]);
        result[i + 2] = correctPixelIntensity(pixels[i + 2]);
        // result[i + 3] не обрабатывается
    }
}
```

Доступ к памяти с шагом

Версия	Время, сек
Без векторизации	0,209
С векторизацией	0,038

Ускорение
5,5 раз

Function Call Sites and Loops	Performance Issues	CPU Time		Type	Wh. No.	Vectorized Loops				Instruction Set Analysis	
		Total Time	Self Time			Vector...	Efficiency	Gain...	VL ...	Traits	Data Types
<input checked="" type="checkbox"/> f_svm1_powf16_z0	<input type="checkbox"/>	0,139s	0,124s	Vector Function		AVX512			16	2-Source Permut ...	Float32; UInt32
<input checked="" type="checkbox"/> [loop in nonlinearCorrection at ga	<input type="checkbox"/> 1 Possible inefficient memory access ...	0,218s	0,078s	Vectorized (Body)		AVX512	~85%	13,59x	16	Gathers; Scatters	Float32; UByte

Непоследовательный
доступ к памяти

Эффективность
<100%

Специальные
инструкции

Расчет средней интенсивности (редукция)

```
const int N_CHANNELS = 3; // RGB

std::tuple<float, float, float> average(
    int height, int width, float* pixels)
{
    float avgR = 0.0f, avgG = 0.0f, avgB = 0.0f;

    #pragma ivdep
    for (int i = 0; i < height * width; i++) {
        avgR += pixels[N_CHANNELS * i];
        avgG += pixels[N_CHANNELS * i + 1];
        avgB += pixels[N_CHANNELS * i + 2];
    }

    avgR /= height * width;
    avgG /= height * width;
    avgB /= height * width;

    return std::make_tuple(avgR, avgG, avgB);
}
```

- ❑ Считаем среднее по каждому из цветовых каналов. На основе данной информации можно выполнять коррекцию цвета.

Версия	Время, сек
Без векторизации	0,011
С векторизацией	0,009

- ❑ Цикл векторизован.
- ❑ Результат корректный, несмотря на зависимости по данным.
- ❑ Память – «узкое горлышко» (bottleneck) приложения.

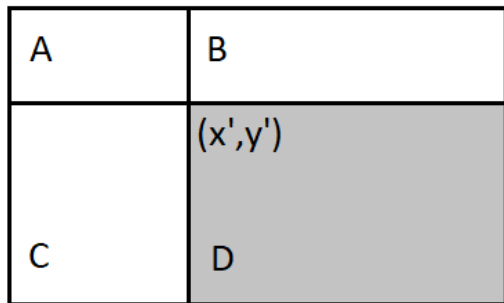
Интегральное изображение

- ❑ Еще один пример – вычисление интегрального изображения.
- ❑ Пусть $I(x, y)$ – исходное изображение, $L(x, y)$ - его интегральное представление.

$$L(x, y) = I(x, y) + L(x, y - 1) + L(x - 1, y) - L(x - 1, y - 1)$$

- ❑ Интегральное изображение используется, например, в задачах распознавания лиц на изображении.
- ❑ С его помощью быстро вычисляется сумма пикселей в прямоугольнике.

(0,0)



(x,y)

$$\begin{aligned} S(D) = \\ S(ABCD) - S(AB) - S(AC) + S(A) = \\ L(x, y) - L(x', y) - L(x, y') + L(x', y') \end{aligned}$$

Интегральное изображение

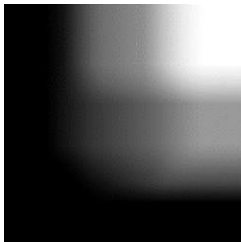
```
void integral(int height, int width, float* pixels)
{
    for (int i = 0; i < height-1; i++) {
        #pragma omp simd // внутренний (после разворачивания следующего) цикл
        for (int j = 0; j < width-1; j++)
            #pragma unroll // разворачивание цикла из 3-х итераций
            for (int k = 0; k < N_CHANNELS; k++) {
                int index_i0_j0 = (i * width + j) * N_CHANNELS + k;
                int index_i0_j1 = (i * width + (j+1)) * N_CHANNELS + k;
                int index_i1_j0 = ((i+1) * width + j) * N_CHANNELS + k;
                int index_i1_j1 = ((i+1) * width + (j+1)) * N_CHANNELS + k;
                // запись результата в тот же массив
                pixels[index_i1_j1] = pixels[index_i1_j1] - pixels[index_i0_j0] +
                    pixels[index_i1_j0] + pixels[index_i0_j1];
            }
    }
}
```

Интегральное изображение

Исходное
изображение



Корректное
интегральное
изображение,
полученное
с отключенной
векторизацией



Интегральное
изображение,
полученное
со включенной
векторизацией



- ❑ В одномерном случае задача вырождается в подсчет накапливаемых сумм массива.

```
#pragma omp simd  
for (int i = 0; i < N - 1; i++)  
    arr[i + 1] += arr[i];
```

**ОТВЕТ
НЕПРАВИЛЬНЫЙ!
ПОЧЕМУ?**

- ❑ Всегда ли такой цикл будет векторизован?

Заключение

- ❑ Векторные вычисления могут ускорить выполнение программы в несколько раз.
- ❑ Ускорение зависит от длины векторного регистра, типа элементов, их расположения в памяти, типа выполняемых инструкций. Иногда задействовать векторные вычисления невозможно или бессмысленно.
- ❑ В случае зависимостей по данным между итерациями цикла векторизация может быть неэффективной и даже приводить к некорректному результату.
- ❑ Современные оптимизирующие компиляторы хорошо справляются с автоматической векторизацией циклов.
- ❑ В некоторых компиляторах предоставляются дополнительные вспомогательные средства.

Литература

1. Amiri, Hossein, and Asadollah Shahbahrami. "SIMD programming using Intel vector extensions." *Journal of Parallel and Distributed Computing* 135 (2020): 83-100.
2. Sabahi, M. "A Guide to Vectorization with Intel C++ Compilers." (2010).
3. Intel, R. "Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4." (2018).

Авторский коллектив

- ❑ Мееров Иосиф Борисович, к.т.н., доцент, зам. зав. каф. МОСТ
- ❑ Сысоев Александр Владимирович, к.т.н., доцент каф. МОСТ
- ❑ Линев Алексей Владимирович, зав. лаб. интернета вещей, каф. ПРИН
- ❑ Волокитин Валентин Дмитриевич, программист лаборатории СТиВВ, каф. МОСТ
- ❑ Козинов Евгений Александрович, к.т.н., преподаватель каф. МОСТ
- ❑ Панова Елена Анатольевна, инженер лаборатории СТиВВ, каф. МОСТ

Контакты

Нижегородский государственный университет

<http://www.unn.ru>

Центр компетенций oneAPI в ННГУ

<http://hpc-education.unn.ru/ru/центр-компетенций-oneapi-в-ннгу>

Институт информационных технологий, математики и механики

<http://www.itmm.unn.ru>