Центр микро- и наномасштабной динамики дисперсных систем

Введение в CUDA

Марьин Д. Ф.

Уфа, 2011г.

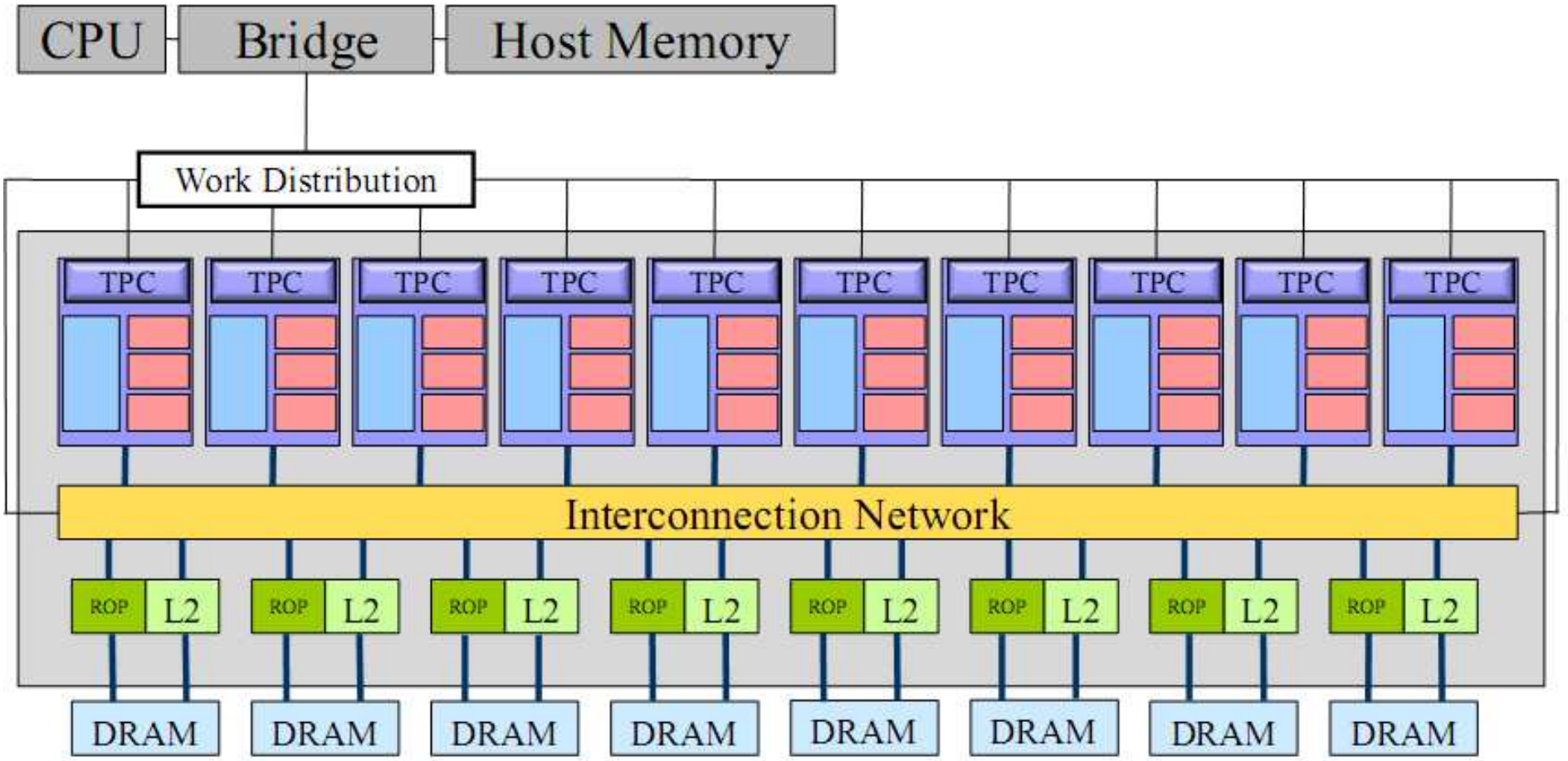
GPU — Graphics Processing Units.
GPGPU — General–Purpose computation on GPU (вычисления общего назначения на графических процессорах).
Средства разработки на GPU:

- Графические API (Direct3D, OpenGL) и шейдерные языки (GLSL, HLSL, Cg);
- Сторонние средства (BrookGPU, Sh, RapidMind);
- Средства от производителей:
  — NVIDIA CUDA (февраль 2007);
  — ATI Stream Technology (AMD FireStream) от AMD/ATI (ранее ATI FireStream и AMD Stream Processor)) (ноябрь 2007);
- OpenCL — Open Computing Langugage, открытый стандарт параллельных вычислений. Поддерживает широкий класс вычислительных устройств за счет введения обобщенных моделей. Разрабатывается Khronos Group. (ноябрь 2008)

CUDA — Compute Unified Design Architecture, технология от компании NVidia, предназначенная для разработки приложений для массивно-параллельных вычислительных устройств.
Общие положения:

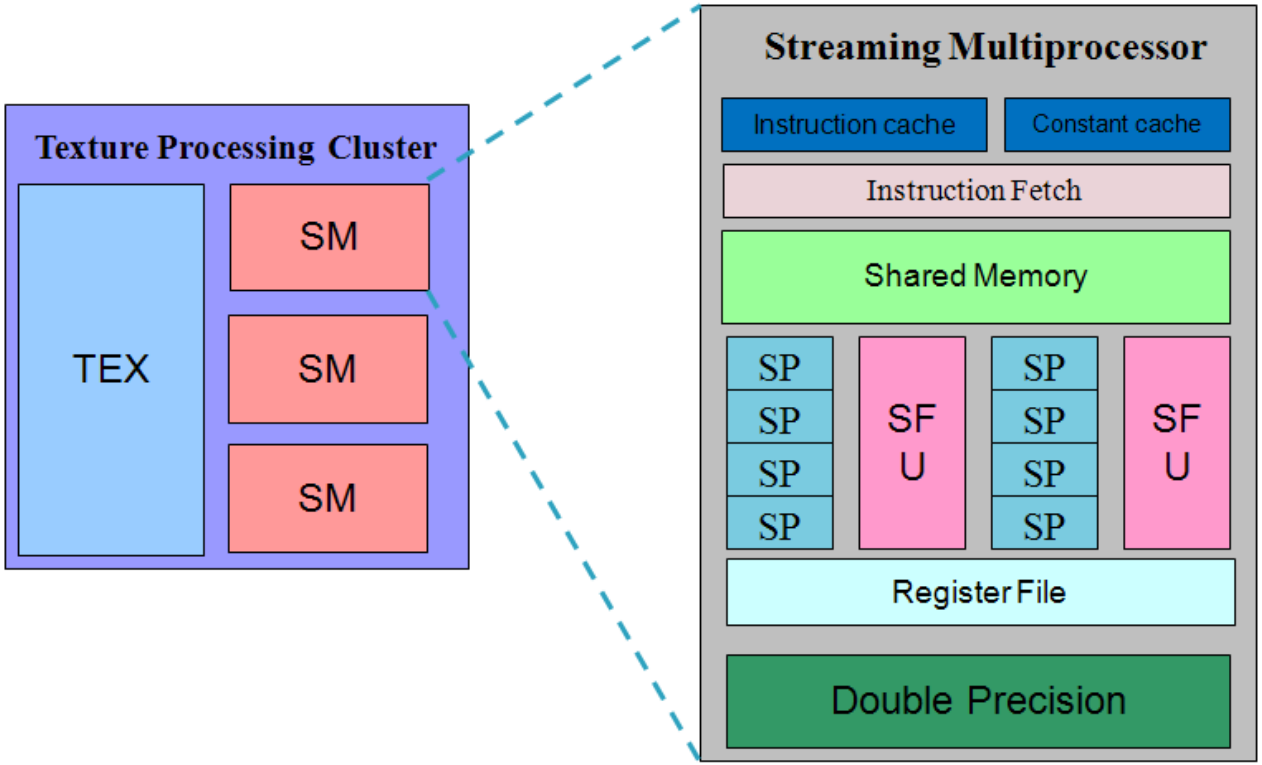
- GPU — сопроцессор для CPU (host);
- GPU обладает собственной памятью (DRAM);
- GPU обладает возможностью параллельного выполнения огромного количества отдельных нитей (threads);
- потоки GPU (в отличие от CPU) очень просты (обладают крайне "небольшой стоимостью") и многочисленны ($\sim$1000 для полной загрузки GPU);
- для осуществления расчётов при помощи GPU хост должен осуществить запуск вычислительного ядра, который определяет конфигурацию GPU в вычислениях и способ получения результатов (алгоритм).

Общая архитектура GPU Tesla 10 series



SPA (Streaming Processor Array) — набор TPC

Архитектура TPC (Texture Processing Cluster) Tesla 10

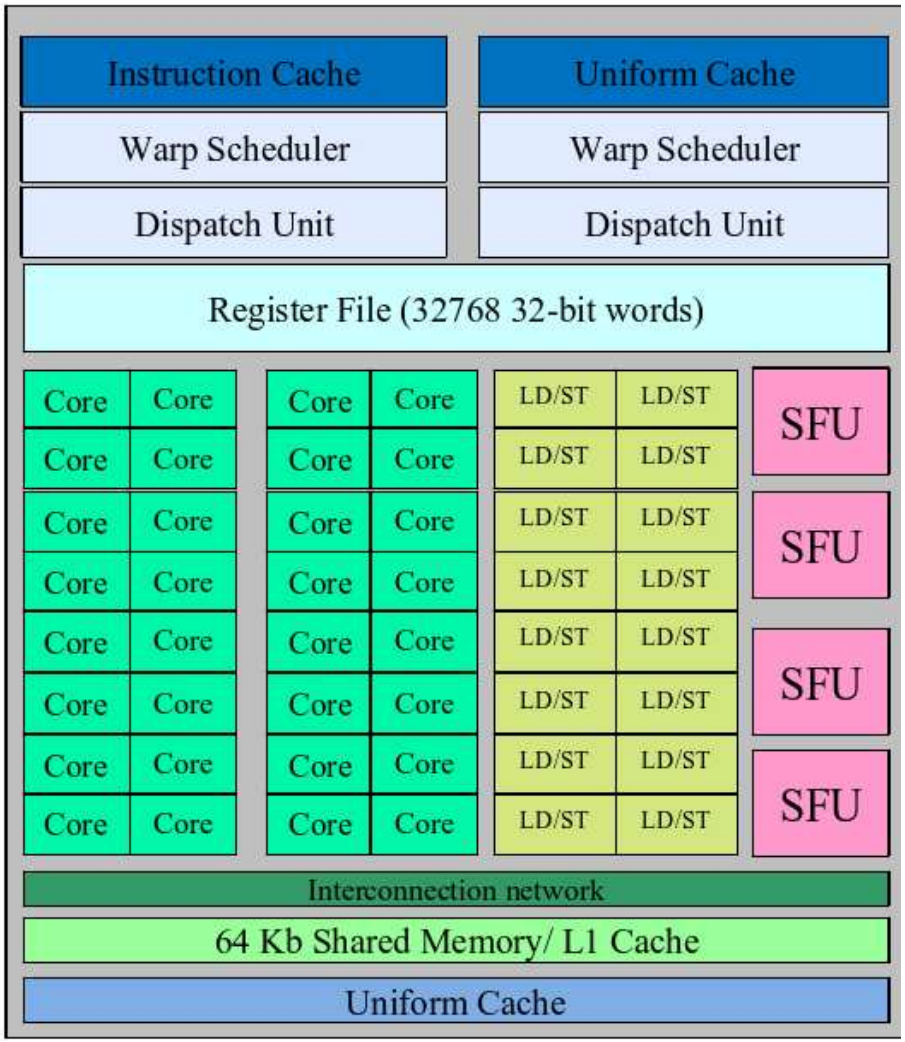


TEX — Texture block
SM — Streaming Multiprocessor
SP — Scalar Processor (CUDA–ядро)
SFU — Super Function Unit

Архитектура Tesla 20

- Объединенный L2 кэш (768 Kb)
- До 1 Tb памяти (64-битная адресация)
- Общее адресное пространство памяти
- Одновременное исполнение ядер, копирования памяти (CPU→GPU, GPU→CPU)
- Одновременное исполнение ядер (до 16)

Архитектура SM Tesla 20



- 32 ядра на SM
- Одновременное исполнение 2x варпов
- 48 Kb разделяемой памяти + 16 Kb кэш;
  или 16 Kb разделяемой + 48 Kb кэш

Compute capability служит для классификации устройств по качественным архитектурным особенностям. Определяется при помощи номеров главной (major) и второстепенной (minor) ревизий (архитектурных версий), записывается в виде major.minor.

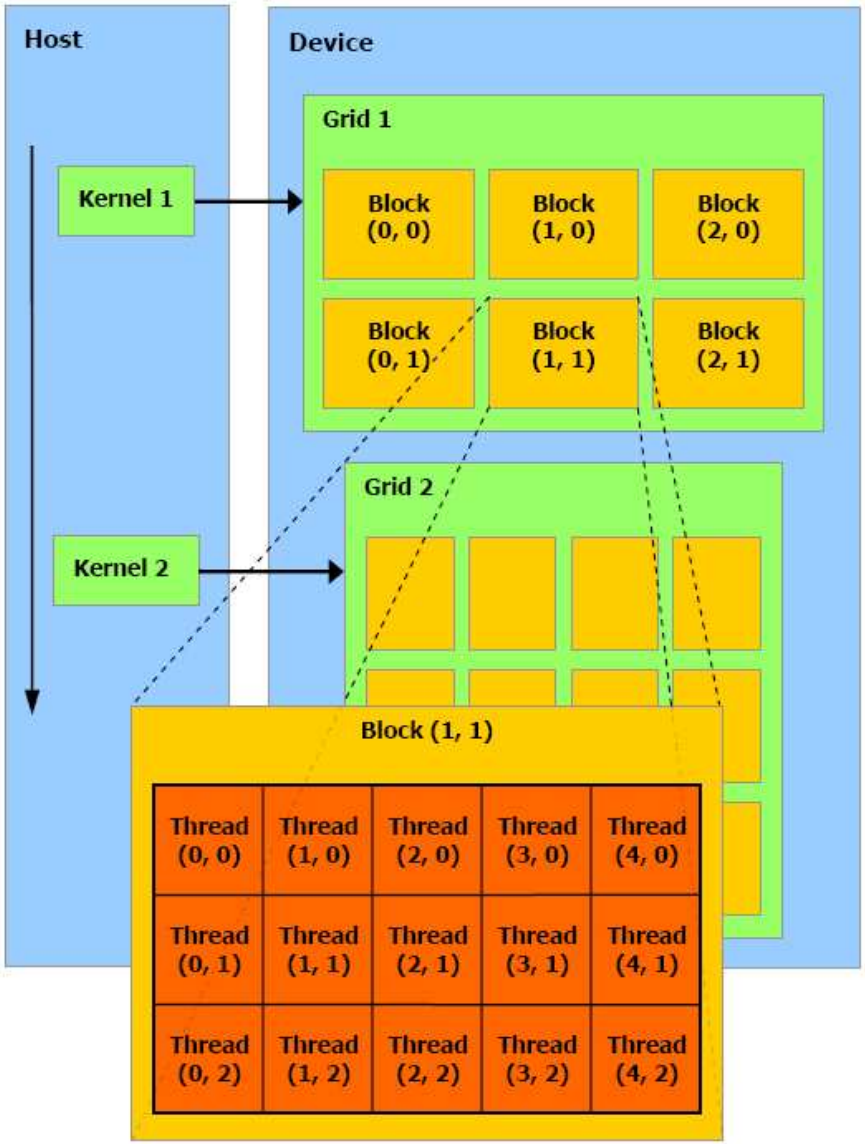
Поддержка вычислений с плавающей точкой в двойной точности, начиная с compute capability 1.3.

Поддержка классов (C++), начиная с compute capability 2.0.

Ключевые термины

Host — CPU — программа в обычной оперативной памяти компьютера, использующая CPU и выполняющая управляющие функции по работе с устройством.
Device — GPU — специализированное устройство, предназначенное для исполнения программ, использующих CUDA.
Kernel/Ядро — код, исполняемый на GPU.

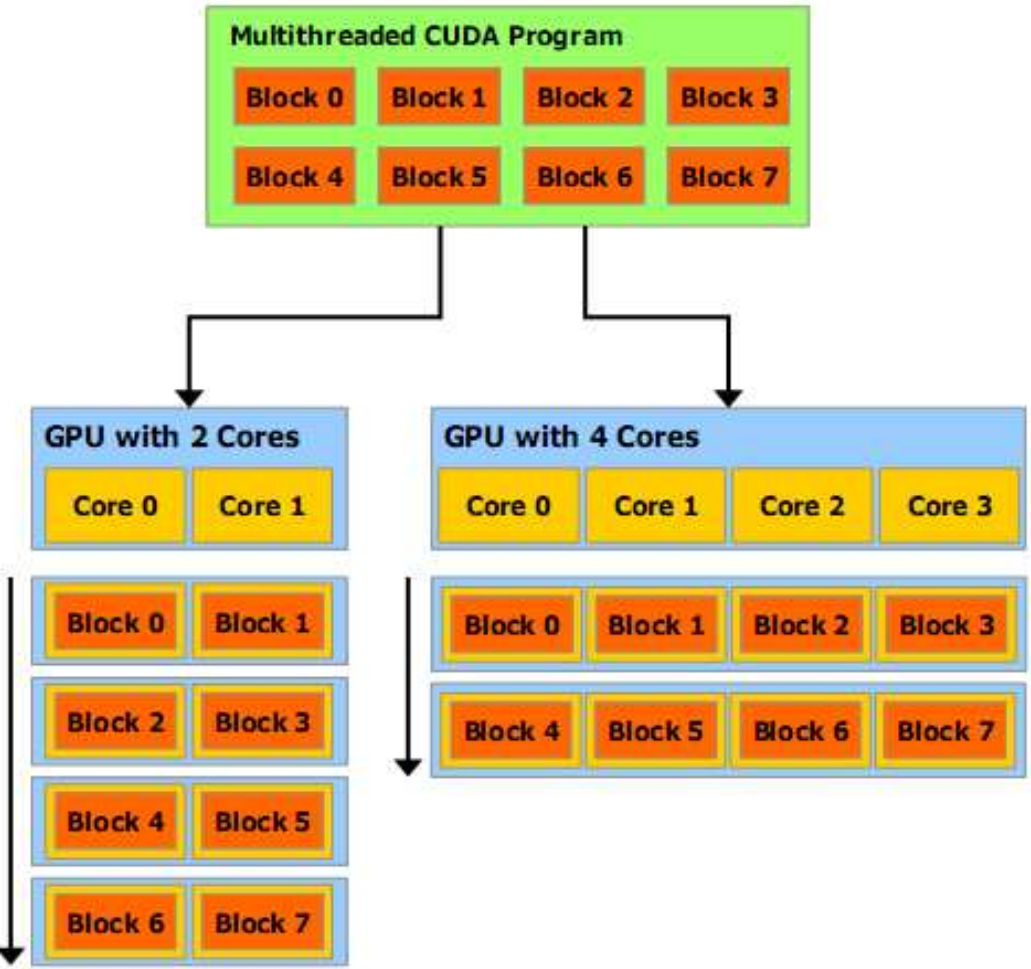
Thread/Поток/Нить — базовый набор данных, которые требуется обработать (отличается от понятия потока на CPU).
Warp/Свёртка — группа из 32 потоков, минимальный объём данных, обрабатываемый SIMD-способом (физически параллельно) в мультипроцессорах CUDA. Все нити warp'а выполняют одну и ту же команду, причем нити разных warp'ов могут выполнять разные команды.
Block — набор потоков, которые могут кооперироваться вместе для эффективного обмена данными через быструю разделяемую память и синхронизировать свое выполнение для координации доступа к памяти (исполняется на одном мультипроцессоре усройства).
Grid/Решетка — набор всех блоков.
Размеры block'а и grid'а задаются при вызове kernel'а.



Каждый поток и блок потоков имеют идентификаторы.
blockIdx — индекс *block*'а внутри *grid*'а (1D, 2D).
threadIdx — индекс *thread*'а внутри *block*'а (1D, 2D, 3D).
Многомерная индексация упрощает декомпозицию многомерных данных!

- Автоматическое распределение блоков на мультипроцессоры.
- Каждый блок целиком выполняется одним мультипроцессором.
- При наличии достаточного количества ресурсов, несколько блоков могут «одновременно» исполняться на одном мультипроцессоре.
  Но число активных блоков на мультипроцессор не может превышать восьми.

Масштабируемость



Наличие большого количества блоков открывает возможности для автоматической масштабируемости с ростом числа мультипроцессоров.

Взаимодействие потоков

Потоки внутри одного блока выполняются на одном мультипроцессоре, они способны взаимодействовать между собой посредством:

- разделяемой памяти (shared memory);
- точек синхронизации (как на стороне host'а, так и на device).

Два потока из различных блоков могут взаимодействовать лишь через глобальную память и точки синхронизации на стороне host'а.

- Все потоки, выполняющиеся на одном мультипроцессоре, группируются в warp'ы. В warp попадают потоки с последовательными идентификаторами.
  Число активных warp'ов на мультипроцессоре ограничено 32 (1024 потока). Поэтому рекомендуется использовать блоки по 128 или 256 потоков, чтобы достичь оптимального компромисса между снижением задержек и числом регистров для большинства ядер.
- Выполнение производится warp'ами. Аппаратный планировщик warp'ов каждый такт определяет warp, готовый к выполнению. Переключение от одного к другому происходит без потерь в тактах.
- Скалярные процессоры данного мультипроцессора параллельно выполняют одну и ту же инструкцию для одного warp'а (SIMT, Single Instruction Multiple Thread)

Примечание

В современных архитектурах количество скалярных процессоров внутри одного мультипроцессора равно 8, а не 32 (размер warp). Из этого следует, что не весь warp исполняется одновременно, он разбивается на 4 части (пул потоков warp'а), которые выполняются последовательно (т.к. процессоры скалярные). Тем самым, для того чтобы покрыть латентность доступа в память в 200 тактов, достаточно 50 warp'ов — таким образом, когда 50ый warp выполнит обращение, данные для первого warp'а уже будут готовы.

$50 \times 32 = 1600$

На GPU медленные обращения к памяти скрывают, используя параллельные вычисления. Пока одни задачи ждут данных, работают другие, готовые к вычислениям. Это один из основных принципов CUDA, позволяющих сильно поднять производительность системы в целом.

Ветвление потоков

A;
if (condition) B;
else C;
D;

В случае ветвлений внутри пула потоков warp'а выполнение сериализуется (это выполняется динамически драйвером CUDA).

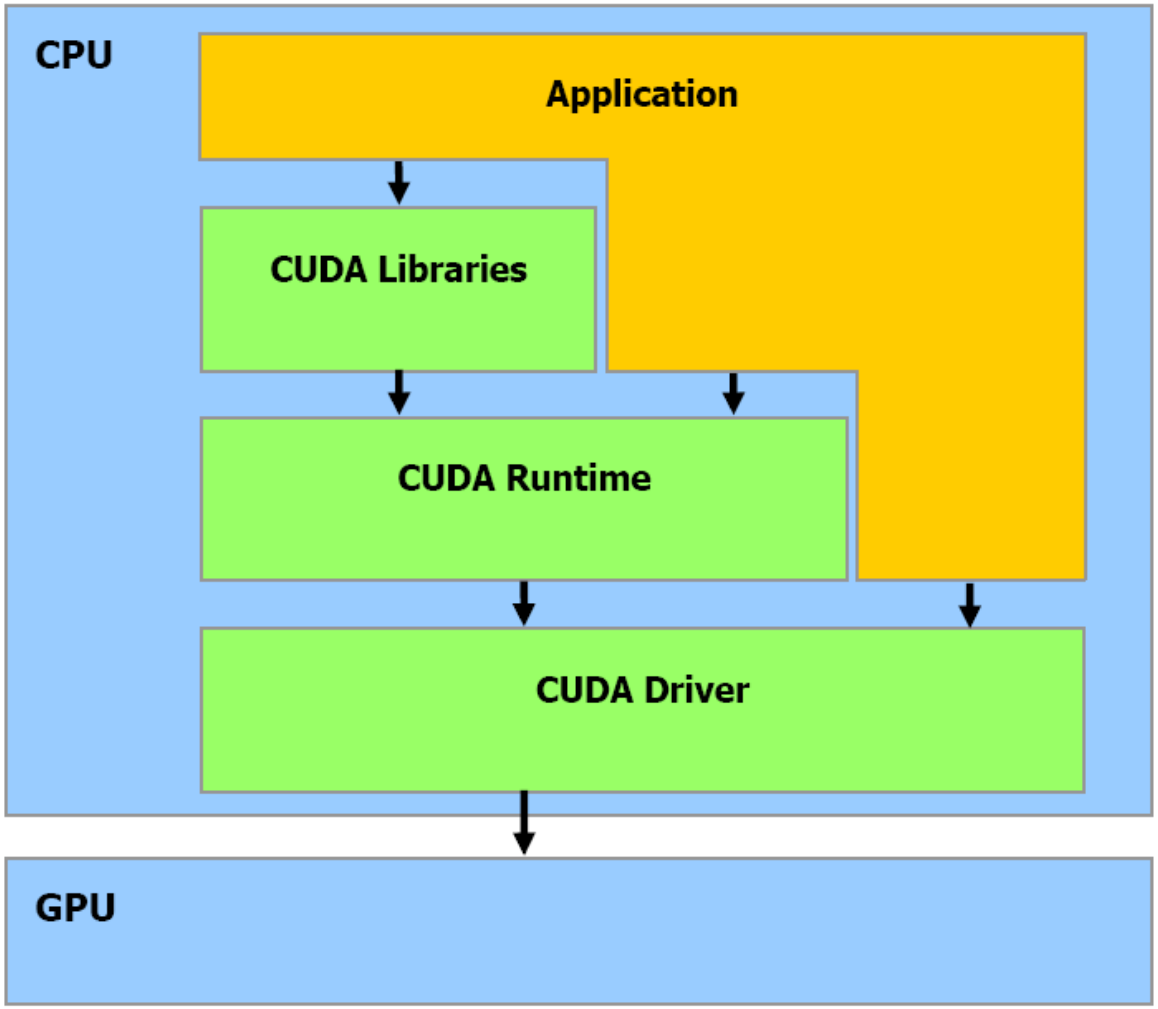
Ветвления не являются причиной падения производительности сами по себе. Вредны только те ветвления, на которых потоки расходятся внутри одного пула потоков warp. При этом если потоки разошлись внутри одного блока, но в разных пулах warp, или внутри разных блоков, это не оказывает никакого эффекта.

Архитектура CUDA

CUDA предоставляет в распоряжение программиста ряд функций, которые могут быть использованы только CPU (CUDA host API). Эти функции отвечают за:

- управление GPU
- работу с памятью
- работу с модулями
- управление выполнением кода
- работу с контекстом
- работу с текстурами
- взаимодействие с OpenGL и Direct3D

Программный стек CUDA



Программы могут использовать GPU посредством:

1. Использования CUDA driver API
2. Использования CUDA runtime API
3. Обращения к стандартным функциям библиотек (CUBLAS, CUFFTW, CUDPP, CUSPARSE, CURAND, CULA...)

CUDA Driver API — низкоуровневый API, реализован в динамической библиотеке nvcuda, и все имена в нём начинаются с префикса cu.
Плюсы:

- низкоуровневый API более гибок, предоставляя программисту дополнительный контроль, если нужно.

Минусы:

- большой объём кода;
- необходимость явных настроек, явной инициализации;
- отсутствие поддержки эмуляции (позволяющего компилировать, запускать и отлаживать коды на CUDA с CPU).

CUDA Runtime API — высокоуровневый API, реализован в динамической библиотеке cudart, и все имена в нём начинаются с префикса cuda. У каждой функции CUDA Runtime API есть прямой аналог в CUDA Driver API, то есть переход не очень сложен, в обратную сторону сложнее.
Плюсы:

- не требует явной инициализации — она происходит автоматически при первом вызове какой-либо его процедуры;
- поддерживает эмуляцию;
- возможность использования дополнительных библиотек.

Взаимодействие API

- Высокоуровневый API реализован над низкоуровневым, каждый вызов функции уровня Runtime API разбивается на более простые инструкции, которые обрабатывает Driver API.
- Два API взаимно исключают друг друга.
- Оба API способны работать с ресурсами OpenGL или Direct3D. Преимущество заключается в том, что ресурсы продолжают храниться в памяти GPU.
- Совместное использование ресурсов в видеопамяти не всегда проходит без ошибок — графические данные приоритетнее!

Библиотеки

- CUBLAS — (BLAS — Basic Linear Algebra Subprograms) — вычисление задач линейной алгебры;
- CUFFT — (FFT — Fast Fourier Transform) — расчёт быстрого преобразования Фурье;
- CUDPP — CUDA Data-Parallel Primitives Library;
- CUSPARSE — функции работы с разряженными матрицами;
- CURAND — генератор случайных чисел;
- CULA — CUDA вариант LAPACK.