ЛЕКЦИЯ 3_3. ПАТТЕРНЫ ПАРАЛЛЕЛЬНОГО ПРОЕКТИРОВАНИЯ ПРИЛОЖЕНИЙ, ВЫПОЛНЕНИЕ ЧАСТИ ПРОГРАММ НА GPU.

Шаблон проектирования или **паттерн** (англ. *design pattern*) в разработке программного обеспечения - повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некотор-ого часто возникающего контекста.

«Низкоуровневые» шаблоны, учитывающие специфику конкретного языка программирования, называются идиомами.

На наивысшем уровне существуют **архитектурные шаблоны**, они охватывают собой архитектуру всей программной системы.

Типы шаблонов проектирования

1. **Основные**
2. **Частные**

Основные шаблоны (Fundamental)

- **Шаблон делегирования** (Delegation pattern) Объект внешне выражает некоторое поведение, но в реальности передаёт ответственность за выполнение этого поведения связанному объекту.

-**Шаблон функционального дизайна** (Functional design)
Гарантирует, что каждый модуль компьютерной программы имеет только одну обязанность и исполняет её с минимумом побочных эффектов на другие части программы.

-**Неизменяемый интерфейс** (Immutable interface)
Создание неизменяемого объекта.

-**Интерфейс** (Interface)
Общий метод для структурирования компьютерных программ для того, чтобы их было проще понять.

-**Интерфейс-маркер** (Marker interface)
В качестве атрибута (как пометки объектной сущности) применяется наличие или отсутствие реализации интерфейса-маркера. В современных языках программирования вместо этого могут применяться атрибуты или аннотации.

-**Контейнер свойств** (Property container)
Позволяет добавлять дополнительные свойст-ва для класса в контейнер (внутри класса), вме-сто расширения класса новыми свойствами.

-**Канал событий** (Event channel)
Расширяет шаблон Publish/Subscribe, создавая централизованный канал для событий.
Использует объект-представитель для подпис-ки и объект-представитель для публикации со-бытия в канале.

1. Порождающие шаблоны (Creational)

Шаблоны проектирования, которые абстрагируют процесс инстанцирования. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов. Шаблон, порождающий классы, использует наследование, чтобы изменять инстанцируемый класс, а шаблон, порождающий объекты, делегирует инстанцирование другому объекту.

-**Абстрактная фабрика** (Abstract factory)
Класс, который представляет собой интерфейс для создания компонентов системы.

-**Строитель** (Builder)
Класс, который представляет собой интерфейс для создания сложного объекта.

-**Фабричный метод** (Factory method)
Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать.

-**Отложенная инициализация** (Lazy initialization)
Объект, инициализируемый во время первого обращения к нему.
-**Мультитон** (Multiton)
Гарантирует, что класс имеет поименованные экземпляры объекта и обеспечивает глобальную точку доступа к ним.
-**Объектный пул** (Object pool)
Класс, который представляет собой интерфейс для работы с набором инициализированных и готовых к использованию объектов.

-**Прототип** (Prototype)
Определяет интерфейс создания объекта через клонирование другого объекта вместо создания через конструктор.

-**Получение ресурса есть инициализация** (Resource acquisition is initialization (RAII))
Получение некоторого ресурса совмещается с инициализацией, а освобождение — с уничтожением объекта.

-**Одиночка** (Singleton)
Класс, который может иметь только один экземпляр.

2. Структурные шаблоны (Structural)

определяют различные сложные структуры, которые изменяют интерфейс уже существующих объектов или его реализацию, позволяя облегчить разработку и оптимизировать программу.

-**Адаптер** (Adapter / Wrapper)
Объект, обеспечивающий взаимодействие двух других объектов, один из которых использует, а другой предоставляет несовместимый с первым интерфейс.

-**Мост** (Bridge)
Структура, позволяющая изменять интерфейс обращения и интерфейс реализации класса независимо.

-**Компоновщик** (Composite)
Объект, который объединяет в себе объекты, подобные ему самому.

-**Декоратор или Wrapper/Обёртка** (Decorator)
Класс, расширяющий функциональность другого класса без использования наследования.
-**Фасад** (Facade)
Объект, который абстрагирует работу с несколькими классами, объединяя их в единое целое.
-**Единая точка входа** (Front controller)
Обеспечивает унифицированный интерфейс для интерфейсов в подсистеме. Front Control-ler определяет высокоуровневый интерфейс, упрощающий использование подсистемы.

-**Приспособленец** (Flyweight)
Это объект, представляющий себя как уни-
кальный экземпляр в разных местах програм-
мы, но фактически не являющийся таковым.

-**Заместитель** (Proxy)
Объект, который является посредником между
двумя другими объектами, и который реали-
зует/ограничивает доступ к объекту, к которо-
му обращаются через него.

3. Поведенческие шаблоны (Behavioral)

Определяют взаимодействие между объек-
тами, увеличивая таким образом гибкость.

-**Цепочка обязанностей** (Chain of responsibility)
Предназначен для организации в системе уровней ответственности.

-Команда, Action, Transaction (Command)
Представляет действие. Объект команды заключает в себе само действие и его параметры.

-**Интерпретатор** (Interpreter)
Решает часто встречающуюся, но подверженную изменениям, задачу.

-Итератор, Cursor (Iterator)
Представляет собой объект, позволяющий получить последовательный доступ к элементам объекта-агрегата без использования описаний каждого из объектов, входящих в состав агрегации.

-Посредник (Mediator)
Обеспечивает взаимодействие множества объектов, формируя при этом слабую связанность и избавляя объекты от необходимости явно ссылаться друг на друга.

-Хранитель (Memento)
Позволяет не нарушая инкапсуляцию зафиксировать и сохранить внутренние состояния объекта так, чтобы позднее восстановить его в этих состояниях.
-Null Object (Null Object)
Предотвращает нулевые указатели, предоставляя объект «по умолчанию».
-Наблюдатель или Издатель-подписчик (Observer) Определяет зависимость типа «один ко многим» между объектами так, что при изменении состояния одного объекта все зависящие от него оповещаются об этом событии.

-**Слуга** (Servant)
Используется для обеспечения общей функциональности группе классов.

-Состояние (State)
Используется,когда во время выполнения программы объект должен менять своё поведение в зависимости от своего состояния.

-Стратегия (Strategy)
Предназначен для определения семейства алгоритмов, инкапсуляции каждого из них и обеспечения их взаимозаменяемости.

-**Шаблонный метод** (Template method)
Определяет основу алгоритма и позволяет наследникам переопределять некоторые шаги алгоритма, не изменяя его структуру в целом.
-**Посетитель** (Visitor)
Описывает операцию, которая выполняется над объектами других классов. При изменении класса Visitor нет необходимости изменять обслуживаемые классы.
-Спецификация (Specification)
Служит для связывания бизнес-логики.

Частные шаблоны параллельного программирования (Concurrency)

Concurrency — Параллелизм

Используются для более эффективного написания многопоточных программ, и предоставляет готовые решения проблем синхронизации

-**Active Object** (Active object)
Служит для отделения потока выполнения ме-тода от потока, в котором он был вызван. Использует шаблоны асинхронный вызов методов и планировщик

-**Balking** (Balking)
Служит для выполнения действия над объек-отом только тогда, когда тот находится в кор-ректном состоянии.

-**Обмен сообщениями** (Messaging pattern, Messaging design pattern (MDP))
Позволяет компонентам и приложениям обмениваться информацией (сообщениями).

-**Блокировка с двойной проверкой** (Double checked locking)

Предназначен для уменьшения накладных расходов, связанных с получением блокировки.

-**Блокировка** (Lock)

Один поток блокирует ресурс для предотвращения доступа или изменения его другими потоками.

-**Монитор** (Monitor)

Объект, предназначенный для безопасного использования более чем одним потоком.

-Reactor (Reactor)
Предназначен для синхронной передачи запросов сервису от одного или нескольких источников.
-Read/write lock (Read/write lock)
Позволяет нескольким потокам одновременно считывать информацию из общего хранилища, но позволяя только одному потоку в текущий момент времени её изменять.
-Планировщик (Scheduler)
Обеспечивает механизм реализации политики планирования, но при этом не зависящих ни от одной конкретной политики.

-**Однопоточное выполнение** (Single thread execution)
Препятствует конкурентному вызову метода, тем самым запрещая параллельное выполнение этого метода.

-**Кооперативный паттерн** (Cooperative pattern)
Обеспечивает механизм безопасной остановки потоков исполнения, используя общий флаг для сигнализирования прекращения работы потоков.

Другие типы шаблонов

Также на сегодняшний день существует ряд других шаблонов:
-Carrier Rider Mapper описывают предоставление доступа к хранимой информации.

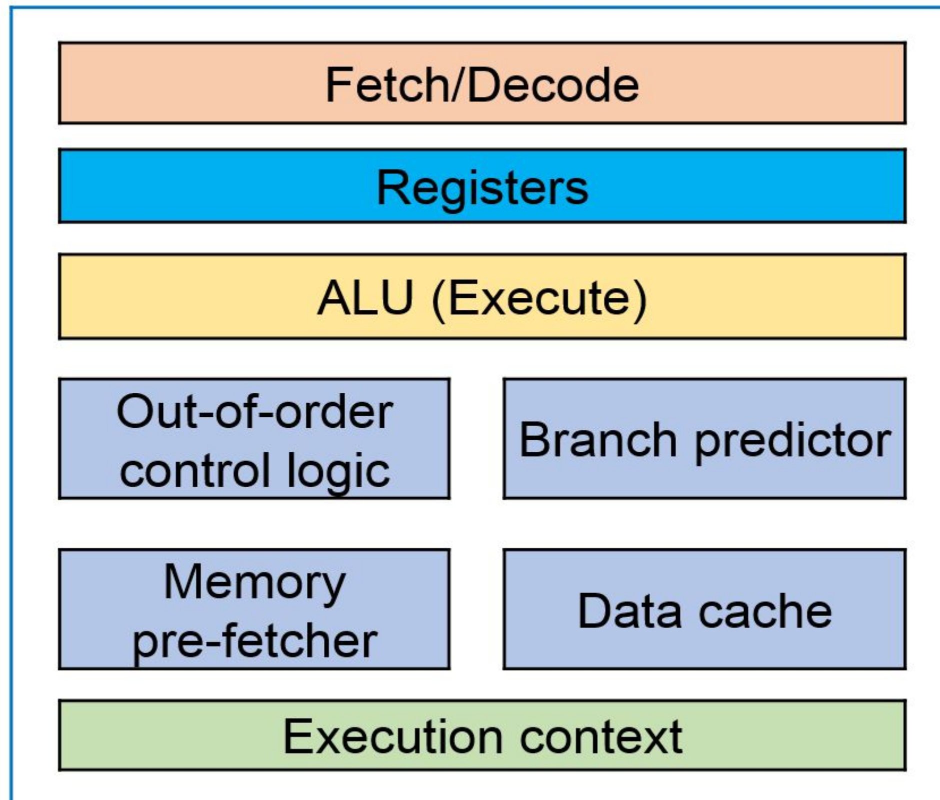
-Аналитические шаблоны описывают основной подход для составления требований для программного обеспечения (requirement analysis) до начала самого процесса программной разработки.

-Коммуникационные шаблоны описывают процесс общения между отдельными участниками/сотрудниками организации.

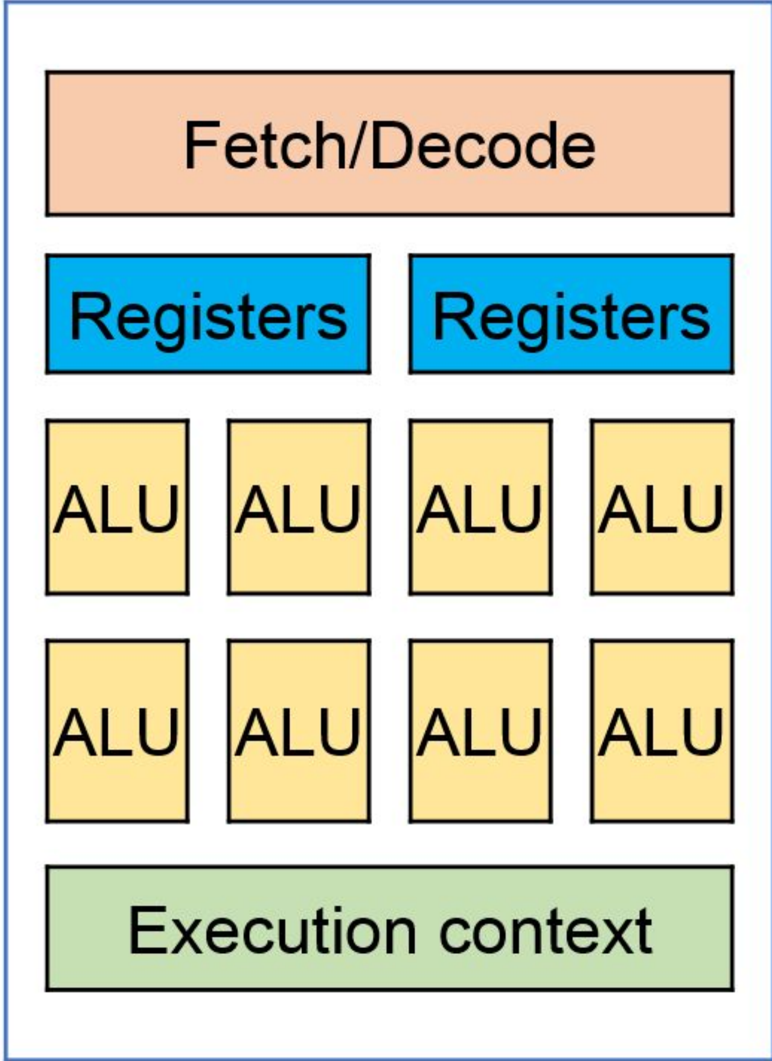
-Организационные шаблоны описывают организационную иерархию предприятия/фирмы

-Антипаттерны (Anti-Design-Patterns) описывают, как не следует поступать при разработке программ, показывая характерные ошибки в дизайне и в реализации.

Архитектура GPU и ее сравнение с CPU



CPU Core



GPU Core

Ограничения и возможности при работе с GPU

Ограничения:

-При выполнении расчетов на GPU, будет выделен целый блок ядер (32 для NVIDIA).

-Все ядра выполняют одни и те же инструкции, но с разными данными, такие вычисления называются Single-Instruction-Multiple-Data или SIMD (хотя NVIDIA вводит свое уточнение).

-GPU очень не любит ветвлений, да и в целом сложной логики в алгоритмах.

Возможности:

Собственно, ускорение тех самых SIMD-вычислений. Простейшим примером может служить поэлементное сложение матриц.

Наиболее распространены две технологии, которые можно использовать для программирования под GPU:

OpenCL
CUDA

OpenCL – это стандарт, который поддерживают большинство производителей видеокарт, в т.ч. и на мобильных устройствах, также код, написанный на **OpenCL**, можно запускать на CPU.

CUDA – это проприетарная технология и SDK от компании NVIDIA. Писать можно на C/C++ или использовать биндинги к другим языкам.

Сравнивать OpenCL и CUDA несколько не корректно, т.к. одно — стандарт, второе — целое SDK. Тем не менее многие выбирают CUDA для разработки под видеокарты несмотря на то, что технология проприетарная, хоть и бесплатная и работает только на картах NVIDIA.

Тому есть несколько причин:

-Более продвинутое API;
-Проще синтаксис и инициализация карты;
-Подпрограмма, выполняемая на GPU, является частью исходных текстов основной (host) программы;
-Собственный профайлер, в т.ч. и визуальный;
-Большое количество готовых библиотек;
-Более живое комьюнити.

К особенностям стоит отнести то, что **CUDA** поставляется с собственным компилятором, который так же может скомпилировать стандартный C/C++ код.

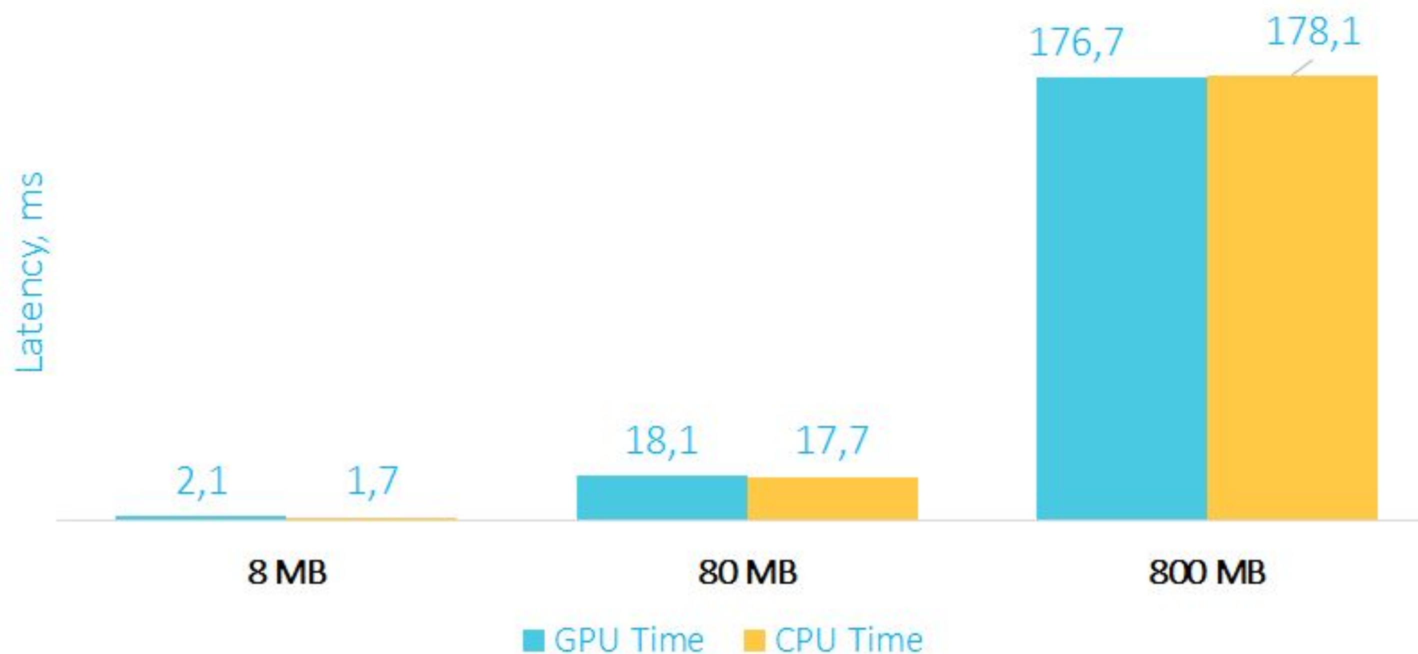
Результаты выполнения алгоритмов на GPU

Для тестирования GPU взят инстанс в AWS с видеокартой Tesla k80, это далеко не самая мощная серверная карта на сегодняшний день, но наиболее доступная и имеет:

4992 CUDA ядра
24 GB памяти
480 Gb/s — пропускная способность памяти

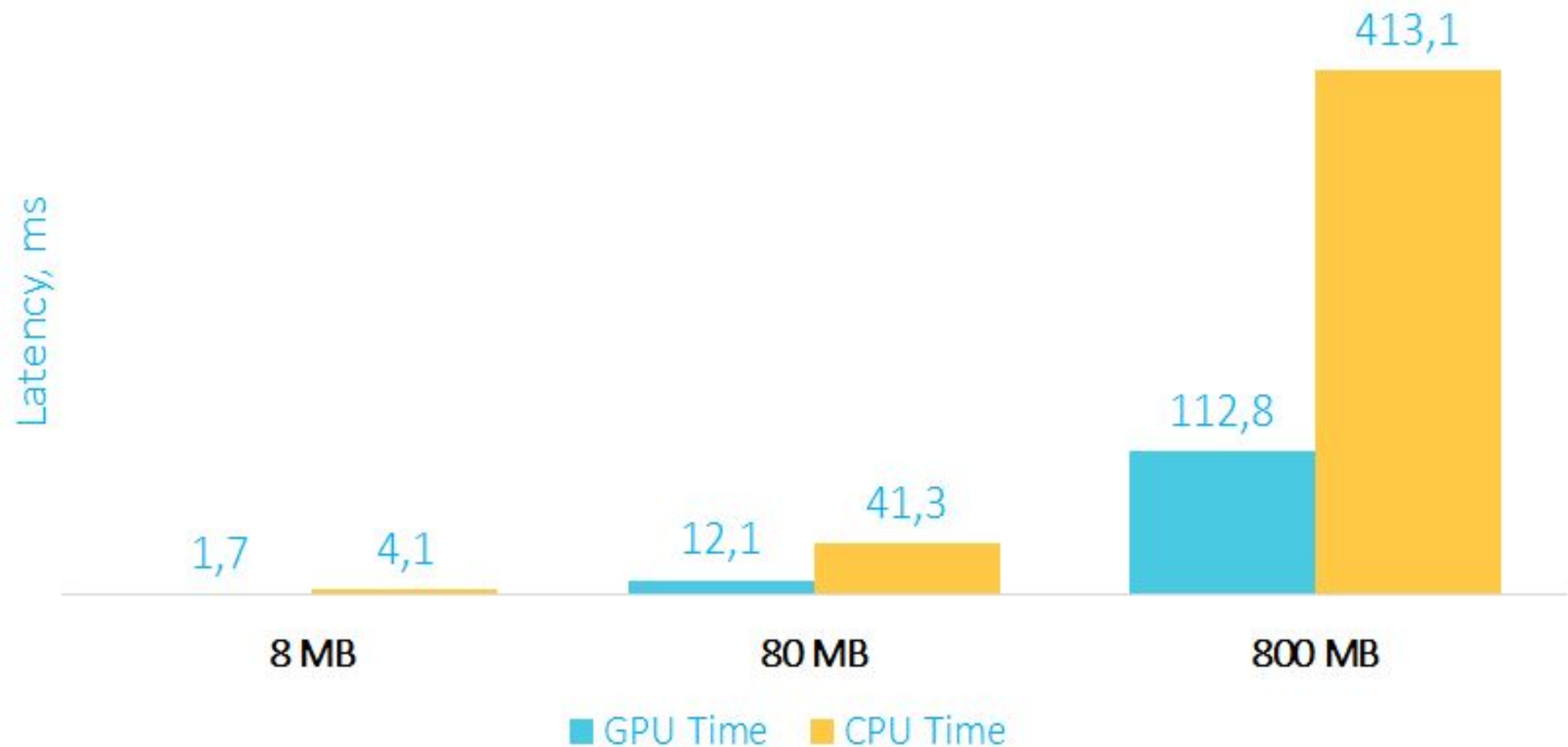
И для тестов на CPU взят инстанс с процессором Intel Xeon CPU E5-2686 v4 @ 2.30GHz

Трансформация



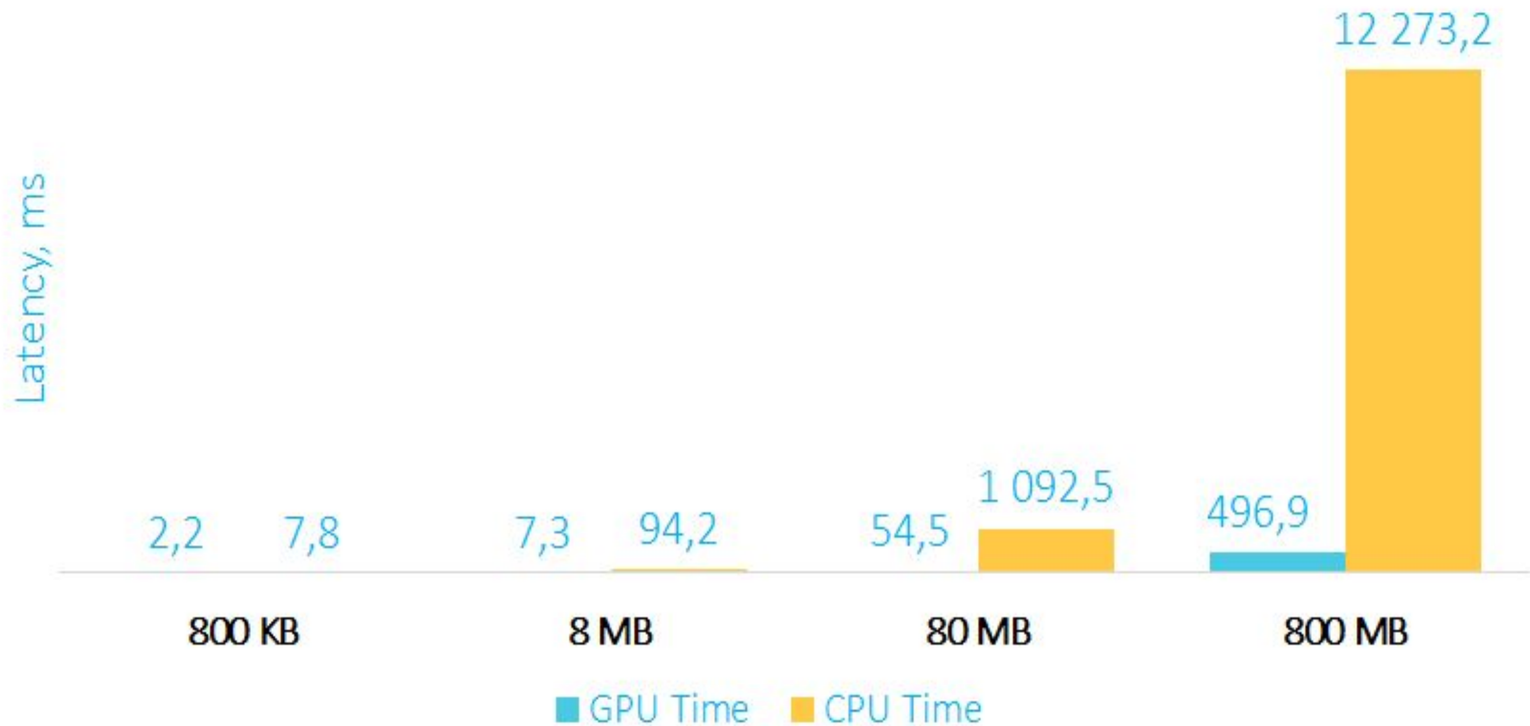
Время выполнения трансформации на GPU и CPU в мс

Агрегация



Время выполнения агрегации на GPU и CPU в мс

Сортировка



Время выполнения сортировки на GPU и CPU в мс

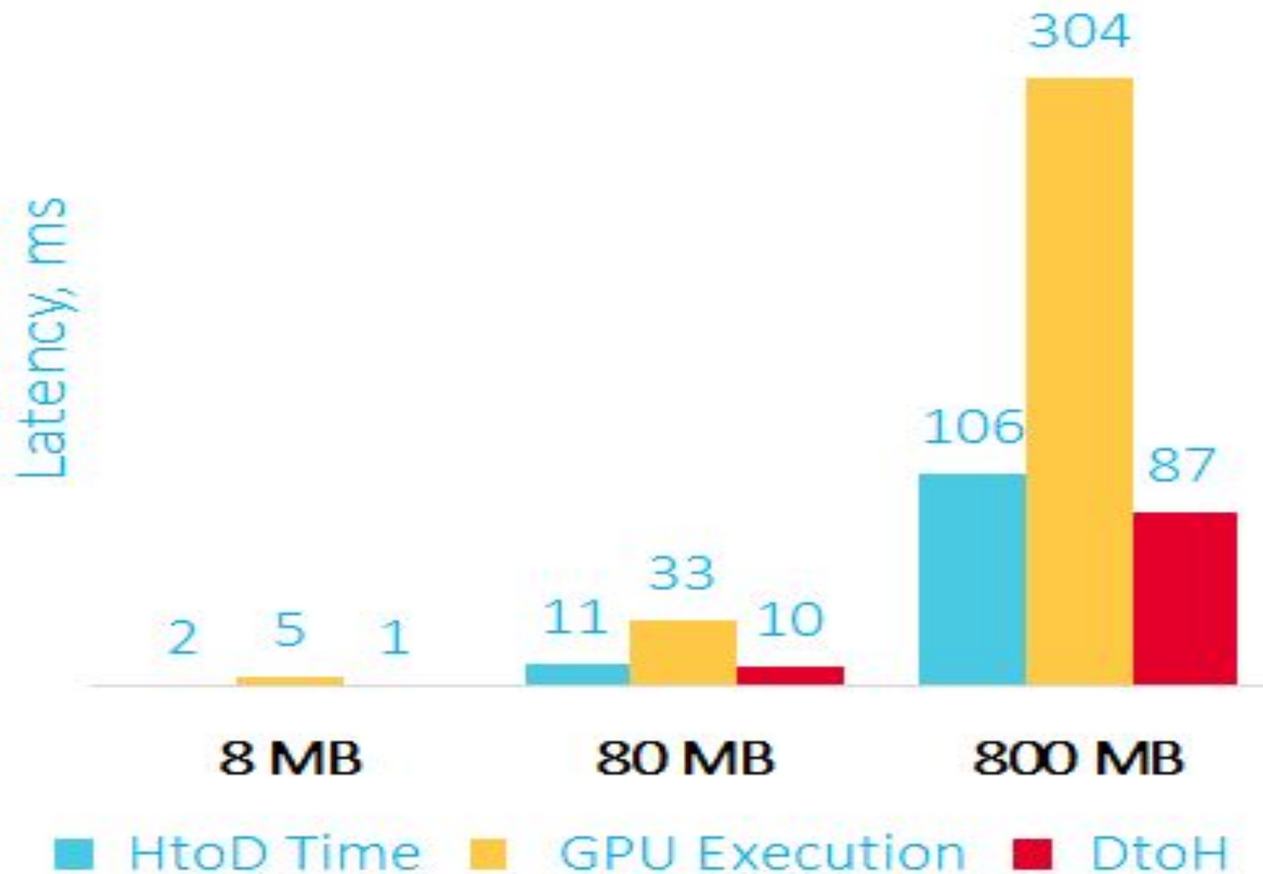
Оверхед на пересылку данных

Время пересылки данных на GPU, сортировки и пересылки данных обратно в RAM в мс

HtoD – передаем данные на видеокарту
GPU Execution – сортировка на видеокарте
DtoH – копирование данных из видеокарты в оперативную память

Первое, что можно отметить – считывать данные из видеокарты получается быстрее, чем записывать их туда.
Второе – при работе с видеокартой можно получить latency от 350 микросекунд, а этого уже может хватить для некоторых low latency приложений.

Оверхед для большого объема данных



Время пересылки данных на GPU, сортировки и пересылки данных обратно в RAM в мс

Серверное использование

Игровые

NVIDIA GeForce RTX 2080 TI

Серверные

NVIDIA Tesla k80

Пример игровой и серверной видеокарт

Основные отличия серверной (NVIDIA) и игровой карты:

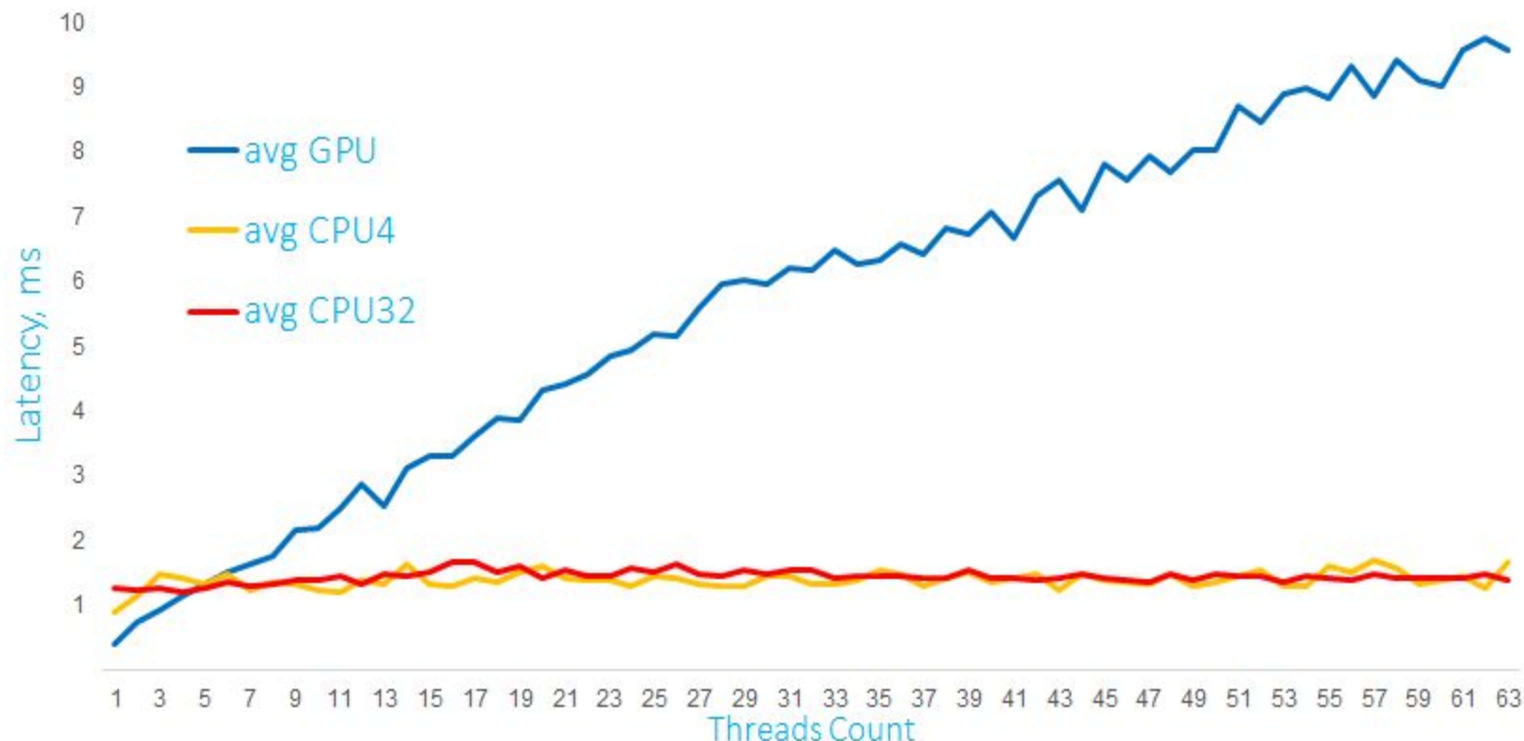
-Гарантия производителя (игровая карта не рассчитана на серверное использование).

-Возможные проблемы с виртуализацией для потребительской видеокарты.

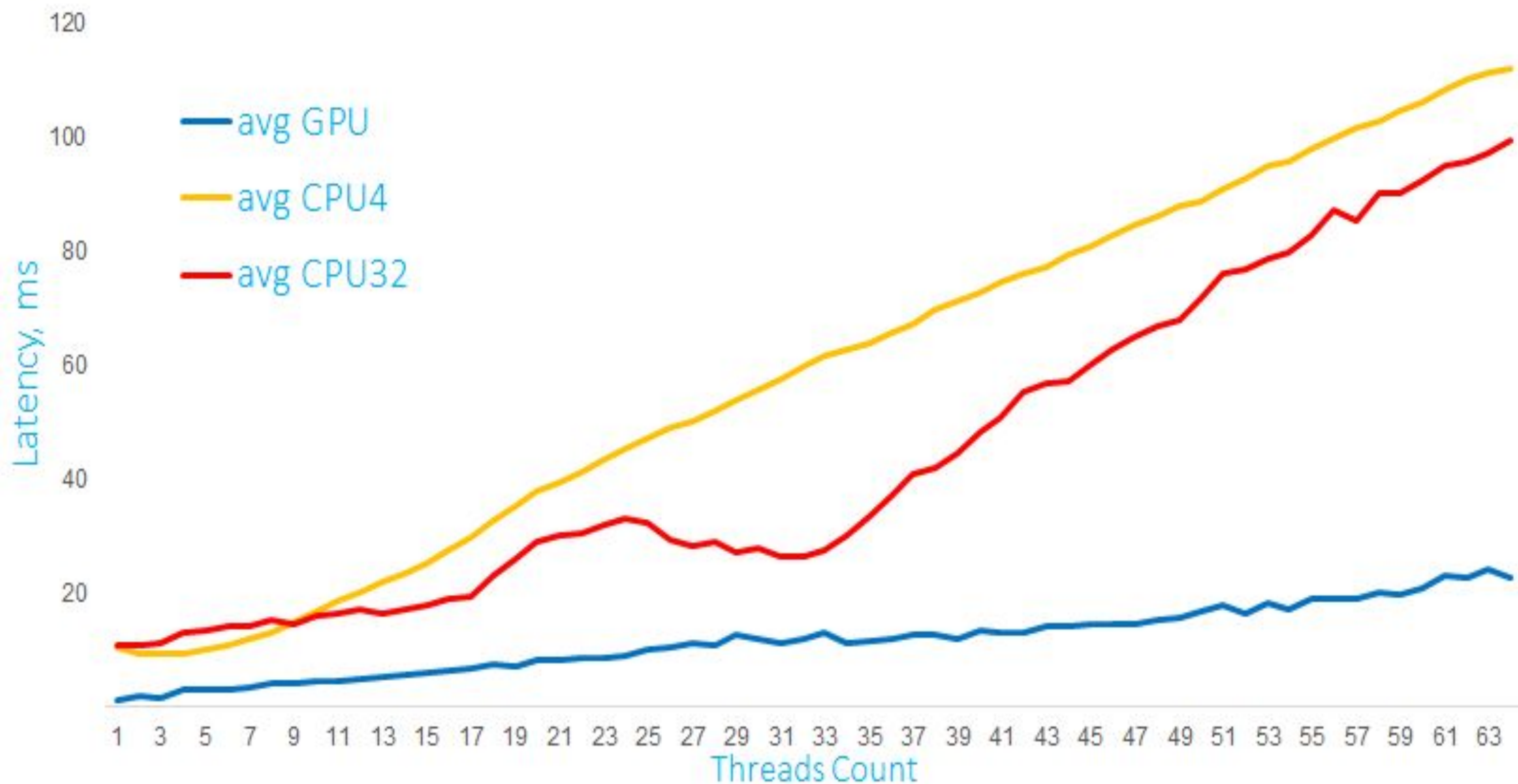
-Наличие механизма коррекции ошибок на серверной карте.

-Количество параллельных потоков (не CUDA ядер) или поддержка Hyper-Q, которая позволяет из нескольких потоков на CPU работать с картой, например, из одного потока закачивать данные на карту, а из другого запускать вычисления.

Многопоточность



Время выполнения математических расчетов на GPU и CPU с матрицами размером 1000 x 60 в мс



Время выполнения математических расчетов на GPU и CPU с матрицами 10 000 x 60 в мс

Ограничение ресурсов

Как мы уже говорили, два основных ресурса видеокарты – это вычислительные ядра и память.

Поэтому, если планируете использовать GPU в своих проектах, стоит рассчитывать на то, что приложение будет использовать видеокарту монопольно, либо вы будете программно контролировать объем выделяемой памяти и количество ядер, используемых для вычислений

Контейнеры и GPU

А если в сервере несколько видеокарт?

Опять же, можно на уровне приложения ре-шать, какой GPU оно будет использовать. Другой способ – это Docker-контейнеры. Можно использовать и обычные контейнеры, но NVIDIA предлагает свои контейнеры NGC, с оптимизированными версиями различного софта, библиотек и драйверов.Оверхед на использования контейнера около 3%.

Работа в кластере

Другой вопрос, что делать, если вы хотите выполнять одну задачу на нескольких GPU в рамках одного сервера или кластера?

Если вы выбрали библиотеку на подобии thrust или более низкоуровневое решение, то задачу придется решать вручную. Высокоуровневые фреймворки, например, для машинного обуче-ния или нейронных сетей, обычно поддержи-вают возможность использования нескольких карт из коробки.

Рекомендации

Если вы размышляете об использовании GPU в своих проектах, то GPU, скорее всего, вам подойдет если:

-Вашу задачу можно привести к SIMD-виду
-Есть возможность загрузить большую часть данных на карту до вычислений (закеширо-вать)
-Задача подразумевает интенсивные вычисле-ния

Заранее также стоит задаться вопросами:

-Сколько будет параллельных запросов;

-На какое latency вы рассчитываете;

-Достаточно ли вам одной карты для вашей нагрузки, нужен сервер с несколькими картами или кластер GPU-серверов.

Спасибо за внимание!