

Автовекторизация в LLVM

Артём Скребков

- Ведущий инженер в области инфраструктуры искусственного интеллекта (AI Frameworks Engineer)
- Закончил ННГУ им. Лобачевского, ИТММ, Прикладная математика и информатика
- Профессиональные интересы
 - Глубокое обучение, OpenVINO, оптимизация под VPU (vision processor unit), графовые компиляторы, C++, Python, Groovy, *Rust*
- Контакты
 - e-mail: skrebkov.art@gmail.com
 - discord сервер курса: <https://discord.gg/4GWv3vPFAQ>

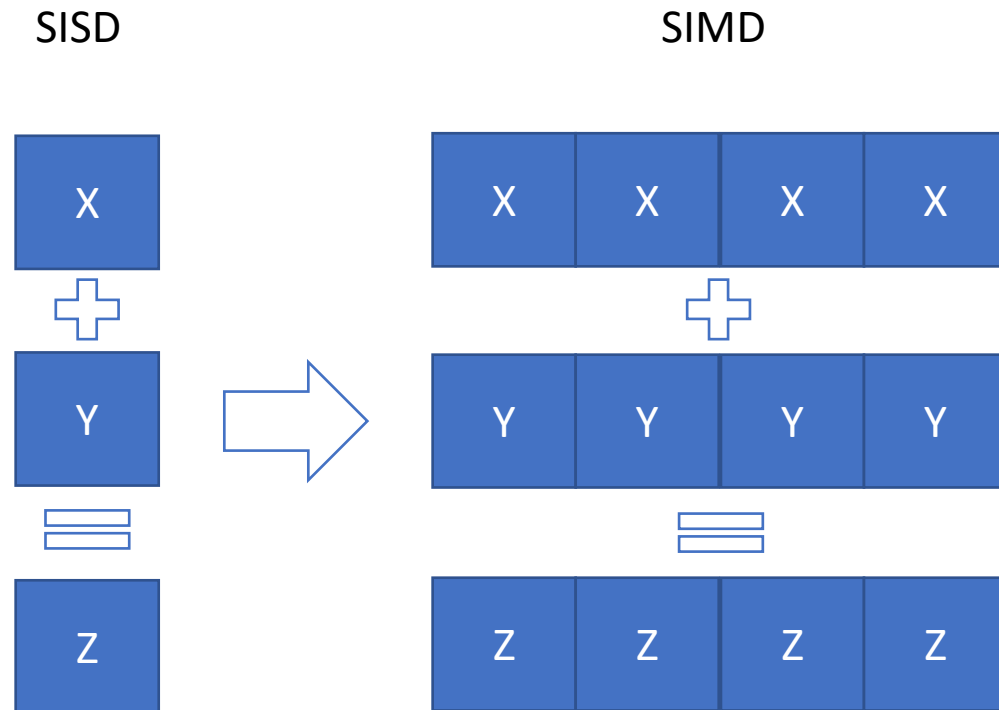
План

- Введение. SIMD
- Автовекторизация в LLVM
 - Как?
 - Настройки векторизации
 - Диагностика
- Как писать автовекторизуемый код
- Постановка задачи

Введение. SIMD

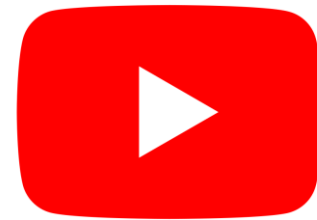
SIMD – Single Instruction Multiple Data

- Одна и та же операция работает параллельно над множеством однородных данных
- Количество данных в векторе не влияет на время исполнения
- Векторные расширения
 - MMX – 64 бита
 - SSE, SSE2, SSE3, SSE4 – 128 бит
 - AVX, AVX2 – 256 бит
 - AVX512 – 512 бит



Применение

- SIMD позволяет добиться прироста производительности в задачах, где требуется обработка большого количества однородных данных
 - 3D графика
 - Физическое/математическое моделирование
 - Обработка изображений/видео
 - Кодирование/декодирование видео



Как использовать SIMD

- Ассемблерные вставки

```
vmovdqu ymm0, ymmword ptr [4*rdx + y]
```

```
vpaddq ymm0, ymm0, ymmword ptr [4*rdx + x]
```

```
vmovdqu ymmword ptr [4*rdx + y], ymm0
```

- Сложен в разработке и поддержке
- Нет переносимости
 - Зависимость на платформу и компилятор



Как использовать SIMD

- Интринсики (intrinsics)
 - Функции, которые компилятор заменяет на соответствующие инструкции

```
int64_t cnt = 0;
auto sseVal = _mm_set1_epi16(VAL);
for (int i = 0; i < ARR_SIZE; i += 8) {
    auto sseArr = _mm_set_epi16(arr[i + 7], arr[i + 6], ..., arr[i + 1], arr[i]);
    cnt += _popcnt32(_mm_movemask_epi8(_mm_cmpeq_epi16(sseVal, sseArr)));
}
```

- Проще чем уровень ассемблера
 - Но всё ещё сложен для разработки и поддержки
- Всё ещё нет переносимости
 - Зависимость на процессор



Как использовать SIMD

- Механизм автовекторизации в компиляторе
 - clang -O3 ...
- Прост в поддержке и разработке
 - Сравнительно с предыдущими способами
- Портируем
- Разработчик должен писать автовекторизуемый код



Раскрутка цикла (loop unrolling)

- Техника оптимизации, состоящая в искусственном увеличении количества инструкций, исполняемых в течение одной итерации цикла
- Потенциально
 - Параллелизм инструкции
 - Большая нагрузка кэша и регистров

```
#pragma clang loop unroll_count(2)
for (int i = 0; i < 64; i++) {
    data[i] = input[i] * other[i];
}
```

Превращается

```
for (int i = 0; i < 64; i += 2) {
    data[i] = input[i] * other[i];
    data[i+1] = input[i+1] * other[i+1];
}
```

Loop interleaving

- Особый случай раскрутки цикла применяемый при векторизации
- Цель та же самая как в случае раскрутки цикла

```
#pragma clang loop interleave_count(2) vectorize_width(2)
for (int i = 0; i < 64; i++) {
    data[i] = input[i] * other[i];
}
```

Превращается

```
for (int i = 0; i < 64; i +=4) {
    data[i..(i+1)] = input[i..(i+1)] * other[i..(i+1)];
    data[(i+2)..(i+3)] = input[(i+2)..(i+3)] * other[(i+2)..(i+3)];
}
```

Автовекторизация в LLVM

Как включить?

- Авто-векторизация включена по-умолчанию (-O2, -Os, -O3)
- Может быть выключена с помощью -fno-vectorize

Настройки

- Параметры векторизации
 - Ширина вектора (-force-vector-width)
 - `$ clang -force-vector-width=8 ...`
 - Значение раскрутки цикла (-force-vector-interleave)
 - `$ clang -force-vector-interleave=2`
- Если значение не указаны явно компилятор подбирает их автоматически

Отладка автовекторизации

- Во время шага векторизации компилятор генерирует векторизационный отчёт
- Информация из отчёта может быть получена с помощью опций
 - -Rpass=loop-vectorize сообщит о векторизованных циклах
 - -Rpass-missed=loop-vectorize сообщит о не векторизованных циклах
 - -Rpass-analysis=loop-vectorize покажет причины почему цикл не был векторизован

Отладка автовекторизации

```
int m = len;
for (int i = 0; i < len; i++)
{
    if (x[i] == 42) {
        m = i;
        break;
    }
}
for (int i = 0; i < m; i++) {
    y[i] += x[i];
}
```

example.cpp:8:5: **remark:** loop not vectorized: loop control flow is not understood by vectorizer [-Rpass-analysis=loop-vectorize]

for (int i = 0; i < len; i++) {

^

example.cpp:8:5: **remark:** loop not vectorized: loop control flow is not understood by vectorizer [-Rpass-analysis=loop-vectorize]

example.cpp:8:5: **remark:** loop not vectorized: loop control flow is not understood by analyzer [-Rpass-analysis=loop-vectorize]

example.cpp:8:5: **remark:** loop not vectorized [-Rpass-missed=loop-vectorize]

example.cpp:14:5: **remark:** vectorized loop (vectorization width: 8, interleaved count: 4) [-Rpass=loop-vectorize]

for (int i = 0; i < m; i++) {

^

Как помочь компилятору?

Общие рекомендации

- Цикл вероятнее может быть векторизован если выполнены следующие условия
 - Количество итераций должно быть известно до исполнения цикла
 - Отсутствие операторов `break` и `continue`
 - Векторизуется только самый вложенный цикл (**innermost loop**)
 - Нет вызовов функций
 - Встраиваемые функции (**inline functions**) без побочных эффектов разрешены
 - Базовые математические функции (`pow()`, `sqrt()`, `exp()`, ...) разрешены
 - Не ветвящийся поток управления (**single control flow**)
 - Отсутствие оператора `switch`

Невекторизируемые циклы

- Оператор break внутри цикла

```
for (int i = 0; i < len; i++)  
{  
    if (x[i] == 42) break;  
    y[i] += x[i];  
}
```

- [Goldbolt](#)

• example.cpp:7:5: **remark:** loop not vectorized: loop control flow is not understood by vectorizer
[-Rpass-analysis=loop-vectorize]

```
for (int i = 0; i < len; i++) {
```

^

example.cpp:7:5: **remark:** loop not vectorized: loop control flow is not understood by vectorizer
[-Rpass-analysis=loop-vectorize]

example.cpp:7:5: **remark:** loop not vectorized: loop control flow is not understood by analyzer [-Rpass-analysis=loop-vectorize]

example.cpp:7:5: **remark:** loop not vectorized [-Rpass-missed=loop-vectorize]

Невекторизируемые циклы

- Второй цикл теперь векторизуемый

```
int m = len;
for (int i = 0; i < len; i++) {
    if (x[i] == 42) {
        m = i;
        break;
    }
}
for (int i = 0; i < m; i++) {
    y[i] += x[i];
}
```

- [Goldbolt](#)

example.cpp:8:5: **remark:** loop not vectorized: loop control flow is not understood by vectorizer [-Rpass-analysis=loop-vectorize]

```
for (int i = 0; i < len; i++) {
```

^

example.cpp:8:5: **remark:** loop not vectorized: loop control flow is not understood by vectorizer [-Rpass-analysis=loop-vectorize]

example.cpp:8:5: **remark:** loop not vectorized: loop control flow is not understood by analyzer [-Rpass-analysis=loop-vectorize]

example.cpp:8:5: **remark:** loop not vectorized [-Rpass-missed=loop-vectorize]

example.cpp:14:5: **remark:** vectorized loop (vectorization width: 8, interleaved count: 4) [-Rpass=loop-vectorize]

```
for (int i = 0; i < m; i++) {
```

^

Невекторизируемые циклы

- Проверка внешнего состояния “errno”

```
for (int i = 0; i < len; i++)  
{  
    x[i] = exp(y[i]);  
    if (errno == ERANGE) {  
        x[i] = 0;  
    }  
}
```

- [Goldbolt](#)

example.cpp:10:16: **remark:** loop not vectorized: call instruction cannot be vectorized [-Rpass-analysis=loop-vectorize]

x[i] = exp(y[i]);

^

example.cpp:9:5: **remark:** loop not vectorized: instruction cannot be vectorized [-Rpass-analysis=loop-vectorize]

for (int i = 0; i < len; i++) {

^

example.cpp:9:5: **remark:** loop not vectorized [-Rpass-missed=loop-vectorize]

Невекторизируемые циклы

- Второй цикл теперь векторизуемый (-fno-math-errno)

```
for (int i = 0; i < len; i++)  
{  
    x[i] = exp(y[i]);  
}
```

- [Compiler Explorer](#)

example.cpp:9:5: **remark:** the cost-model indicates that interleaving is not beneficial [-Rpass-analysis=loop-vectorize]

```
for (int i = 0; i < len; i++) {
```

^

example.cpp:9:5: **remark:** vectorized loop (vectorization width: 4, interleaved count: 1) [-Rpass=loop-vectorize]

Невекторизируемые циклы

- Оператор switch внутри цикла

```
for (int i = 0; i < len; i++)  
{  
    switch(x[i]) {  
        case 0: x[i] = i*2; break;  
        case 1: x[i] = i;   break;  
        default: x[i] = 0;  
    }  
}
```

- [Goldbolt](#)

example.cpp:7:9: **remark:** loop not vectorized: loop contains a switch statement [-Rpass-analysis=loop-vectorize]

```
switch (x[i]) {
```

^

example.cpp:6:5: **remark:** loop not vectorized [-Rpass-missed=loop-vectorize]

```
for (int i = 0; i < len; i++) {
```

^

Невекторизируемые циклы

- Количество итераций неизвестно до исполнения

```
while (*p != 0) {  
    *q++ = *p++;  
}
```

- [Goldbolt](#)

example.cpp:9:5: **remark:** loop not vectorized: value that could not be identified as reduction is used outside the loop [-Rpass-analysis=loop-vectorize]

```
while (*p != 0) {
```

^

example.cpp:9:5: **remark:** loop not vectorized: could not determine number of loop iterations [-Rpass-analysis=loop-vectorize]

example.cpp:9:5: **remark:** loop not vectorized [-Rpass-missed=loop-vectorize]

Постановка задачи

Постановка задачи

- Требуется провести анализ заданного цикла в форме отчёта
- Подробная инструкция
 - <https://github.com/ArtemSkrebkov/autovectorization-practise/blob/main/README.md>
- Кратко что нужно сделать:
 1. Определить конфигурацию тестовой платформы
 2. Исследовать цикл
 1. Получить LLVM IR в следующих конфигурациях:
 1. Без векторизации и раскрутки
 2. Без векторизации, с раскруткой
 3. С векторизацией, без раскрутки
 4. С векторизацией и раскруткой
 2. Провести измерения времени исполнения цикла в тех же самых конфигурациях
 3. Проанализировать полученные результаты

Пререквезиты

- Инструменты
 - clang-11 (or later), git
- Репозиторий
 - <https://github.com/ArtemSkrebkov/autovectorization-practise.git>

Цель практики

- Проанализировать LLVM IR в различных конфигурациях
- Убедиться что компилятор может автовекторизовать код

Определение конфигурации

- Поиск в интернете
 - Информация об процессорах Intel может быть найдена [здесь](#)
 - Информация об процессорах AMD может быть найдена, например, [здесь](#)
- Windows
 - [CPU-Z](#)
- Linux
 - \$ lscpu
- Пример конфигурации
 - Intel® Core™ i5-8350U Processor
 - Instruction Set Extensions Intel® SSE4.1, Intel® SSE4.2, Intel® AVX2

Исследуемый цикл

- Пример

```
void run(){  
    for (int i=0; i < N; i++) {  
        r[i] = a[i] + b[i];  
    }  
}
```

Как получить LLVM IR для заданной функции

- Генерируем llvm-байт код
 - `clang -I./loops/vectorizable/ -DINCLUDE_TEST=\"testN.hpp\" test.cpp -emit-llvm -c -o testN.bc`
- Извлекаем llvm-байт код интересующей нас функции
 - `llvm-extract -func=?run@testFunc@@QEAAXXZ testN.bc -o run-fn.bc`
- Дизассемблирует llvm-байт код в текстовое представление
 - `llvm-dis run-fn.bc -o run-fn.ll`
- `run-fn.ll` содержит текстовое представление LLVM IR

Скалярный vs “раскрученный” код

Параметры: -O3 -fno-vectorize -fno-unroll-loops

for.body:

%0 = load float, float* %arrayidx, align 4, !tbaa !3

%1 = load float, float* %arrayidx3, align 4, !tbaa !3

%add = fadd float %0, %1

store float %add, float* %arrayidx5, align 4, !tbaa !3

- *В целях улучшения читаемости вспомогательные инструкции были удалены из листингов*

Параметры: -O3 -fno-vectorize

for.body:

%0 = load float, float* %arrayidx, align 4, !tbaa !3

%1 = load float, float* %arrayidx3, align 4, !tbaa !3

%add = fadd float %0, %1

store float %add, float* %arrayidx5, align 4, !tbaa !3

%2 = load float, float* %arrayidx.1, align 4, !tbaa !3

%3 = load float, float* %arrayidx3.1, align 4, !tbaa !3

%add.1 = fadd float %2, %3

store float %add.1, float* %arrayidx5.1, align 4, !tbaa !3

%4 = load float, float* %arrayidx.2, align 4, !tbaa !3

%5 = load float, float* %arrayidx3.2, align 4, !tbaa !3

%add.2 = fadd float %4, %5

store float %add.2, float* %arrayidx5.2, align 4, !tbaa !3

%6 = load float, float* %arrayidx.3, align 4, !tbaa !3

%7 = load float, float* %arrayidx3.3, align 4, !tbaa !3

%add.3 = fadd float %6, %7

store float %add.3, float* %arrayidx5.3, align 4, !tbaa !3

**В 4 раза больше
инструкции за 1
итерацию**

Скалярный vs векторизованный код

В 4 раза больше
данных за 1
итерацию

Параметры: -O3 -fno-vectorize -fno-unroll-loops

for.body:

%0 = load float, float* %arrayidx, align 4, !tbaa !3

%1 = load float, float* %arrayidx3, align 4, !tbaa !3

%add = fadd float %0, %1

store float %add, float* %arrayidx5, align 4, !tbaa !3

- *В целях улучшения читаемости вспомогательные инструкции были удалены из листингов*

Параметры: -O3 -fno-unroll-loops

vector.body:

%0 = getelementptr inbounds %struct.testFunc, %struct.testFunc* %this, i64 0, i32 0, i64 %index

%1 = bitcast float* %0 to <4 x float>*

%wide.load = load <4 x float>, <4 x float>* %1, align 4, !tbaa !3

%2 = getelementptr inbounds %struct.testFunc, %struct.testFunc* %this, i64 0, i32 1, i64 %index

%3 = bitcast float* %2 to <4 x float>*

%wide.load12 = load <4 x float>, <4 x float>* %3, align 4, !tbaa !3

%4 = fadd <4 x float> %wide.load, %wide.load12

%5 = getelementptr inbounds %struct.testFunc, %struct.testFunc* %this, i64 0, i32 2, i64 %index

%6 = bitcast float* %5 to <4 x float>*

store <4 x float> %4, <4 x float>* %6, align 4, !tbaa !3

Скалярный vs векторизованный и “раскрученный” код

В 4 раза больше
данных, в 4 раза
больше инструкций
за 1 итерацию

Параметры: -O3 -fno-vectorize -fno-unroll-loops

for.body:

%0 = load float, float* %arrayidx, align 4, !tbaa !3

%1 = load float, float* %arrayidx3, align 4, !tbaa !3

%add = fadd float %0, %1

store float %add, float* %arrayidx5, align 4, !tbaa !3

Параметры: -O3

vector.body:

%index = phi i64 [%index.next.1, %vector.body], [0, %entry]

%3 = getelementptr inbounds float, float* %0, i64 %index

%4 = bitcast float* %3 to <4 x float>*

%wide.load = load <4 x float>, <4 x float>* %4, align 4, !tbaa !13, !alias.scope !15

%5 = getelementptr inbounds float, float* %3, i64 4

%6 = bitcast float* %5 to <4 x float>*

%wide.load24 = load <4 x float>, <4 x float>* %6, align 4, !tbaa !13, !alias.scope !15

%7 = getelementptr inbounds float, float* %1, i64 %index

%8 = bitcast float* %7 to <4 x float>*

%wide.load25 = load <4 x float>, <4 x float>* %8, align 4, !tbaa !13, !alias.scope !18

%9 = getelementptr inbounds float, float* %7, i64 4

%10 = bitcast float* %9 to <4 x float>*

%wide.load26 = load <4 x float>, <4 x float>* %10, align 4, !tbaa !13, !alias.scope !18

%11 = fadd <4 x float> %wide.load, %wide.load25

%12 = fadd <4 x float> %wide.load24, %wide.load26

%13 = getelementptr inbounds float, float* %2, i64 %index

%14 = bitcast float* %13 to <4 x float>*

store <4 x float> %11, <4 x float>* %14, align 4, !tbaa !13, !alias.scope !20, !noalias !22

%15 = getelementptr inbounds float, float* %13, i64 4

%16 = bitcast float* %15 to <4 x float>*

store <4 x float> %12, <4 x float>* %16, align 4, !tbaa !13, !alias.scope !20, !noalias !22

%17 = getelementptr inbounds float, float* %0, i64 %index.next

%18 = bitcast float* %17 to <4 x float>*

%wide.load.1 = load <4 x float>, <4 x float>* %18, align 4, !tbaa !13, !alias.scope !15

%19 = getelementptr inbounds float, float* %17, i64 4

%20 = bitcast float* %19 to <4 x float>*

%wide.load24.1 = load <4 x float>, <4 x float>* %20, align 4, !tbaa !13, !alias.scope !15

%21 = getelementptr inbounds float, float* %1, i64 %index.next

%22 = bitcast float* %21 to <4 x float>*

%wide.load25.1 = load <4 x float>, <4 x float>* %22, align 4, !tbaa !13, !alias.scope !18

%23 = getelementptr inbounds float, float* %21, i64 4

%24 = bitcast float* %23 to <4 x float>*

%wide.load26.1 = load <4 x float>, <4 x float>* %24, align 4, !tbaa !13, !alias.scope !18

%25 = fadd <4 x float> %wide.load.1, %wide.load25.1

%26 = fadd <4 x float> %wide.load24.1, %wide.load26.1

%27 = getelementptr inbounds float, float* %2, i64 %index.next

%28 = bitcast float* %27 to <4 x float>*

store <4 x float> %25, <4 x float>* %28, align 4, !tbaa !13, !alias.scope !20, !noalias !22

%29 = getelementptr inbounds float, float* %27, i64 4

%30 = bitcast float* %29 to <4 x float>*

store <4 x float> %26, <4 x float>* %30, align 4, !tbaa !13, !alias.scope !20, !noalias !22

- *В целях улучшения читаемости вспомогательные инструкции были удалены из листингов*

Измерение времени исполнения

- benchmark_app/main.cpp
 - Приложения для измерения времени исполнения
- Запуск
 - `./clang -O3 ./benchmark_app/main.cpp -I ./loops/vectorizable/ -DINCLUDE_TEST=\"testN.hpp\" -o ./benchmark_app.exe`
 - Где **N** – номер вашего варианта
 - `./benchmark_app.exe`

- Пример вывода:

Test-1 duration = 55.407 ms

Test-2 duration = 54.482 ms

Test-3 duration = 56.467 ms

Test-4 duration = 59.506 ms

Test-5 duration = 58.993 ms

Test-6 duration = 56.75 ms

Test-7 duration = 56.647 ms

Test-8 duration = 64.19 ms

Test-9 duration = 61.116 ms

Test-10 duration = 61.616 ms

average duration = 58.5174 ms

Выводы

- Необходимо заполнить секции
 - Анализ автоматически векторизованного LLVM IR
 - Анализ времени исполнения автоматически векторизованного кода

Опциональные задачи

- Дополнительно взять для анализа один из циклов содержащихся в папке `not_vectorizable`
 - Требуется внести правки реализацию цикла чтобы компилятор мог его векторизовать
- Желаящие так же могут
 - Выполнить отчёт в формате Markdown/latex/etc
 - Выполнить отчёт на английском языке

Полезные ссылки

- [Goldbolt](#)
- [LLVM auto vectorization guide](#)
- [Intel Intrinsics Guide](#)
- [Interleaving and unrolling](#)
- [LLVM Language Reference Manual](#)

Вопросы