

Архитектура и программирование массивно-параллельных вычислительных систем

⌘ Лекторы:

☑ Боресков А.В. (ВМиК МГУ)

☑ Харламов А. (NVidia)

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



Легко видно, что роста частоты
практически нет

- ☒ Энерговыведение \sim четвертой степени частоты
- ☒ Ограничения техпроцесса
- ☒ Одноядерные системы зашли в тупик

Существующие многоядерные системы



- ⌘ Таким образом, повышение быстродействия следует ждать именно от параллельности.
- ⌘ Уже давно CPU используют параллельную обработку для повышения производительности
 - ☑ Конвейер
 - ☑ Multithreading
 - ☑ SSE

Intel Core 2 Duo



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

- ⌘ 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



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

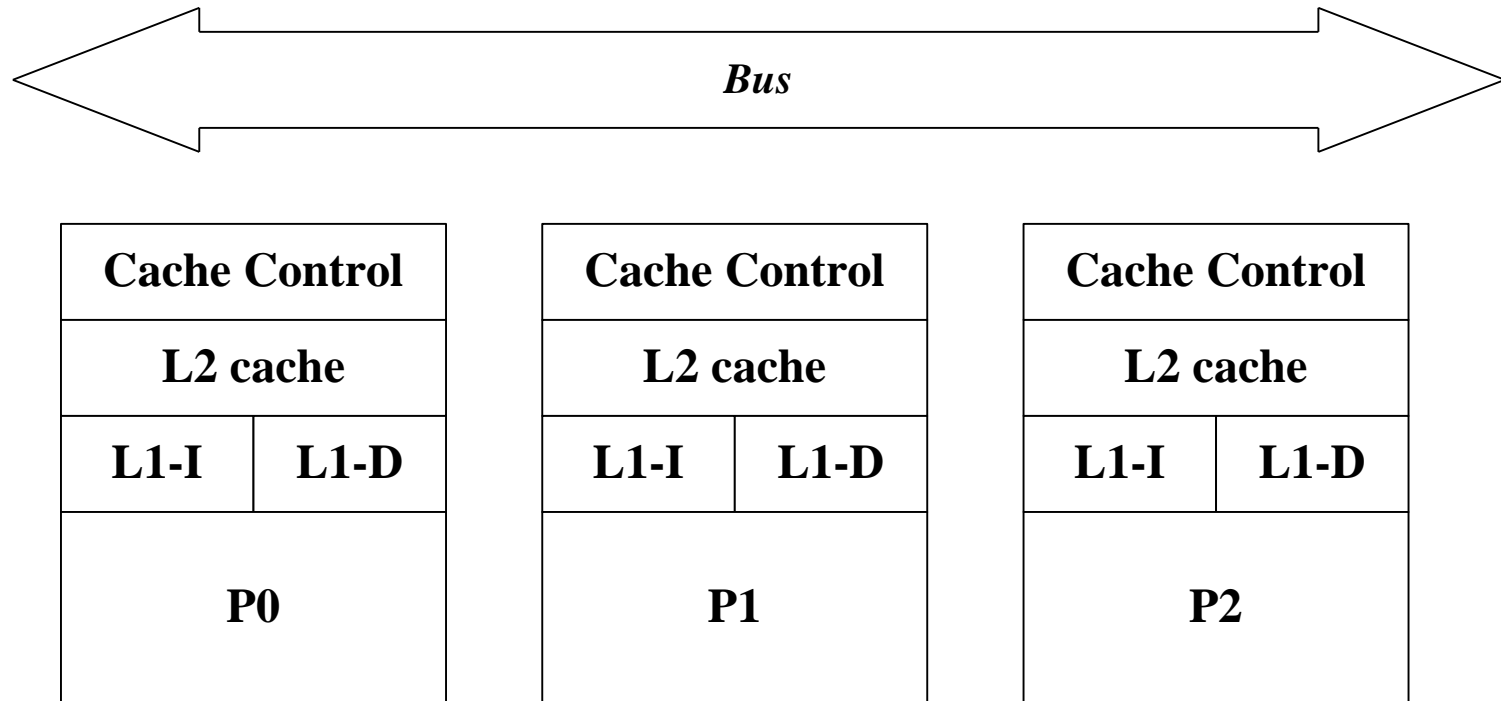
Memory Bus Controller			
L2 cache			
L1-I	L1-D	L1-I	L1-D
P2		P3	

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)



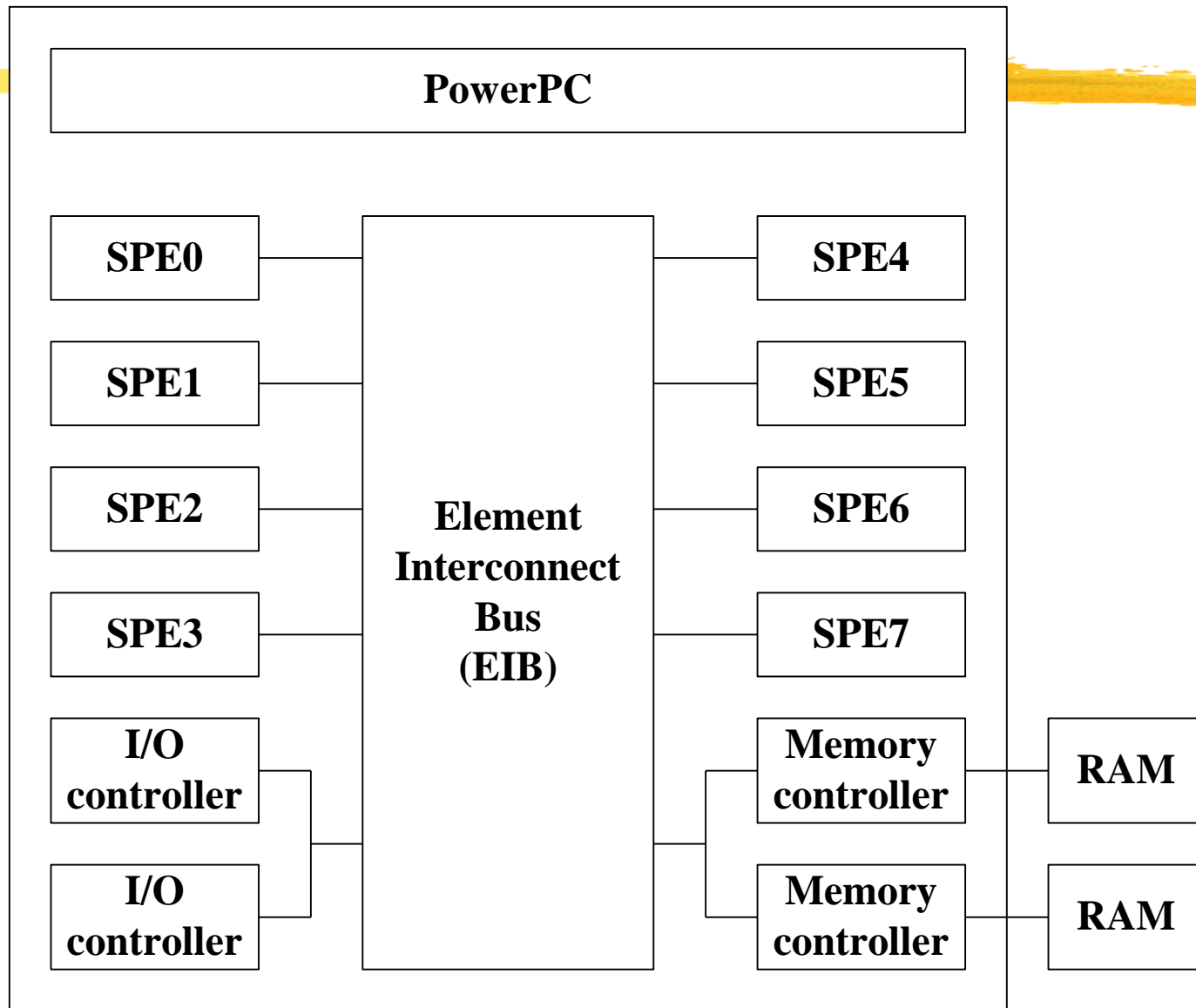
Каждый процессор

- ✂ 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)

Cell

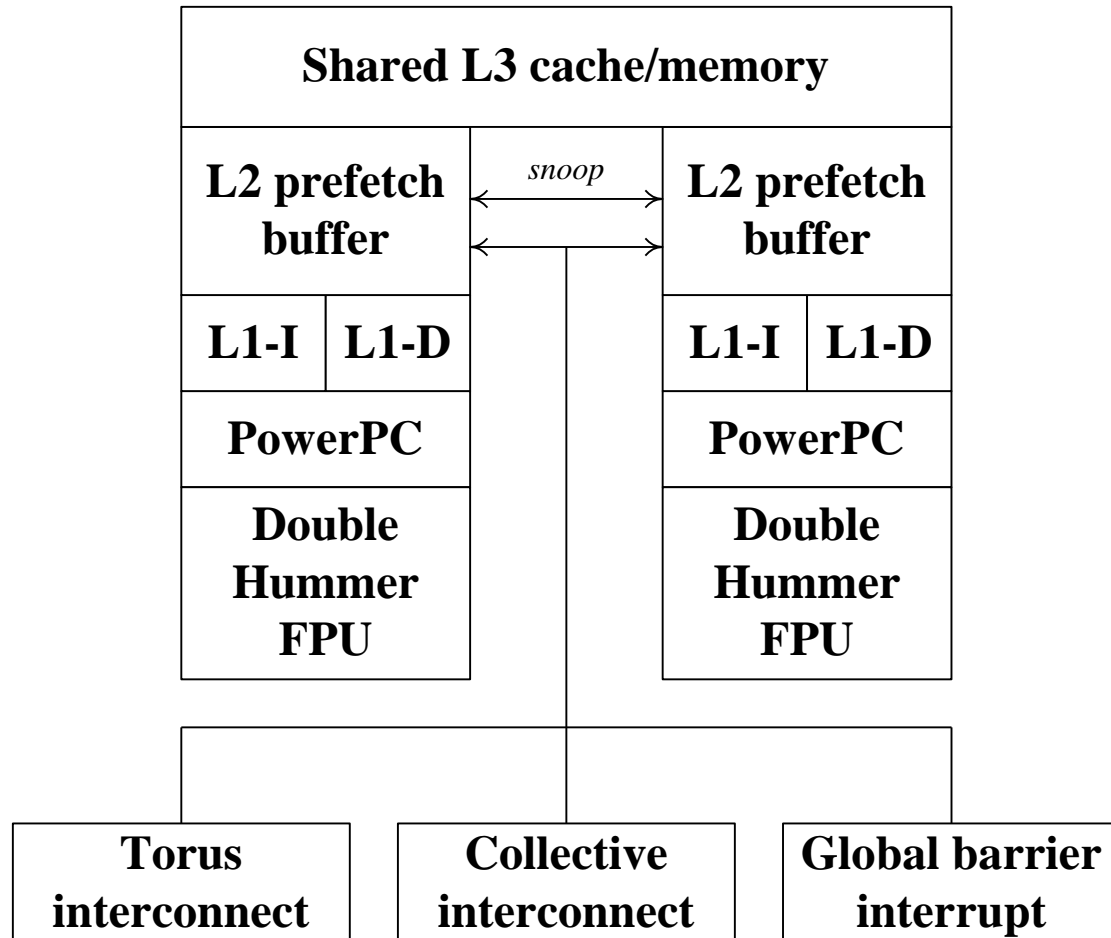


Cell



- ⌘ 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

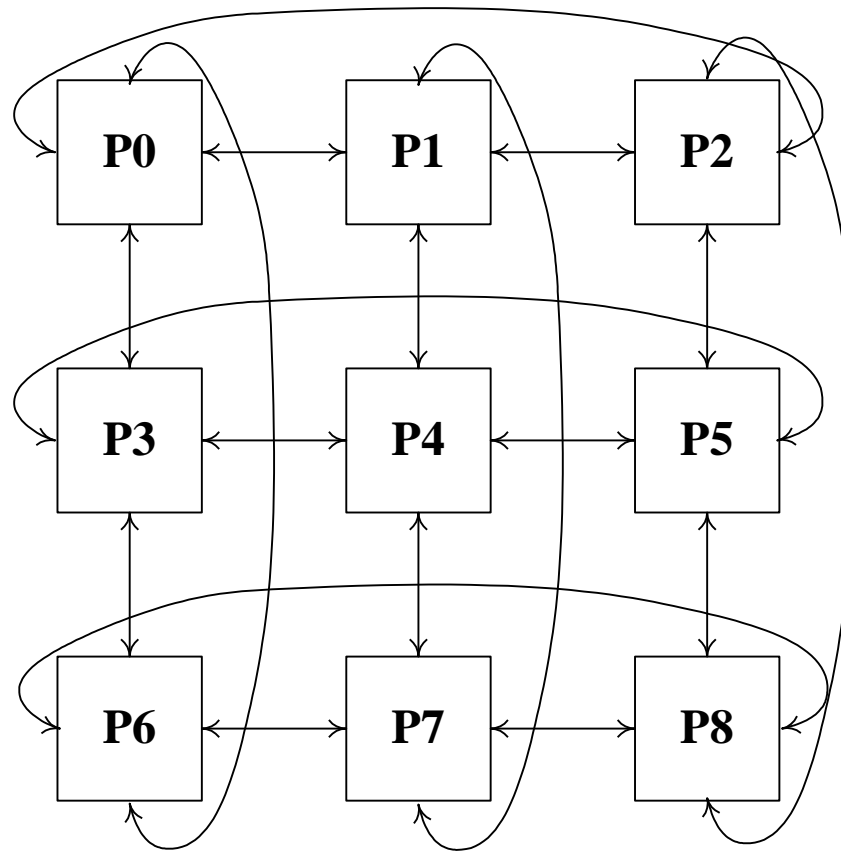
- ⌘ 512 Mb off-chip RAM

- ⌘ 6 двухсторонних портов для 3D-тора

- ⌘ 3 двухсторонних порта для collective network

- ⌘ 4 двухсторонних порта для barrier/interrupt

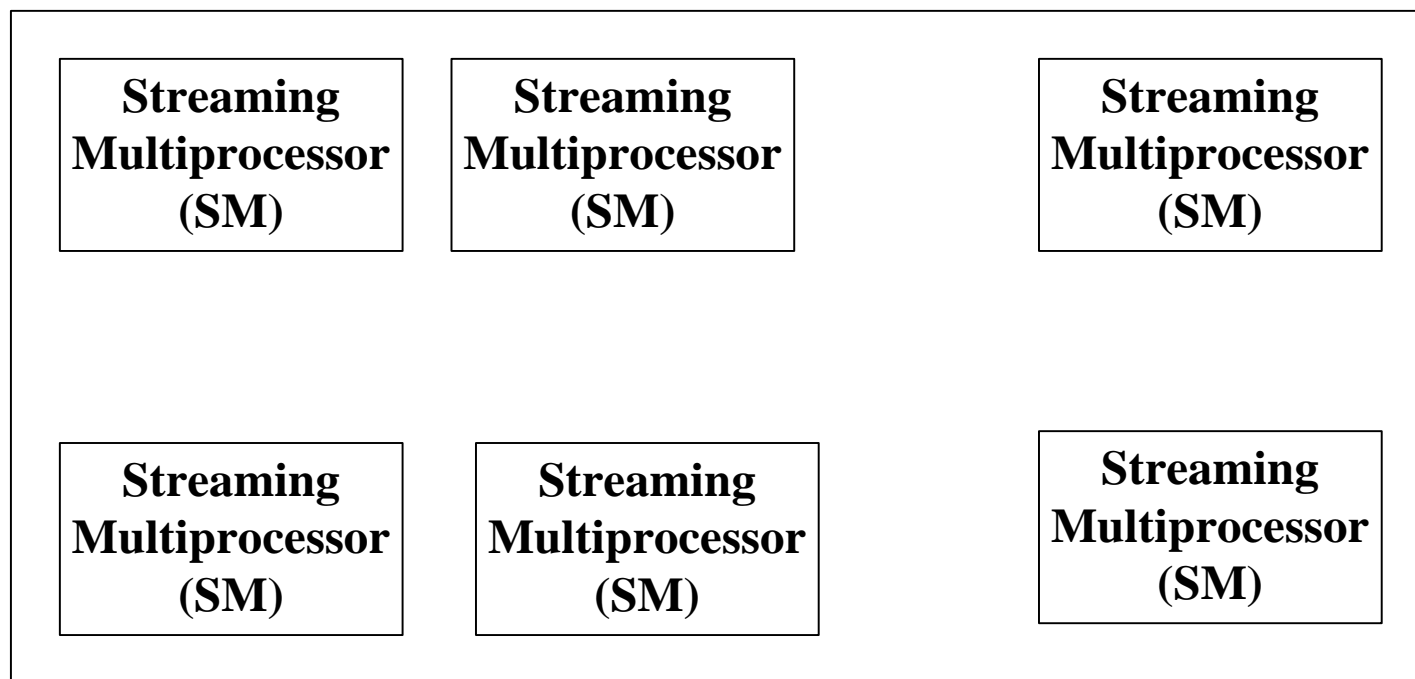
BlueGene/L



Архитектура G80

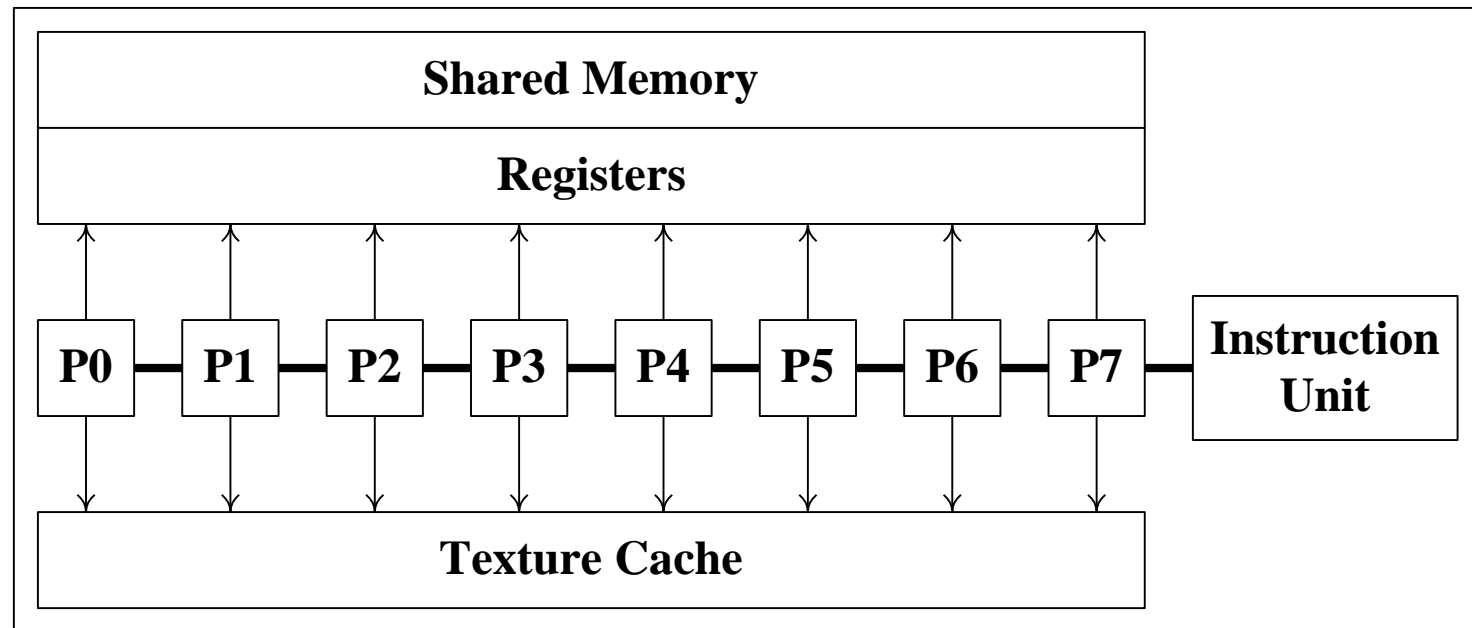


Массив из потоковых мультипроцессоров



Архитектура G80

Streaming Multiprocessor



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*

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 a = {5.0f, 2.0f, 1.0f, 3.0f};
    vec4 b = {5.0f, 3.0f, 9.0f, 7.0f};
    vec4 c;

    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]);

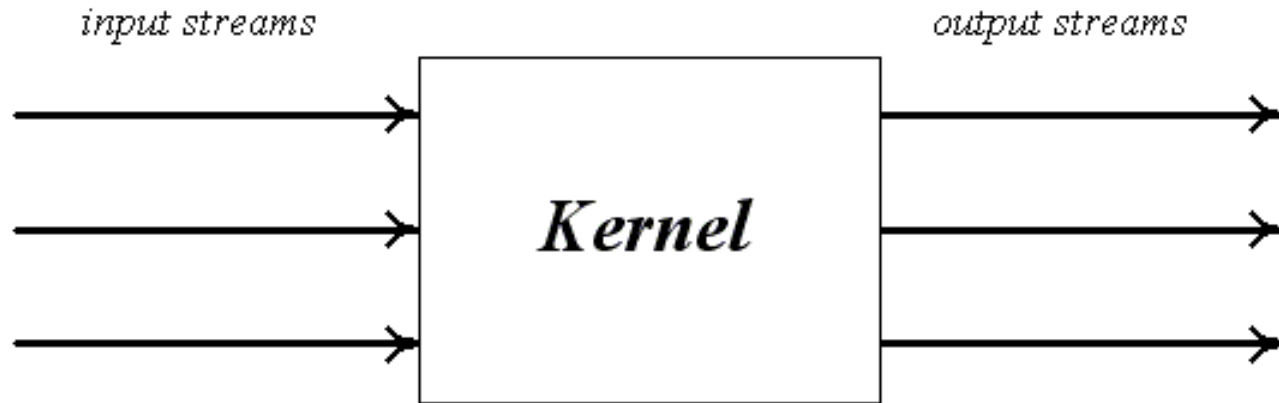
    return 0;
}
```

SIMD



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

SIMD



- Так как каждый элемент может быть обработан независимо от других, то их можно обрабатывать параллельно
- Можно соединять между собой отдельные ядра для получения более сложной схемы обработки

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

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

Отличия CPU от GPU




- ⌘ Очень высокая степень параллелизма
- ⌘ Основная часть чипа занята логикой, а не кэшем
- ⌘ Ограничения по функциональности

GP GPU



- ⌘ 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

CUDA (Compute Unified Device Architecture)



- ⌘ Программирование массивно-параллельных систем требует специальных систем/языков.
- ⌘ Программирование ресурсов CPU ограничено
 - ☑ Multithreading
 - ☑ SSE
 - ☑ Часто bottleneck – в пропускной способности памяти
- ⌘ CUDA - система (библиотеки и расширенный C) для программирования GPU

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 * (float) idx / (float) N;      // значение аргумента

    data [idx] = sinf ( sqrtf ( x ) );                             // найти значение и
                                                                    // записать его 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;

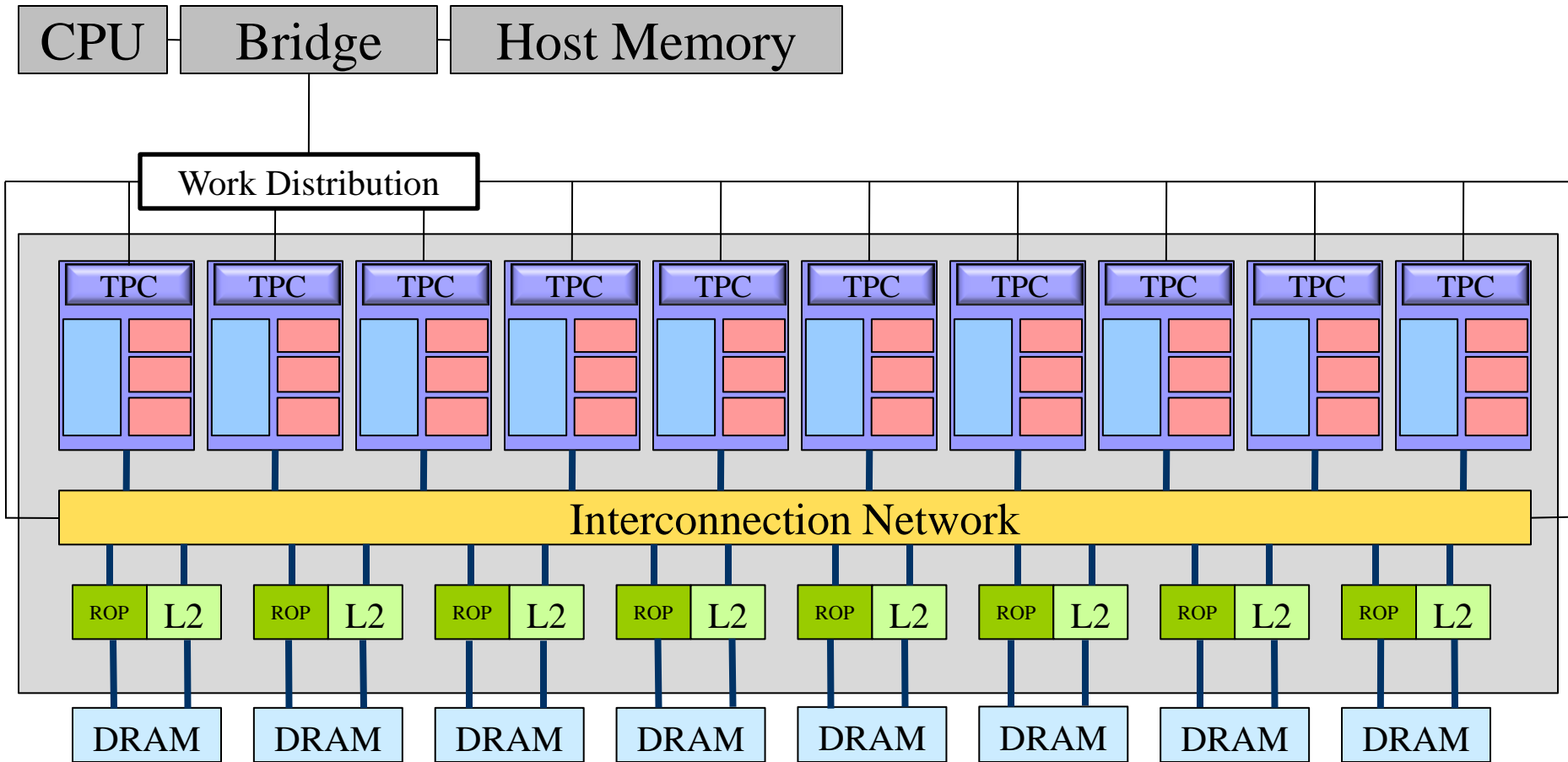
        // выделить память на GPU под N элементов
cudaMalloc ( (void**)&dev, N * sizeof ( float ) );

        // 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 );

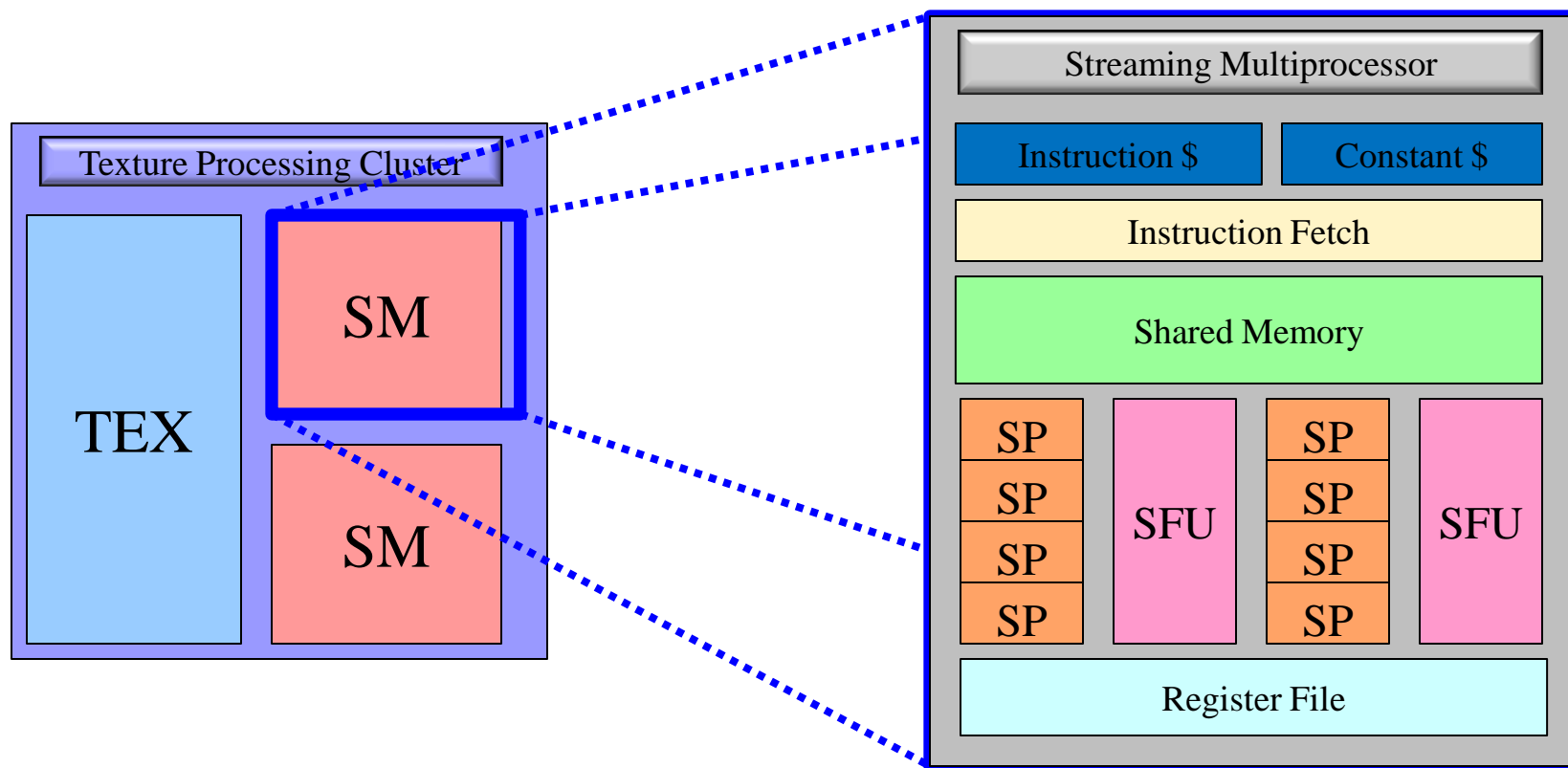
        // освободить память GPU
cudaFree   ( dev   );
```

Архитектура Tesla 10



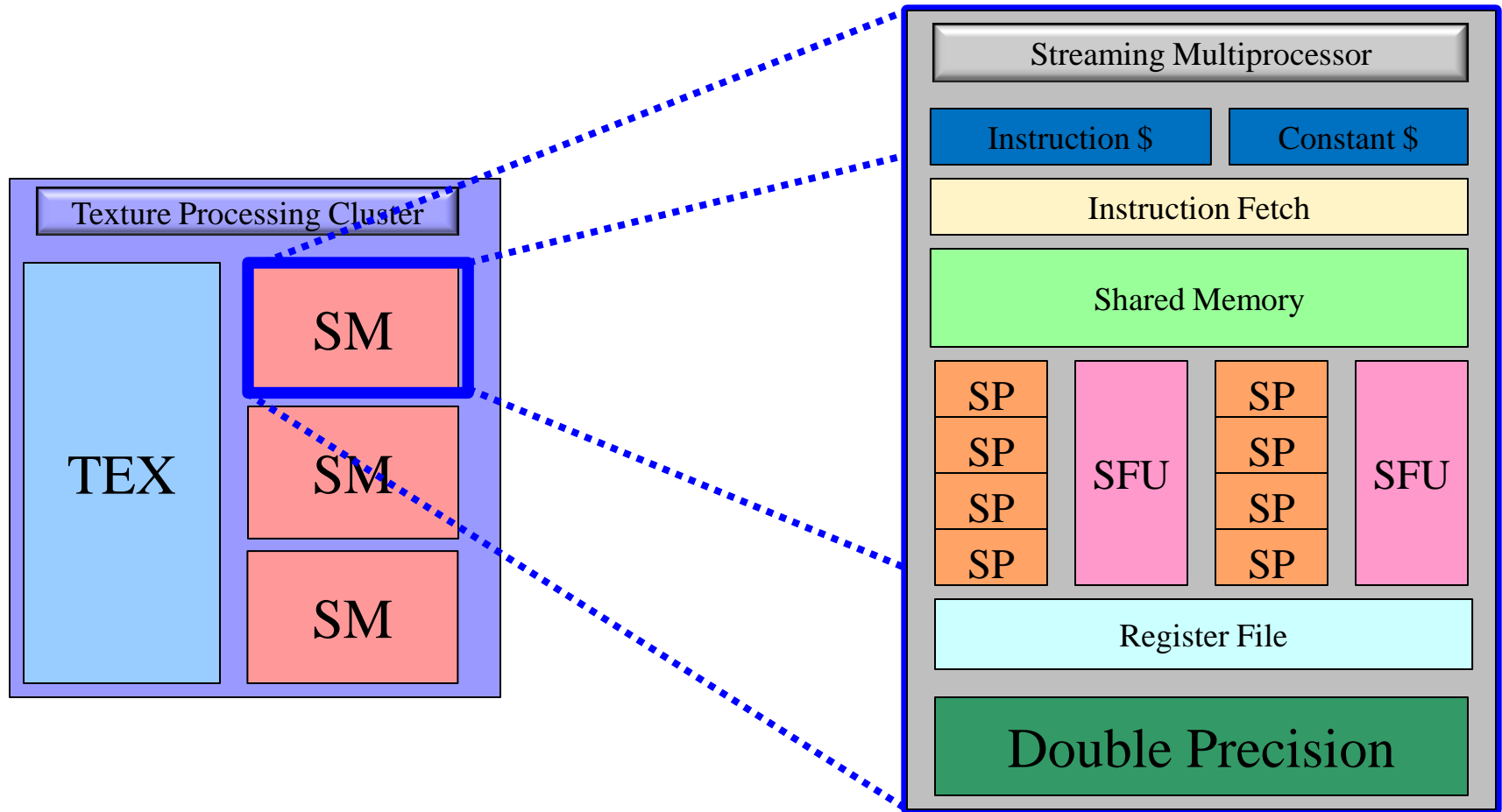
Архитектура Tesla:

Мультипроцессор Tesla 8



Архитектура Tesla

Мультипроцессор Tesla 10



Архитектура



⌘ Масштабируемость:

☑ [+][-] SM внутри TPC

☑ [+][-] TPC

☑ [+][-] DRAM партии

⌘ Схожие архитектуры:

☑ Tesla 8: 8800 GTX

☑ Tesla 10: GTX 280

SIMT (Single Instruction, Multiple Threads)



- ⌘ Параллельно на каждом SM выполняется большое число отдельных нитей (threads)
- ⌘ Нити подряд разбиваются на warp'ы (по 32 нити) и SM управляет выполнением warp'ов
- ⌘ Нити в пределах одного warp'а выполняются физически параллельно
- ⌘ Большое число warp'ов покрывает латентность

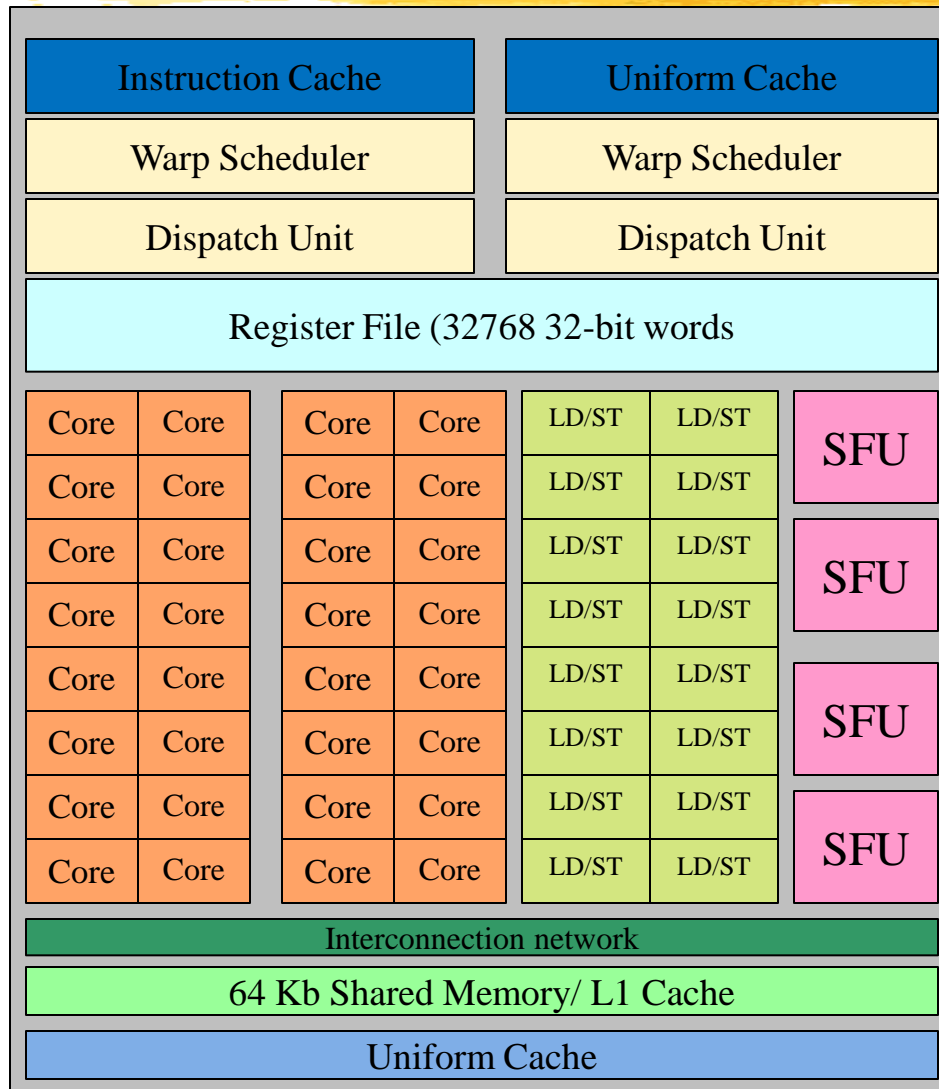
Архитектура Fermi



- ⌘ Unified L2 cache (768 Kb)
- ⌘ Up to 1 Tb of memory (64 bit addressing)
- ⌘ Unified address space
- ⌘ ECC protection (DRAM, registers, shared, cache)
- ⌘ Simultaneous CPU->GPU, GPU->CPU, kernel execution
- ⌘ 10x faster context switching, concurrent kernel execution (up to 16 kernels)

Архитектура Fermi.

Потоковый мультипроцессор



Архитектура Fermi



- ⌘ 32 cores per SM
- ⌘ Dual Thread schedule – simultaneous execution of 2 warps
- ⌘ 48 Kb shared + 16 Kb cache или 16 Kb shared + 48 Kb cache
- ⌘ Cheap atomics

Ресурсы нашего курса



⌘ [CUDA.CS.MSU.SU](https://cuda.cs.msu.su)

- ☑ Место для вопросов и дискуссий
- ☑ Место для материалов нашего курса
- ☑ Место для ваших статей!
 - ☒ Если вы нашли какой-то интересный подход!
 - ☒ Или исследовали производительность разных подходов и знаете, какой из них самый быстрый!
 - ☒ Или знаете способы сделать работу с CUDA проще!

⌘ steps3d.narod.ru

⌘ www.nvidia.ru

Ресурсы нашего курса

⌘ К той лекции:

☑ CUDA / MT / SSE "hello world" проекты

☑ CUDA / MT / SSE "Hello World" проекты

☒ Чуть более насыщенные чем маленькие "hello world" ы 😊

☑ SVN ?

Несколько слов о курсе



- ⌘ Математический спецкурс

- ⌘ 11 лекций

- ⌘ 5 семинарских занятий

 - ☐ Раз в две недели

 - ☐ Цель занятий:

 - ☐ Начать быстро программировать на CUDA

 - ☐ Написать и сдать практические задания

- ⌘ 5 практических заданий

Несколько слов о курсе



⌘ Отчетность по курсу

☑ 5 практических заданий

☑ Задания сдаются на семинаре

☑ Либо по почте

- В течении недели со дня семинара, на котором задание выдано
- Если у вас не получается – дайте нам знать

☑ Альтернатива

☑ Дайте нам знать

