

Профилирование и бенчмаркинг

Никита Манович
Intel 2020

Преждевременная
оптимизация — корень
всех (или большинства)
проблем в
программировании.



Профилирование: основные определения

Оптимизация — модификация программного кода для улучшения его эффективности.

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

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

Профилировка с помощью таймера



```
img1 = cv.imread('messi5.jpg')

e1 = cv.getTickCount()
for i in range(5,49,2):
    img1 = cv.medianBlur(img1,i)
e2 = cv.getTickCount()
t = (e2 - e1)/cv.getTickFrequency()
print( t )

# Result I got is 0.521107655 seconds
```

Профилировка C/C++ с помощью VTune

▼ 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

Профилировка Python с помощью cProfile

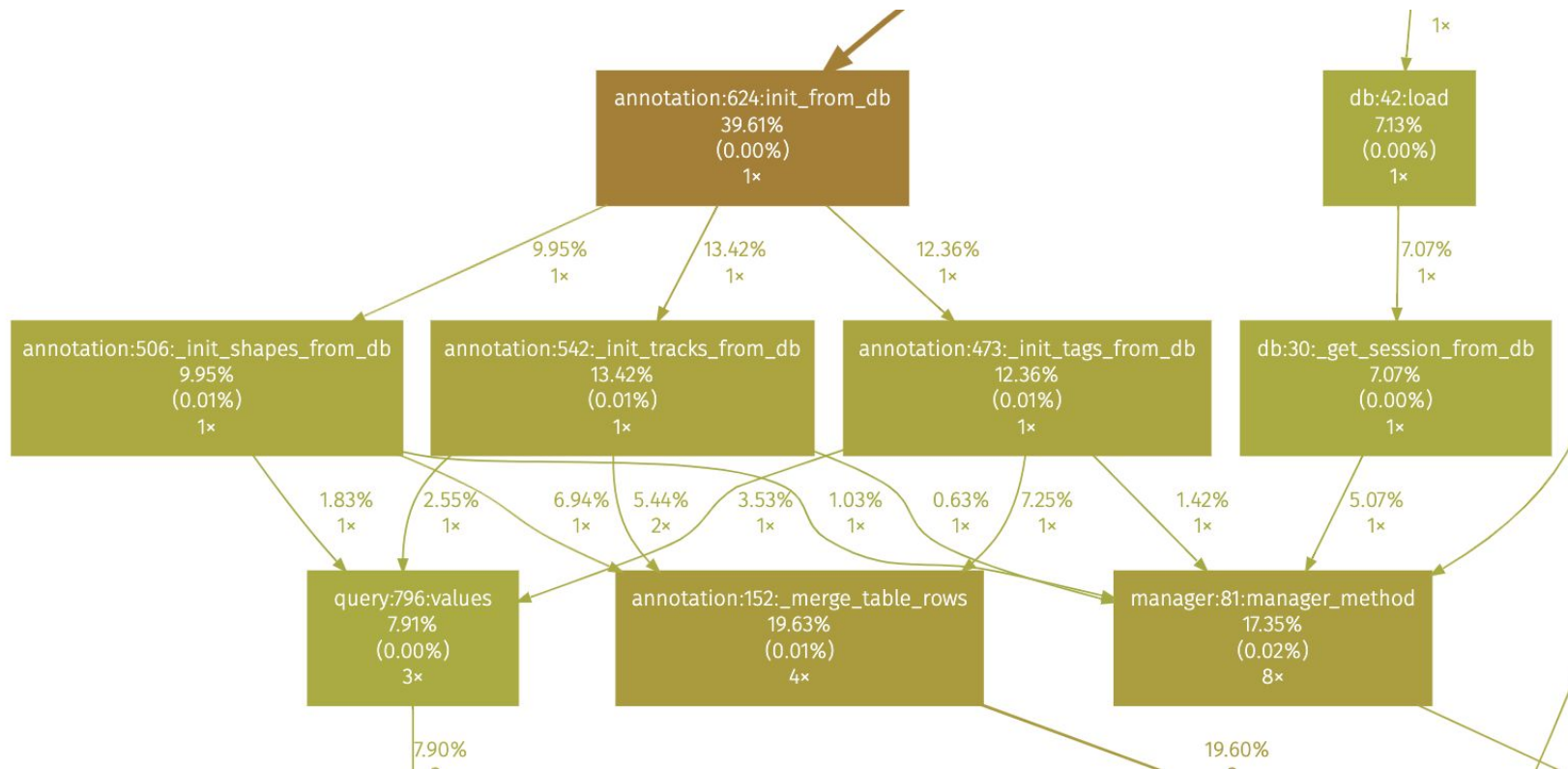
```
1  import cv2 as cv
2  import sys
3
4  img = cv.imread(sys.argv[1])
5  for i in range(1,100):
6      blur1 = cv.blur(img,(5,5))
7      blur2 = cv.GaussianBlur(img,(5,5),0)
8      blur3 = cv.medianBlur(img,5)
9      blur4 = cv.bilateralFilter(img,9,75,75)
```

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
99	20.241	0.204	20.241	0.204	{bilateralFilter}
99	4.033	0.041	4.033	0.041	{medianBlur}
99	1.245	0.013	1.245	0.013	{blur}
1	0.719	0.719	27.219	27.219	opencv_ex1.py:1(<module>)
99	0.462	0.005	0.462	0.005	{GaussianBlur}

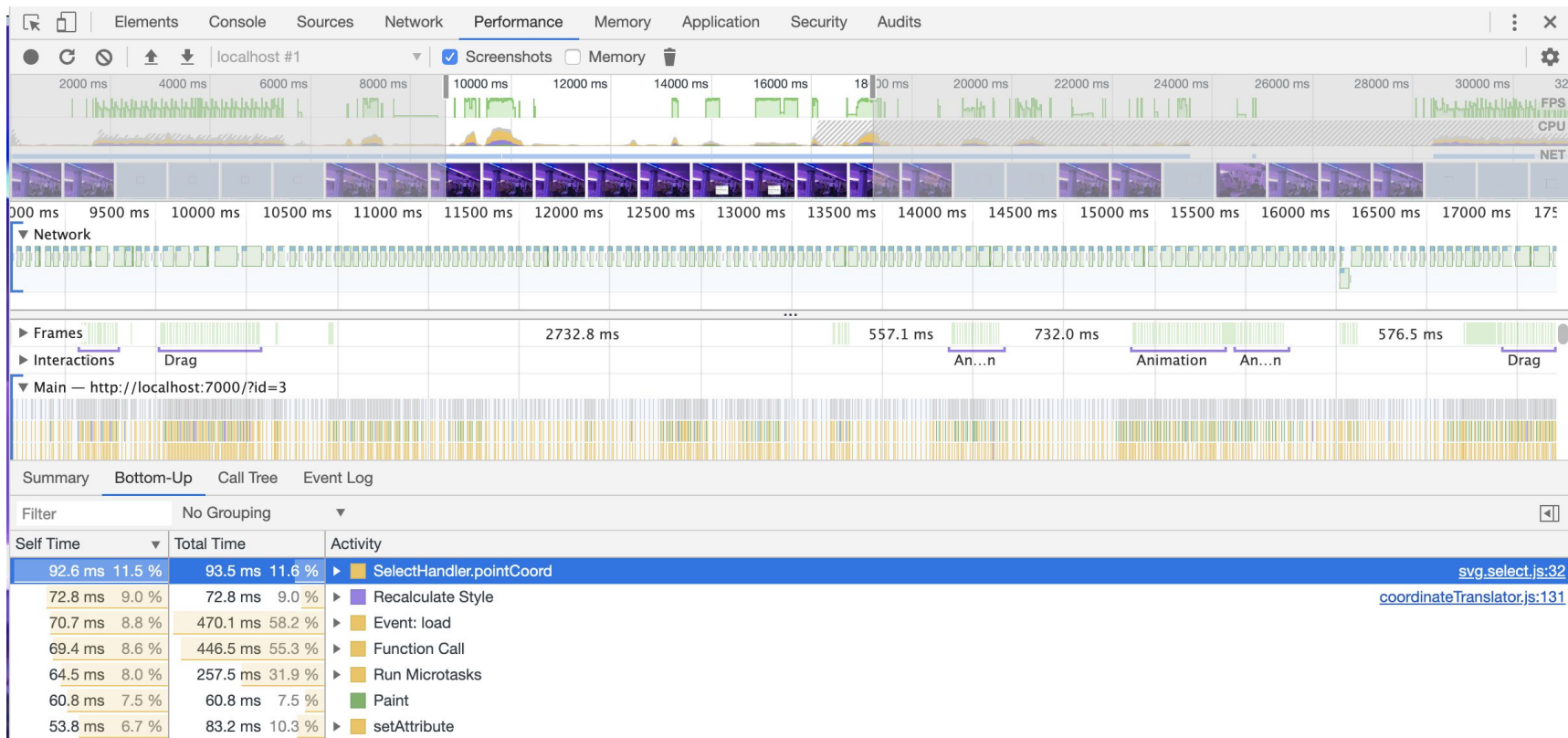
Профилировка Django с помощью Silk

← → ↺ localhost:8080/profiler/requests/						
Summary		Requests		Profiling		
				Show: 25 ▾ Order: Recent		
07:52:35.955	07:52:35.951	07:52:35.947	07:52:35.624	07:52:34.845	07:52:33.104	07:52:29.059
200 GET	200 GET	200 GET	200 GET	200 GET	200 GET	200 GET
/api/v1/server/annotation/formats	/api/v1/tasks/1	/api/v1/jobs/1/annotations	/api/v1/jobs/1	/	/git/repository/get/1	/api/v1/tasks
2012ms overall 360ms on queries 19 queries	1372ms overall 188ms on queries 6 queries	1807ms overall 351ms on queries 10 queries	244ms overall 28ms on queries 4 queries	302ms overall 15ms on queries 2 queries	176ms overall 17ms on queries 3 queries	554ms overall 52ms on queries 8 queries
07:52:28.593	07:52:27.550	07:52:27.364	07:52:27.327	07:52:26.776		
200 GET	200 OPTIONS	404 OPTIONS	404 GET	404 OPTIONS		
/api/v1/server/dataset/formats	/git/repository/meta/get	/auto_annotation/meta/get	/analytics/app/kibana	/tensorflow/annotation/meta/get		
73ms overall 11ms on queries 1 queries	430ms overall 111ms on queries 3 queries	596ms overall 0ms on queries 0 queries	599ms overall 0ms on queries 0 queries	453ms overall 0ms on queries 0 queries		
07:52:26.737	07:52:26.720	07:52:26.707	07:52:26.692	07:52:26.442	07:52:26.235	
404 OPTIONS	200 GET	200 GET	200 GET	200 GET	200 GET	200 GET
/tensorflow/segmentation/meta/get	/api/v1/server/about	/api/v1/users	/api/v1/server/annotation/formats	/api/v1/users/self	/api/v1/users/self	
516ms overall 0ms on queries 0 queries	552ms overall 56ms on queries 1 queries	1079ms overall 163ms on queries 4 queries	1814ms overall 215ms on queries 18 queries	174ms overall 16ms on queries 2 queries	171ms overall 16ms on queries 2 queries	

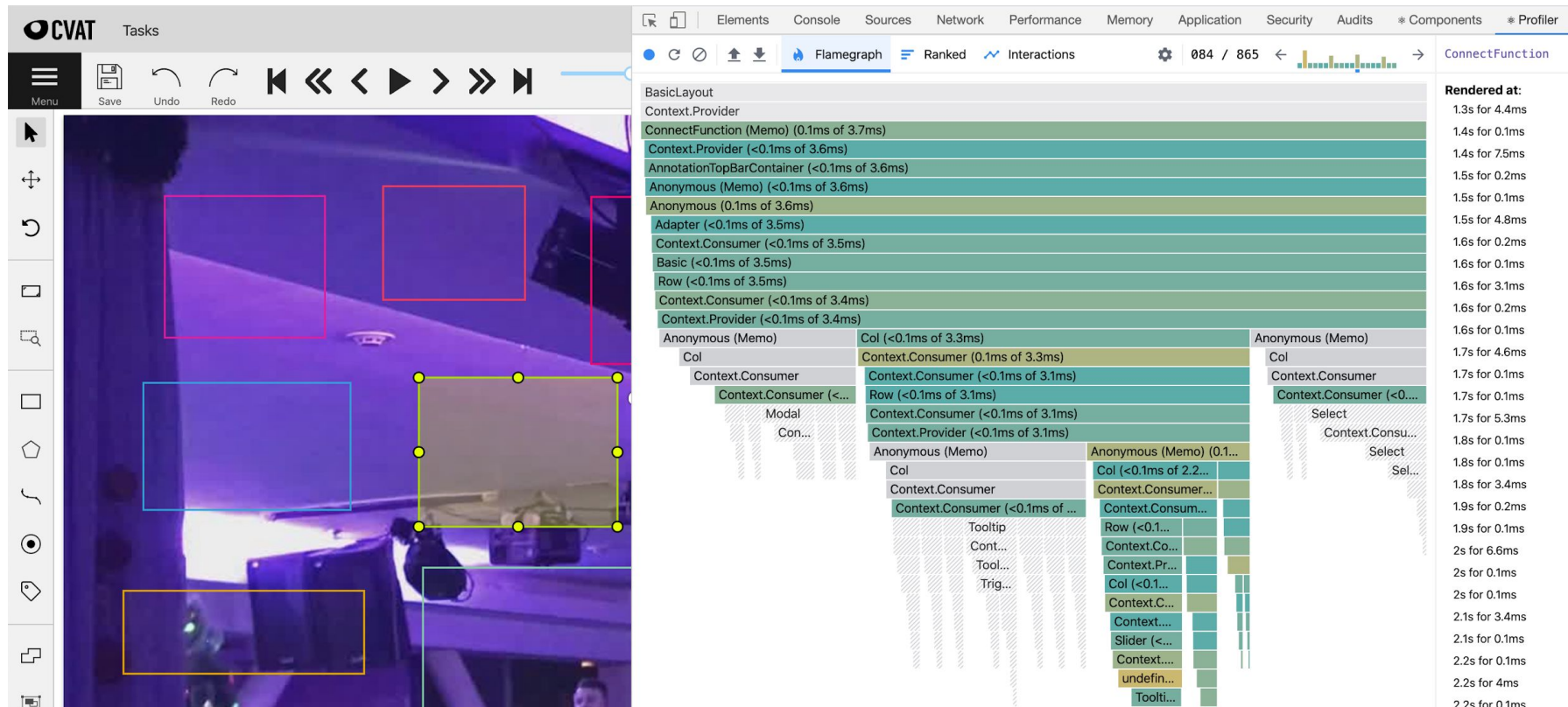
Профилировка Django с помощью Silk



Профилировка JavaScript с помощью Chrome



Профилировка с помощью React Profiler



Цели профилировки

- 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 быстро работает
- 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 кода

- Инструментация исходного кода
- Статическая бинарная инструментация
- Динамическая бинарная инструментация
- С помощью LD_PRELOAD

Статистические методы

- Статистический сэмплинг
- Статистический граф-вызовов



Гранулярность информации

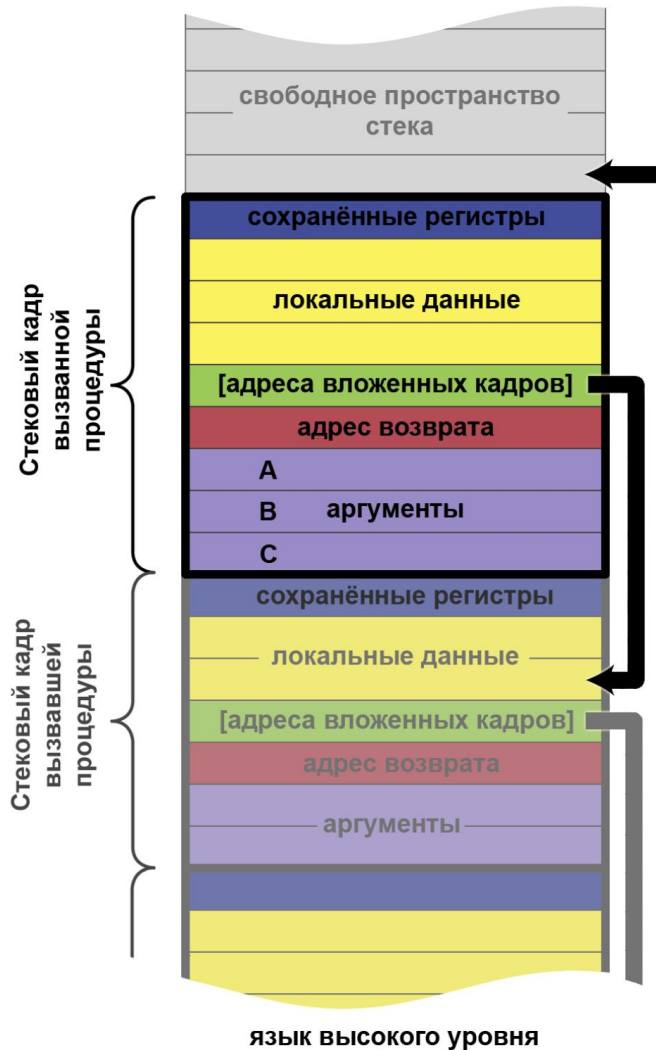


- Модуль или программа
- Файл с исходным кодом
- Функция или метод класса
- Строчка кода в исходном коде
- Байт код, машинная инструкция

Собираемая информация

- Процессорные события
- События операционной системы
- Время выполнения частей программы
- Время ожидания каких-либо событий
- Количество вызов функций
- Стек вызовов

Собираемая информация атрибутируется к различным интересным участкам программы.



Инструментация исходного кода

Преимущества

- Переносим и работает на всех платформах
- Полный контроль над собираемой информацией
- Легко интерпретируемые результаты

Недостатки

- В общем случае существенно замедляет код
- Плохо расширяется и тяжело поддерживается
- Требуется пересборки программы

Статическая бинарная instrumentation

Преимущества

- Полный контроль над собираемой информацией
- Легко интерпретируемые результаты
- Не требует пересборки программы

Недостатки

- В общем случае существенно замедляет код
- Требуется наличие специальных инструментов
- Instrumentation необходима после каждой пересборки программы

Динамическая бинарная instrumentation

Преимущества

- Полный контроль над собираемой информацией
- Легко интерпретируемые результаты
- Не требует пересборки программы
- Можно настроить под входные данные

Недостатки

- В общем случае существенно замедляет код
- Требуется наличие специальных инструментов

Подмена функций с помощью LD_PRELOAD

Преимущества

- Полный контроль над собираемой информацией
- Легко интерпретируемые результаты
- Не требует пересборки программы
- Стандартный механизм, встроенный в операционную систему

Недостатки

- В общем случае существенно замедляет код
- Зависит от операционной системы
- Можно подменять функции только из динамических библиотек

Статистический сэмплинг

Преимущества

- Работает на большинстве платформ
- Легко интерпретируемые результаты
- Отображение только нужных данных
- Не замедляет код (зависит от технологии)

Недостатки

- Невозможно собрать статистику без специальных инструментов
- Не всегда позволяет выявить проблемы
- Требуется наличие отладочной информации

Бенчмаркинг: основные определения

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

Тест производительности, бенчмарк - контрольная задача, необходимая для определения сравнительных характеристик производительности компьютерной системы.

Бенчмаркинг: задачи и типы

Типичные задачи

- **Тестирование производительности**, то есть сравнение запуска с предыдущей версией или ожидаемым результатом
- **Сравнение с конкурентами** для демонстрации преимущества своего решения

Основные типы (*)

- **Микро бенчмаркинг** - похоже на юнит тест и тестирует небольшую часть системы или отдельную подпрограмму
- **Макро бенчмаркинг** - напоминает интеграционное тестирование и покрывает какой-то сценарий использования системы

(*) Разделение по типам условное и зависит от контекста. Одна и та же бенчмарка в разных контекстах может быть как микро, так и макро.

Бенчмаркинг: основные принципы

- **Актуальность:** измерять полезные характеристики
- **Репрезентативность:** метрики производительности должны быть приняты академией и индустрией
- **Справедливость:** честное сравнение
- **Повторяемые результаты:** результаты должны быть воспроизводимы
- **Прозрачность:** метрики должны быть интерпретируемыми и понятными
- **Экономическая целесообразность:** легко поддерживать, работает разумное время
- **Масштабируемость:** возможность расширения, поддержка разных сценариев

Бенчмаркинг: проблемы

- **Зависимость от окружения**, таких как операционная система, процессор, подсистема ввода-вывода и т.п.
- **Ложные результаты**, когда регрессия с производительностью не может быть найдена или происходит большое количество ложных срабатываний
- **Воспроизводимость результатов**, то есть получение одних и тех же результатов с заданной погрешностью при нескольких запусках
- **Неприемлемое время работы**, например, невозможно быстро оценить результаты оптимизации
- **Неадекватные метрики**, например, среднее время не всегда является хорошим показателем (системы реального времени)

Бенчмаркинг: советы

- **Запуск в изолированном окружении** (например, отключение антивируса и других тяжелых сервисов)
- **Сравнение с предельным случаем** (например, чтение с парсингом)
- **Запуск множества раз и усреднение результатов**
- **Использование фреймворков для бенчмаркинга** (например, [google/benchmark](https://github.com/google/benchmark), Celero)

Бенчмаркинг: примеры реализации

- **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 видеопоследовательностях
- **Сравнивать с эталоном** или решением от конкурентов (OpenCV, FastCV, OpenCV for Tegra)

Бенчмаркинг: промышленные примеры

- **MPerf** (<https://mlperf.org>) - тесты для измерения производительности тренировки и вывода DL моделей на GPU, CPU, TPU и т.д.
- **LINPACK** (<http://www.netlib.org/benchmark/>) - для измерения вычислительной производительности компьютеров при обработке чисел с плавающей запятой.
- **AnTuTu** (<http://www.antutu.com>) - программа для тестирования производительности устройств, обычно используемая для тестирования смартфонов и планшетов.
- **CoreMark** (<http://www.eembc.org/coremark/>) - набор синтетических тестов производительности для измерения скорости центральных процессоров во встраиваемых системах.

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

