

# Архитектура и программирование массивно- параллельных вычислительных систем

## Лекторы:

[Боресков А.В. \(ВМиК МГУ\)](#)

[Харламов А. \(NVIDIA\)](#)

# План

- Существующие архитектуры
- Классификация
- CUDA
- Несколько слов о курсе
- Дополнительные слайды

# План

- Существующие архитектуры
  - Intel CPU
  - SMP
  - CELL
  - BlueGene
  - NVIDIA Tesla 10
- 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 E6700, 2.66 GHz
- 2007 г. - Core 2 Duo E6800, 3 GHz
- 2008 г. - Core 2 Duo E8600, 3.33 Ghz
- 2009 г. - Core i7 950, 3.06 GHz

# Существующие многоядерные системы

- Роста частоты практически нет
  - Энерговыведение ~ четвертой степени частоты
  - Ограничения техпроцесса
  - Одноядерные системы зашли в тупик

# Существующие многоядерные системы

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

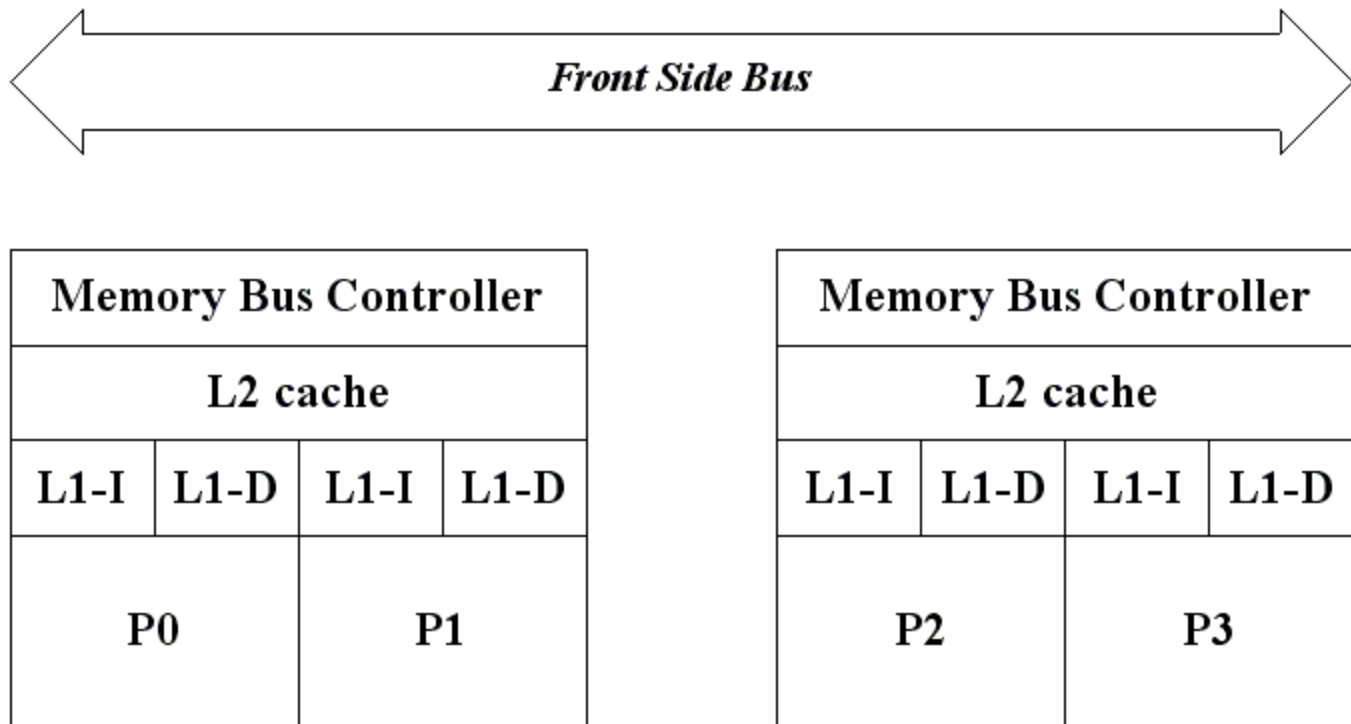
# Intel Core 2 Duo

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



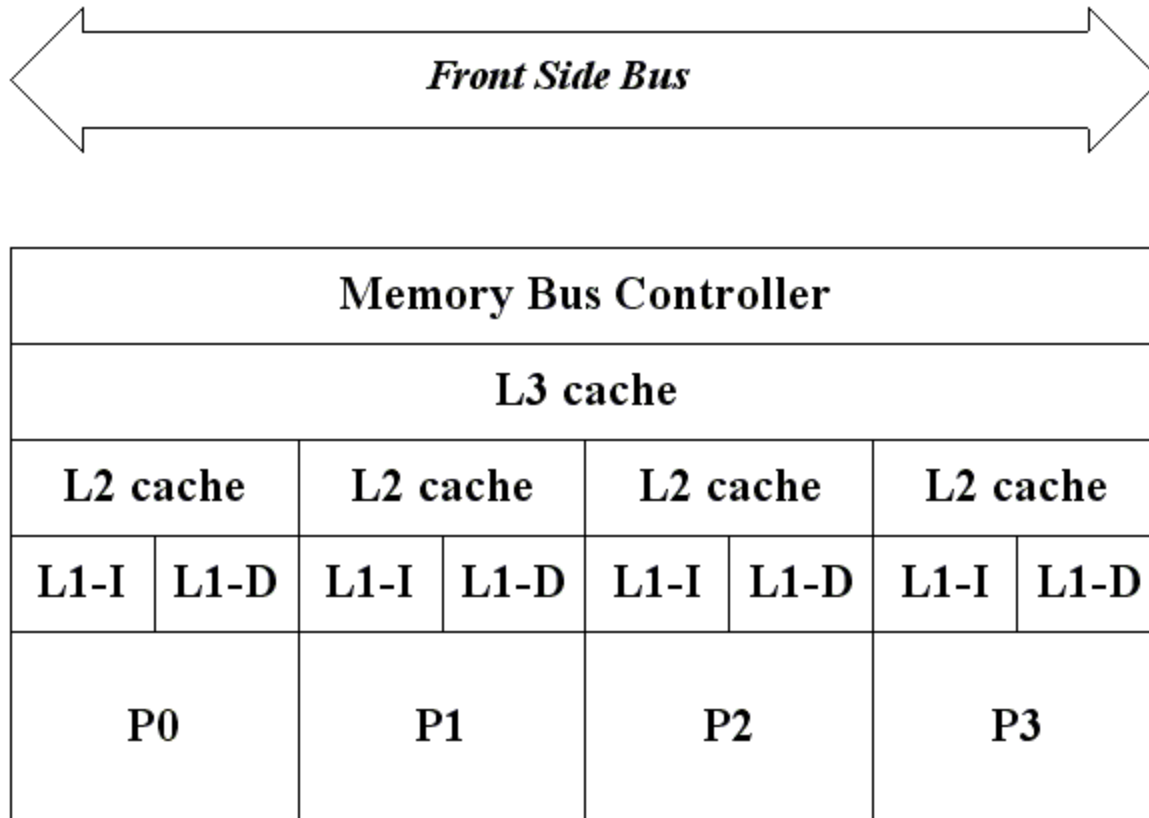
Memory Bus Controller			
L2 cache			
L1-I	L1-D	L1-I	L1-D
P0		P1	

# Intel Core 2 Quad

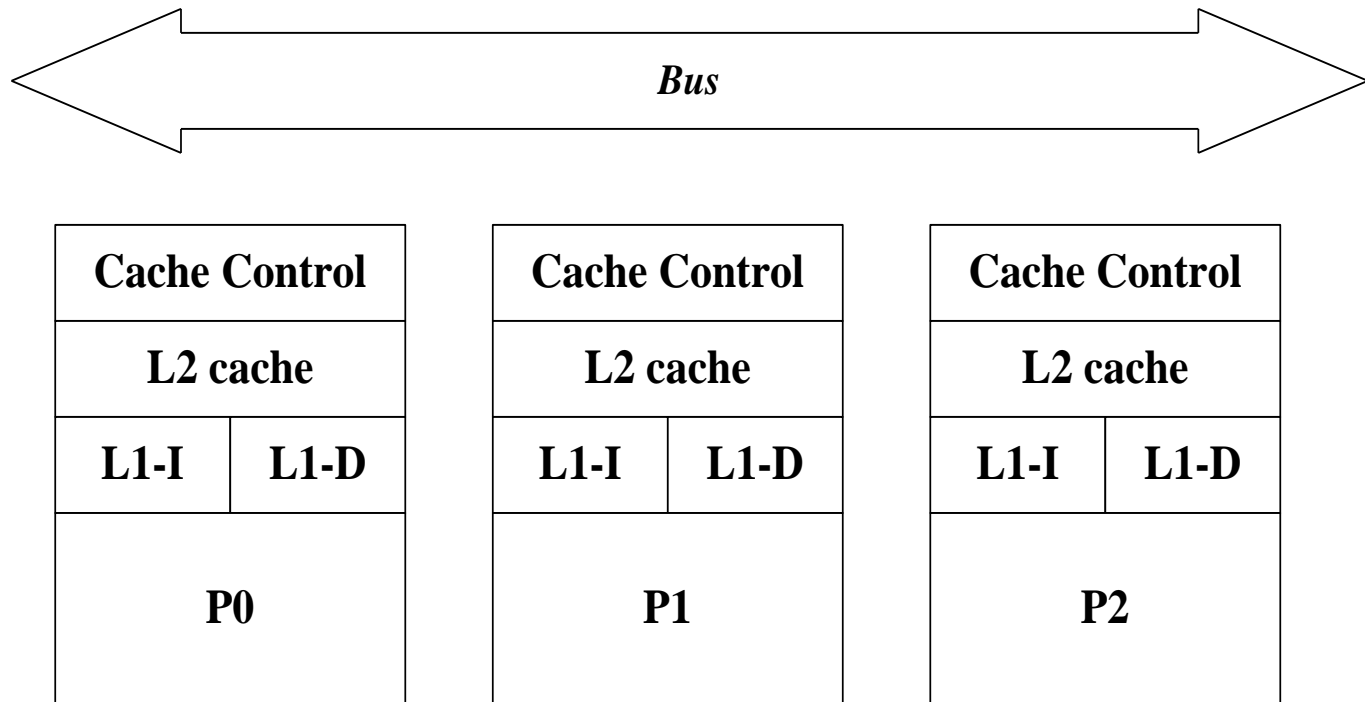




# Intel Core i7



# Symmetric Multiprocessor Architecture (SMP)

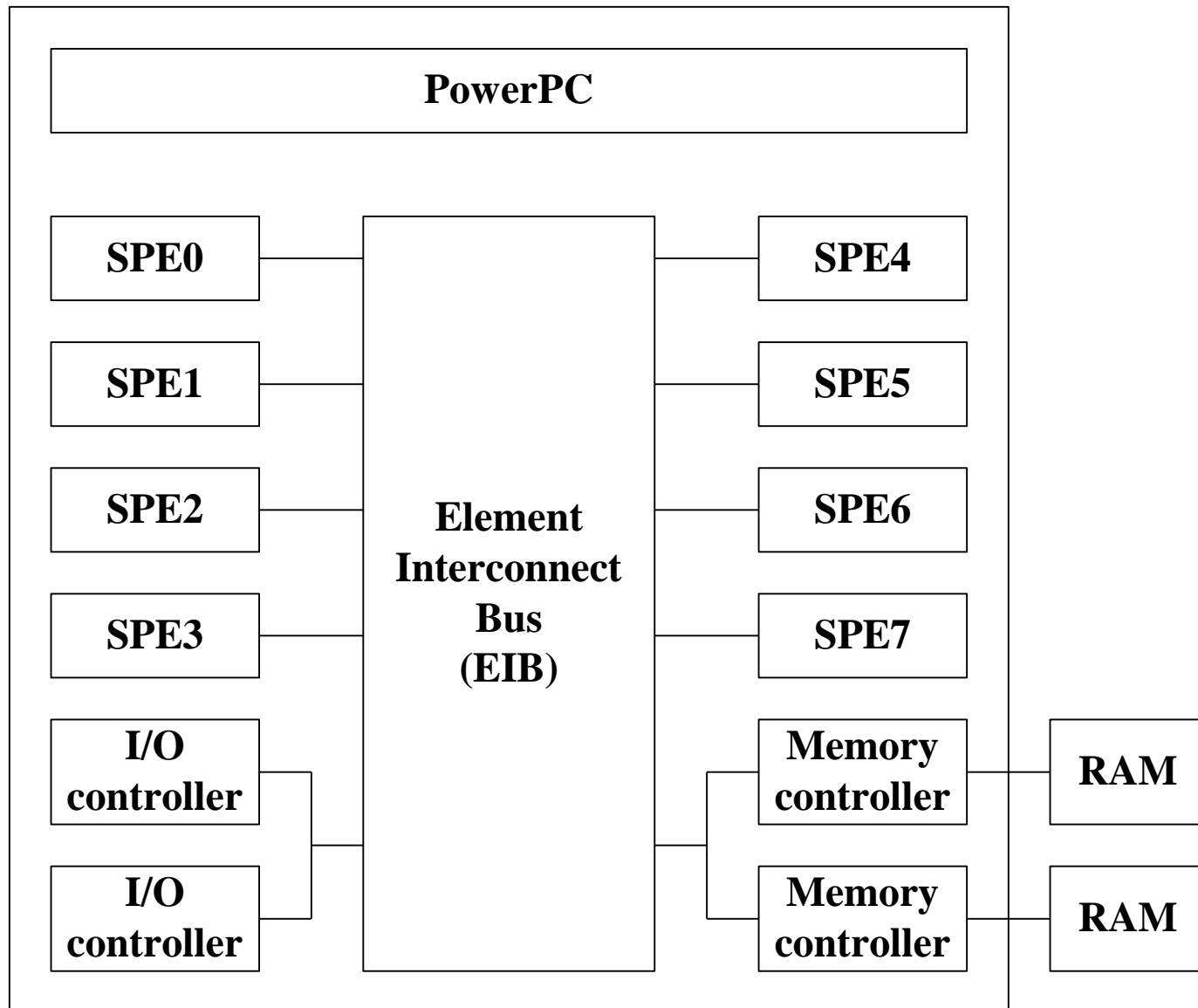


# Symmetric Multiprocessor Architecture (SMP)

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)

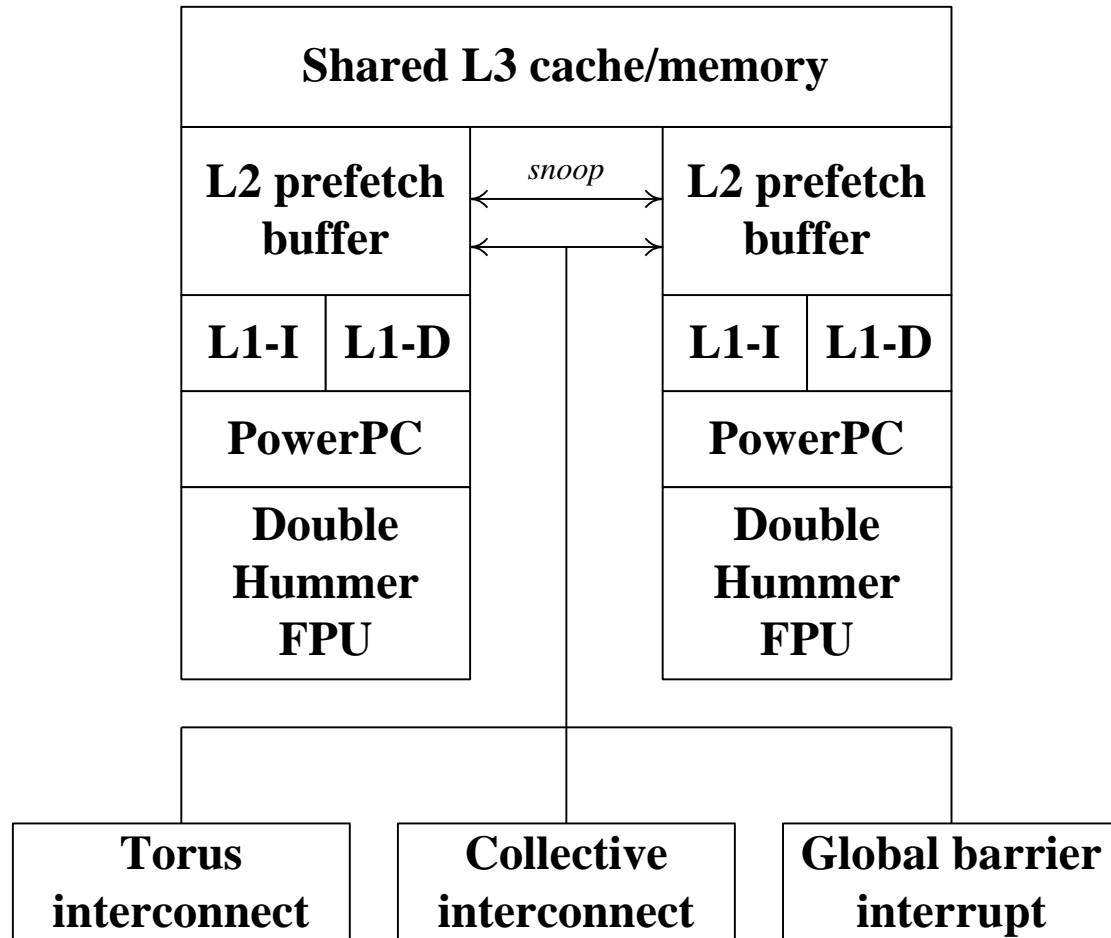
# Cell



# Cell

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

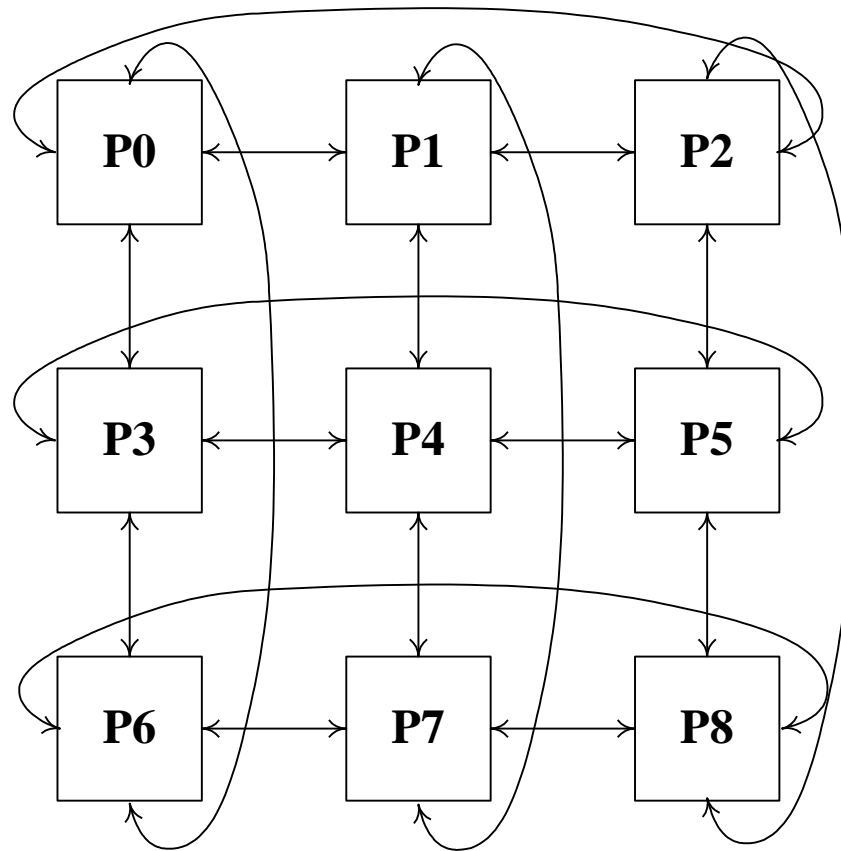
# BlueGene/L



# BlueGene/L

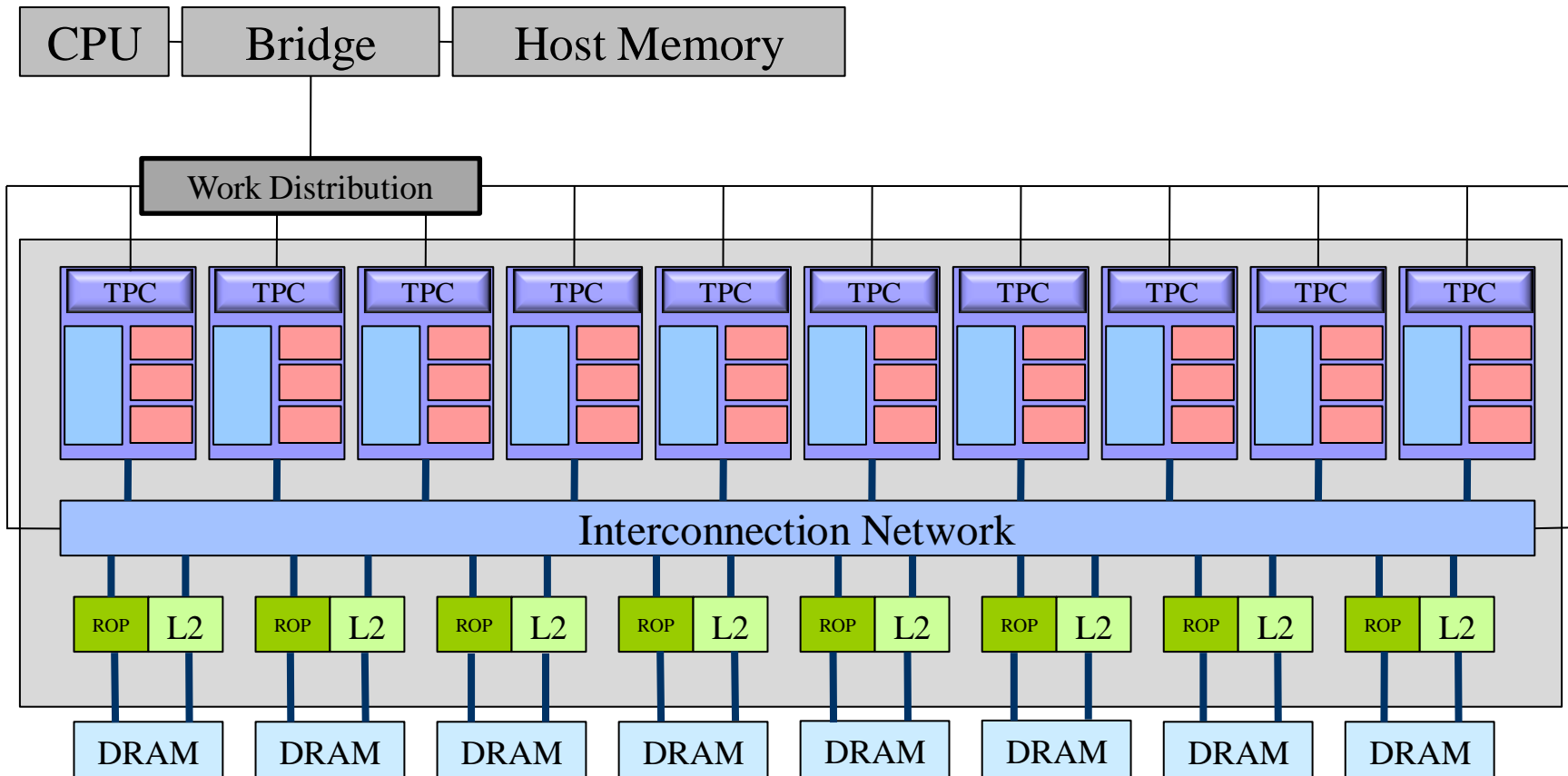
- 65536 dual-core nodes
- node
  - 770 Mhz PowerPC
  - Double Hammer FPU (4 Flop/cycle)
  - 4 Mb on-chip L3 кэш
  - 512 Mb off-chip RAM
  - 6 двухсторонних портов для 3D-тора
  - 3 двухсторонних порта для collective network
  - 4 двухсторонних порта для barrier/interrupt

# BlueGene/L



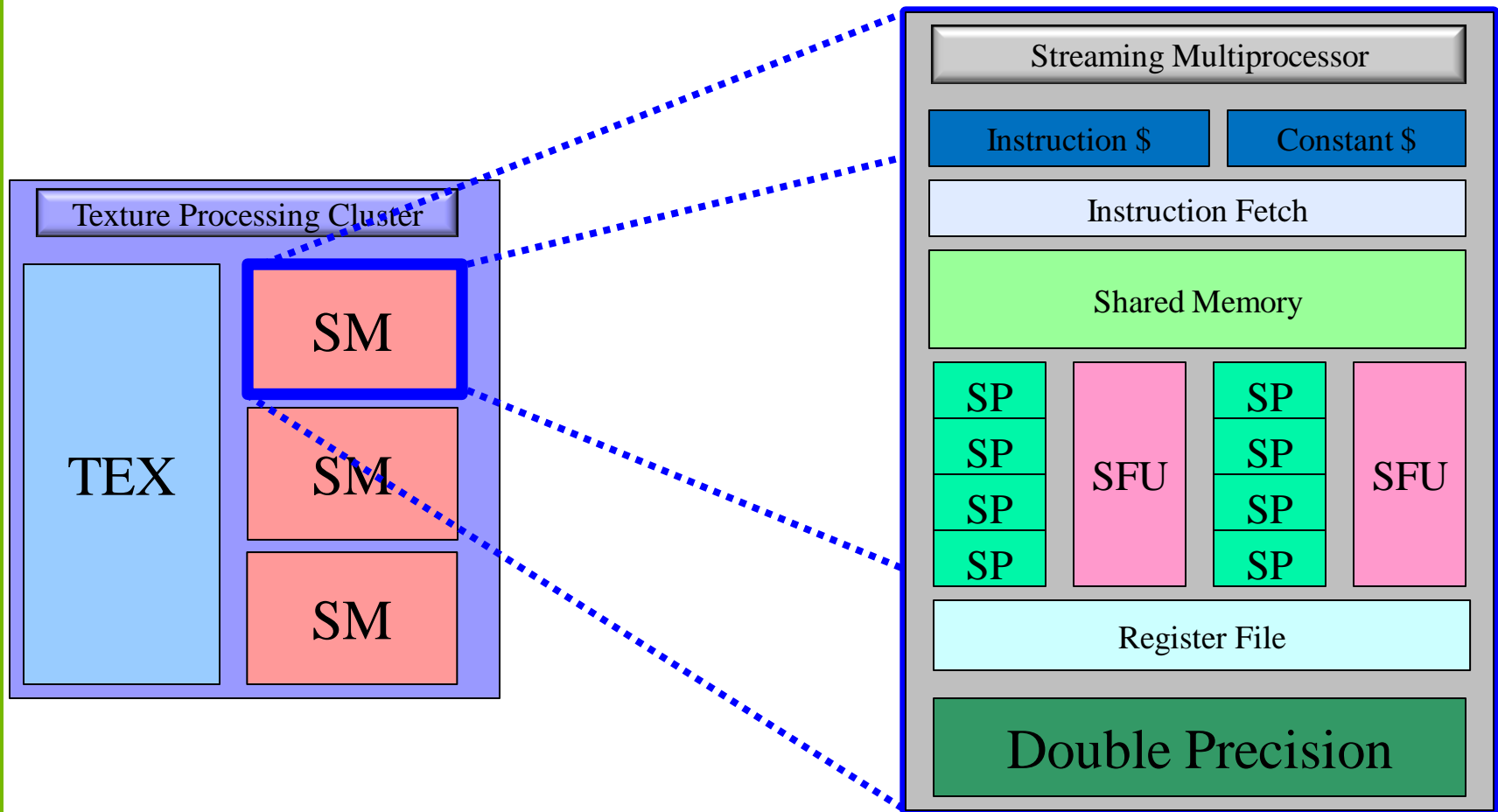


# Архитектура Tesla 10



# 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'а
    - Кол-во регистров на SM
    - Т.д.

# План

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

# Классификация

	<b>Single Instruction</b>	<b>Multiple Instruction</b>
<b>Single Data</b>	SISD	MISD
<b>Multiple Data</b>	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\*

# MultiThreading “Hello World”

```
#include <stdio.h>

#include <windows.h>

#include <process.h>    // для beginthread()

void mtPrintf( void * pArg );

int main()
{
    int t0 = 0; int t1 = 1;

    _beginthread(mtPrintf, 0, (void*)&t0 );

    mtPrintf( (void*)&t1);

    Sleep( 100 );

    return 0;
}

void mtPrintf( void * pArg )
{
    int * pIntArg = (int *) pArg;

    printf( "The function was passed %d\n", (*pIntArg) );
}
```

# MultiThreading “Hello World”

```
// создание нового потока
// необходимо указать:
// entry point функцию,
// размер стека, при 0 – OS выберет сама
// (void *) – указатель на аргументы функции
_beginthread(mtPrintf, 0, (void*)&t1 );

// напечатать из основного потока
mtPrintf( (void*)&t0 );

// подождать 100 мс
// создание потока windows требует времени
// если основной поток закончит выполнение
// то и все дочерние потоки будут прерваны
Sleep( 100 );
```



# 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;
}
```

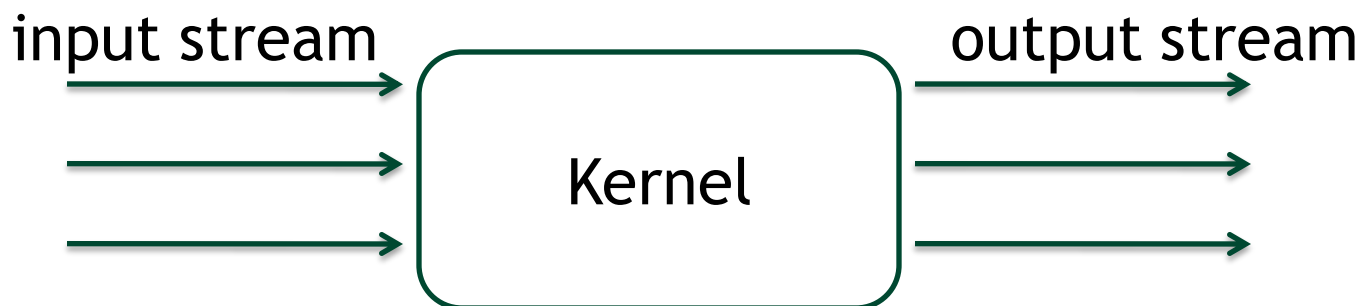
# CPU

- Параллельное программирование CPU требует специальных 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)

# SIMD



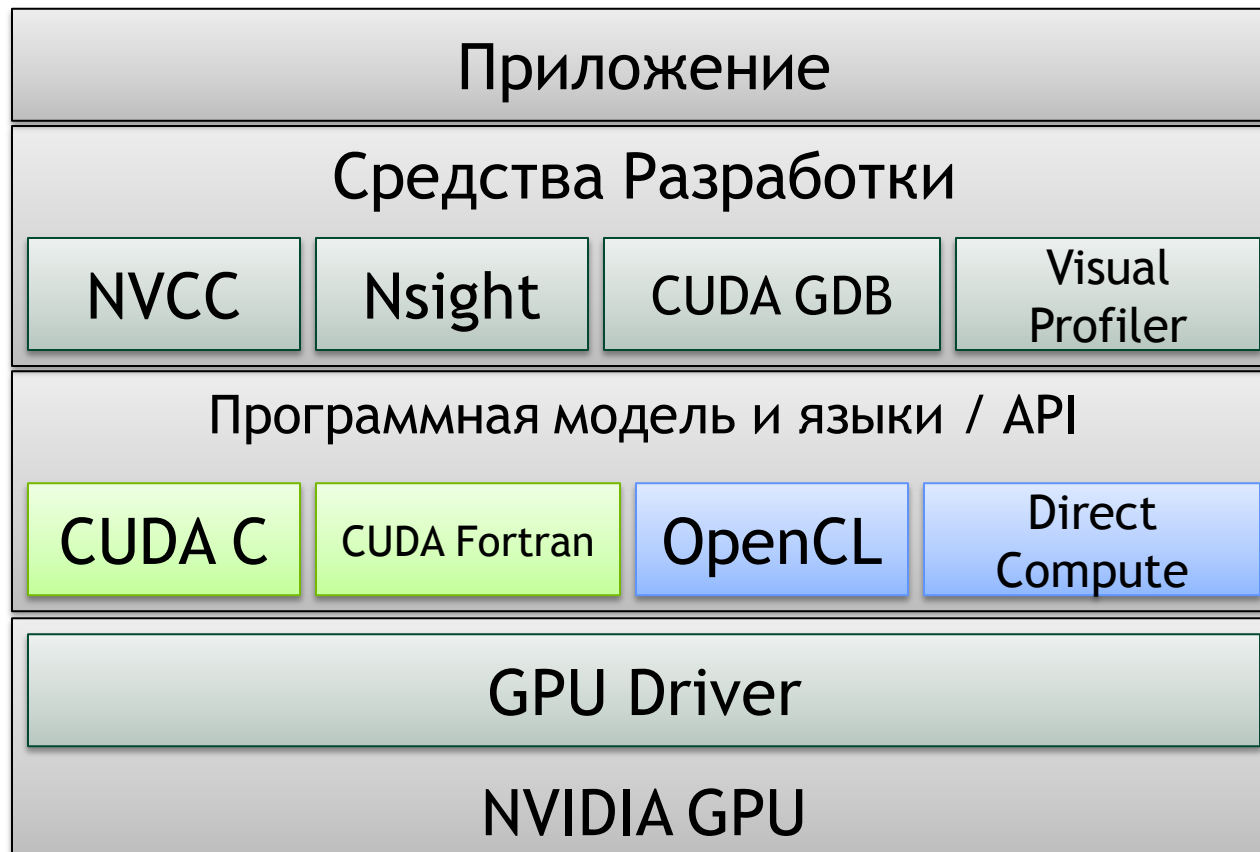
- Каждый элемент может быть обработан независимо от других
  - Их можно обрабатывать параллельно
- Можно соединять между собой отдельные ядра для получения более сложного конвейера обработки

# 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

# Compute Unified Device Architecture

- CUDA - программно-аппаратный стек для программирования GPU



# Программная модель CUDA

- Код состоит из последовательных и параллельных частей
- Последовательные части кода выполняются на CPU (*host*)
- Массивно-параллельные части кода выполняются на GPU (*device*)
  - Является сопроцессором к CPU (*host*)
  - Имеет собственную память (DRAM)
  - Выполняет одновременно **очень много** нитей

# Программная модель CUDA

- Параллельная часть кода выполняется как большое количество нитей (*threads*)
- Нити группируются в блоки (*blocks*) фиксированного размера
- Блоки объединяются в сеть блоков (*grid*)
- Ядро выполняется на сетке из блоков
- Каждая нить и блок имеют свой уникальный идентификатор



# Программная модель CUDA

- Десятки тысяч нитей

```
for ( int ix = 0; ix < nx; ix++ )  
{  
    pData[ix] = f(ix);  
}
```

```
for ( int ix = 0; ix < nx; ix++ ){  
    for ( int iy = 0; iy < ny; iy++ )  
    {  
        pData[ix+iy*nx] = f(ix)*g(iy);  
    }  
}
```

```
for ( int ix = 0; ix < nx; ix++ ){  
    for ( int iy = 0; iy < ny; iy++ ){  
        for ( int iz = 0; iz < nz; iz++ )  
        {  
            pData[ix+(iy+iz*ny)*nx] = f(ix)*g(iy)*h(iz);  
        }  
    }  
}
```

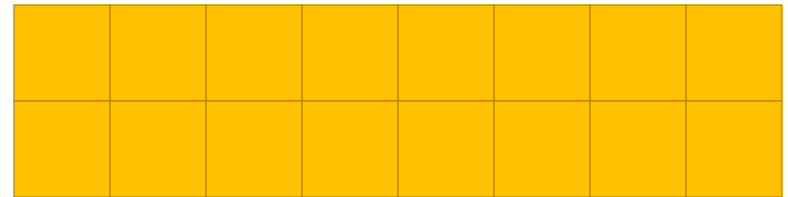
# Программная модель CUDA

- Нити в CUDA объединяются в блоки:

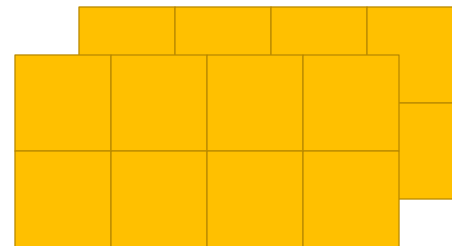
- 1D топология блока



- 2D топология блока



- 3D топология блока



- Общее кол-во нитей в блоке ограничено
- В текущем HW это 512\* нитей

\* В Tesla 20 ограничение на 1024 нити в блоке

# Программная модель CUDA

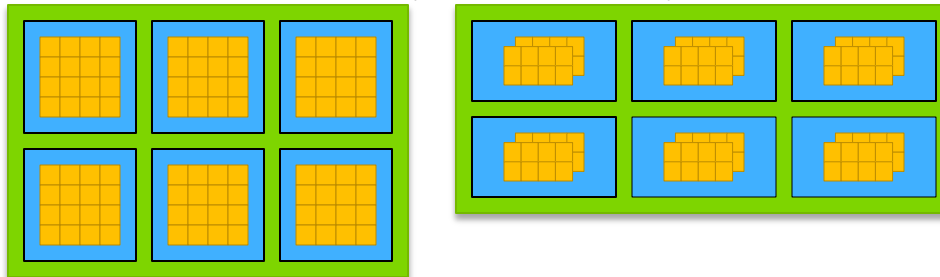
- Блоки могут использовать *shared* память
  - Нити могут обмениваться общими данными
- Внутри блока потоки могут синхронизоваться

# Программная модель CUDA

- Блоки потоков объединяются в сеть (*grid*) блоков потоков
  - 1D топология сетки блоков потоков






- 2D топология сетки блоков потоков



- Блоки в сети выполняются независимо друг от друга

Легенда:

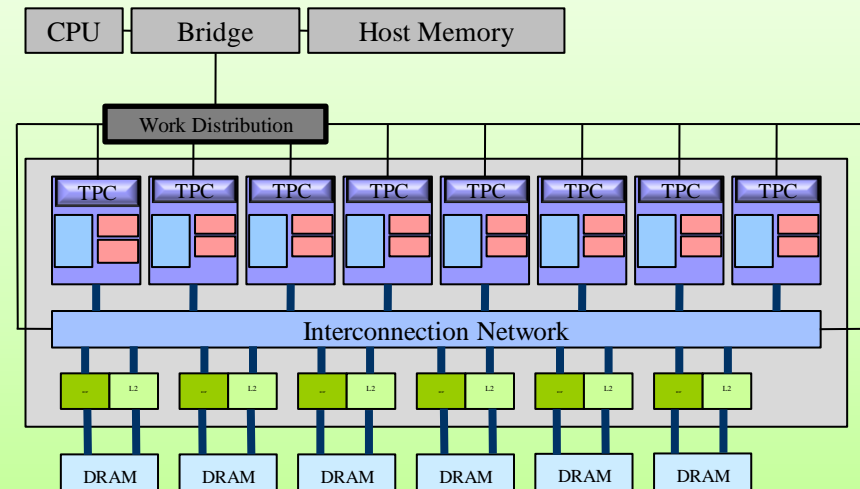
-нить   
-блок   
-сеть 

# Связь программной модели с HW

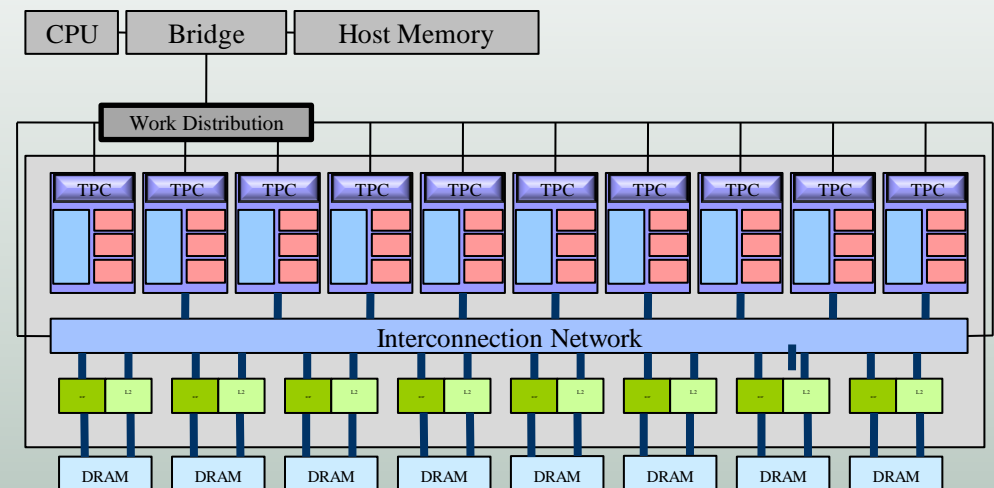
- Блоки могут использовать *shared* память
  - Т.к. блок целиком выполняется на одном SM
  - Объем *shared* памяти ограничен и зависит от HW
- Внутри блока нити могут синхронизоваться
  - Т.к. блок целиком выполняется на одном SM
- Масштабирование архитектуры и производительности

# Масштабирование архитектуры Tesla

Tesla 8

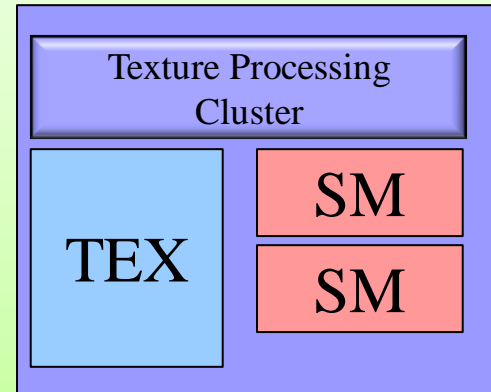


Tesla 10

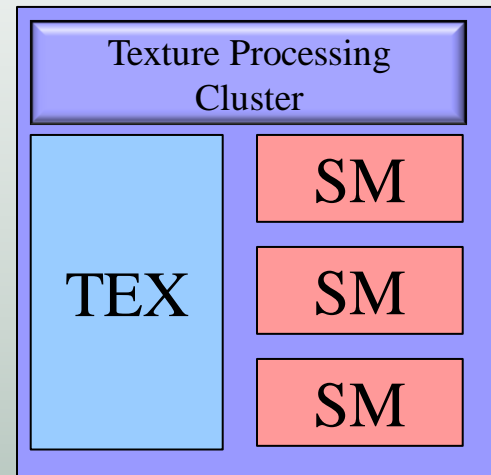


# Масштабирование мультипроцессора Tesla 10

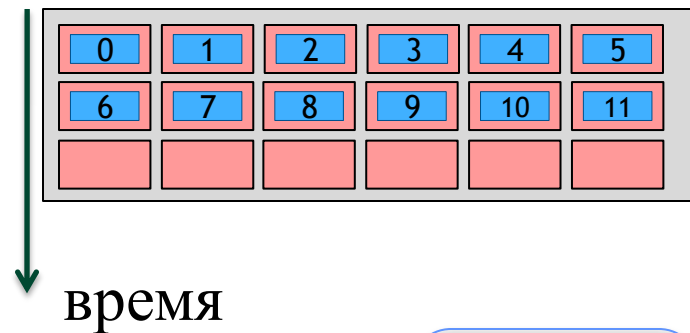
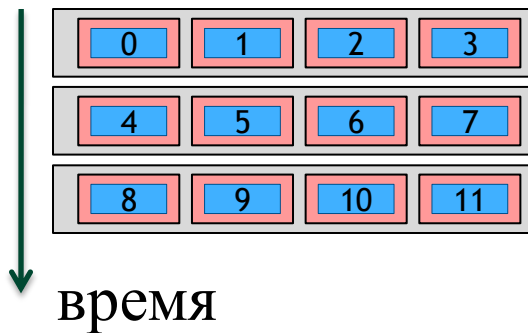
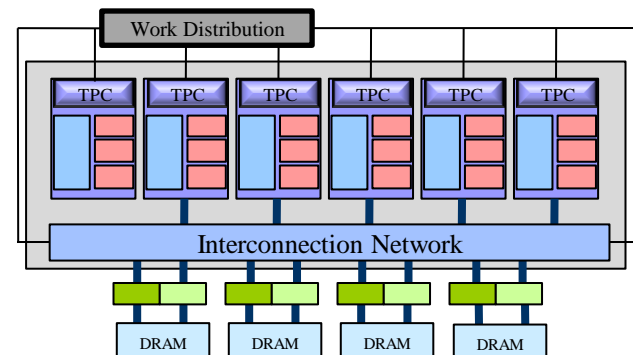
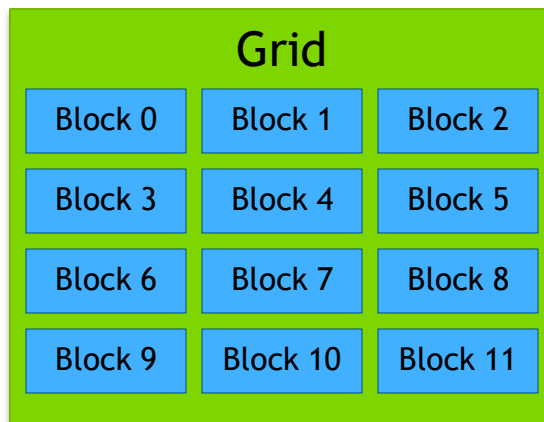
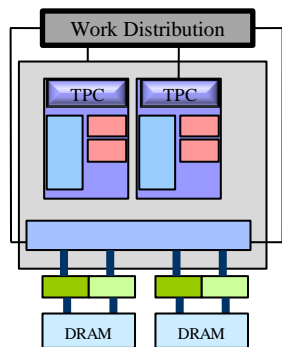
Tesla 8



Tesla 10



# Масштабирование производительности



Легенда:

-блок

-сеть

-SM

-HW планировщик



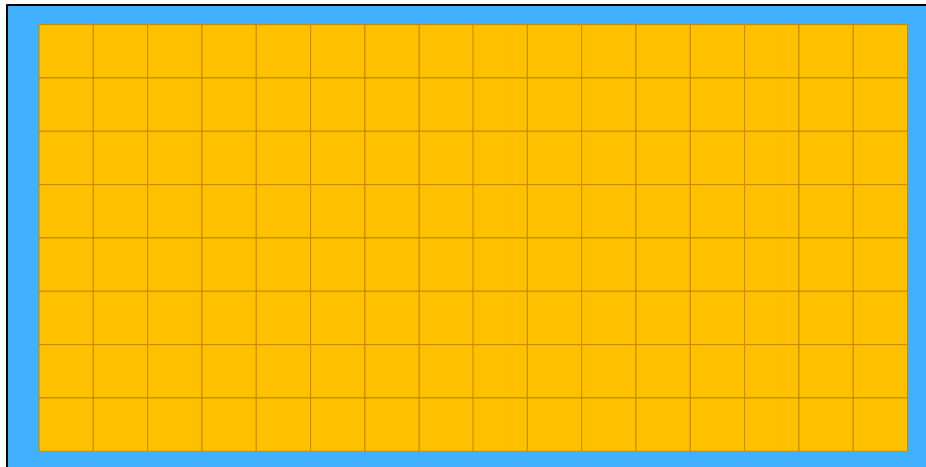


# Связь программной модели с HW

- Очень высокая степень параллелизма
  - Десятки тысяч потоков на чипе
  - Потоки на GPU очень «легкие»
  - HW планировщик задач
- Основная часть чипа занята логикой, а не кэшем
- Для полноценной загрузки GPU нужны тысячи потоков
  - Для покрытия латентностей операций чтения / записи
  - Для покрытия латентностей sfu инструкций

# Блоки и warp'ы?




- Блоки - абстракция программной модели
- Warp - реальная единица исполнения HW



Warp 0
Warp 1
Warp 2
Warp 3

Half-warp 1
Half-warp 2
Half-warp 1
Half-warp 2
Half-warp 1
Half-warp 2
Half-warp 1
Half-warp 2

Легенда:

-нить   
-блок   
-сеть 

# Single Instruction Multiple Threads (SIMT)

- Параллельно на каждом SM выполняется большое число отдельных нитей (*threads*)
- Нити в пределах одного *warp*'а выполняются физически параллельно (SIMD)
- Разные *warp*'ы могут исполнять разные команды
- Большое число *warp*'ов покрывает латентность

# Язык CUDA C

- CUDA C - это расширение языка C/C++
  - спецификаторы для функций и переменных
  - новые встроенные типы
  - встроенные переменные (внутри ядра)
  - директива для запуска ядра из C кода
- Как скомпилировать CUDA код
  - nvcc компилятор
  - .cu расширение файла

# Язык CUDA C

## Спецификаторы

### Спецификатор функций

Спецификатор	Выполняется на	Может вызываться из
__device__	device	device
__global__	device	host
__host__	host	host

### Спецификатор переменных

Спецификатор	Находится	Доступна	Вид доступа
__device__	device	device	RW
__constant__	device	device / host	R / W
__shared__	device	block	RW / __syncthreads()

# Язык CUDA C

## Спецификаторы

- Спецификатор `__global__` соответствует ядру
  - Может возвращать только `void`
- Спецификаторы `__host__` и `__device__` могут использоваться одновременно
  - Компилятор сам создаст версии для CPU и GPU
- Спецификаторы `__global__` и `__host__` не могут быть использованы одновременно

# Язык CUDA C

## Ограничения

- Ограничения на функции, выполняемые на GPU:
  - Нельзя брать адрес (за исключением `__global__`)
  - Не поддерживается рекурсия
  - Не поддерживаются `static`-переменные внутри функции
  - Не поддерживается переменное число входных аргументов

# Язык CUDA C

## Ограничения

- Ограничения на спецификаторы переменных:
  - Нельзя применять к полям структуры или `union`
  - Не могут быть `extern`
  - Запись в `__constant__` может выполнять только CPU через специальные функции
  - `__shared__` - переменные не могут инициализироваться при объявлении



# Язык CUDA C

## Типы данных

- Новые типы данных:
  - 1/2/3/4-мерные вектора из базовых ТИПОВ
    - (u)char, (u)int, (u)short, (u)long, longlong
    - float, double
  - dim3 – uint3 с нормальным конструкторов, позволяющим задавать не все компоненты
    - Не заданные инициализируются единицей

# Язык CUDA C

## Встроенные переменные

Сравним CPU vs CUDA C код:

```
float * data;  
for ( int i = 0; i < n; i++ )  
{  
    data [x] = data[i] + 1.0f;  
}
```

Пусть  $n_x = 2048$   
Пусть в блоке 256  
ПОТОКОВ

→ кол-во блоков =  
 $2048 / 256 = 8$

```
__global__ void incKernel ( float * data )  
{  
    [ 0 .. 7 ]      [ == 256 ]      [ 0 .. 255 ]  
    int idx = blockIdx.x * blockDim.x + threadIdx.x;  
    data [idx] = data [idx] + 1.0f;  
}
```

# Язык CUDA C

## Встроенные переменные

- В любом CUDA kernel'е доступны:

- `dim3` `gridDim;`
- `uint3` `blockIdx;`
- `dim3` `blockDim;`
- `uint3` `threadIdx;`
- `int` `warpSize;`

`dim3` – встроенный тип, который используется для задания размеров kernel'а  
По сути – это `uint3`.

# Язык CUDA C

## Директивы запуска ядра

- Как запустить ядро с общим кол-во тредов равным nx?

```
float * data;  
dim3 threads ( 256 );  
dim3 blocks ( nx / 256 );  
incKernel<<<blocks, threads>>> ( data );
```

<<< , >>> угловые скобки, внутри которых задаются параметры запуска ядра

# Язык CUDA C

## Директивы запуска ядра

- Общий вид команды для запуска ядра  
`incKernel<<<bl, th, ns, st>>> ( data );`
- *bl* – число блоков в сетке
- *th* – число нитей в сетке
- *ns* – количество дополнительной shared-памяти, выделяемое блоку
- *st* – поток, в котором нужно запустить ядро

# Как скомпилировать CUDA код

- NVCC – компилятор для CUDA
  - Основными опциями команды `nvcc` являются:
  - `--use_fast_math` - заменить все вызовы стандартных математических функций на их быстрые (но менее точные) аналоги
  - `-o <outputFileName>` - задать имя выходного файла
- CUDA файлы обычно носят расширение `.cu`

# 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;
}
```

# CUDA “Hello World”

```
__global__ void kernel ( float * data )
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x; // номер текущей нити
    float x = 2.0f * 3.1415926f * idx / N; // значение аргумента
    data [idx] = sinf ( sqrtf ( x ) ); // найти значение и записать в массив
}
```

- Для каждого элемента массива (всего N) запускается отдельная нить, вычисляющая требуемое значение.
- Каждая нить обладает уникальным id



# CUDA “Hello World”

```
float    a [N];
float * dev = NULL;

        // выделить память на GPU под N элементов
cudaMalloc ( (void**)&dev, N * sizeof ( float ) );

        // запустить N нитей блоками по 512 нитей
        // выполняемая на нити функция - kernel
        // массив данных - dev
kernel<<<dim3((N/512),1), dim3(512,1)>>> ( dev );

        // скопировать результаты из памяти GPU (DRAM) в
        // память CPU (N элементов)
cudaMemcpy ( a, dev, N * sizeof ( float ), cudaMemcpyDeviceToHost );

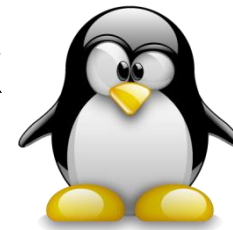
        // освободить память GPU
cudaFree   ( dev   );
```

# План


- Существующие архитектуры
- Классификация
- CUDA
- Несколько слов о курсе
  - Отчетность по курсу
  - Ресурсы нашего курса
- Дополнительные слайды

# Несколько слов о курсе

- Математический спецкурс
- 5 семинарских занятий
  - Раз в две недели
  - Цель занятий:
    - Начать быстро программировать на CUDA
    - Написать и сдать практические задания
  - На удаленной машине \*nix
    - Тренируйте shell-skill
- 5 практических заданий



# Отчетность по курсу

- 5 практических заданий
  - Задания сдаются на семинаре
  - 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**

# Отчетность по курсу

- Если тема email отличается от **CUDA Assignment #**
  - MINOR FAIL
- Если ваш код не собирается или не запускается
  - MAJOR FAIL
- Если обнаруживается дубликат
  - EPIC FAIL

Недвусмысленный  
намек от кэпа



# Отчетность по курсу

- Если вы не сдадите задания и не предупредите **заранее**  
— FAIL
- Если вы выбрали альтернативу, но нас не предупредили **заранее**  
— EPIC FAIL

Недвусмысленный  
намек от кэпа



# Ресурсы нашего курса

- [Steps3d.Narod.Ru](#)
- [Google Site CUDA.CS.MSU.SU](#)
- [Google Group CUDA.CS.MSU.SU](#)
- [Google Mail CS.MSU.SU](#)
- [Google SVN](#)
- [Tesla.Parallel.Ru](#)
- [Twirpx.Com](#)
- [Nvidia.Ru](#)





# 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

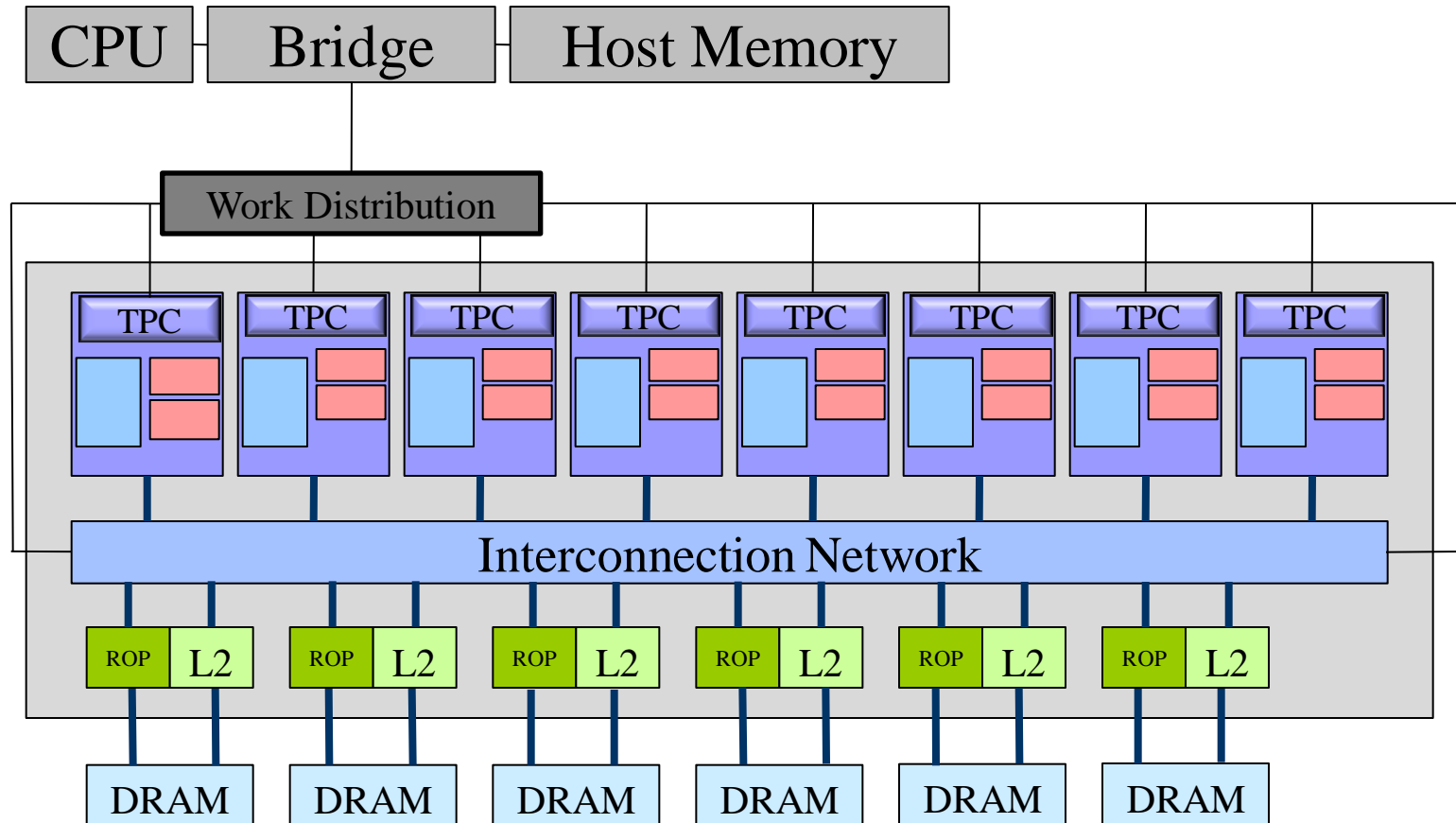
# GPGPU

- 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

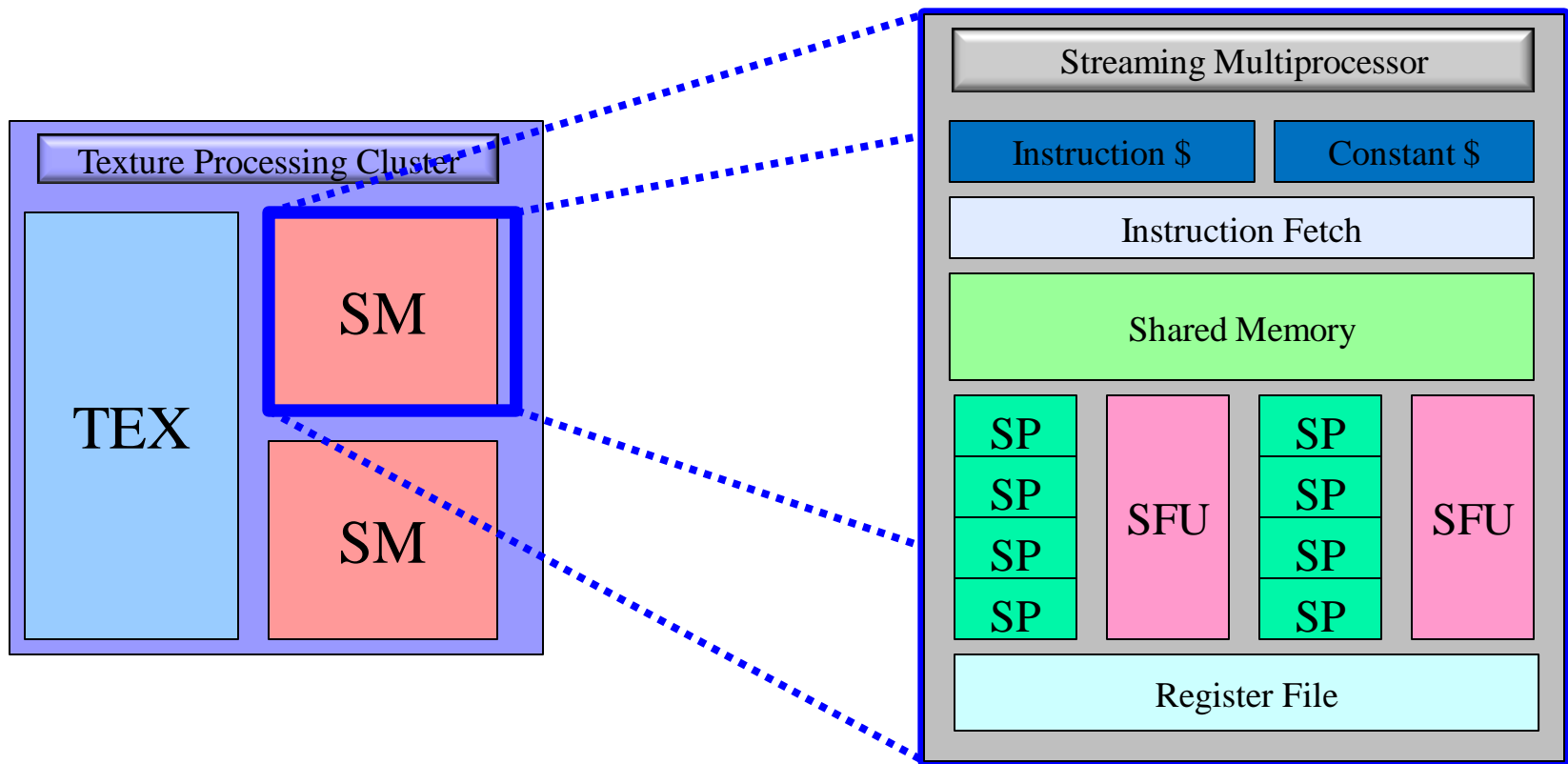
- Появление шейдерных языков высокого уровня (Cg, GLSL, HLSL)
- Поддержка ветвлений и циклов (GeForce 6xxx)
- Появление GPU, превосходящие CPU в 10 и более раз по Flop'ам

# Архитектура Tesla 8



# Архитектура Tesla:

## Мультипроцессор Tesla 8



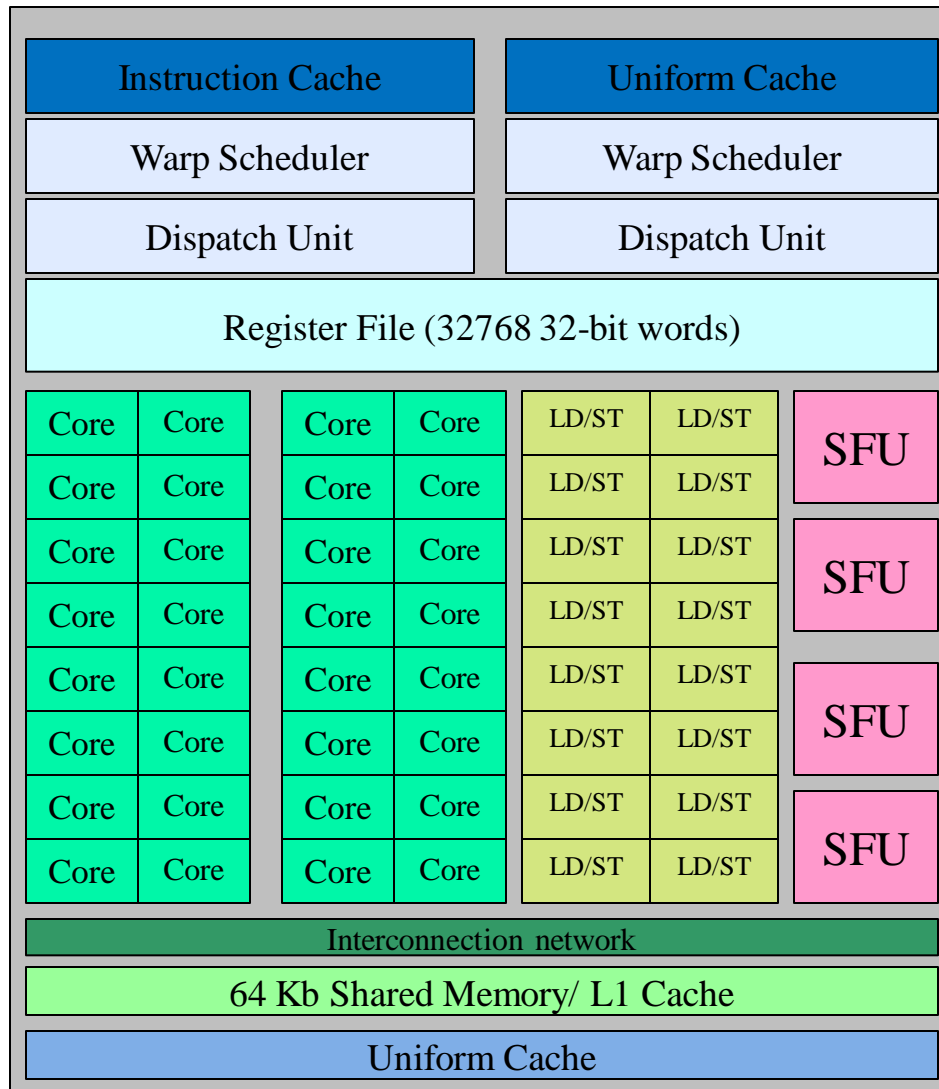


# Архитектура Tesla 20

- 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