Профилирование и бенчмаркинг

Никита Манович
Itseez, 2016

Основные определения

Профилирование - сбор характеристик работы программы, таких как время выполнения отдельных фрагментов (обычно подпрограмм), число верно предсказанных условных переходов, число кэш-промахов и т. д.

Профилировщик - это программа, которая собирает характеристики работы приложения для дальнейшего анализа.

Бенчмаркинг - это процесс измерения производительности разных частей программы и сравнение результатов с эталоном.

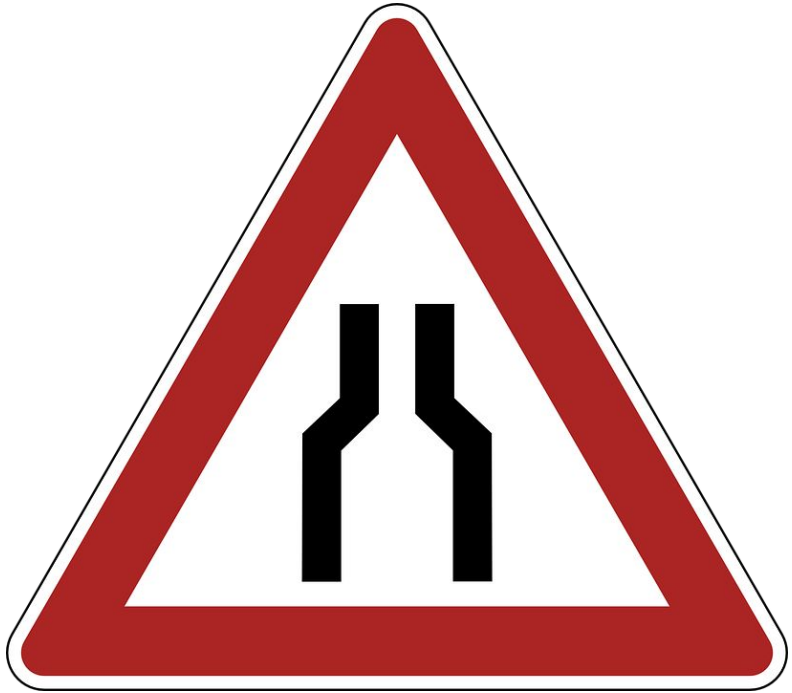
Зачем профилировать код?

- Лучше понимать приложение и его архитектуру
- Находить “узкие” места в программе и понимать из-за чего они “тормозят”
- Рассчитать потенциал “разгона” приложения
- Не тратить время на оптимизацию кода, который и так работает достаточно быстро
- Не усложнять код там, где это не нужно
- Экономить на “железе”

Правило 80/20 - обычно 20% кода потребляют 80% ресурсов системы

Типичные “узкие” места

- Процессор
- Подсистема ввода-вывода
- Оперативная память
- Сетевые задержки
- Разделяемые ресурсы
- Частые системные вызовы
- Внешние ресурсы
  - Базы данных
  - Web-сервисы



Типы профилировщиков

- Инструментация исходного кода
- Статистический сэмплинг
- Статистический граф-вызовов
- Статическая бинарная инструментация
- Подмена функций с помощью LD_PRELOAD
- Динамическая бинарная инструментация

Инструментация исходного кода

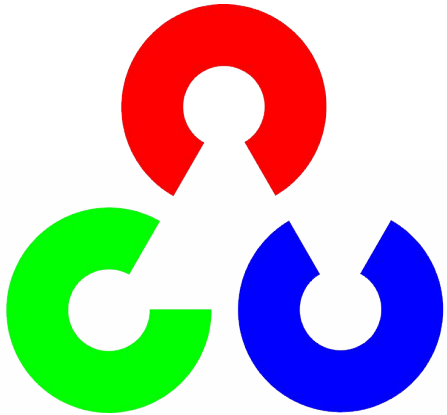
Преимущества

- Переносим и работает на всех платформах
- Легко интерпретируемые результаты
- Отображение только нужных данных

Недостатки

- Замедляет код
- Плохо расширяется
- Требует пересборки

Пример getTickCount из OpenCV



```
270     int64 t = getTickCount();
271     if( alg == STEREO_BM )
272         bm->compute(img1, img2, disp);
273     else if( alg == STEREO_VAR ) {
274         var(img1, img2, disp);
275     }
276     else if( alg == STEREO_SGBM || alg == STEREO_HH )
277         sgbm->compute(img1, img2, disp);
278     t = getTickCount() - t;
279     printf("Time elapsed: %fms\n", t*1000/getTickFrequency());
280
```

Профилировка с помощью GNU gprof

Необходимо выполнить несколько шагов:

- Скомпилировать и слинковать программу с -pg ключом
- Запустить свою программу как обычно. Файл gmon.out будет записан по завершению программы автоматически
- Запустить утилиту gprof для анализа профилировочных данных
  - $ gprof <binary> gmon.out (плоский профиль и граф-вызовов)
  - $ gprof <binary> gmon.out -A (аннотация исходного кода)

Пример вывода gprof для minigzip

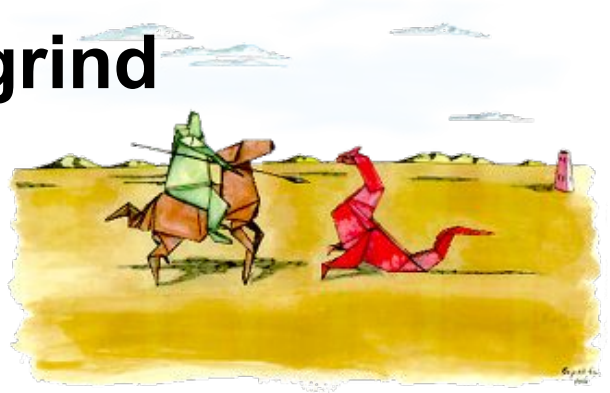
% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
47.03	46.03	46.03	488169	0.00	0.00	deflate_slow
28.39	73.82	27.78	2514884599	0.00	0.00	longest_match
6.54	80.22	6.40	224116	0.00	0.00	_tr_stored_block
6.48	86.56	6.34	488687	0.00	0.00	fill_window
5.35	91.80	5.24	163388023	0.00	0.00	pqdownheap
3.06	94.79	2.99	244216	0.00	0.00	crc32
1.45	96.21	1.42	731871	0.00	0.00	build_tree
0.90	97.09	0.88	19841	0.00	0.00	compress_block
0.45	97.53	0.44				crc32_combine64
0.13	97.66	0.13	487914	0.00	0.00	scan_tree
0.06	97.72	0.06	976585	0.00	0.00	deflate
0.05	97.77	0.05				deflateCopy
0.04	97.81	0.04	243957	0.00	0.00	_tr_flush_block
0.04	97.85	0.04	39682	0.00	0.00	send_tree
0.02	97.87	0.02	244213	0.00	0.00	gz_comp
0.02	97.89	0.02	732373	0.00	0.00	flush_pending
0.00	97.89	0.00	732373	0.00	0.00	_tr_flush_bits
0.00	97.89	0.00	732373	0.00	0.00	bi_flush
0.00	97.89	0.00	244213	0.00	0.00	gzwrite

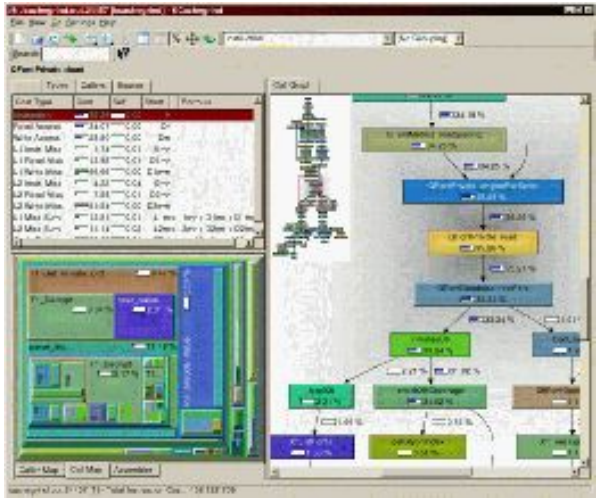
Ограничения gprof

- Требует пересобрать приложение с -pg ключом
- Накладные расходы могут быть большими (от 30% до 260%)
- Как любой статистический подход дает приближенные результаты
- Не считает время, которое было проведено вне приложения (системные вызовы, I/O, переключение контекста)
- Не работает с разделяемыми библиотеками

Профилировка с помощью valgrind

- Не требуется никаких специальных опций компиляции
- Программа запускается под valgrind в режиме callgrind
  - $ valgrind --tool=callgrind <binary> <options>
- Для визуализации данных можно использовать kcachegrind
- Позволяет оценить работу приложения с памятью (упрощенная модель)

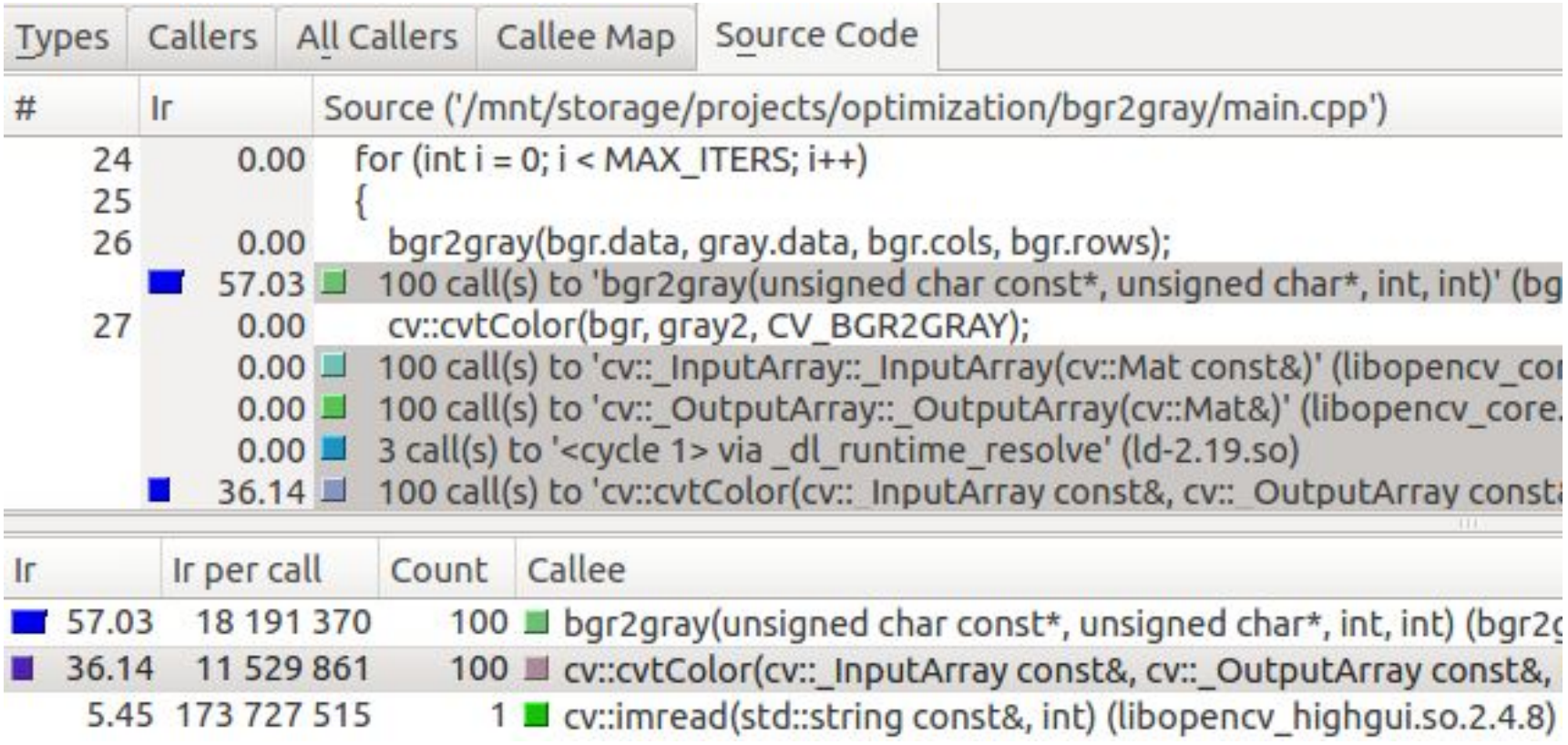




Пример преобразования цветов

```
3  void bgr2gray(unsigned char const* bgr, unsigned char* gray,
4      int width, int height)
5  {
6      enum { R, G, B };
7      static const float cr = 0.114f;
8      static const float cg = 0.587f;
9      static const float cb = 0.299f;
10
11     int n = width * height;
12     for (int i = 0; i < n; i++, bgr += 3)
13     {
14         int v = bgr[R] * cr + bgr[G] * cg + bgr[B] * cb;
15         gray[i] = v > 255? 255 : v;
16     }
17 }
```

Пример вывода KCachegrind



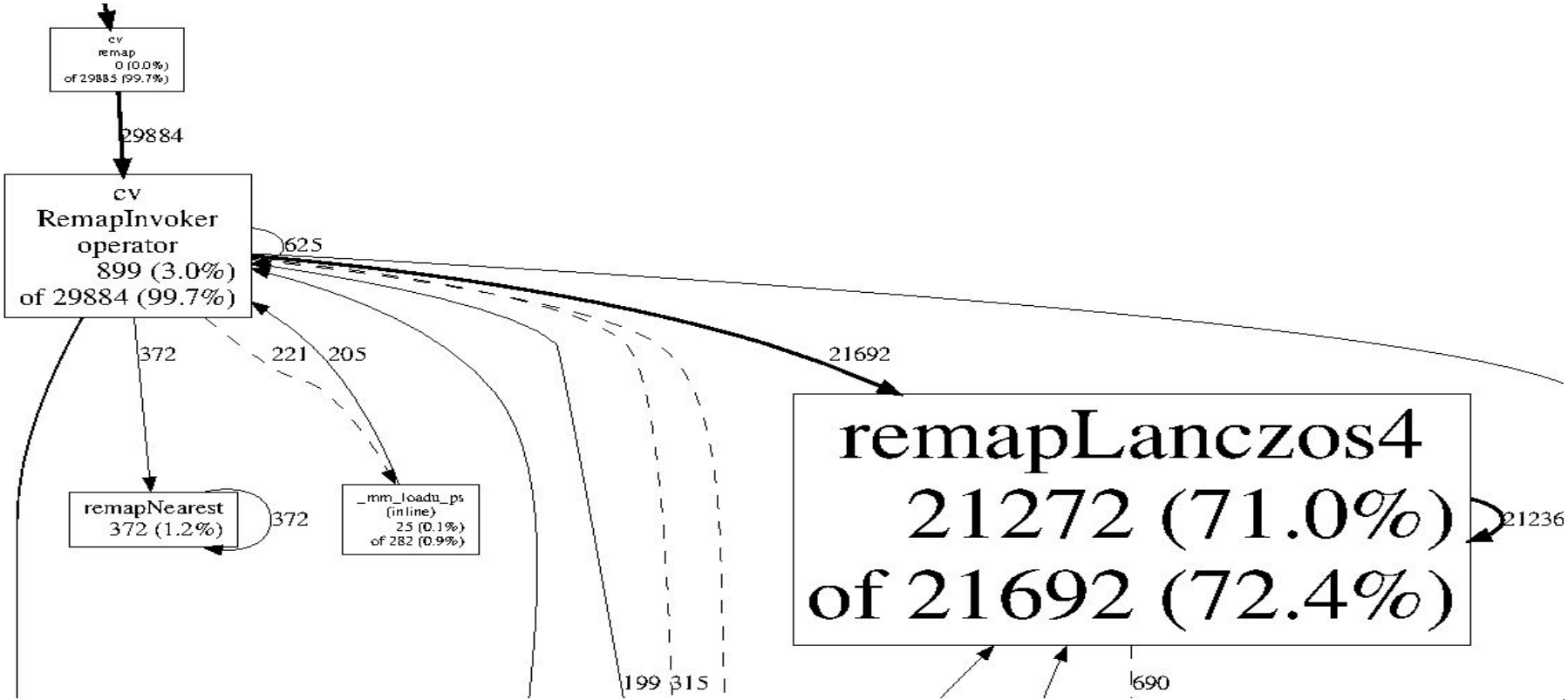
Ограничения valgrind

- Замедляет исследуемое приложение в 10x раз (в данном примере в 0.535s против 8.194s, в 15.32x)
- Можно анализировать только небольшие программы
- Позволяет проанализировать работу с памятью, но для других целей его использование сомнительно

Профилировка с помощью Google perf tools

- В коде можно использовать специальный API, который позволяет указать что профилировать: ProfilerStart() и ProfilerStop()
- Собрать программу с -lprofiler или использовать LD_PRELOAD
- Определить переменную окружения CPUPROFILE и установить её значение в файл, куда будут сохраняться результаты
- Запустить программу как обычно
- С помощью скрипта pprof проанализировать результаты
  - `$ pprof --text <binary> $CPUPROFILE` (вывести плоский профиль на консоль)
  - `$ pprof --gv <binary> $CPUPROFILE` (сгенерировать граф вызовов)
  - `$ pprof --callgrind <binary> $CPUPROFILE` (сгенерировать данные в формате callgrind)
- Возможная гранулярность: адрес / линия / функция / файл

Пример вывода Google perf tools



Ограничения Google perf tools

- Можно собрать только информацию о CPU-bound приложениях
- Нет возможности собрать CPU события
- Нет удобного GUI для визуализации результатов
- Как любой статистический подход дает приближенные результаты
- Не работает в случае, если программа завершается по сигналу

Профилировка с Intel VTune Amplifier

Коммерческий профилировщик, который поддерживает Linux, Windows, Android и некоторые другие ОС. Работает преимущественно на x86 архитектуре.

Имеет графический интерфейс, но большинство возможностей можно запустить из командной строки.

Пример вывода Intel VTune Amplifier

libldw-1.4.so	76.875s
itseez::ldw::RidgeDetector::process	29.752s
itseez::ldw::find	9.419s
itseez::arguscv::(anonymous namespace)::remapNearest<unsigned char>	8.894s
itseez::ldw::MotionHistoryFilter::process	5.014s
itseez::ldw::ComponentTracker::track	3.219s

Module / Function / Call Stack	Clockticks	Instructions Retired	CPI Rate	Retiring uOps	Wasted Work	Back-end Issues			Front-end Issues
					Branch Mispredict	Divider	Memory Latency		
							LLC Miss	LLC Hit	
libldw-1.4.so	177,032,265,548	142,840,214,260	1.239	0.441	0.109	0.082	0.031	0.020	0.041
itseez::ldw::RidgeDetector::process	68,460,102,690	35,986,053,979	1.902	0.442	0.076	0.189	0.000	0.002	0.017
itseez::arguscv::(anonymous namespace)::remapNearest	20,844,031,266	21,990,032,985	0.948	0.412	0.031	0.000	0.109	0.039	0.004
itseez::ldw::find	20,258,030,387	20,736,031,104	0.977	0.466	0.078	0.017	0.000	0.007	0.039
itseez::ldw::MotionHistoryFilter::process	12,342,018,513	13,758,020,637	0.897	0.532	0.171	0.032	0.000	0.017	0.060
itseez::ldw::ComponentTracker::track	7,306,010,959	7,064,010,596	1.034	0.440	0.322	0.003	0.069	0.018	0.142

Ограничения Intel VTune Amplifier

- Высокая стоимость
- Содержит несколько технологий профилирования, каждая из которых имеет свои недостатки, которые не всегда очевидны
- Работает преимущественно на Intel архитектурах
- Для продвинутого анализа приложений требуется установка драйверов, что может приводить к краху всей системы

Тестирование производительности

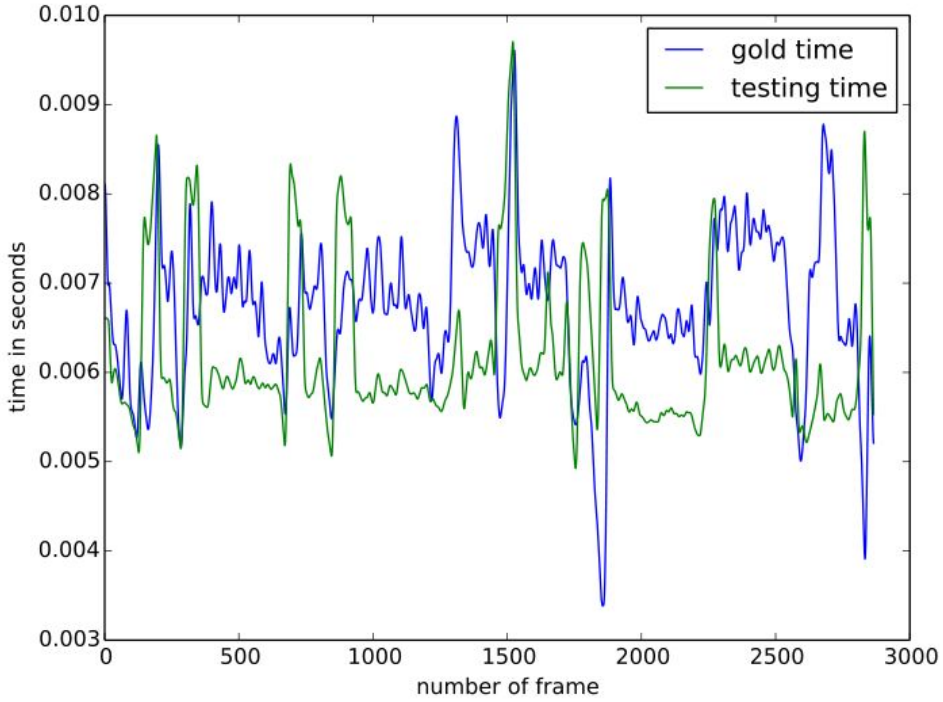
- Для крупных проектов так же необходимо, как и тестирование качества
- Цель - это объективная оценка производительности приложения на разных платформах
- Создание плацдарма для дальнейшей оптимизации

```
Note: Google Test filter = Size_MatType_countNonZero.*
[==========] Running 28 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 28 tests from Size_MatType_countNonZero
[ RUN   ] Size_MatType_countNonZero.countNonZero/0
[ VALUE ]       (640x480, 8UC1)
[       OK ] Size_MatType_countNonZero.countNonZero/0 (14 ms)
[ RUN   ] Size_MatType_countNonZero.countNonZero/1
[ VALUE ]       (640x480, 8SC1)
[       OK ] Size_MatType_countNonZero.countNonZero/1 (14 ms)
[ RUN   ] Size_MatType_countNonZero.countNonZero/2
[ VALUE ]       (640x480, 16UC1)
[       OK ] Size_MatType_countNonZero.countNonZero/2 (80 ms)
```

Как тестировать производительность?

- **Gtest фреймворк с расширением** + скрипты для удобного запуска, сбора и сравнения результатов (подход, который использует OpenCV)
- **С помощью профилировщиков** сравнивать основные “горячие” точки по разным характеристикам (например, Intel VTune Amplifier может выдавать разницу по двум результатам)
- В сложных алгоритмах компьютерного зрения можно **коррелировать время обработки одного фрейма в зависимости от номера** кадра на видео последовательностях
- **Сравнивать с эталоном** или решением от конкурентов (OpenCV, FastCV, OpenCV for Tegra, AcceleratedCV)

Сравнение производительности в Itseez/ADAS



Основные выводы

- Существует большое количество инструментов, которые помогают собирать необходимые метрики работы приложения
- Нужно понимать как достоинства так и недостатки инструментов, а также умело пользоваться ими
- Простой анализ можно сделать “руками” (замеряем / печатаем)
- Сложный анализ приложения возможен только с помощью специальных инструментов
- Тестирование производительности необходимо, как основа для развития и роста кодовой базы

Вопросы