Федеральное государственное автономное образовательное учреждение высшего профессионального образования
«Национальный исследовательский университет «Высшая школа экономики»

Московский институт электроники и математики им. А.Н. Тихонова

Прикладная математика, Информатика и вычислительная техника, Компьютерная безопасность, Системный анализ и математические технологии
Бакалавриат, Специалитет, Магистратура

О Т Ч Е Т

по исследованию улучшения производительности

по проектной работе

Разработка системы распознавания (ML) и интеллектуального контроля ручных операций в промышленном производстве

Выполнил студент:
Ляпунова Софья Александровна,
группа БИВ214

Москва 2024

Оглавление

1. Цель

Повышение количества кадров в секунду (FPS) в приложении для контроля ручных операций на производстве. Улучшение FPS предполагается осуществить путем оптимизации вычислений, в том числе через более эффективное распределение задач между графическим (GPU) и центральным (CPU) процессорами. Высокий FPS обеспечит более точное и быстрое отслеживание операций в реальном времени, что улучшит качество контроля и безопасность трудовых процессов.

2. Теоретическая основа

2.1.FPS

FPS (Frames Per Second) является мерой частоты смены кадров в приложении, определяя, насколько плавно отображается видео и графика. Чем выше FPS, тем более плавным для пользователя становится визуальный опыт. Низкий FPS может привести к рывкам и задержкам, что негативно сказывается на взаимодействии пользователя с приложением.

2.2.Параллельные и конкурентные вычисления

Параллельное выполнение (Parallel execution) подразумевает физическое одновременное выполнение нескольких задач. Это достигается за счет использования многоядерных процессоров, где каждое ядро может обрабатывать отдельную задачу. Параллелизм идеально подходит для задач, которые можно разделить на независимые подзадачи, выполняемые одновременно без необходимости взаимодействия между ними.

Конкурентное выполнение (Concurrent execution) предполагает, что система может обрабатывать несколько задач в один и тот же момент времени. Это не обязательно означает, что задачи выполняются физически одновременно. В одноядерных системах это достигается за счет быстрого переключения контекста между задачами, создавая иллюзию одновременности. В многоядерных системах конкурентность может быть реализована с помощью параллелизма, но ключевой особенностью является возможность взаимодействия и совместной работы задач.

3. Методы исследования

3.1.Перевод вычислений на GPU

3.1.1. Теория

Использование GPU для выполнения вычислений позволяет значительно ускорить обработку данных благодаря его способности выполнять тысячи параллельных операций. GPU имеют значительно больше ядер по сравнению с CPU, что делает их идеальными для задач, требующих массового параллелизма.

Библиотеки, такие как CUDA (для графических процессоров NVIDIA) и OpenCL, предоставляют средства для разработки программ, использующих мощность GPU.

3.1.2. Практика

В данном исследовании был выполнен перевод вычислительной части приложения на GPU с использованием библиотеки CUDA .

В таблице 17 представлено сравнение производительностей различных моделей, выполняемых на CPU и двух GPU: NVIDIA GeForce RTX 3080 (обозначена как 0) и NVIDIA GeForce GTX 1660 (обозначена как 1). В последнем столбце таблицы указан процент увеличения FPS (кадров в секунду) по сравнению с выполнением на CPU.

Таблица 17. Производительность моделей при выполнении на различных комбинациях GPU и CPU

Назначение моделей на процессоры			Результаты	
palm model	drone tools model	drone details model	Количество кадров в секунду, обработанных системой при	Процентное изменение FPS по сравнению с выполнением на CPU,

			выполнении моделей, FPS	%
cpu	cpu	cpu	10.585795500815214	0
0	0	0	10.760937562563763	1,65450071
0	0	1	10.795655313251011	1,982466149
0	1	0	10.97882685224355	3,712818289
0	1	1	9.892921512065698	-6,545318098
1	0	0	10.902879716688143	2,99537447
1	0	1	9.942854624773434	-6,073618898
1	1	0	10.01346692060467	-5,406571289
1	1	1	9.745736339833135	-7,935720664

Наилучшей комбинацией оказалось выполнение моделей на GPU 0 (NVIDIA GeForce RTX 3080), GPU 1 (NVIDIA GeForce GTX 1660) и снова GPU 0, что дало прирост FPS на 3,71% по сравнению с выполнением на CPU.

Однако этот прирост производительности является незначительным. В нашем случае возможно, что вычислительные задачи не подходили для архитектуры GPU. Если задачи содержат множество последовательных операций и зависимостей, то использование GPU может не дать ожидаемого прироста производительности.

3.2.Многопоточность (Threading и ThreadPoolExecutor)

3.2.1. Теория

Многопоточность - это способность программы выполнять несколько потоков (нитей) одновременно. Каждый поток представляет собой отдельный путь выполнения внутри одного процесса.

Поток (Thread) - это наименьшая единица выполнения программы, которой операционная система выделяет время процессора. Каждый поток имеет собственный стек, регистры и счетчик команд, но разделяет общую область памяти с другими потоками того же процесса. Потоки позволяют эффективно использовать ресурсы процессора и повысить отзывчивость приложений.

Пул потоков (Thread Pool) - это набор предварительно созданных потоков, которые ожидают поступления задач для выполнения. Вместо создания нового потока для каждой задачи, задача назначается одному из свободных потоков в пуле. Это позволяет избежать накладных расходов на создание и уничтожение потоков, повышая производительность приложения. Пул потоков обычно управляется специальным планировщиком, который распределяет задачи между потоками.

GIL (Global Interpreter Lock) - это механизм в реализации Python, который позволяет только одному потоку выполнять байт-код в любой момент времени. Это ограничение было введено для упрощения реализации интерпретатора Python и предотвращения проблем с потоками при работе с объектами в памяти. Однако GIL также ограничивает возможности истинного параллелизма в Python, поскольку только один поток может выполняться одновременно.

3.2.2. Практика

В рамках исследования были созданы три отдельных потока, каждый из которых был назначен для обработки одной из следующих моделей:

- Palm Model
- Drone Tools Model
- Drone Details Model

Каждая модель была назначена на отдельный поток с целью распараллелить вычисления и повысить производительность приложения.

Однако, из-за ограничений, накладываемых GIL, только один поток мог выполнять байт-код в любой момент времени. Это означает, что потенциальные

преимущества параллелизма были ограничены, и многопоточность не смогла обеспечить увеличения FPS.

Результаты исследования показали, что использование многопоточности (Threading и ThreadPoolExecutor) не привело к увеличению FPS в приложении. Это связано с тем, что GIL не позволяет реализовать параллелизм в Python.

3.3.Многопроцессная обработка (Multiprocessing)

3.3.1. Теория

Многопроцессная обработка, или multiprocessing, в Python — это метод параллельного выполнения задач, использующий отдельные процессы вместо потоков. Это позволяет эффективно использовать многопроцессорные и многоядерные системы, поскольку каждый процесс может выполняться на отдельном ядре процессора.

Процесс в контексте многопроцессной обработки представляет собой независимую единицу выполнения в компьютерной системе с собственным набором инструкций и состоянием. Каждый процесс обладает собственным виртуальным адресным пространством, что означает изоляцию процессов друг от друга и невозможность прямого взаимодействия с памятью или ресурсами других процессов без использования специальных механизмов межпроцессного взаимодействия.

Основное преимущество многопроцессной обработки перед многопоточностью заключается в том, что процессы не ограничены GIL (Global Interpreter Lock), характерным для потоков в Python. Это дает возможность параллельной работы на разных ядрах процессора, обеспечивая истинный параллелизм и улучшение производительности задач. Каждый процесс имеет отдельное пространство памяти, что уменьшает риск конфликтов данных и упрощает синхронизацию, в отличие от потоков, которые работают в рамках одного процесса и делят общее пространство памяти. Однако управление процессами требует больше системных ресурсов, чем управление потоками, так

как операционной системе нужно выделить дополнительные ресурсы для каждого запущенного процесса, что приводит к увеличению накладных расходов.

Несмотря на то, что GIL применяется в каждом процессе, созданном через многопроцессность, он не ограничивает параллелизм между процессами, так как они функционируют независимо друг от друга.

Однако при использовании механизмов межпроцессной коммуникации для обмена данными между процессами возникают следующие проблемы:

Гонки данных (Data races): Это ситуация, когда несколько процессов пытаются одновременно получить доступ к одним и тем же данным для чтения или записи. Может привести к тому, что один процесс перезапишет изменения, сделанные другим, что приведет к потере данных или непредсказуемому поведению программы. Например, если два процесса одновременно пытаются обновить одну и ту же запись в базе данных, результат может зависеть от того, какой процесс завершит свою операцию последним.

Deadlocks (Взаимные блокировки): Взаимная блокировка возникает, когда два или более процесса ожидают освобождения ресурсов, которые уже заняты другими процессами. Каждый из процессов ждет, пока другой освободит ресурс, но ни один из них не может продолжить выполнение, так как они заблокированы. Это может привести к полной остановке всех процессов, участвующих в блокировке.

3.3.2. Практика

Были проведены два эксперимента с использованием многопроцессности (multiprocessing).

В первом эксперименте один процесс отвечал за выполнение всех трех моделей, а второй — за захват изображения и отрисовку. Однако это не привело к увеличению скорости работы, поскольку выполнение моделей, осуществляемое последовательно в одном процессе, оказалось самой ресурсоемкой задачей, и параллельная работа второго процесса не смогла компенсировать этот недостаток.

Во втором эксперименте задачи были распределены между тремя

процессами, каждый из которых выполнял одну из моделей, в то время как основной процесс занимался захватом и отрисовкой. Это позволило каждой модели работать независимо и параллельно, что должно было способствовать повышению общей производительности.

Использование разделяемой памяти и очередей предотвратило гонки данных, обеспечивая упорядоченный и контролируемый доступ к данным. Однако, когда процессы должны совместно использовать результаты друг друга — например, когда один процесс должен отрисовывать фреймы, полученные после детекции другими процессами, — возникает взаимная блокировка. Это происходит, потому что каждый процесс зависит от результатов работы других процессов, и если один из процессов задерживается или ожидает ресурс, это блокирует всю систему.

3.4. Асинхронные вычисления (Asyncio)

3.4.1. Теория

Асинхронные вычисления в Python, реализуемые через модуль asyncio, представляют собой мощный инструмент для организации конкурентных вычислений. Позволяет программе продолжать выполнение, в то время как она ожидает завершения операций ввода-вывода или других длительных процессов. Это увеличивает общую эффективность, поскольку процессор может выполнять другие задачи во время ожидания.

Async I/O, Asynchronous Input and Output — асинхронный ввод/вывод подразумевает выполнение вычислений во время ожидания результата от ввода и вывода данных.

При использовании модуля asyncio программа сама принимает решение о том, когда ей нужно переключиться между задачами. Каждая задача взаимодействует с другими задачами, передавая им управление тогда, когда она к этому готова. Поэтому такая схема работы называется **«кооперативной многозадачностью»** (cooperative multitasking), так как каждая задача должна

взаимодействовать с другими, передавая им управление в момент, когда она уже не может сделать ничего полезного.

Сопрограммы — это специальные функции, помеченные ключевым словом async, которые могут быть приостановлены и возобновлены. Они позволяют использовать оператор await для ожидания результата другой сопрограммы или операции ввода-вывода, не блокируя при этом основной поток выполнения программы. Это позволяет программе эффективно переключаться между задачами, улучшая производительность и отзывчивость.

Цикл событий (event loop) управляет выполнением сопрограмм и обработкой I/O операций, позволяя задачам переходить из состояния ожидания в состояние готовности и наоборот. Имеется единственный **цикл событий**, который занимается управлением всеми задачами. Задачи могут пребывать в некотором количестве различных состояний, самыми важными из которых можно назвать состояние готовности (ready) и состояние ожидания (waiting). Цикл событий на каждой итерации проверяет, имеются ли задачи, пребывающие в состоянии ожидания, которые завершены и оказались в состоянии готовности. Затем цикл берёт задачу, находящуюся в состоянии готовности, и выполняет её до тех пор, пока она не завершится, либо — до тех пор, пока не окажется, что ей нужно дождаться завершения другой задачи.

Футуры (futures) же представляют собой объекты, которые обещают предоставить результат асинхронной операции в будущем.

В целом, асинхронное программирование в Python позволяет избежать блокировок и взаимоблокировок задач, увеличивая общую эффективность использования процессора. Программа сама решает, когда переключаться между задачами, что делает код конкурентным и кооперативным. Асинхронный ввод/вывод позволяет выполнять вычисления во время ожидания результата от ввода и вывода данных, что особенно полезно при решении задач, зависящих от подсистемы ввода/вывода, и обеспечивает потокобезопасность.

3.4.2. Практика

В рамках исследования была внедрена асинхронная обработка с использованием asyncio, что способствовало оптимизации управления операциями ввода-вывода. Асинхронные методы применялись для параллельного распознавания объектов на изображениях из видеопотока, что достигалось благодаря использованию ключевых слов async и await, а также функции asyncio.gather, обеспечивающей одновременное выполнение множества асинхронных задач.

Асинхронные операции инициировались в функции run_three_models, где с помощью await asyncio.gather(...) запускались множественные экземпляры функции detect_objects, каждый из которых независимо обрабатывал кадр и выявлял объекты.

Использование await перед asyncio.gather указывает на приостановку выполнения текущей функции — в данном случае run_three_models — до завершения всех переданных в asyncio.gather асинхронных задач. В бесконечном цикле while True происходит непрерывное получение кадров и их асинхронная обработка, что позволяет обрабатывать изображения по мере поступления, минуя ожидание завершения предыдущих операций.

Применение асинхронности в процессе реального времени для распознавания объектов способствует увеличению производительности и эффективности, так как позволяет выполнять несколько задач одновременно, сокращая время ожидания окончания длительных операций ввода-вывода. Это критически важно для систем, требующих быстрой обработки большого объема данных, как в случае с видеопотоком. Благодаря этому достигается значительное повышение частоты кадров (FPS), поскольку приложение может обрабатывать другие задачи, не ожидая окончания операций ввода-вывода.

4. Результаты исследования

В таблице 18 представлены результаты исследования, где сравниваются различные эффективные методы вычислений по их производительности (FPS) и процентному приросту по сравнению с базовой моделью. Как видно из таблицы, перевод модели на ГПУ (графический процессор) и использование асинхронных вычислений приводит к увеличению FPS и процентного прироста производительности на 38%.

Таблица 18. Результаты исследования

	Скорость обработки, FPS	Улучшение производительности, %
Исходная модель кода	10.585795500815214	0
Модель, оптимизированная для ГПУ	10.97882685224355	3,712818289
Модель с асинхронными вычислениями	14.180416196292715	33,95702000
Модель, оптимизированная для ГПУ с асинхронными вычислениями	14.628262387698413	38,18765332

5. Заключение

Исследование было направлено на повышение количества кадров в секунду (FPS) в приложении для контроля ручных операций на производстве. Целью было улучшение FPS за счет оптимизации вычислений, включая более эффективное распределение задач между графическим (GPU) и центральным (CPU) процессорами. Высокий FPS необходим для точного и быстрого отслеживания операций в реальном времени, что способствует улучшению качества контроля и безопасности трудовых процессов.

В ходе исследования вычислительная часть приложения была переведена на GPU с использованием библиотеки CUDA. Сравнение производительности различных моделей, выполняемых на CPU и двух GPU (NVIDIA GeForce RTX 3080 и NVIDIA GeForce GTX 1660), показало, что наилучший прирост FPS достигается при выполнении модели Palm Model на GPU NVIDIA GeForce RTX 3080, затем Drone Tools Model на GTX 1660 и Drone Details Model снова на RTX 3080, что дало прирост на 3,71%. Однако этот прирост оказался незначительным, что может быть связано с несоответствием вычислительных задач архитектуре GPU, особенно если задачи содержат множество последовательных операций и зависимостей.

Также были созданы три отдельных потока для обработки трех разных моделей: Palm Model, Drone Tools Model и Drone Details Model. Каждая модель обрабатывалась в отдельном потоке для параллелизации вычислений. Однако ограничение GIL в Python, которое позволяет выполнять байт-код только в одном потоке за раз, не позволяет реализовать параллелизм.

Были проведены эксперименты с многопроцессностью. В первом эксперименте один процесс выполнял все модели, а второй занимался захватом изображения и отрисовкой, но это не увеличило скорость работы. Во втором эксперименте задачи были распределены между тремя процессами, каждый из которых выполнял одну модель, в то время как основной процесс занимался захватом и отрисовкой. Однако, такая схема приводила к проблемам взаимной

блокировки, поскольку каждый процесс зависел от результатов, обработанных другими процессами.

Применение асинхронной обработки через asyncio значительно повысило эффективность работы. Асинхронные методы включали запуск нескольких задач по обнаружению объектов, каждая из которых независимо анализировала кадры. Благодаря асинхронности, система могла обрабатывать новые изображения параллельно, не дожидаясь окончания предыдущих задач.

В результате, использование асинхронных вычислений и перевод вычислений на графический процессор привело к увеличению FPS и процентного прироста производительности на 38%. Это подтверждает, что асинхронная обработка является наиболее эффективным методом для приложений, требующих быстрой обработки данных в реальном времени, и позволяет достичь высокого FPS, улучшая качество контроля и безопасность производственных процессов.

6. Резюме

Моя работа была сосредоточена на поиске эффективного метода увеличения скорости работы программы за счет оптимизации вычислений, включая более оптимальное распределение задач между графическим (GPU) и центральным (CPU) процессорами. Основные направления исследования представлены на слайде.

Первый метод:

Он заключался в перенаправлении вычислительной части приложения на GPU с использованием библиотеки CUDA, что дало прирост производительности на 3,71%. Этот прирост оказался незначительным, что, возможно, связано с несоответствием вычислительных задач архитектуре GPU, особенно если задачи содержат множество последовательных операций и зависимостей.

Второй метод:

Этот метод был направлен на применение многопоточных вычислений. Были созданы три отдельных потока для обработки трех различных моделей. Каждая модель обрабатывалась в отдельном потоке для параллелизации вычислений. Однако в Python существует GIL (Global Interpreter Lock) - механизм, который позволяет только одному потоку выполнять байт-код в любой момент времени, что не позволяет реализовать настоящий параллелизм.

Третий подход:

Были проведены эксперименты с многопроцессностью. Основное преимущество многопроцессной обработки перед многопоточностью заключается

в том, что процессы не ограничены GIL, так как каждый процесс имеет отдельное пространство памяти. В первом эксперименте один процесс выполнял все модели, а второй занимался захватом изображения и отрисовкой, но это не увеличило скорость работы, так как выполнение моделей в одном процессе оказалось самой ресурсоемкой задачей. Во втором эксперименте задачи были распределены между тремя процессами, каждый из которых выполнял одну модель, в то время как основной процесс занимался захватом и отрисовкой. Однако такая схема приводила к проблемам взаимной блокировки, поскольку каждый процесс зависел от результатов, обработанных другими процессами.

Четвертый вариант:

Были исследованы асинхронные методы работы программы. Этот подход включал запуск нескольких детекций, каждая из которых независимо анализировала кадры. Благодаря асинхронности система могла обрабатывать новые изображения, не дожидаясь окончания предыдущих задач.

Итог:

В результате, использование асинхронных вычислений и перевод вычислений на графический процессор привели к увеличению FPS и приросту производительности на 38%. Дополнительно было получено ещё 2% прироста за счет оптимизации вычислений для выполнения на GPU.