

Рисунок 1.2 – Схема *CUDA*-ядра

*CUDA*-ядра имеют блоки общей памяти, но все ядра способны выполнять лишь одну инструкцию. Таким образом, графические сопроцессоры *CUDA*-ядер архитектурно спроектированы для обработки массивов данных одинаковыми операциями.

Используемый в данной работе сопроцессор *NVidia GeForce GTX 1050ti* использует архитектуру *Pascal*. В ней *CUDA*-ядра объединены в блоки обработки (*Processing blocks*). Пара из блоков обработки составляет поточный мультипроцессор (*Streaming Multiprocessor*). В каждом мультипроцессоре есть области памяти для инструкций и так называемая *Shared Memory* – область памяти, к которой могут обратиться все ядра данного стримингового мультипроцессора [4].

Пара из стриминговых мультипроцессоров образует кластер обработки текстур (*Texture Processing Cluster*), в то время как несколько *TPC* образуют кластер обработки графики (*Graphic Processing Unit*). В архитектуре *Pascal* на один кластер обработки текстур приходится 128 *CUDA*-ядер [5].

В графическом сопроцессоре сопроцессор *NVidia GeForce GTX 1050ti* находится два кластера обработки графики, и каждый состоит из трех кластеров обработки текстур. Таким образом, при 128 *CUDA*-ядрах в одном кластере обработки текстур, данная видеокарта обладает 768 *CUDA*-ядрами. Схема графического сопроцессора приведена на рисунке 1.3 [6].

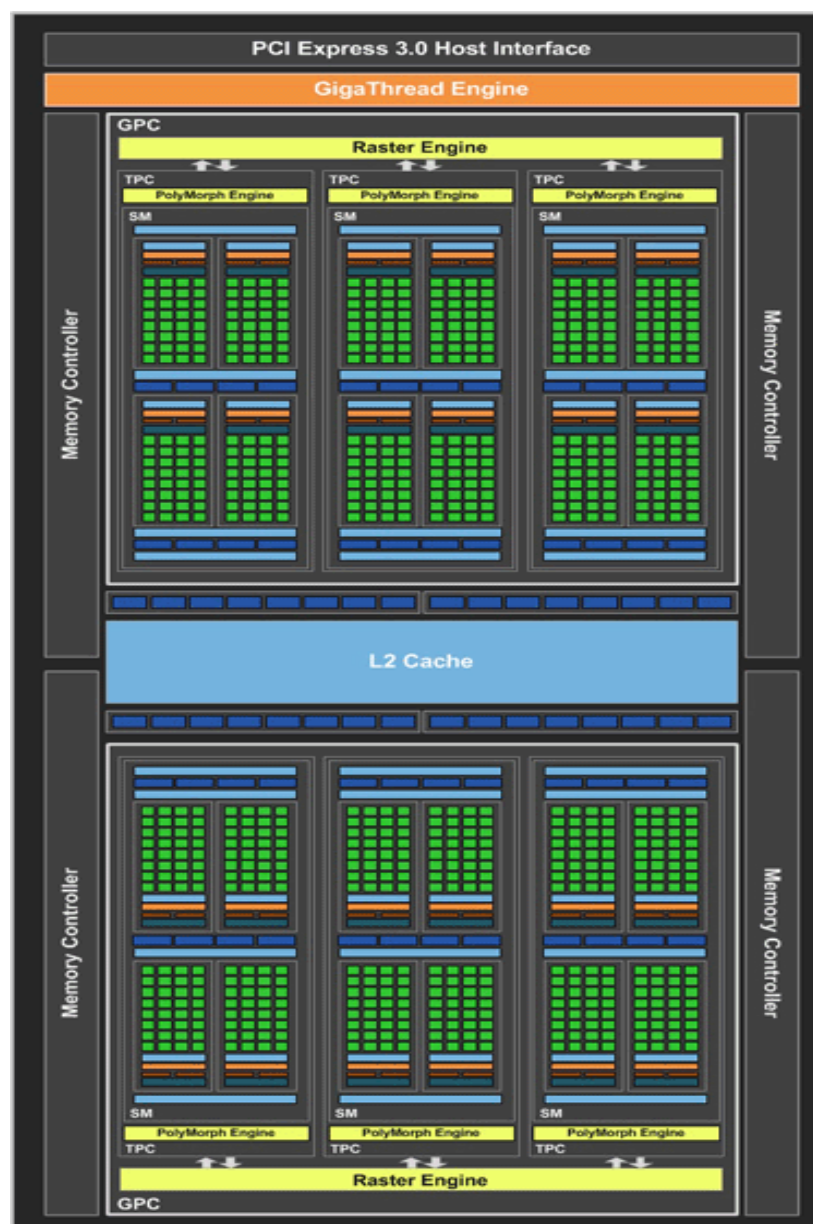


Рисунок 1.3 – Архитектура *Pascal* в видеокарте *NVidia GeForce GTX 1050ti*

**Обоснование выбора вычислительной системы:** данные аппаратные платформы были выбраны для сравнения так как они принципиально различны: процессор от *intel* спроектирован для выполнения максимально возможного числа задач с максимальной скоростью и не большим числом исполняемых на процессоре потоков, в то время как архитектура сопроцессоров с *CUDA*-ядрами подразумевает использование этих сопроцессоров для параллельной однотипной обработки больших объемов данных.

## 2 ПЛАТФОРМА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

**Инструменты разработки на платформе CUDA:** *CUDA* – платформа параллельных вычислений на графических процессорах, разработанная компанией *NVIDIA*. Она позволяет использовать возможности *GPU* для ускорения вычислений и повышения производительности различных задач, таких как научные исследования, анализ данных, глубокое обучение и другие. Основным элементом платформы – язык программирования *CUDA C/C++*, который является расширением стандартного языка *C/C++*. Он позволяет разрабатывать высокопроизводительные приложения, используя специальные конструкции, которые выполняются параллельно на множестве вычислительных потоков на графическом процессоре.

*CUDA* поддерживает различные типы *GPU*, начиная с архитектуры *Tesla* и вплоть до современных моделей. На платформе также доступны различные инструменты разработки, включая компиляторы, отладчики, профилировщики и библиотеки, которые упрощают разработку и оптимизацию приложений, весь этот комплекс объединяется под названием *CUDA Toolkit*.

Как можно видеть на рисунке 2.1, *CUDA* обеспечивает два *API*:

- высокоуровневый *API*: *CUDA Runtime API*;
- низкоуровневый *API*: *CUDA Driver API*.

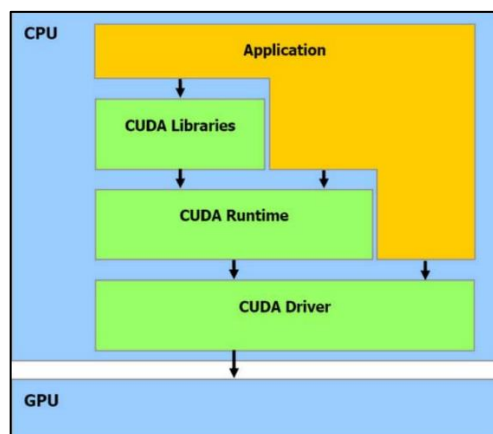


Рисунок 2.1 – схема программной модели *CUDA*

Поскольку высокоуровневый *API* реализован над низкоуровневым, каждый вызов функции уровня *Runtime* разбивается на более простые инструкции, которые обрабатывает *Driver API*. Необходимо обратить внимание, что два *API* взаимно исключают друг друга: программист может использовать один или другой *API*, но смешивать вызовы функций двух *API* не получится. Вообще, термин «высокоуровневый *API*» относителен. Даже

*Runtime API* таков, что многие сочтут его низкоуровневым; впрочем, он всё же предоставляет функции, весьма удобные для инициализации или управления контекстом.

С *Driver API* работать ещё сложнее; для запуска обработки на *GPU* потребуется больше усилий. С другой стороны, низкоуровневый *API* более гибок, при необходимости предоставляя программисту дополнительный контроль.

Библиотека *CUDA Math Library* – это проверенный высокоточный набор стандартных математических функций. Доступная для любого приложения *CUDA C* или *CUDA C++*, просто добавив «*#include math.h*» в исходный код, библиотека *CUDA Math Library* гарантирует, что ваше приложение получит преимущества от высокопроизводительных математических процедур, оптимизированных для каждой архитектуры графического процессора *NVIDIA* [7].

**Средство профилирования *NVIDIA Visual Profiler*:** *NVIDIA Visual Profiler* – это графический инструмент профилирования, который отображает хронологию загрузки *CPU* и *GPU* во время работы вашего приложения. Программа автоматически анализирует *GPU*-ядра и помогает определить возможности для оптимизации.

В *Visual Profiler* можно одновременно открыть несколько временных шкал на разных вкладках. На рисунке 2.2 показана шкала для приложения *CUDA*.

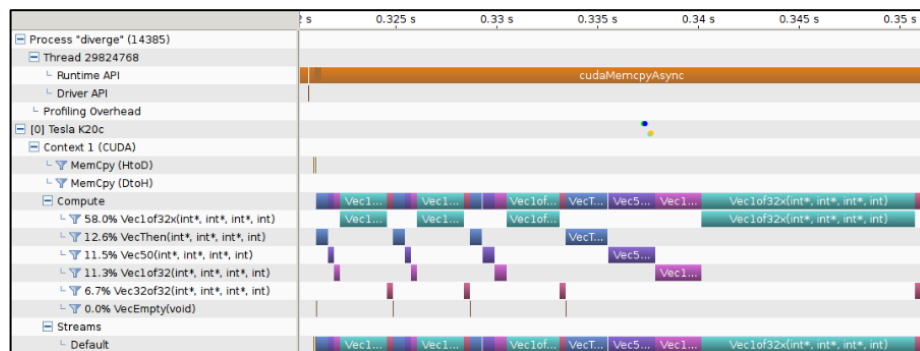


Рисунок 2.2 – Пользовательский интерфейс *NVIDIA Visual Profiler*

В верхней части находятся горизонтальные отметки времени, прошедшего с начала профилирования приложения. В левой части отображены единицы исполнения: процесс (*process*), потоки (*thread*), *GPU* (*device*), контексты (*context*), ядра (*kernel*), стримы (*stream*) и т.д. В центре показаны строки, отражающие активность отдельных элементов. Каждая строка отображает интервалы времени между началом и окончанием каких-

либо процессов. Например, строки напротив ядер показывают время начала и окончания выполнения этого ядра.

*Analysis view* отображает результаты анализа приложения. Доступны два режима: управляемый и неуправляемый. В управляемом режиме система проводит несколько этапов анализа, чтобы помочь понять слабые места в производительности и указать на возможности оптимизации приложения. В неуправляемом режиме можно самостоятельно запустить необходимые этапы и изучить их результаты. На рисунке 2.3 показан вид управляемого анализа. В левой части находятся пошаговые инструкции, которые помогут проанализировать и оптимизировать ваше приложение. Правая часть показывает подробные результаты и аналитическую инфографику.

Неуправляемый анализ содержит список доступных процессов, каждый из которых можно запустить вручную и увидеть результат.

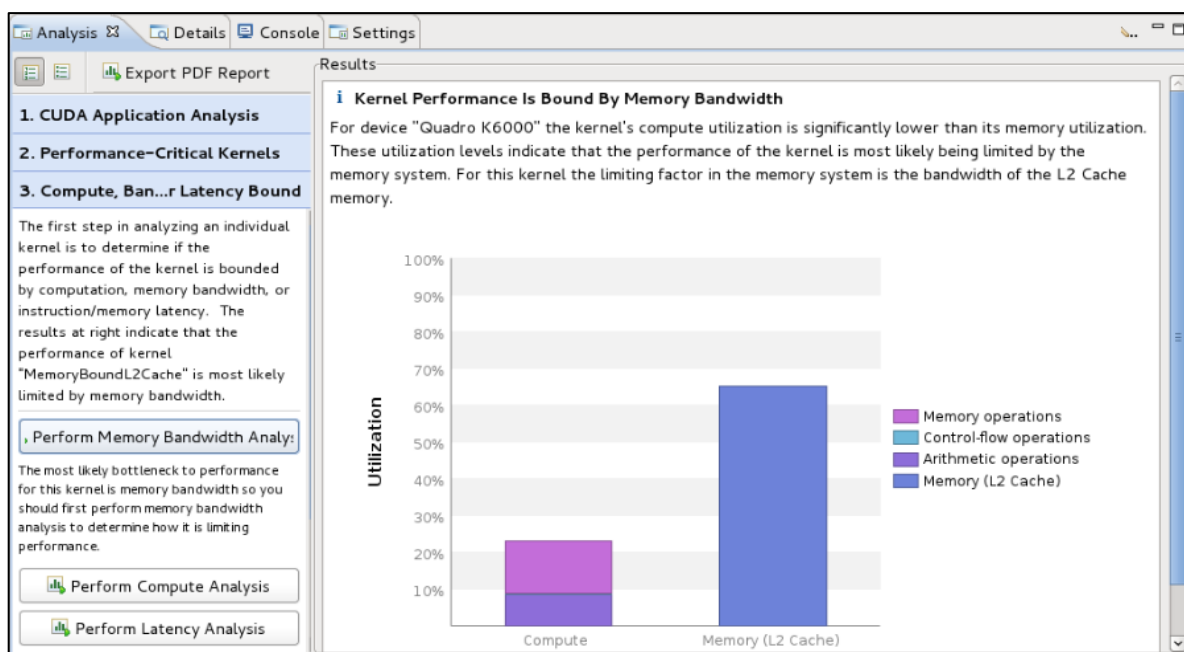
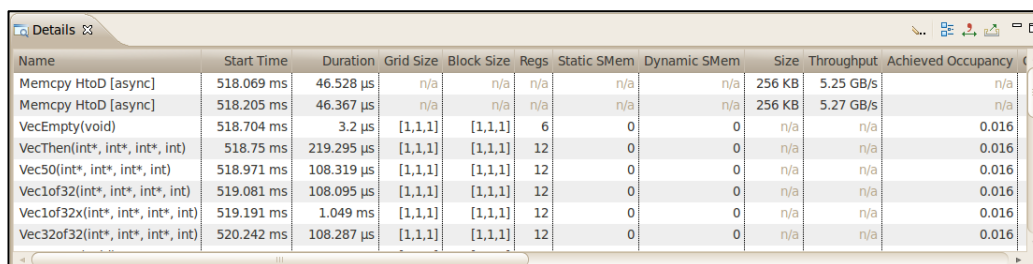


Рисунок 2.3 – Вид управляемого режима

*GPU Details View* показывает таблицу с информацией о каждом копировании памяти (*memcpy*) и запуске ядра (*kernel*) в профилируемом приложении. Для ядер в столбцах показаны соответствующие метрики и события (см. рисунок 2.4).



Name	Start Time	Duration	Grid Size	Block Size	Regs	Static SMem	Dynamic SMem	Size	Throughput	Achieved Occupancy
Memcpy HtoD [async]	518.069 ms	46.528 µs	n/a	n/a	n/a	n/a	n/a	256 KB	5.25 GB/s	n/a
Memcpy HtoD [async]	518.205 ms	46.367 µs	n/a	n/a	n/a	n/a	n/a	256 KB	5.27 GB/s	n/a
VecEmpty(void)	518.704 ms	3.2 µs	[1,1,1]	[1,1,1]	6	0	0	n/a	n/a	0.016
VecThen(int*, int*, int*, int)	518.75 ms	219.295 µs	[1,1,1]	[1,1,1]	12	0	0	n/a	n/a	0.016
Vec50(int*, int*, int*, int)	518.971 ms	108.319 µs	[1,1,1]	[1,1,1]	12	0	0	n/a	n/a	0.016
Vec1of32(int*, int*, int*, int)	519.081 ms	108.095 µs	[1,1,1]	[1,1,1]	12	0	0	n/a	n/a	0.016
Vec1of32x(int*, int*, int*, int)	519.191 ms	1.049 ms	[1,1,1]	[1,1,1]	12	0	0	n/a	n/a	0.016
Vec32of32(int*, int*, int*, int)	520.242 ms	108.287 µs	[1,1,1]	[1,1,1]	12	0	0	n/a	n/a	0.016

Рисунок 2.4 – Пользовательский интерфейс *GPU Details View*

**Операционная система *Ubuntu*:** *Ubuntu* – это операционная система, использующая ядро *Linux*. При выполнении программ на *CPU* эта операционная система дает ряд преимуществ: профилировщик производительности *htop*, позволяющий посмотреть нагрузку на процессор для каждого приложения, и малое количество фоновых процессов. Малое количество фоновых процессов системы позволяет демонстрировать более высокую и стабильную производительность, чем на операционных системах семейства *windows*.

***OpenMP*:** Для программирования многопоточных программ будет использоваться *OpenMP*. *OpenMP* – открытый стандарт для распараллеливания программ на языках *C*, *C++* и *Fortran*. Даёт описание совокупности директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с общей памятью. Для использования *OpenMP* необходимо добавить директивы в код программы, указывающие компилятору, какие участки кода можно выполнять параллельно. При компиляции с поддержкой *OpenMP*, компилятор автоматически генерирует код, который может выполняться параллельно на нескольких ядрах процессора. Это позволяет ускорить выполнение программы и повысить ее производительность [10].

Выбранные платформы программного обеспечения позволят изучить программы с точки зрения использования возможностей аппаратных платформ, благодаря чему можно будет сделать выводы об производительности программ, эффективности использования ресурсов и целесообразности их использования для решения практических задач.