Архитектура и программирование массивно-параллельных вычислительных систем

Лектор:

Боресков А.В. (ВМиК МГУ)

План

- Существующие архитектуры
- Классификация
- Эволюция GPU
- GP GPU
- Несколько слов о курсе
- Дополнительные слайды

План

- Существующие архитектуры
  - Intel CPU
  - SMP
  - CELL
  - BlueGene
  - NVIDIA Tesla/GeForce
- Классификация
- CUDA
- Несколько слов о курсе
- Дополнительные слайды

Существующие многоядерные системы

Посмотрим на частоты CPU:

- 2004 г. - Pentium 4, 3.46 GHz
- 2005 г. - Pentium 4, 3.8 GHz
- 2006 г. - Core Duo T2700, 2333 MHz
- 2007 г. - Core 2 Duo E6800, 3 GHz
- 2009 г. - Core i7 950, 3.06 GHz
- 2010 г. - Core i5, 3.6 GHz
- 2011 г. - Core i7 Extreme 3.9 Ghz
- 2017 г. – Core i7 7700K 4.2 Ghz

Существующие многоядерные системы

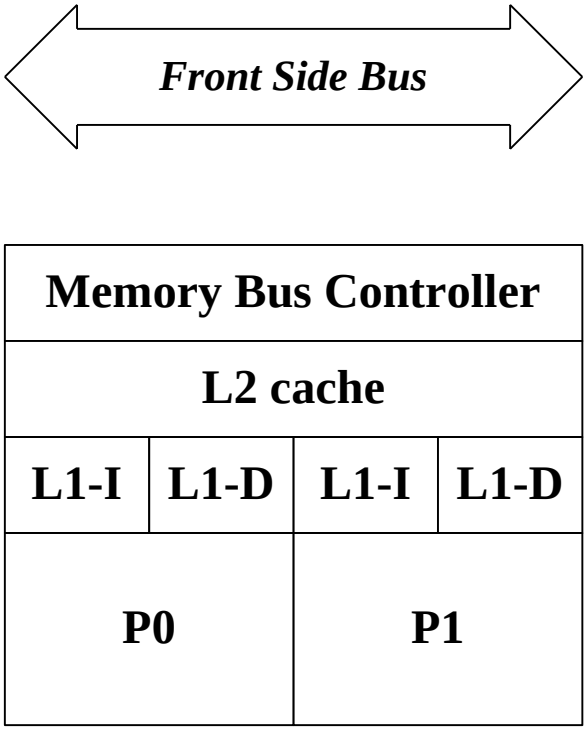
- Роста частоты практически нет
  - Энерговыделение ~ второй степени частоты
  - Ограничения техпроцесса
  - Одноядерные системы зашли в тупик
  - Развитие идет за счет параллельности
    - Многоядерность
    - Hyperthreading

Существующие многоядерные системы

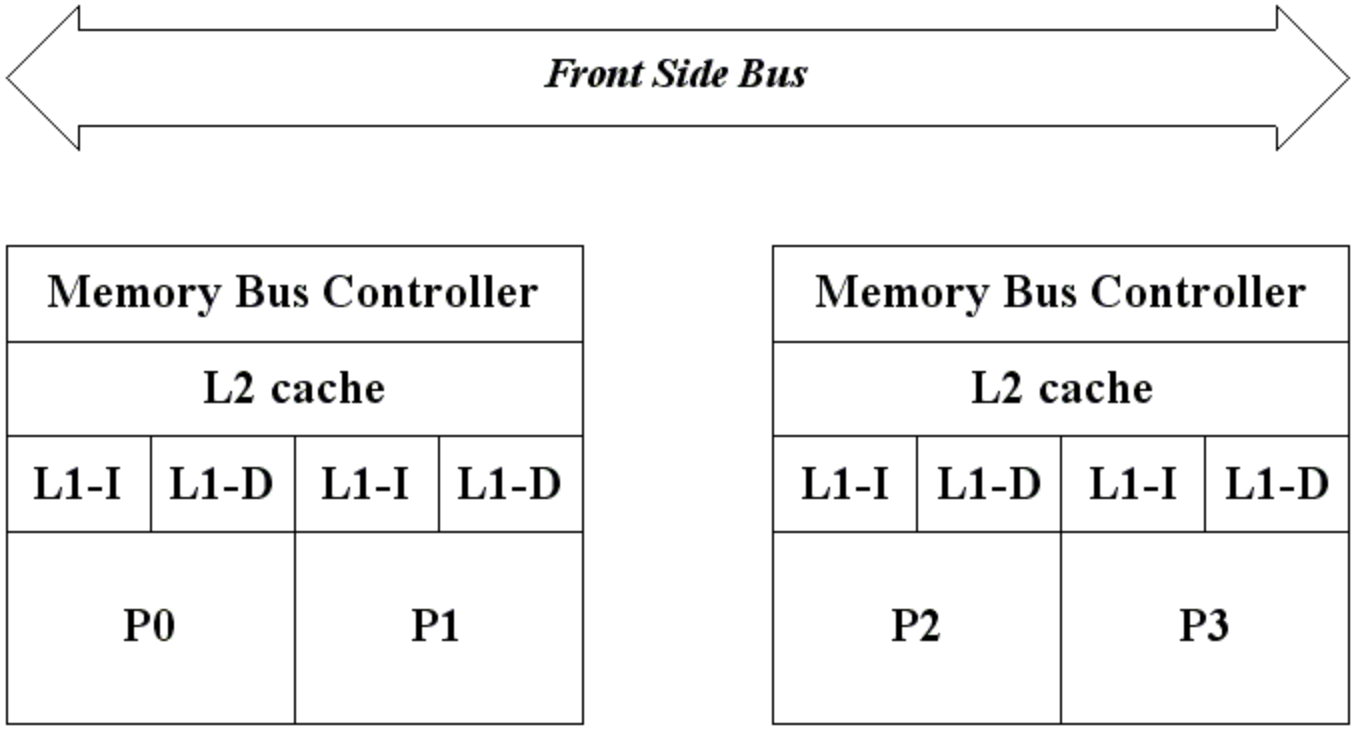
- Повышение быстродействия следует ждать от параллельности.
- CPU используют параллельную обработку для повышения производительности
  - Конвейер
  - Multithreading
  - SSE
  - Hyperthreading

Intel Core 2 Duo

- 32 Кб L1 кэш для каждого ядра
- 2/4 Мб общий L2 кэш
- Единый образ памяти для каждого ядра - необходимость синхронизации кэшей



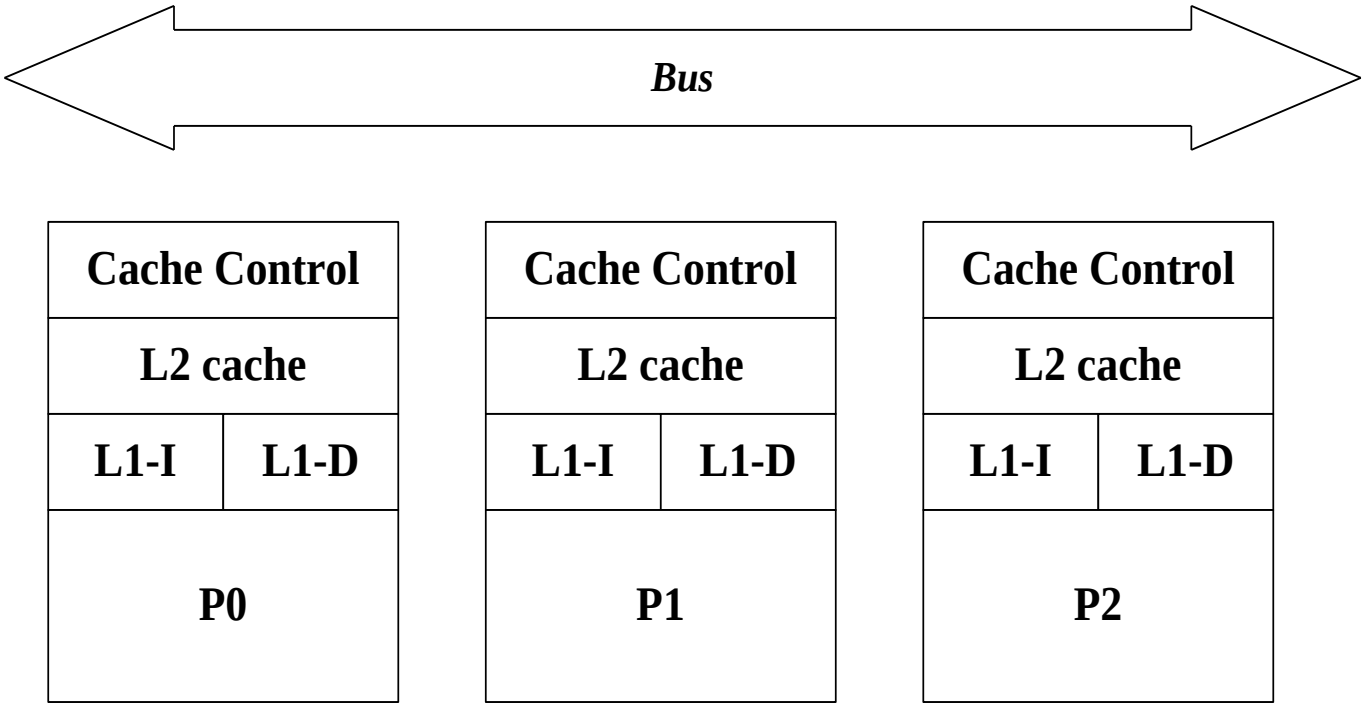
Intel Core 2 Quad



Intel Core i7

Memory Bus Controller							
L3 cache							
L2 cache		L2 cache		L2 cache		L2 cache	
L1-I	L1-D	L1-I	L1-D	L1-I	L1-D	L1-I	L1-D
P0		P1		P2		P3	

Symmetric Multiprocessor Architecture (SMP)



Symmetric Multiprocessor Architecture (SMP)

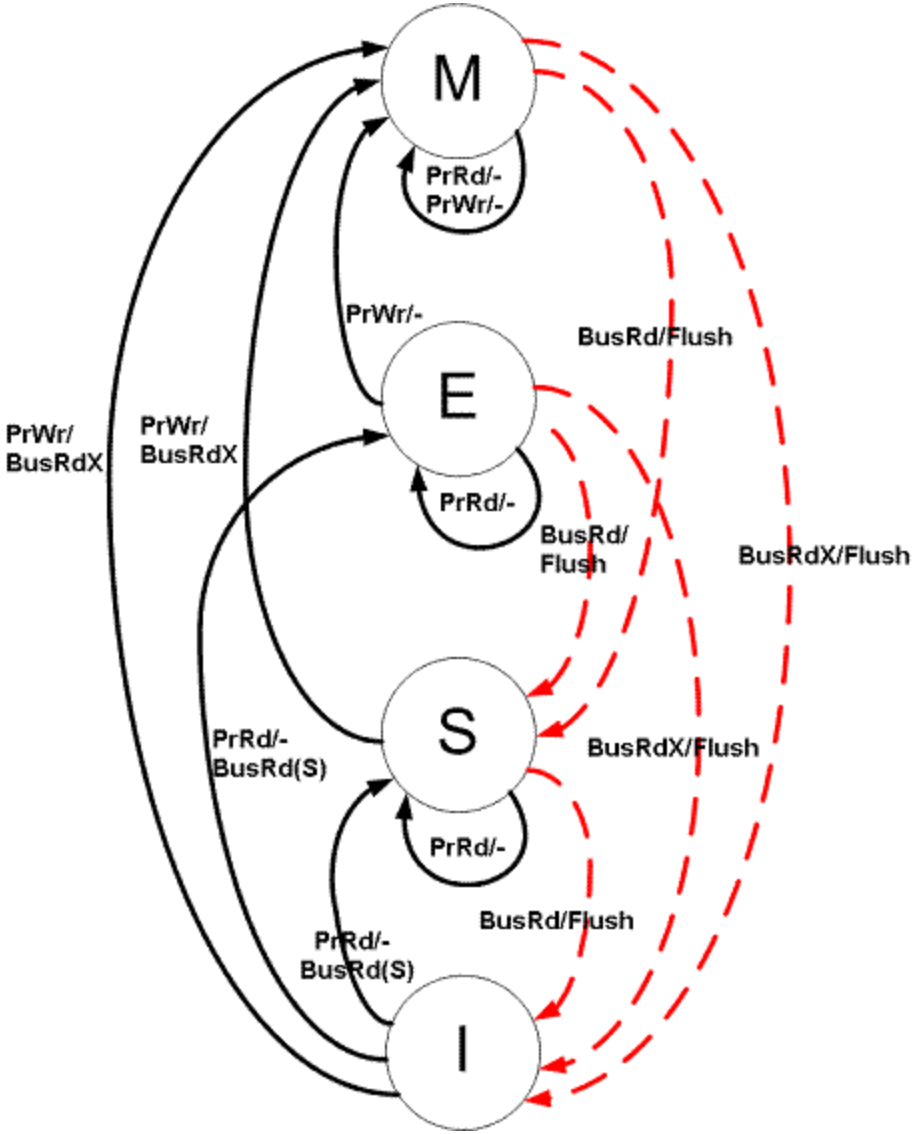
Каждый процессор

- имеет свои L1 и L2 кэши
- подсоединен к общей шине
- **отслеживает доступ других процессоров к памяти** для обеспечения единого образа памяти (например, один процессор хочет изменить данные, кэшированные другим процессором)

Отслеживание кэшей для обеспечения целостности памяти

- Протокол MESI
  - (M)odified – есть только в одном кэше и dirty
  - (E)xclusive – есть только в одном кэше и clean
  - (S)hared – есть в нескольких кэшах и clean
  - (I)nvalid

Конечный автомат для MESI



Пример

- Есть массив данных длины N
- Нужно на 4-ядерном процессоре найти число «7» в этом массиве
- Запускаем 4 нити (по одной на каждый процессор), каждая нить получает четверть массива

Пример

- Общий счетчик для всех нитей
  - На каждой записи будет синхронизация всех кэшей
- Отдельный счетчик на каждую нить
  - Все зависит от их расположения в памяти

Расположение счетчиков в памяти

int counts [4];

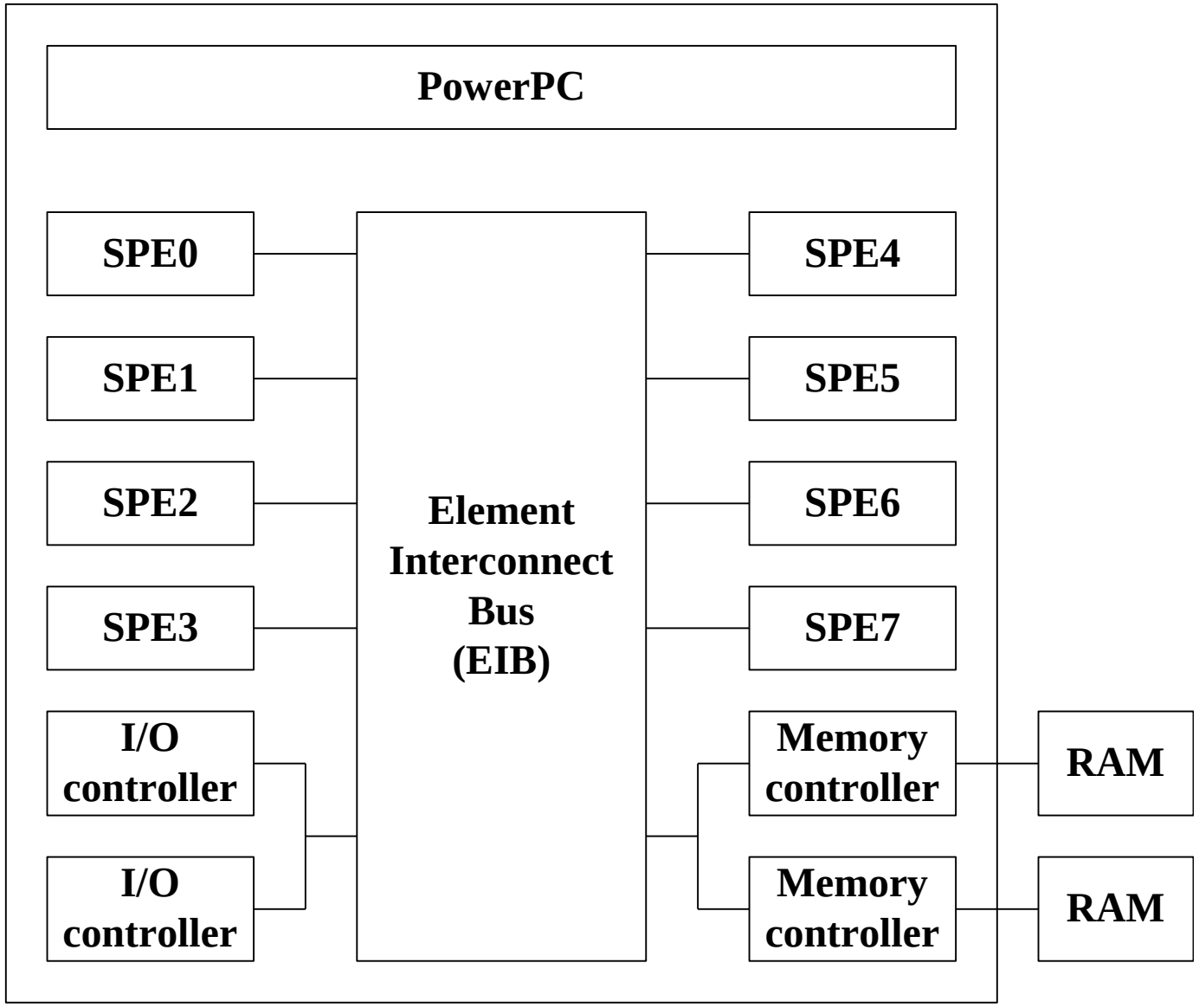
- Кэш работает группами байтов (линейками), обычно 32-64 байта
- Скорее всего они попадут в одну линейку и на каждую запись в один из счетчиков будет идти полная синхронизация всей линейки

Расположение счетчиков в памяти

```
struct {
  int count;
  char dummy [32-4];
} counts [4];
```

- Тогда каждый счетчик попадает в свою 32-байтовую линейку

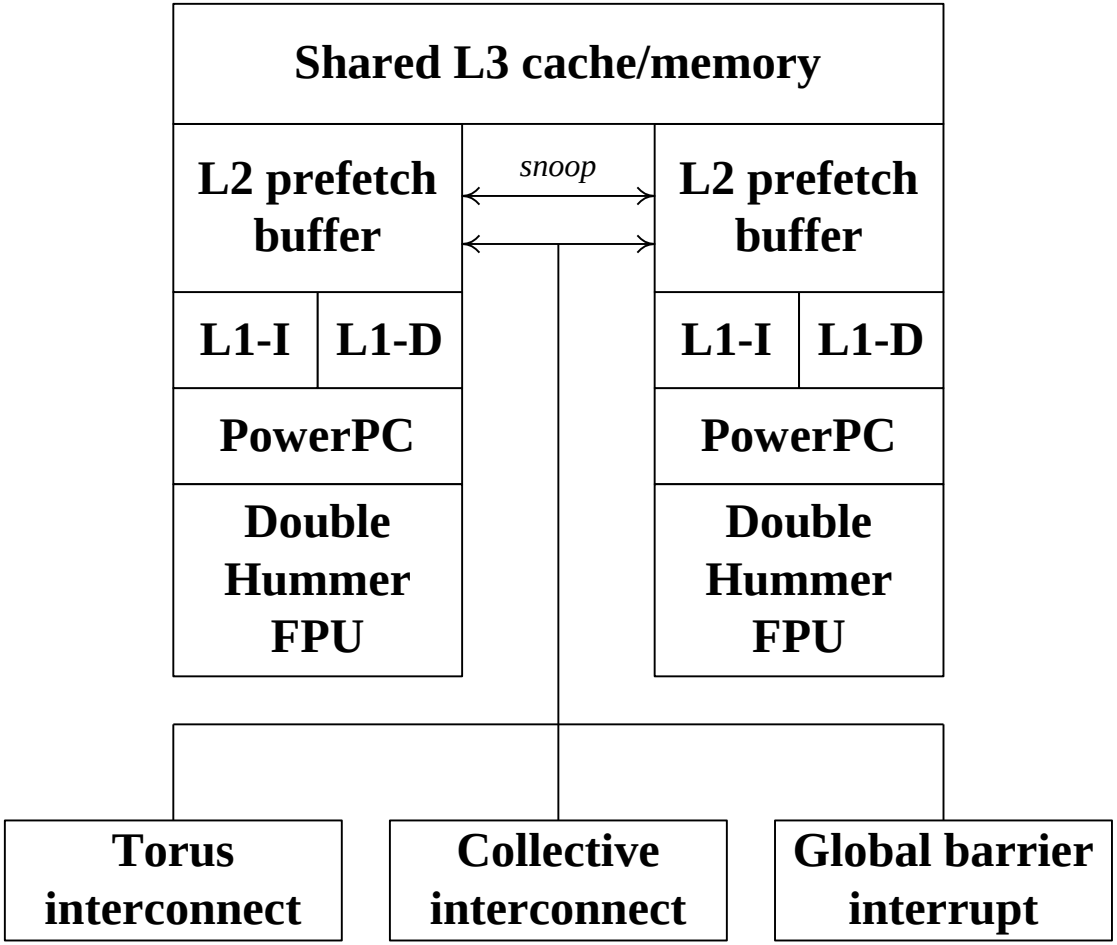
Cell


PowerPC
SPE0
SPE1
SPE2
SPE3
I/O controller
I/O controller
Element Interconnect Bus (EIB)
SPE4
SPE5
SPE6
SPE7
Memory controller
Memory controller
RAM
RAM

Cell

- Dual-threaded 64-bit PowerPC
- 8 Synergistic Processing Elements (SPE)
- 256 Kb on-chip на каждый SPE

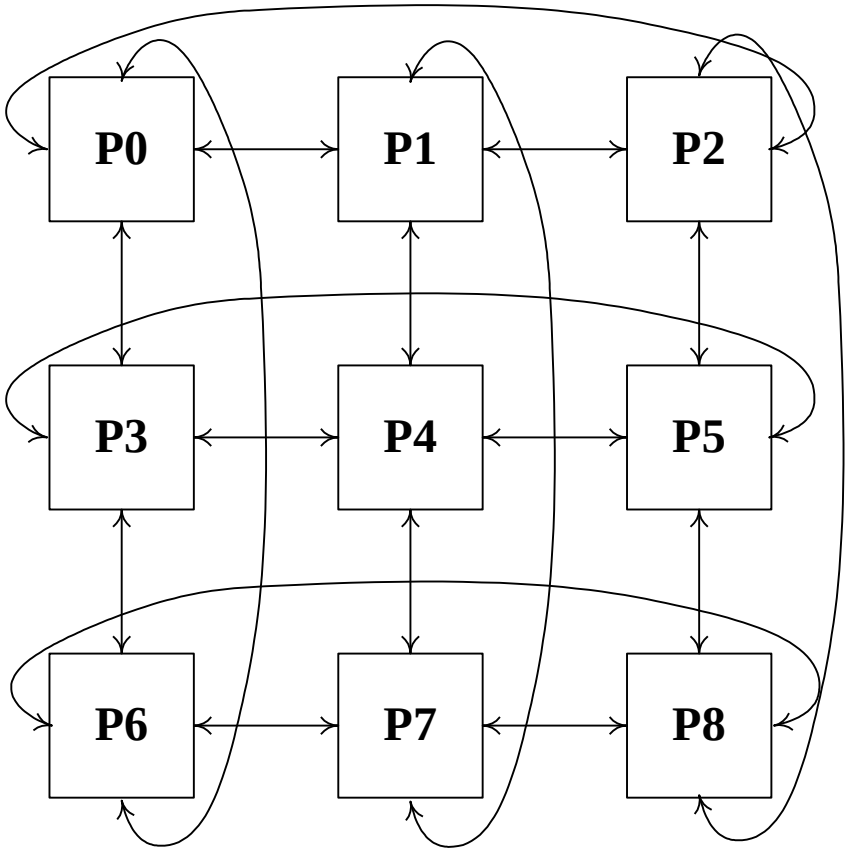
BlueGene/L


Shared L3 cache/memory
L2 prefetch buffer
snoop
L2 prefetch buffer
L1-I
L1-D
L1-I
L1-D
PowerPC
PowerPC
Double Hummer FPU
Double Hummer FPU
Torus interconnect
Collective interconnect
Global barrier interrupt

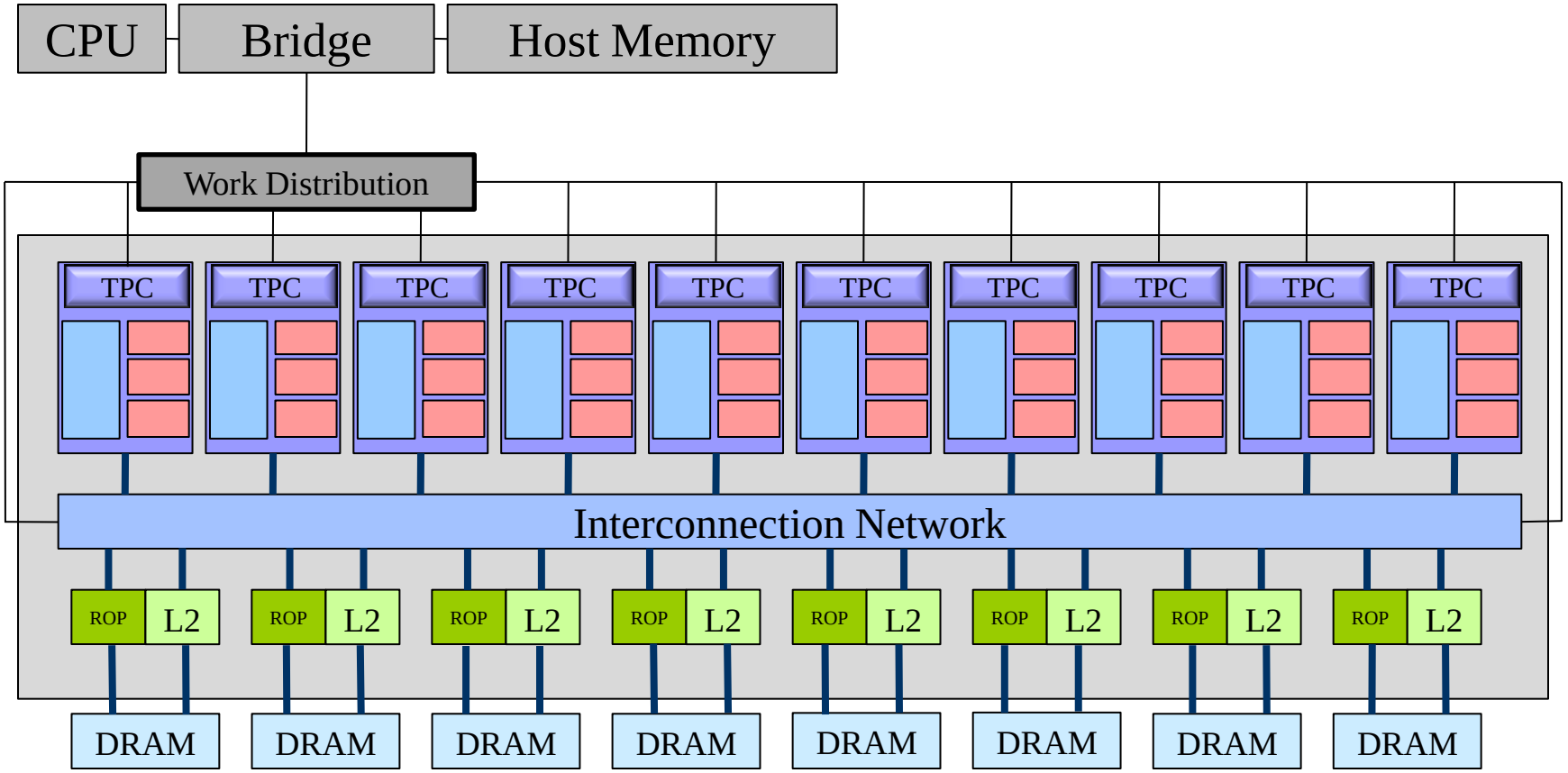
BlueGene/L

- 65536 dual-core nodes
- node
  - 770 Mhz PowerPC
  - Double Hammer FPU (4 Flop/cycle)
  - 4 Mb on-chip L3 кэш
  - 512 Mb off-chip RAM
  - 6 двухсторонних портов для 3D-тора
  - 3 двухсторонних порта для collective network
  - 4 двухсторонних порта для barrier/interrupt

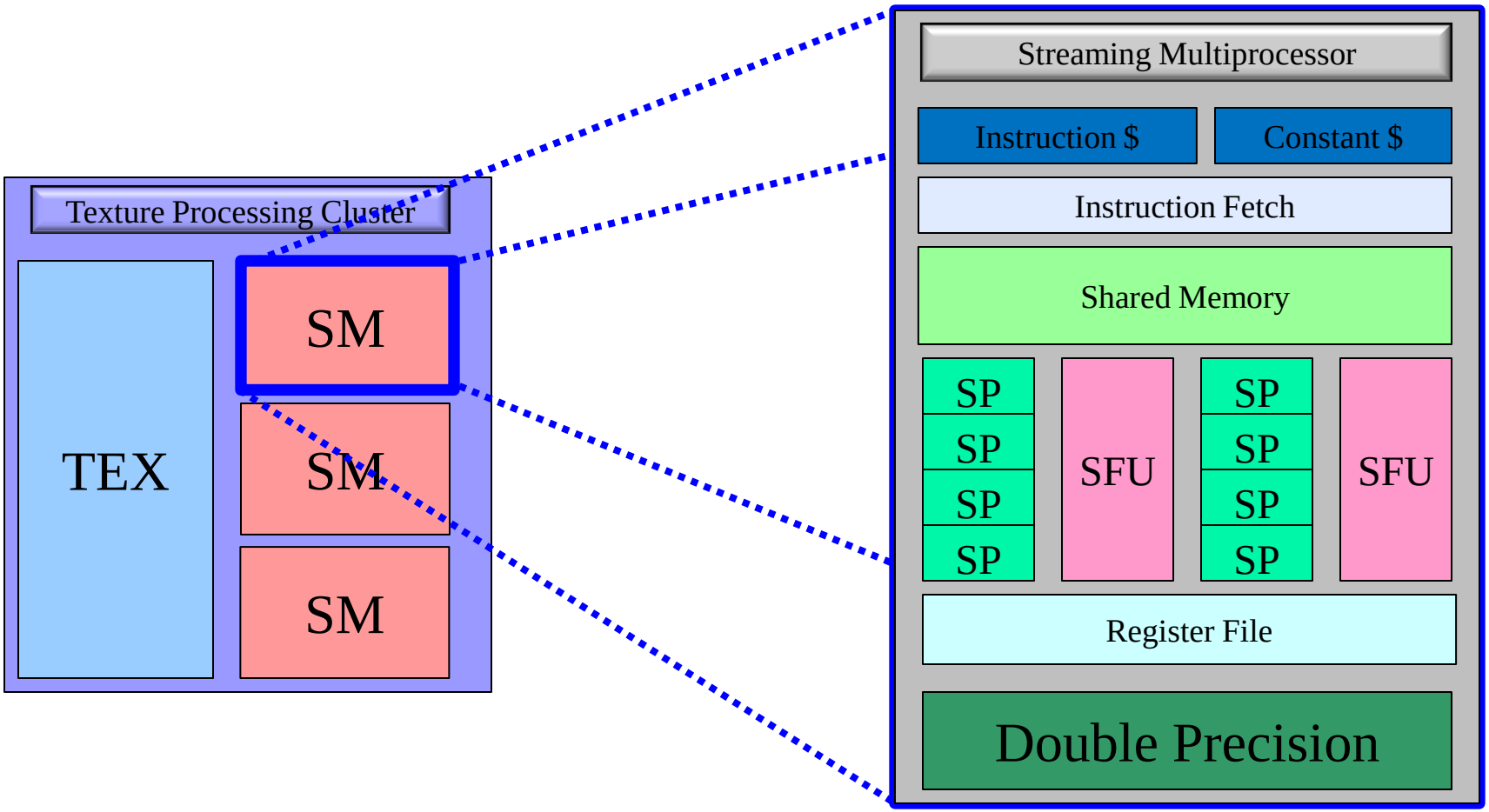
BlueGene/L



Архитектура Tesla 10



Архитектура Tesla
Мультипроцессор Tesla 10



Технические детали

- RTM **CUDA Programming Guide**
- Run CUDAHelloWorld
  - Печатает аппаратно зависимые параметры
    - Размер shared памяти
    - Кол-во SM
    - Размер warp’a
    - Кол-во регистров на SM
    - т.д.

План

- Существующие архитектуры
- Классификация
  - Примеры для CPU
- CUDA
- Несколько слов о курсе
- Дополнительные слайды

Классификация

	Single Instruction	**Multiple Instruction**
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Классификация

- CPU – SISD
  - Multithreading: позволяет запускать множество потоков – параллелизм на уровне задач (MIMD) или данных (SIMD)
  - SSE: набор 128 битных регистров ЦПУ
    - можно запаковать 4 32битных скаляра и проводить над ними операции одновременно (SIMD)
- GPU – SIMD*

SSE “Hello World”

```
#include <xmmintrin.h>
#include <stdio.h>
struct vec4
{
    union
    {
        float   v[4];
        __m128  v4;
    };
};
int main()
{
    vec4 c, a = {5.0f, 2.0f, 1.0f, 3.0f}, b = {5.0f, 3.0f, 9.0f, 7.0f};
    c.v4 = _mm_add_ps(a.v4, b.v4);
    printf("c = {%.3f, %.3f, %.3f, %.3f}\n", c.v[0], c.v[1], c.v[2], c.v[3]);
    return 0;
}
```

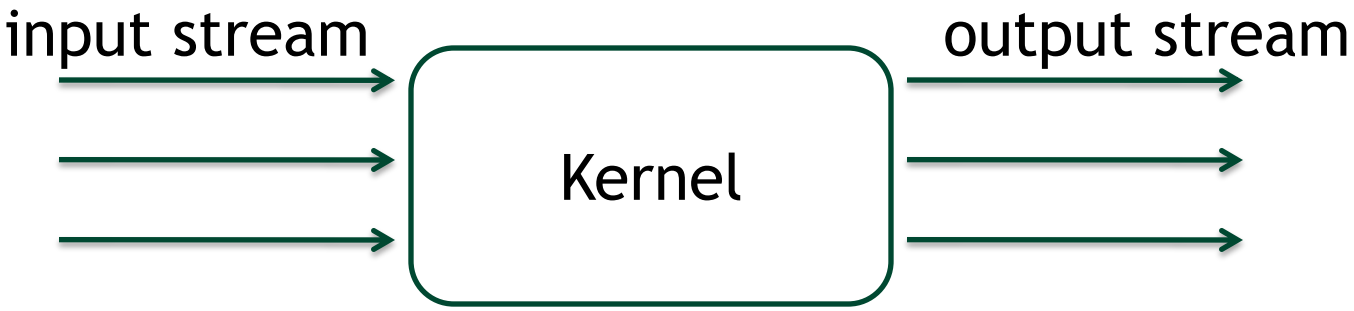
CPU

- Параллельное программирование CPU требует специалльных API
  - MPI, OpenMP
- Программирование ресурсов CPU ограничено
  - Multithreading
  - SSE
  - Ограничивает пропускная способность памяти

SIMD

- На входе поток однородных элементов, каждый из которых может быть обработан независимо
- На выходе – однородный поток
- Обработкой занимается ядро (kernel)
- Легко распараллеливается

SIMD



- Каждый элемент может быть обработан независимо от других
  - Их можно обрабатывать параллельно
- Можно соединять между собой отдельные ядра для получения более сложного конвеера обработки

План

- Существующие архитектуры
- Классификация
- CUDA
  - Программная модель
  - Связь программной модели с HW
  - SIMT
  - Язык CUDA C
  - Примеры для CUDA
- Несколько слов о курсе
- Дополнительные слайды

Эволюция GPU

- 3DFx Voodoo
  - Растеризация треугольников (вершины уже спроектированы CPU)
  - Буфер глубины
  - Текстурирование треугольников
  - Альфа-блендинг

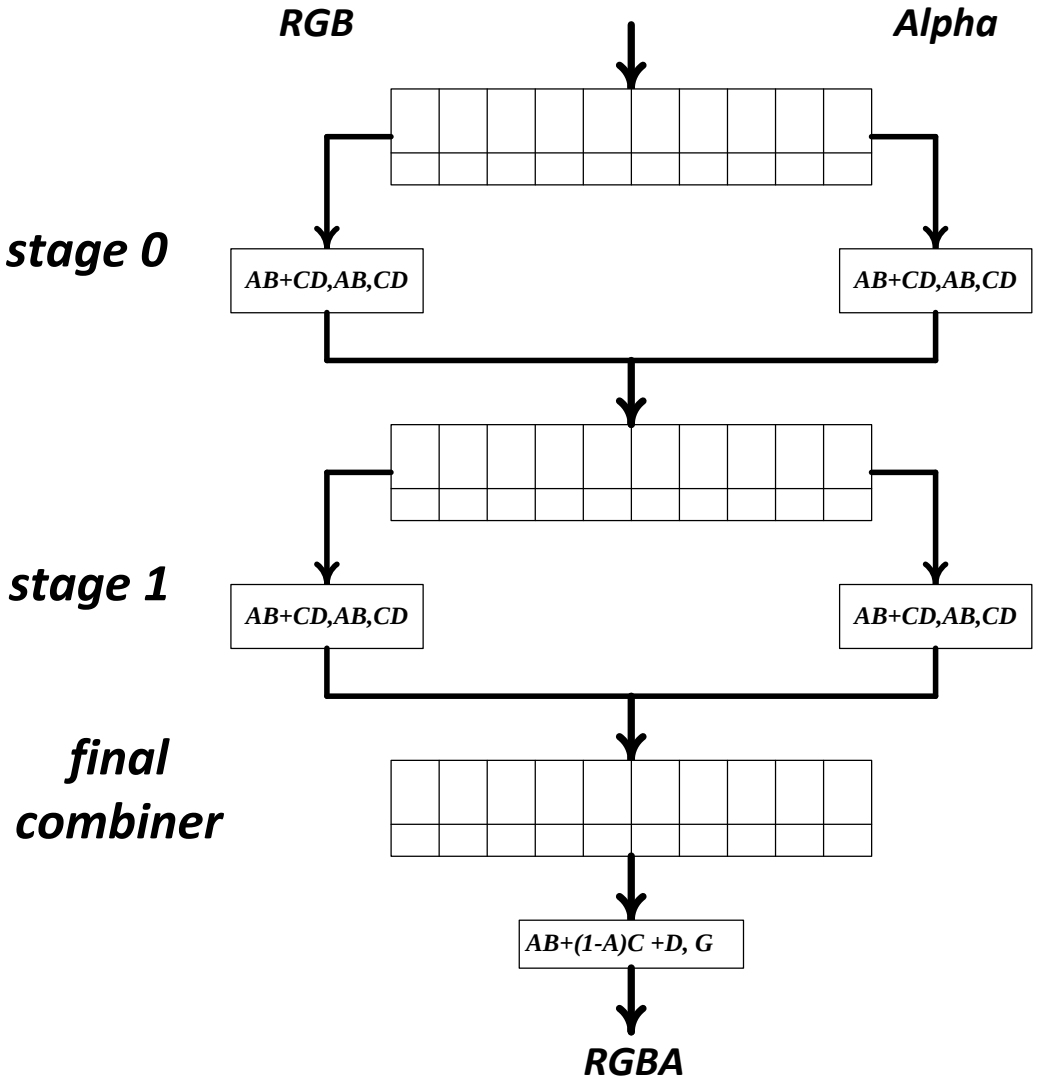
GPU как SIMD

- Потоки вершин и фрагментов
- Каждая вершина может обрабатываться независимо от других, но по одним и тем же правилам
- Каждый фрагмент может обрабатываться независимо от других, но по тем же правилам

Развитие GPU

- Увеличение быстродействия
  - Достаточно увеличить число вершинных и фрагментных блоков
- Увеличение гибкости
  - Хочется задавать законы обработки вершин и фрагментов

Развитие GPU, register combiners



Развитие GPU. Шейдеры

- Наибольшая гибкость – возможность задания программы для обработки (шейдера, shader)
- Первыми появились вершинные шейдеры (GeForce 2/3)
- Затем фрагментные (GeForceFX 5xxx)
- Изначально писались на специальном ассемблере

Пример вершинного шейдера

```
!!ARBvp1.0
ATTRIB pos     = vertex.position;
PARAM  mat [4] = { state.matrix.mvp };

# transform by concatenation of modelview and projection matrices

DP4 result.position.x, mat [0], pos;
DP4 result.position.y, mat [1], pos;
DP4 result.position.z, mat [2], pos;
DP4 result.position.w, mat [3], pos;

# copy primary color

MOV result.color, vertex.color;
END
```

Высокоуровневые шейдерные языки

```
varying   vec3 lt;
varying   vec3 ht;
uniform sampler2D tangentMap;
uniform sampler2D decalMap;
uniform sampler2D anisoTable;
void main (void)
{
   const vec4   specColor = vec4 ( 0, 0, 1, 0 );
   vec3    tang = normalize(2.0*texture2D(tangentMap, gl_TexCoord[0].xy).xyz - 1.0);
   float   dot1 = dot  ( normalize ( lt ), tang );
   float   dot2 = dot  ( normalize ( ht ), tang );
   vec2    arg  = vec2 ( dot1, dot2 );
   vec2    ds   = texture2D ( anisoTable, arg*arg ).rg;
   vec4    color = texture2D ( decalMap, gl_TexCoord [0].xy );

   gl_FragColor   = color * ds.x + specColor * ds.y;
}
```

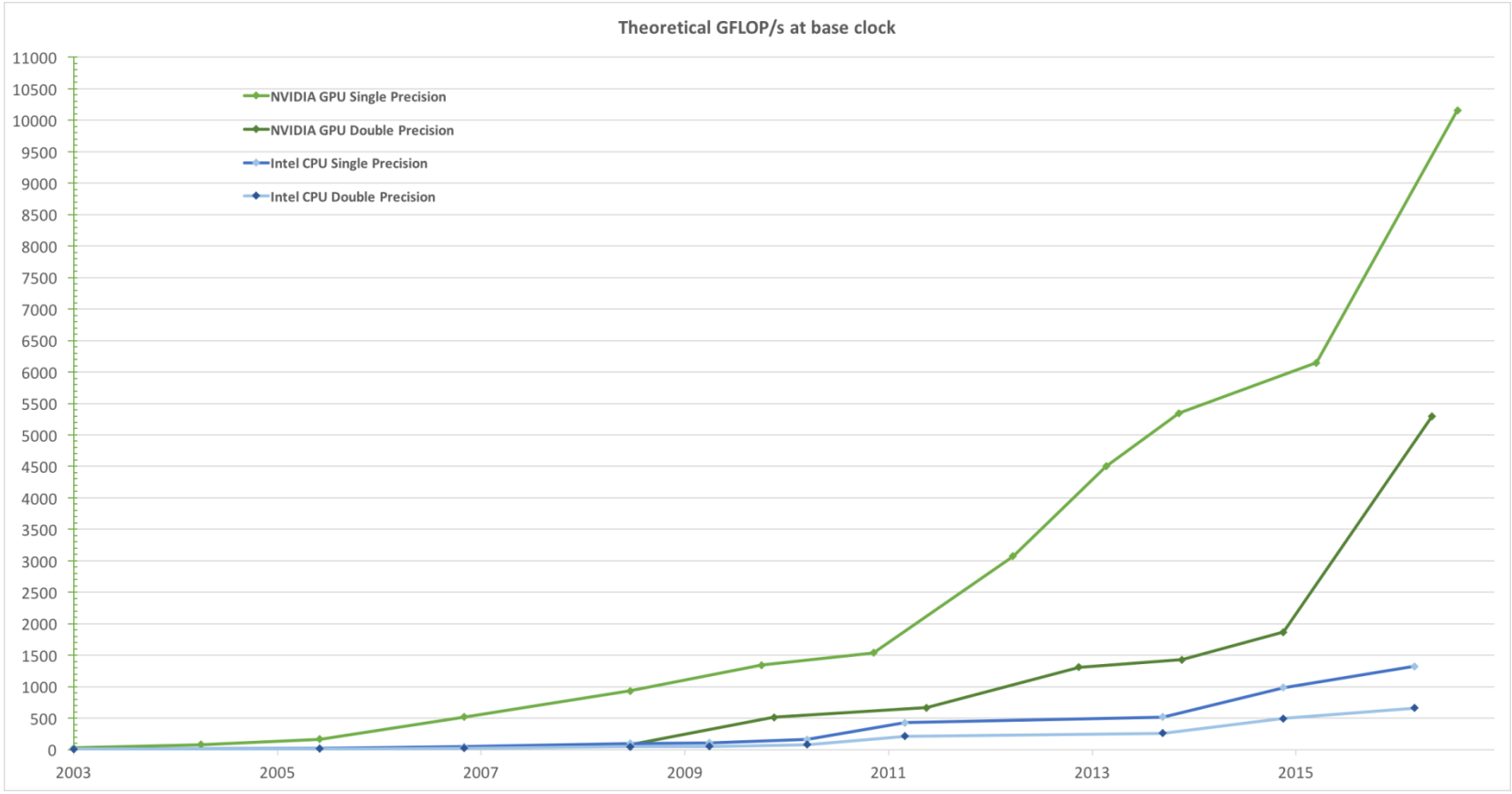
Высокоуровневые шейдерные языки – SPIR-V (OpenCL, Vulkan)

- Бинарный формат (есть и текстовое представление)
- Reference compiler
- Можно использовать любой язык, если его можно компилировать в SPIR-V
- Драйвер сам переводит SPIR-V в код для конкретного GPU

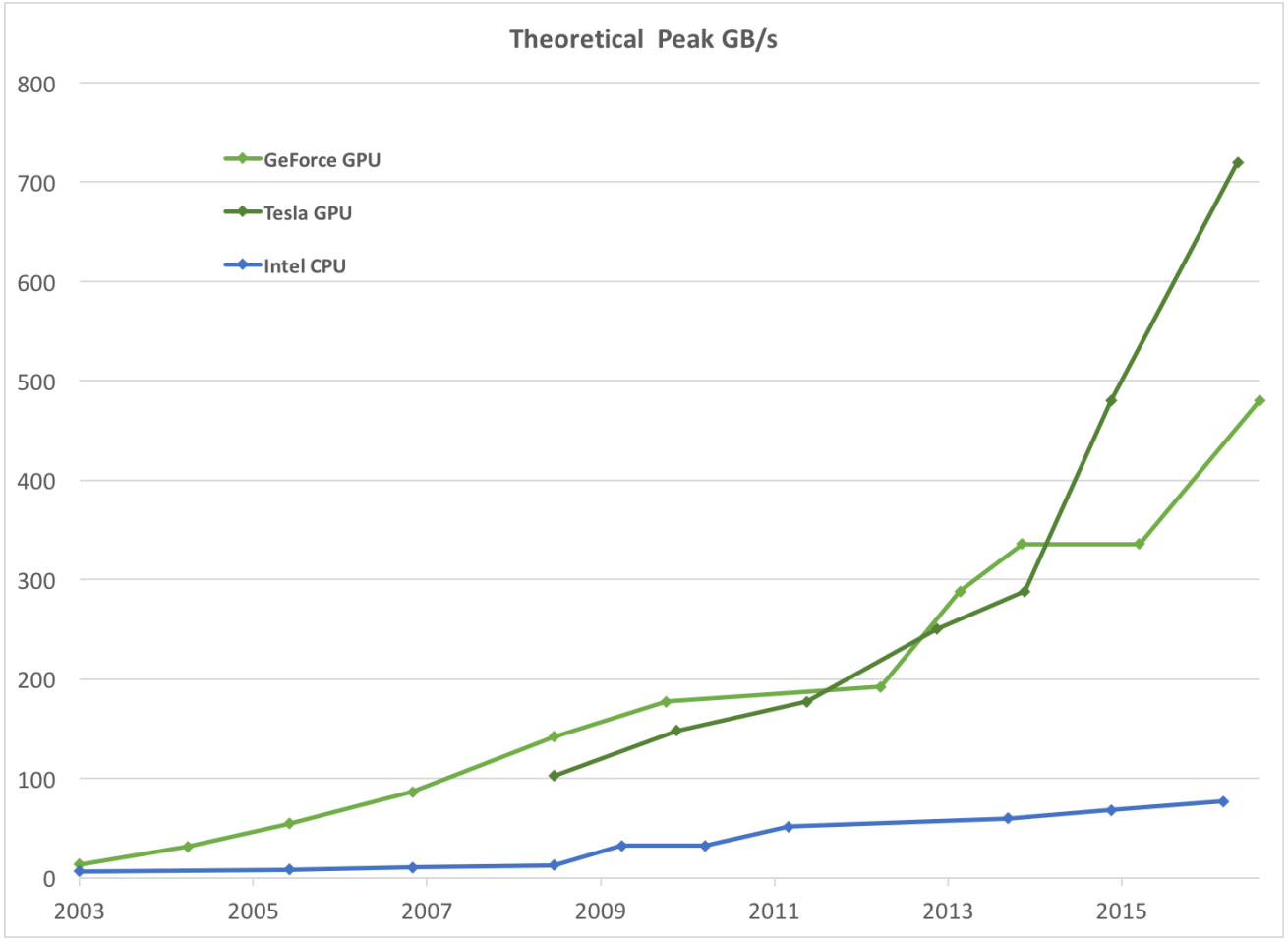
Развитие GPU

- Набор вершинных процессоров
- Набор фрагментных процессоров
- Поддержка текстур с компонентами типа float (полноценные многомерные массивы)
- Все операции выполняются над float’ми
- Итоговое быстродействие на порядок выше чем у CPU

Развитие GPU



Развитие GPU



GPGPU

- Использование вычислительной мощи GPU для решения неграфических задач
  - Сортировка
  - N-частиц
  - Быстрое преобразование Фурье
- Код пишется сразу на двух языках – традиционном (C++) и шейдерном (GLSL)

Ограничение GPGPU

- Необходимость использование графического API (OpenGL, DX)
- Отсутствует возможность взаимодействия между параллельно обрабатываемыми элементами
- Отсутствие поддержки операции типа *scatter* (scatter vs. gather)

Специализированные API для GPGPU

- CUDA
- OpenCL
- DX Compute shaders
- OpenGL compute shaders
- C++ AMP
- Vulkan (one API to rule them all ☺)

Специализированные API для GPGPU

- Основной материал дается на примере CUDA
  - Очень легко начать
  - Просто
  - Базовые концепции те же самые, что и в остальных API
- Также рассмотрим другие API
  - OpenCL
  - Vulkan

CUDA “Hello World”

```
#define   N    (1024*1024)
__global__ void kernel ( float * data )
{
  int   idx = blockIdx.x * blockDim.x + threadIdx.x;
  float x   = 2.0f * 3.1415926f * (float) idx / (float) N;
  data [idx] = sinf ( sqrtf ( x ) );
}

int main ( int argc, char *  argv [] )
{
  float a [N];
  float * dev = NULL;
  cudaMalloc ( (void**)&dev, N * sizeof ( float ) );
  kernel<<<dim3((N/512),1), dim3(512,1)>>> ( dev );
  cudaMemcpy ( a, dev, N * sizeof ( float ), cudaMemcpyDeviceToHost );
  cudaFree   ( dev );
  for (int idx = 0; idx < N; idx++)  printf("a[%d] = %.5f\n", idx, a[idx]);
  return 0;
}
```

Несколько слов о курсе

- Математический спецкурс 0.5 года
- Несколько практических заданий
  - Оценка ставится по практическим заданиям

Отчетность по курсу

- Практические задания
  - Задания сдаются на лекции
  - Либо по почте 
    - с темой **CUDA Assignment #**
    - В течении недели с момента публикации
  - Если у вас не получается – дайте нам знать
    - **Заранее**
- Альтернатива
  - Дайте нам знать
    - **Заранее**

Ресурсы нашего курса

- Steps3d.Narod.Ru
- Tesla.Parallel.Ru
- Twirpx.Com
- Nvidia.Ru
- Developer.nvidia.com

План

- Существующие архитектуры
- Классификация
- CUDA
- Несколько слов о курсе
- Дополнительные слайды
  - Эволюция GPU
  - Архитектура Tesla 8
  - Архитектура Tesla 20

Эволюция GPU

- Voodoo - растеризация треугольников, наложение текстуры и буфер глубины
- Очень легко распараллеливается
- На своих задачах легко обходил CPU

Эволюция GPU

- Быстрый рост производительности
- Добавление новых возможностей
  - Мультитекстурирование (RivaTNT2)
  - T&L
  - Вершинные программы (шейдеры)
  - Фрагментные программы (GeForceFX)
  - Текстуры с floating point-значениями

Эволюция GPU: Шейдеры

- Работают с 4D float-векторами
- Специальный ассемблер
- Компилируется драйвером устройства
- Отсутствие переходов и ветвления
  - Вводились как vendor-расширения

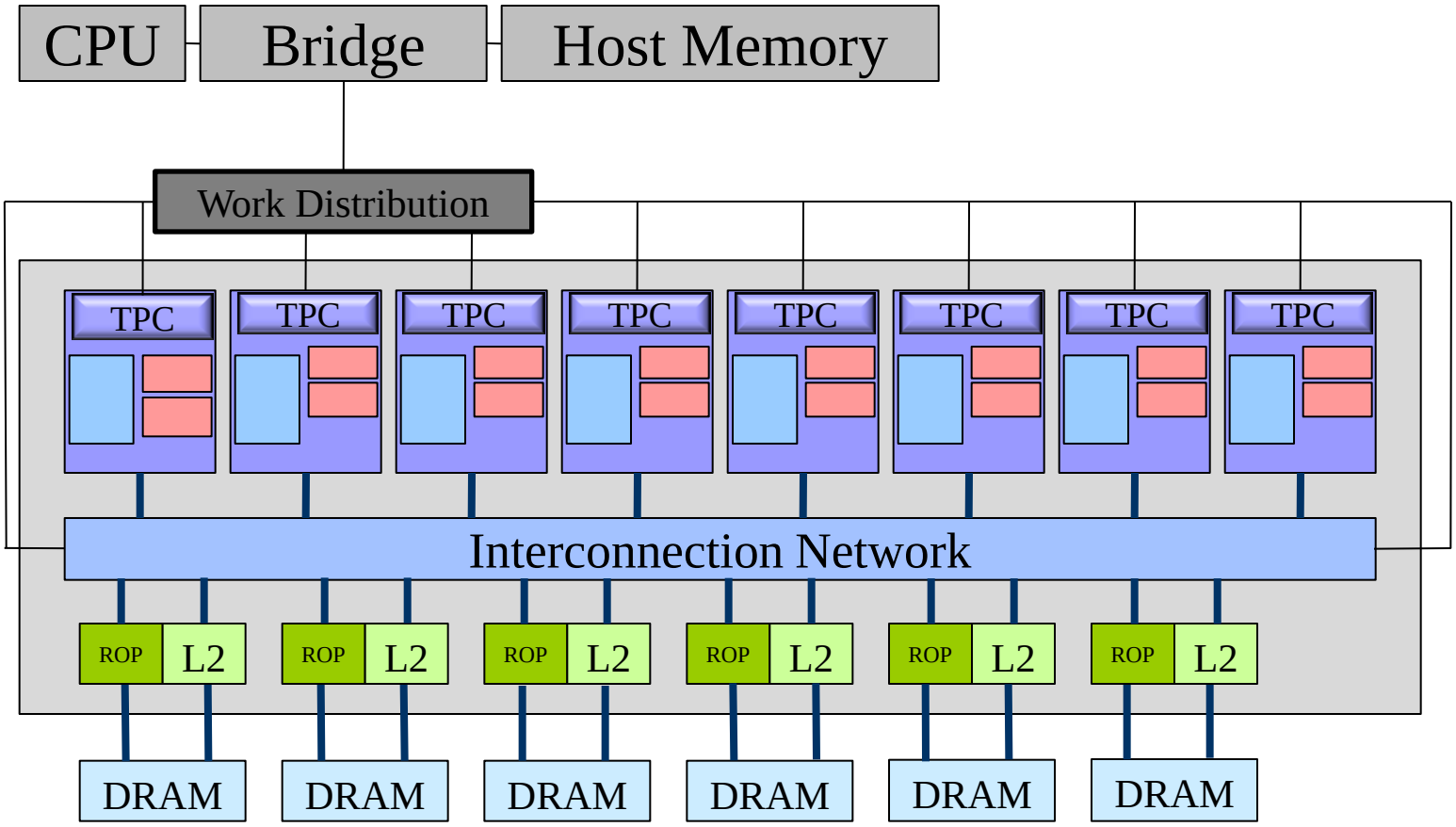
GPGPU

- Использование GPU для решения не графических задач
- Вся работа с GPU идет через графический API (OpenGL, D3D)
- Программы используют сразу два языка – один традиционный (C++) и один шейдерный
- Ограничения, присущие графическим API

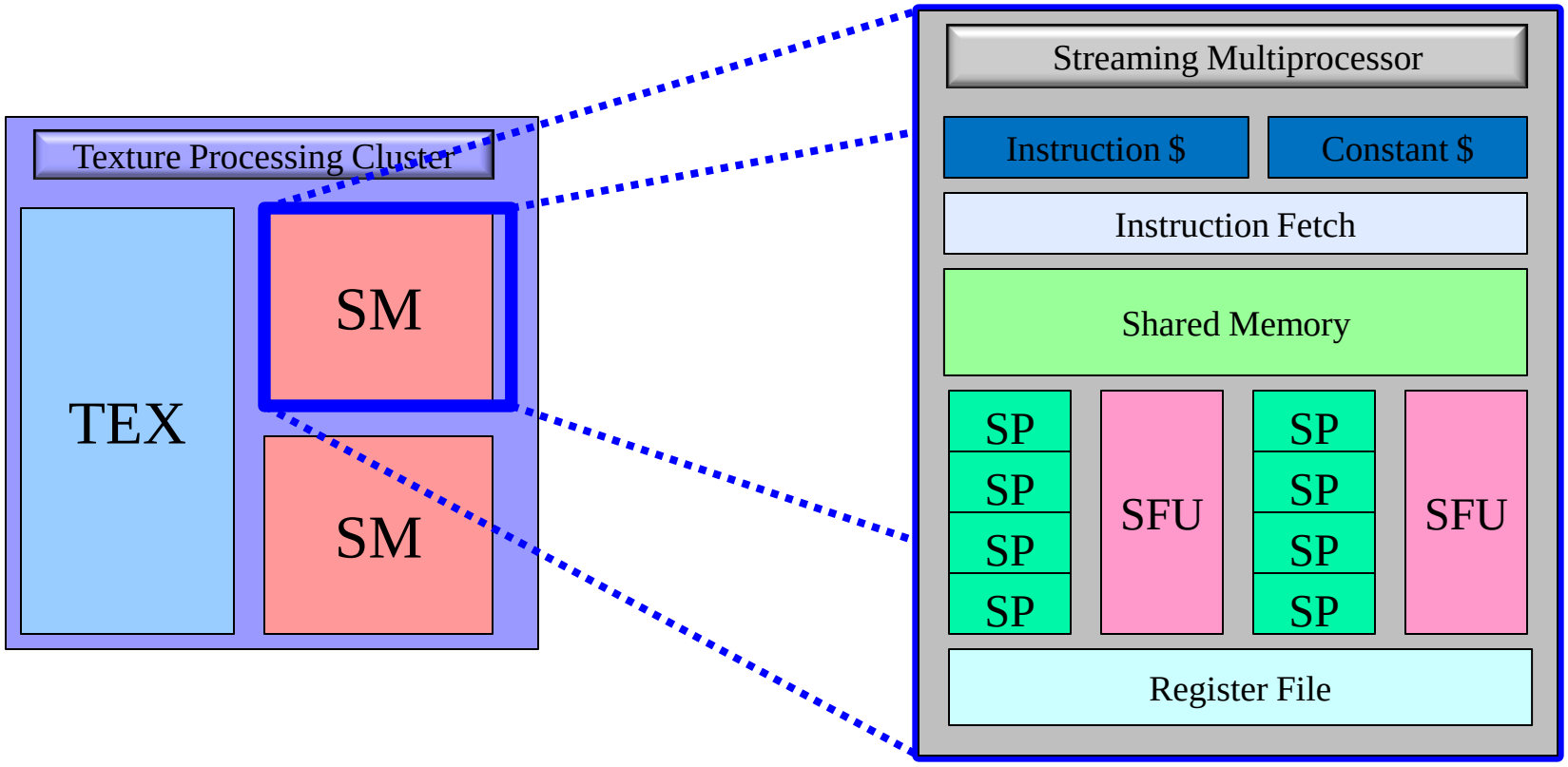
Эволюция GPU: Шейдеры

- Появление шейдерных языков высокого уровня (Cg, GLSL, HLSL)
- Поддержка ветвлений и циклов (GeForce 6xxx)
- Появление GPU, превосходящие CPU в 10 и более раз по Flop’ам

Архитектура Tesla 8



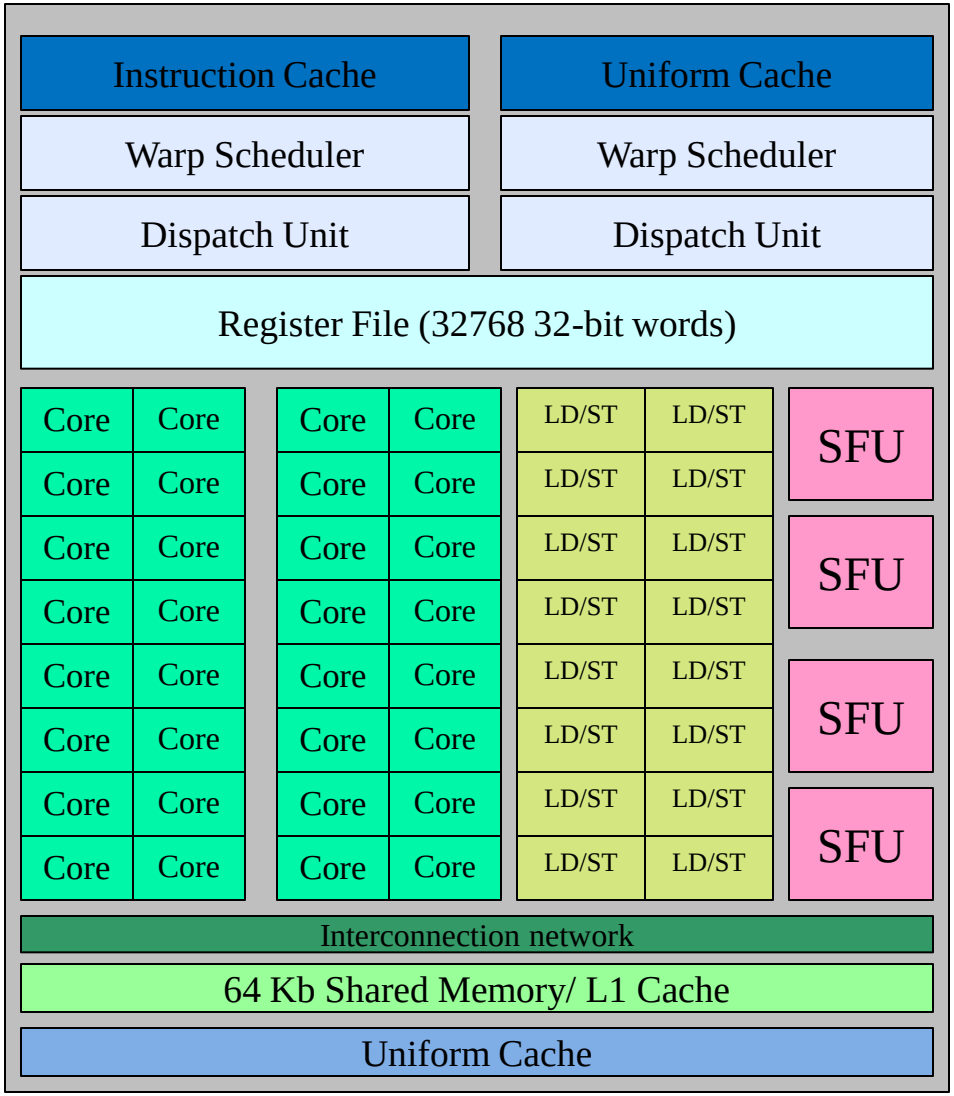
Архитектура Tesla: Мультипроцессор Tesla 8



Архитектура Tesla 20

- Объединенный L2 кэш (768 Kb)
- До 1 Tb памяти (64-битная адресация)
- Общее адресное пространство памяти
- ККО (DRAM, регистры, разделяемая память, кэш)
- Одновременное исполнение ядер, копирования памяти (CPU->GPU, GPU->CPU)
- Быстрая смена контекста (10x)
- Одновременное исполнение ядер (до 16)

Архитектура Tesla 20
Потоковый мультипроцессор



Архитектура Tesla 20

- 32 ядра на SM
- Одновременное исполнение 2х варпов.
- 48 Kb разделяемой памяти
- 16 Kb кэш
  - или 16 Kb разделяемй + 48 Kb кэш
- Дешевые атомарные операции