Лекция 2: Использование графических процессоров для ускорения вычислений. Архитектура графического процессора

Н.Д. Смирнова

Санкт-Петербургский государственный Политехнический университет

24.09.2011

Содержание

Цели

Стремимся

- получить представление о GPU на абстрактном уровне
- интуитивно представить подходящие задачи
- уровень должен позволить писать и оптимизировать код для GPU
- не стремимся понять электронику

Содержание

- Три основные идеи, которые делают GPU быстрым
  - лучше больше простых процессоров, чем один сложный
  - разделять поток инструкций (SIMD)
  - прятать задержки с помощью чередования рабочих групп
- реализация этих идей на двух платформах: NVIDIA Fermi, AMD Cypress

Использованные материалы

С разрешения Dominika Gïoddeke в этой лекции использованы материалы (мысли, схемы) лекции

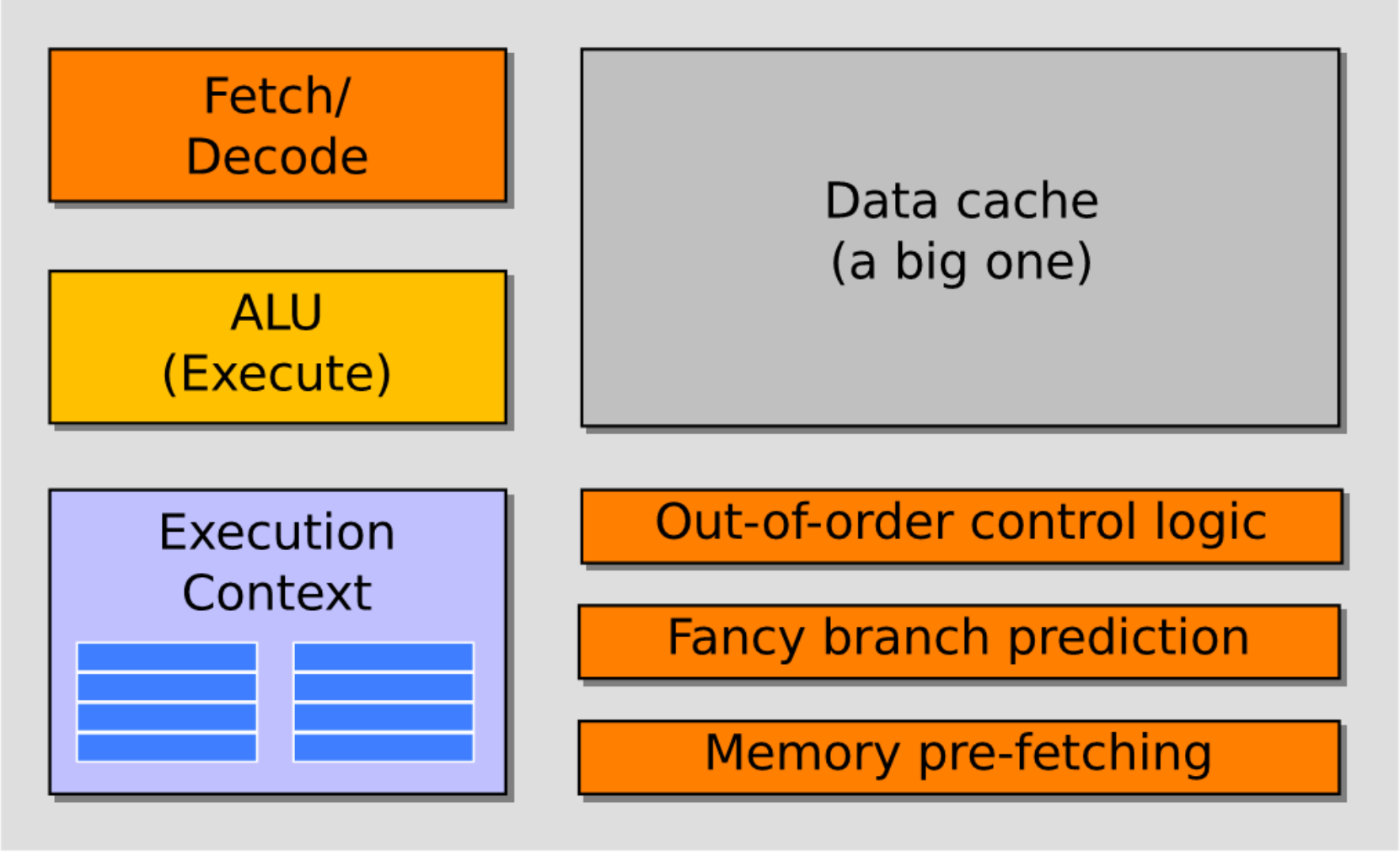
"Scientific Computing on GPUs: GPU Architecture Overview", PPAM 2011 Tutorial

Содержание

Содержание

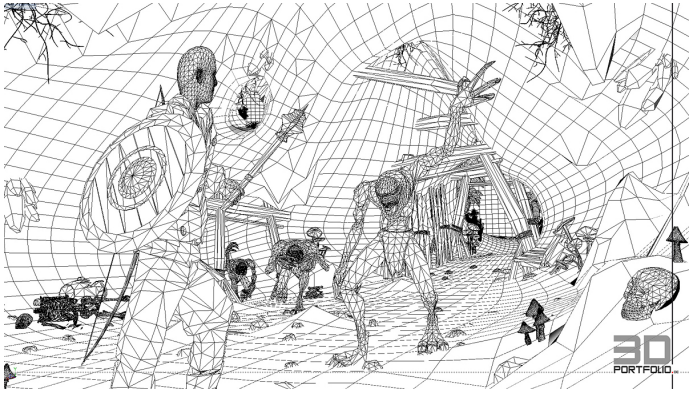
Идея 1: Лучше больше простых процессоров, чем один сложный

Устройство CPU (абстрактно)



Задачи компьютерной графики

Обработка вершины



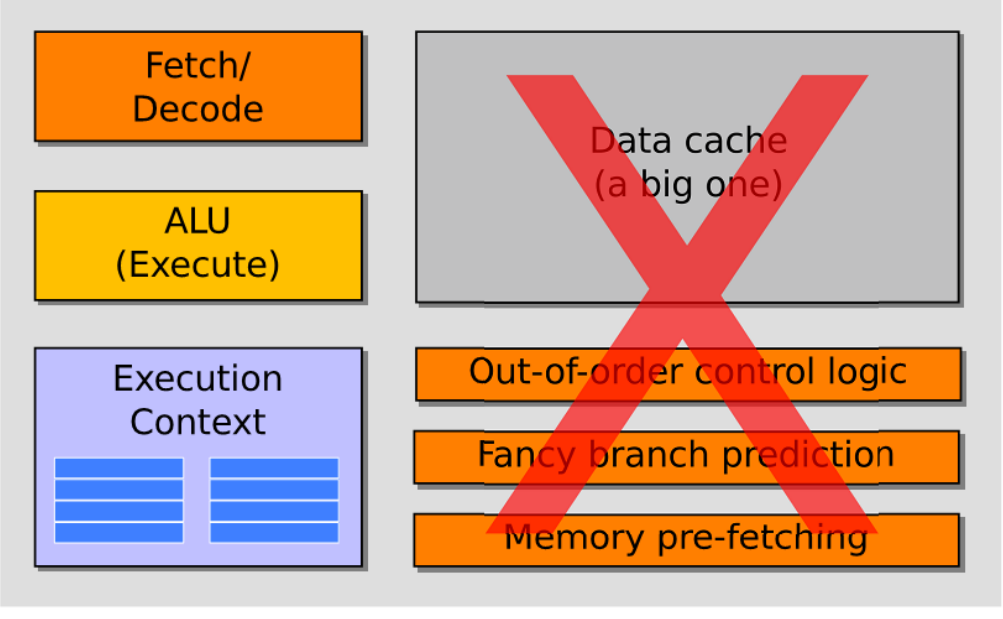
Обработка фрагмента

[1] http://forum.deepsilver.com/forum/showthread.php?t=44299

Упрощение CPU

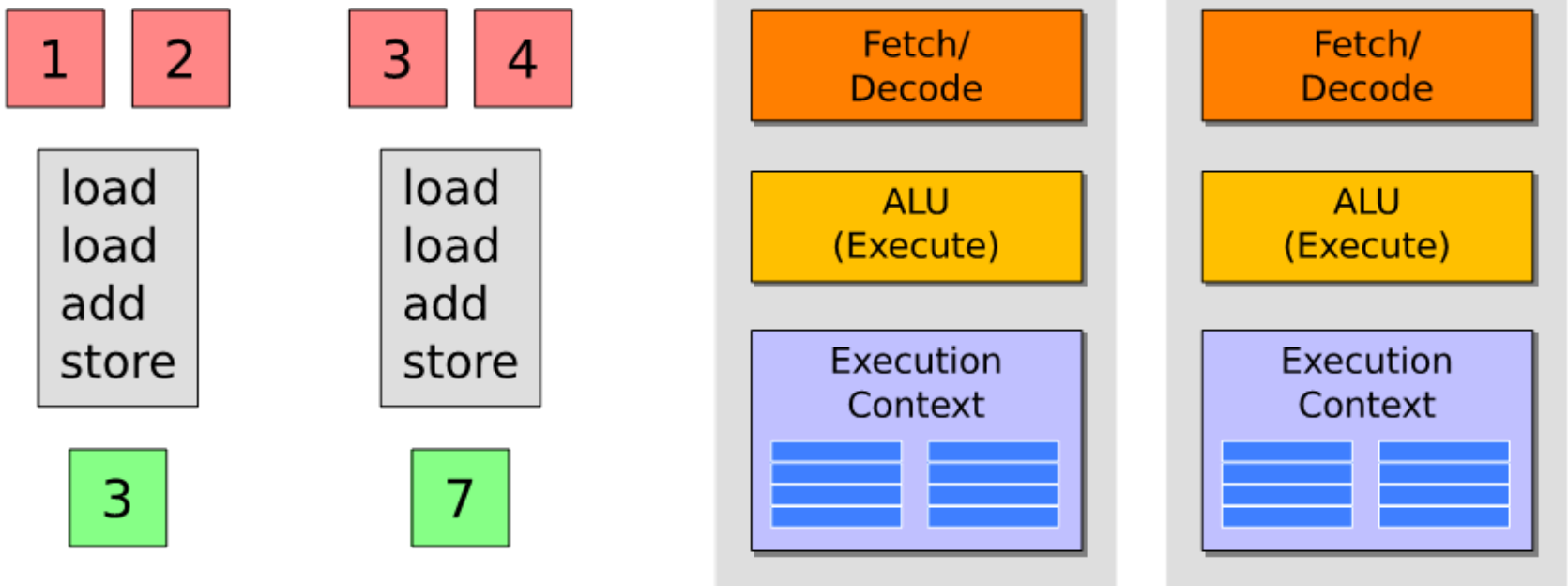
Убираем все, что делает обычный CPU быстрым:

- кэш (все равно используется неактивно)
- микросхемы, реализующие:
  - непоследовательное исполнение инструкций
  - предсказание ветвления
  - опережающая выборка блоков данных из памяти



Увеличиваем количество ядер

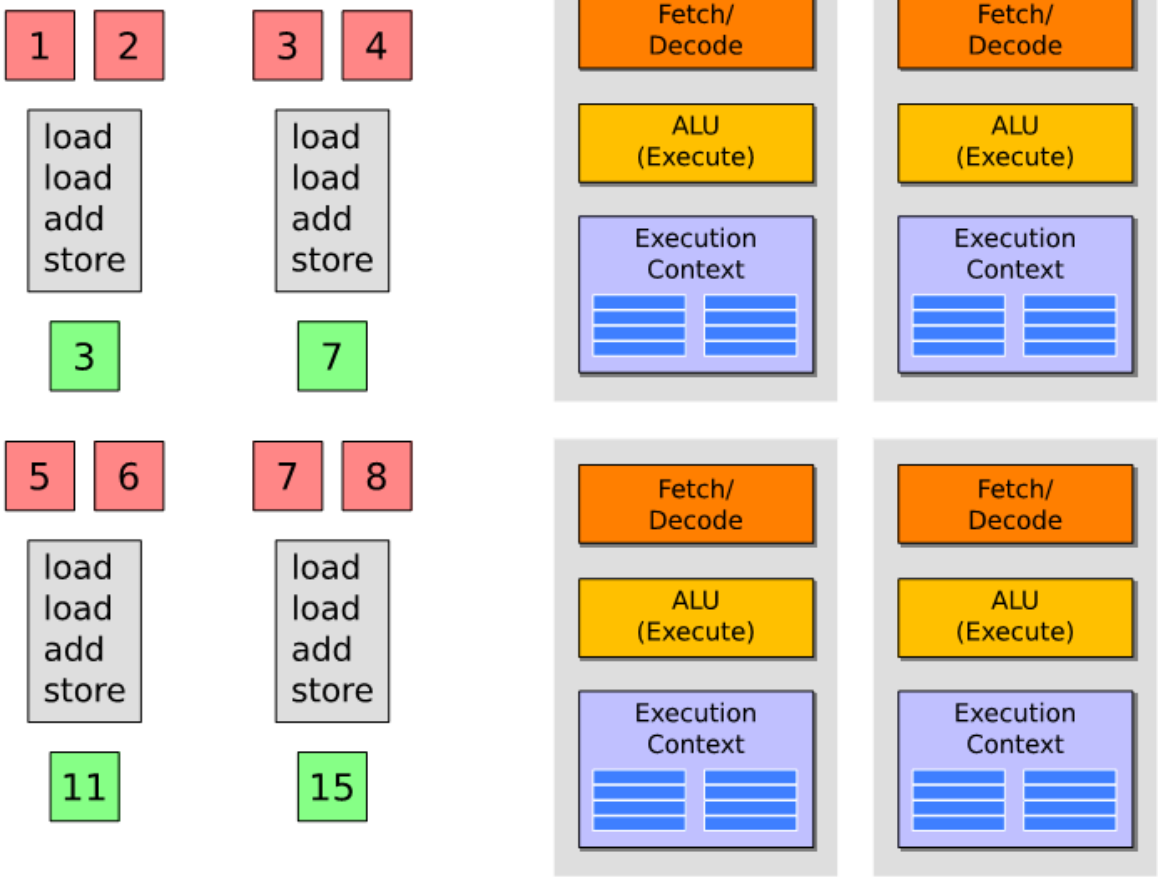
Используем сэкономленные транзисторы для новых ядер



Идея 1: Лучше больше простых процессоров, чем один сложный

Еще больше ядер параллельно

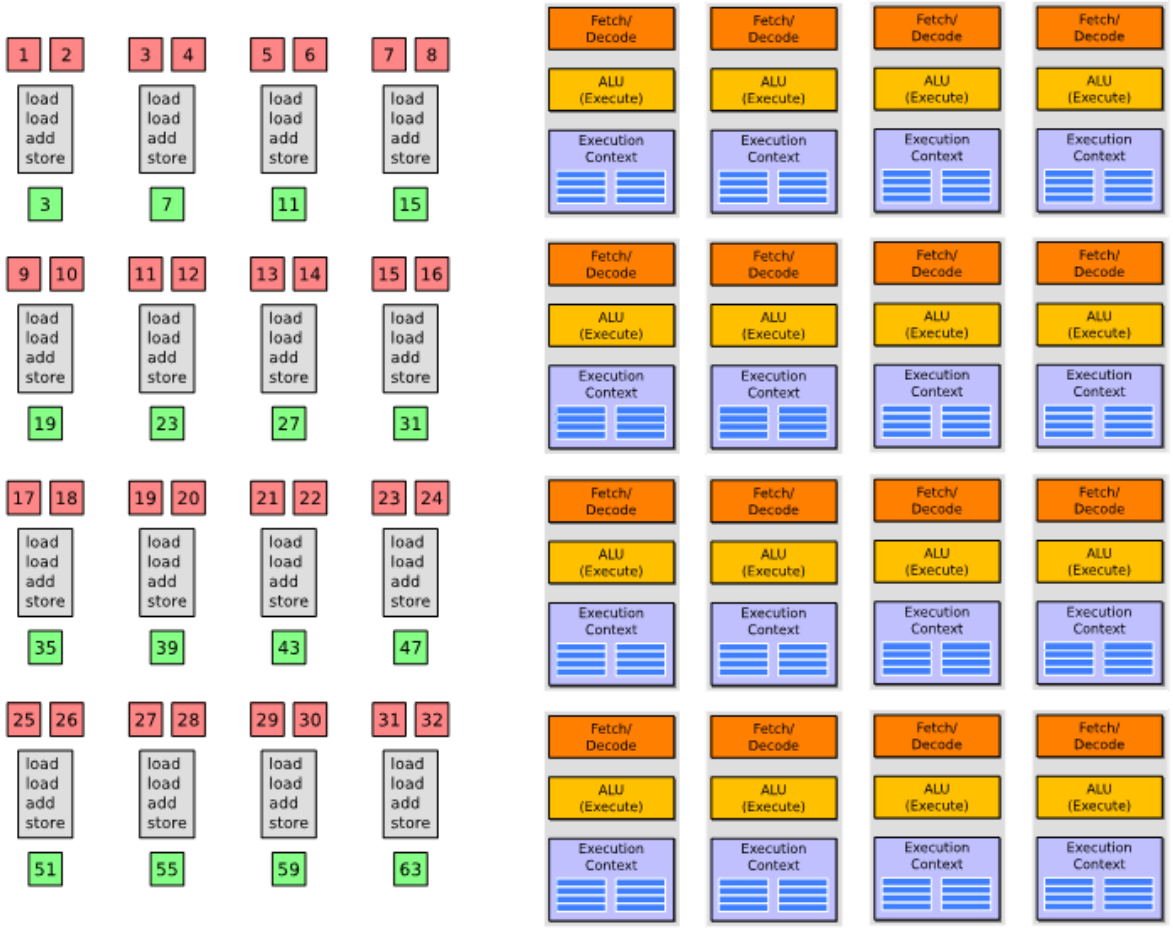
Используем сэкономленные транзисторы для новых ядер



Идея 1: Лучше больше простых процессоров, чем один сложный

Еще больше ядер параллельно

Используем сэкономленные транзисторы для новых ядер



Содержание

Одна инструкция - много данных (SIMD)

Особенности вычислений компьютерной графики

- одна операция (обработка вершины, фрагмента) выполняется над большим количеством данных
- естественный параллелизм: векторные, матричные операции
- множество идентичных потоков инструкций одновременно избыточно, достаточно **одного**
- концепция *информационного параллелизма* (data-parallelism)

SIMD и GPU

Основная идея архитектуры GPU: на уровне конструкции чипов

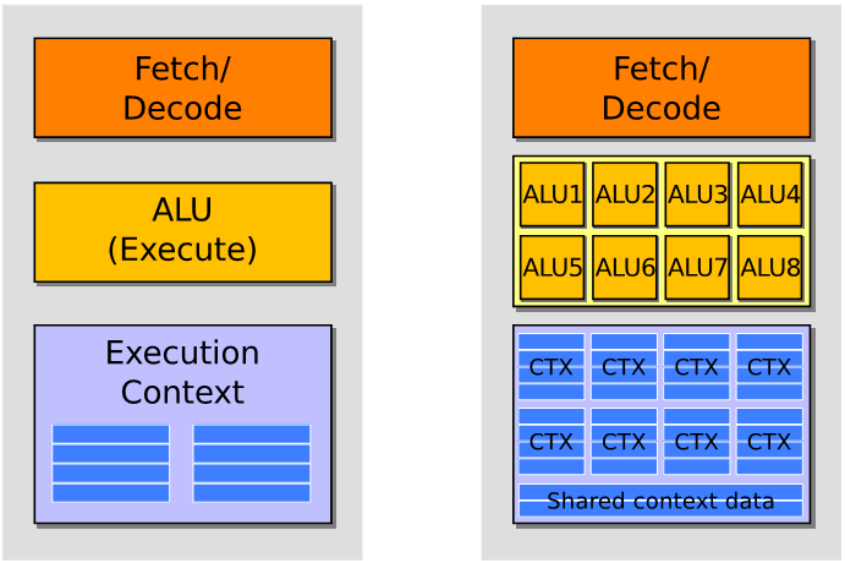
- максимально приспособиться с информационному параллелизму
- понизить стоимость/сложность управления потоками инструкций на множестве ALU [2]
- повысить количество эффективных флопов [3] относительно общего количества транзисторов

[2] ALU (arithmetic logic unit) - вычислитель, цифровое устройство, производящее арифметические и логические операции

[3] FLOP - **fl**oating point **op**erations

Оптимизация для SIMD

Усовершеннствованное ядро



Пример архитектуры: 16 усовершенствованных ядер:

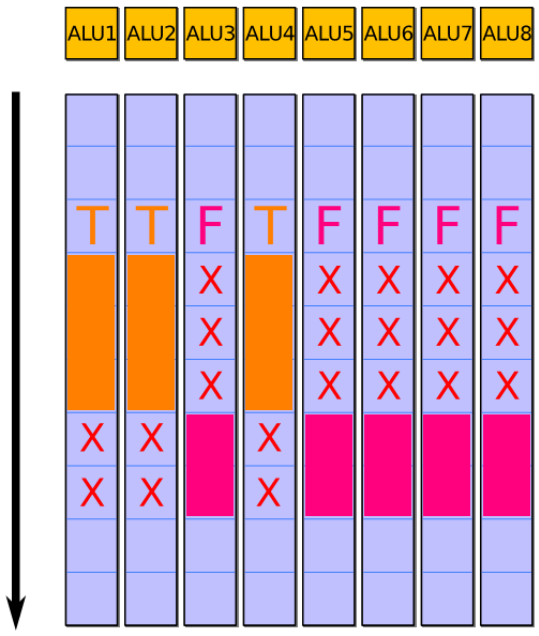
- 128 операций параллельно
- возможно выполнение 16-ти независимых потоков инструкций

Проблемы с SIMD

- SIMD вычисления подразумевают эффективный доступ к памяти
- ветвление ведет к изменению потока инструкций
  - *решение*: вместо ветвления последовательное выполнение ветвей (branch serialization):
    - сперва работают те вычислители, которым досталась ветка **if**
    - затем те, которым досталась - **else**
  - *цена*: в 2 раза дольше, чем без ветвления
  - *бонус*: реализация в железе недорогая
  - *неприятность*: не все ALU используются, в худшем случае 1/8 ALU делает что-то полезное

Branch Serialization

- *пример*: последовательно выполняем обе ветви
  - шаг1: 3 из 8 ALU выполняют ветвь if
  - шаг2: 5 из 8 ALU выполняют ветвь else

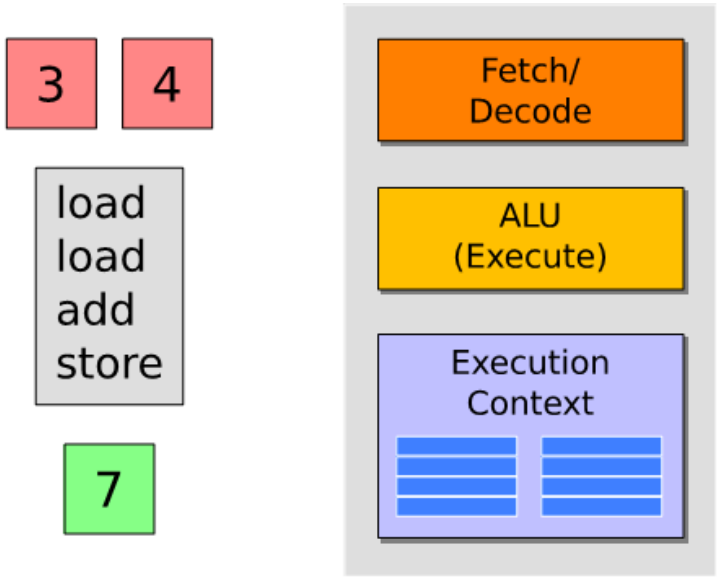


Содержание

Задержки

Задержки(простои) - задержки вычислений из-за зависимостей между инструкциями

- пример: выполнение ADD не может начаться, пока не закончится LOAD
- причина задержки: доступ к памяти - операция медленная
- упрощение CPU (всех наворотов, помогавших убрать простои) является плохой идеей (Идея #1)?



Чередование SIMD-групп

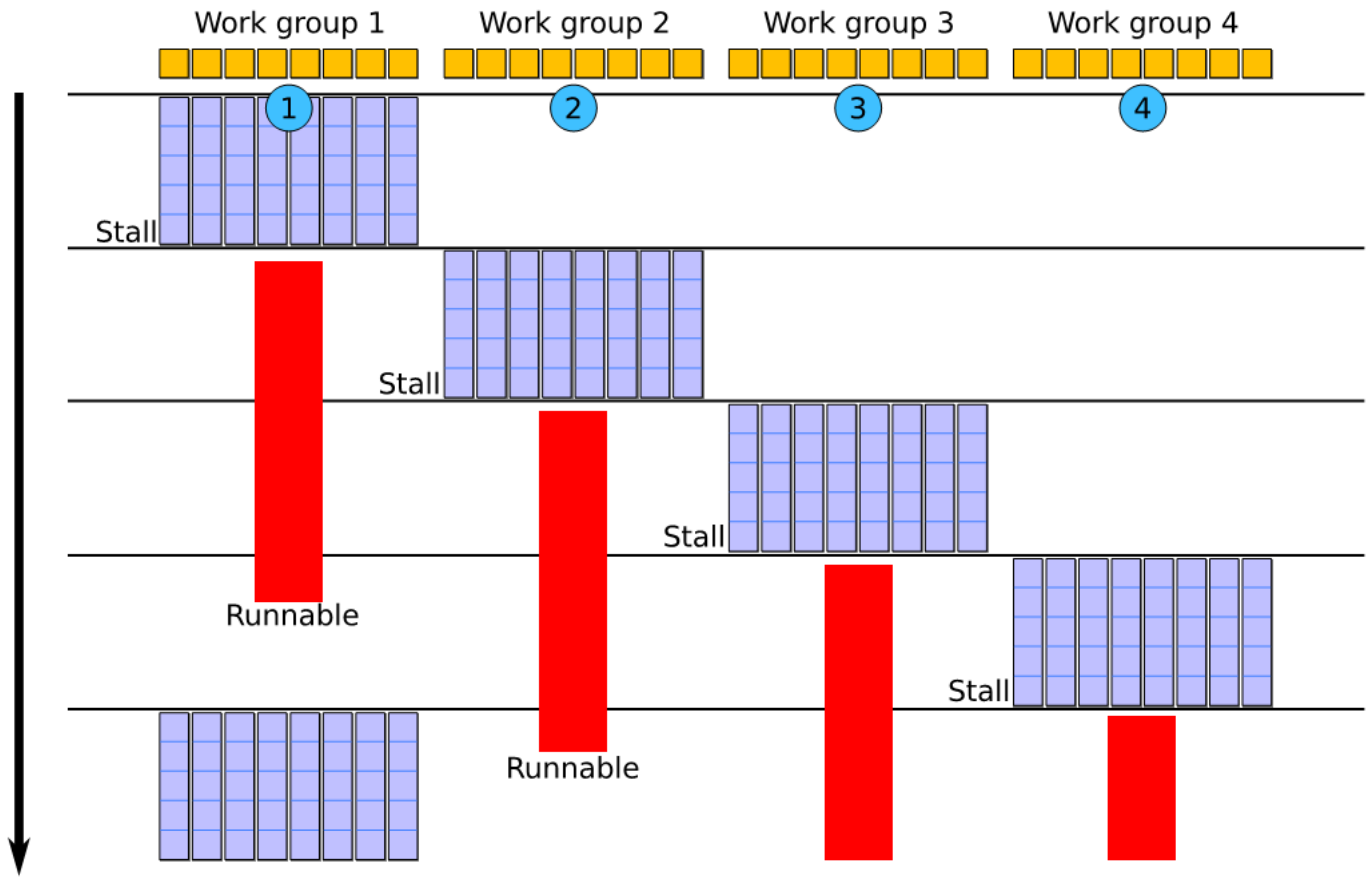
Но ведь GPU подразумевает наличие независимых SIMD-”групп”:

Идея 3: Чередовать обработку SIMD-групп

- переключаемся на выполнение инструкций другой SIMD-группы, если текущая простаивает
- механизм переключения зашит в железе GPU - никаких накладных расходов
- в идеале, все задержки спрятаны, производительность максимальна

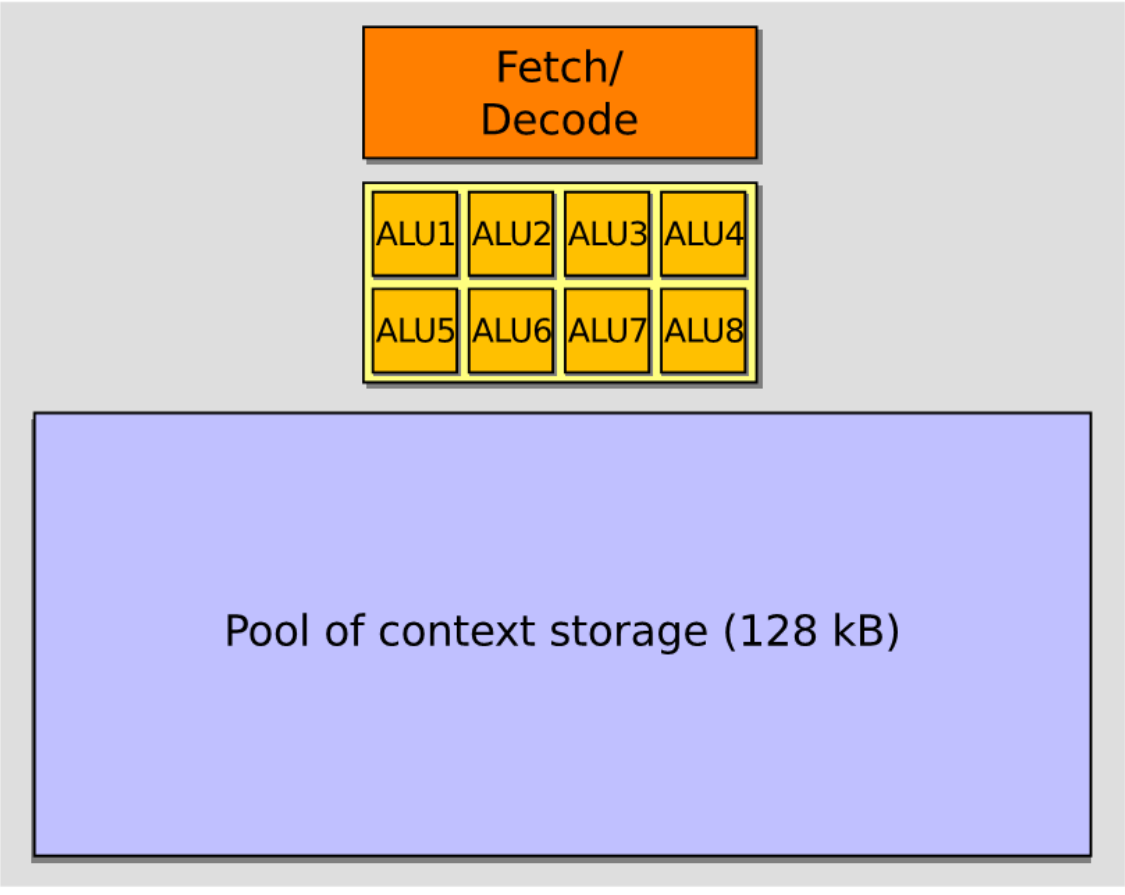
Идея 3: Прятать задержки с помощью чередования рабочих групп

Повышение производительности



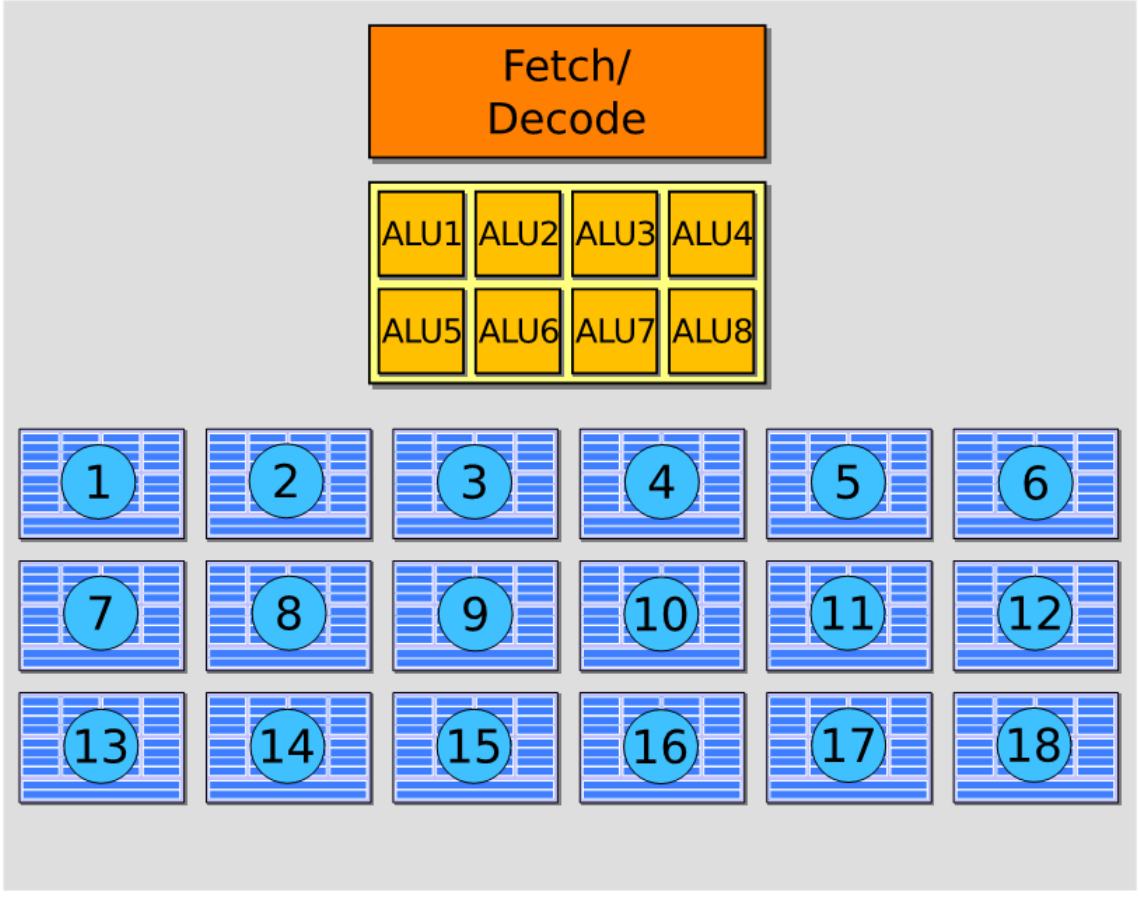
Хранение контекстов

Общее хранилище контекстов



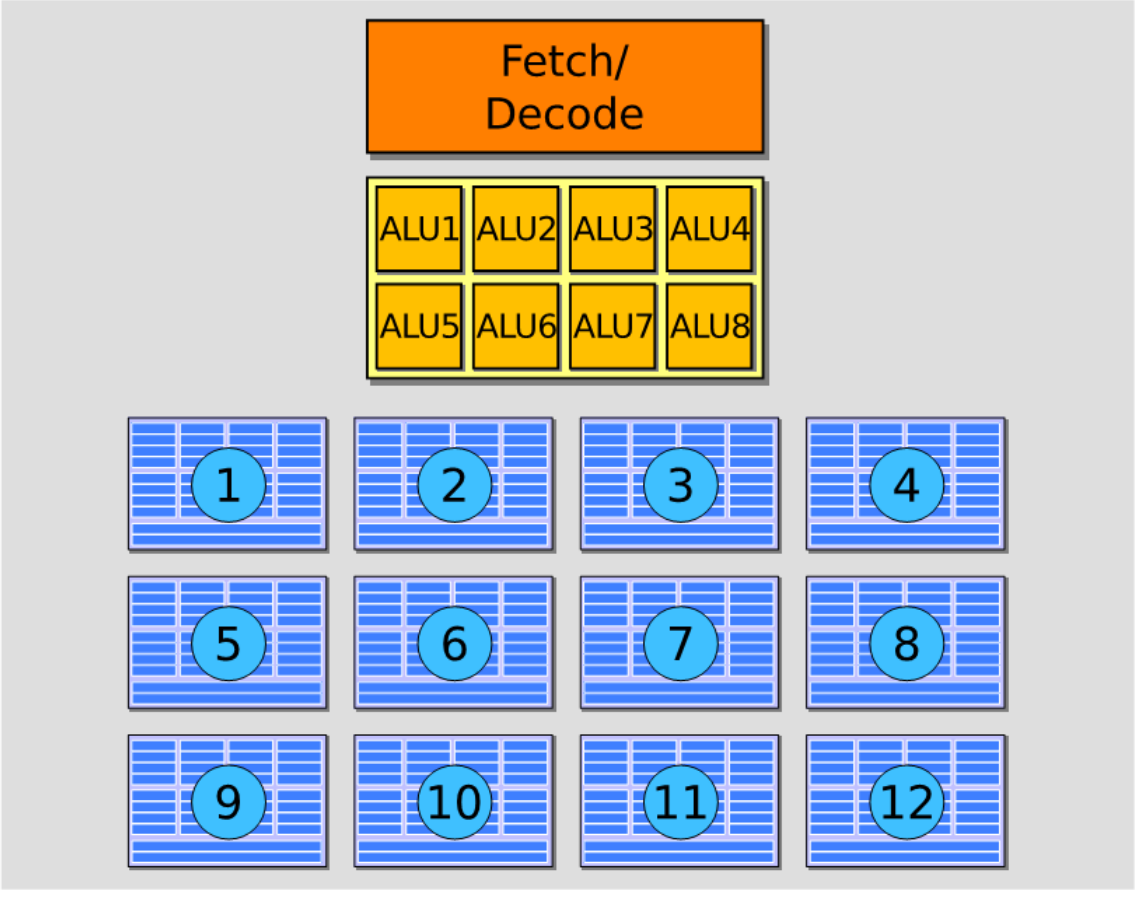
Хранение контекстов

18 маленьких контекстов: простои прячутся эффективно



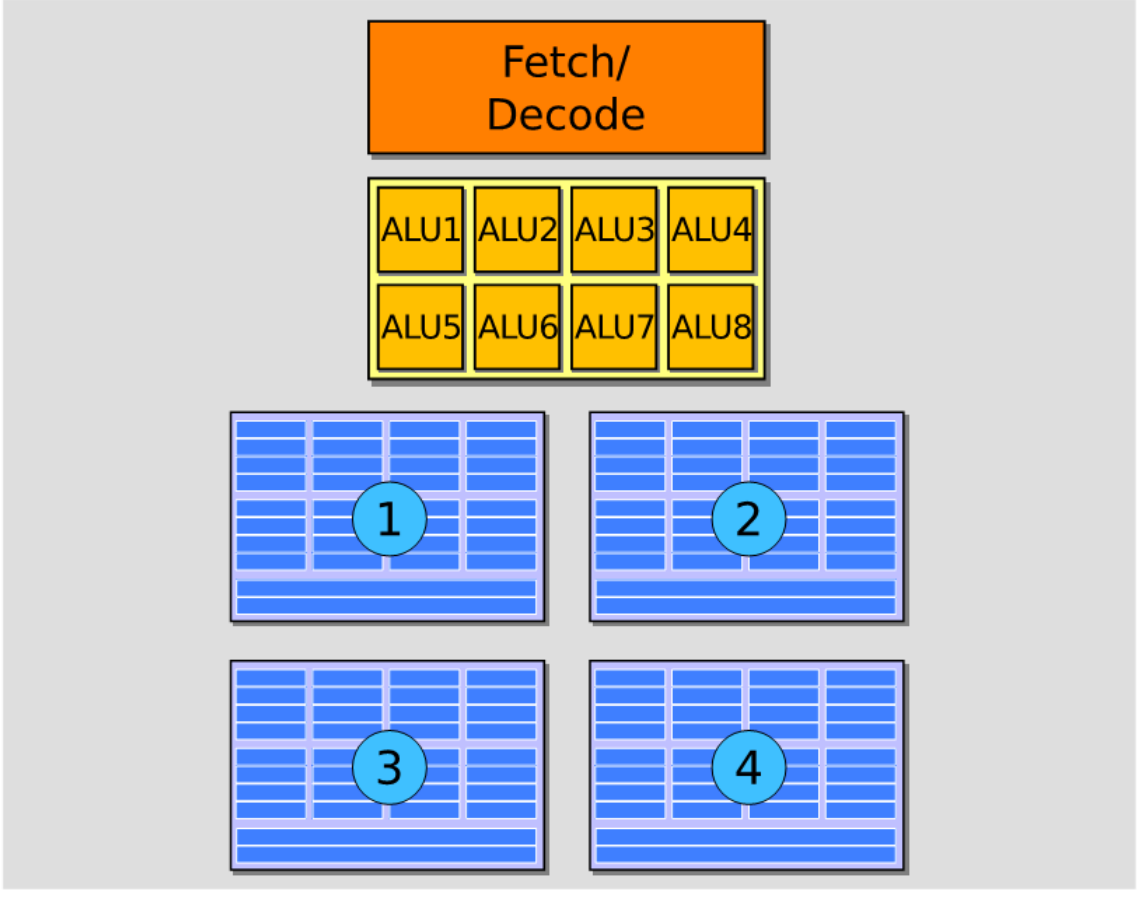
Хранение контекстов

12 средних контекстов



Хранение контекстов

4 больших контекста: простои прячутся плохо



Что имеем?

Основные идеи, увеличивающие "вычислительную плотность" процессора

- Использовать большее количество упрощенных ядер
- Упаковывать в ядро побольше ALU (вычислителей)
- Избегать простоев путем чередования обработки разных SIMD "групп"
- Получили легко **масштабируемую архитектуру любой цены/производительности**

Точка соприкосновения программиста с архитектурой

- OpenCL/CUDA придлагают иерархию для синхронизации SIMD групп

Содержание

Содержание

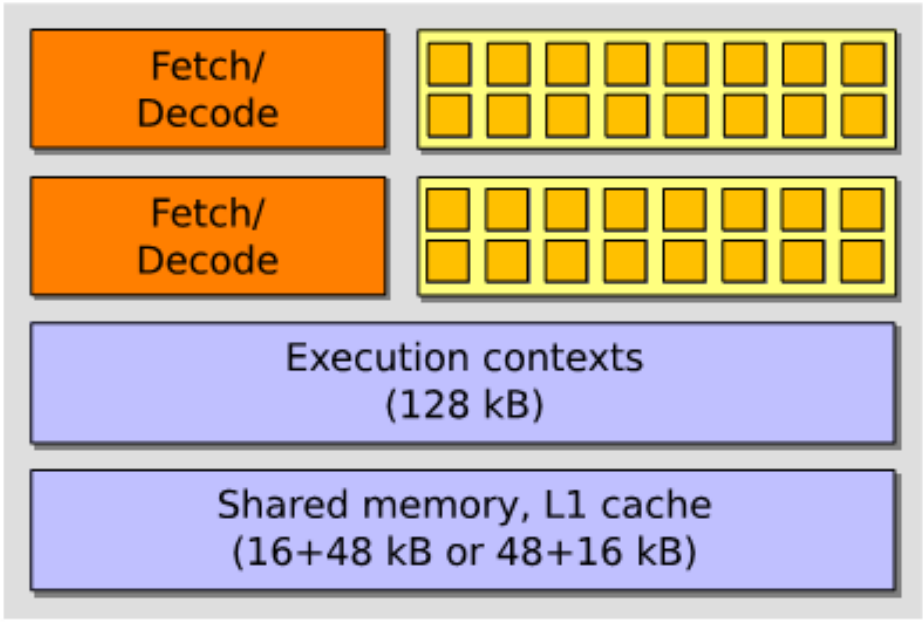
NVIDIA Fermi

Когда NVIDIA говорит

- 480 stream процессора (ядер CUDA)

Подразумевается

- 15 ядер
- в каждой по 2 группы, состоящих из 16 SIMD процессоров



Содержание

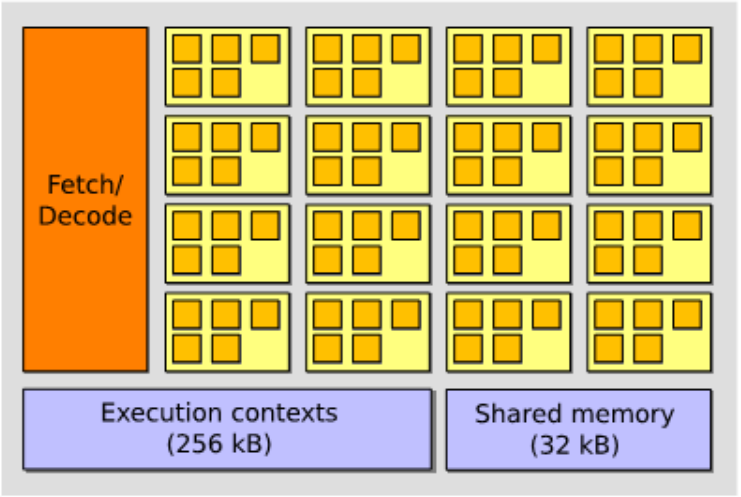
AMD Cypress

Когда AMD говорит

- 1600 stream процессора

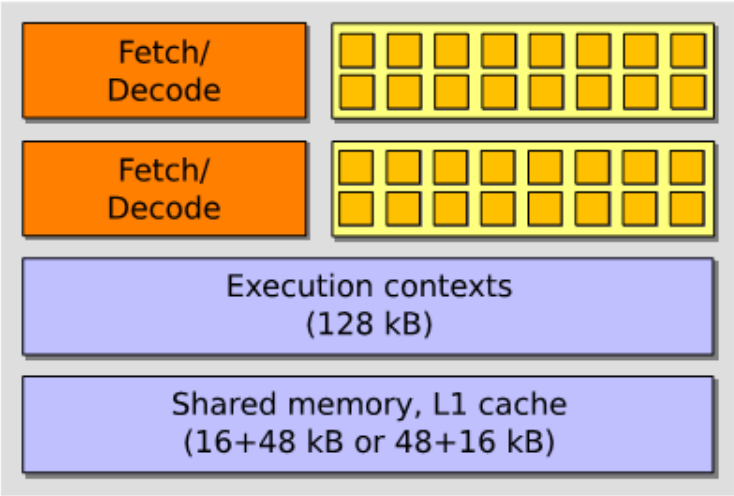
Подразумевается

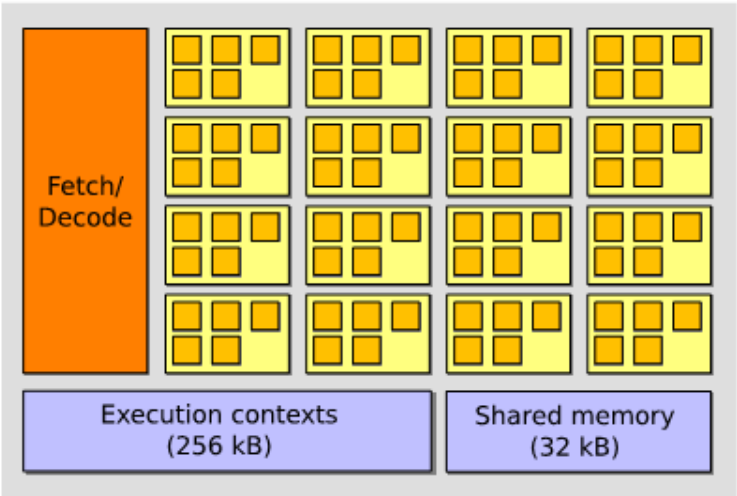
- 20 ядер
- в каждом по 16 ”толстых” SIMD процессора
- каждый из которых исполняет VLIW пакет из 5 независимых интрукций



Содержание

Необходимо учитывать особенности архитектуры





ПРОДОЛЖЕНИЕ СЛЕДУЕТ...