Лекция 1

- Содержание курса.
- Особенности и назначение GP GPU.
- Обзор архитектуры CUDA.
- CUDA API.

Содержание курса

1. Тема 1 (2 ч.)
   1.1. Назначение и особенности ***GP GPU***.
   1.2. Обзор архитектуры ***CUDA***.
   1.3. Установка средств разработки (***CUDA SDK***).
   1.4. ***CUDA API***.

2. Тема 2 (1 ч.)
   2.1. Характеристики *GPU*.
   2.2. Вычислительные возможности и версии *CUDA*.

3. Тема 3 (1 ч.)
   3.1. Обработка ошибок.
   3.2. Анализ производительности на основе объектов событий (*CUDA events*).

4. Тема 4 (4 ч.)
   - ***4.1. cuda-gdb***
   - *4.2. Data Display Debugger (ddd)*
   - *4.3. Nsight Eclipse Plugins*
   - *4.4. Nsight Visual Studio Code Edition*
   - ***4.5. nvprof***
   - *4.6. nvvp*
   - ***4.7. Nsight Compute CLI***
   - *4.8. Nsight Compute*

5. Тема 5 (2 ч.)
   - 5.1. Объединение нитей в блоки и варпы.
   - 5.2. Оптимальная конфигурация нитей.

6. Тема 6 (2 ч.)
   6.1. Иерархия памяти.
   6.2. Регистровая и локальная память.

7. Тема 7 (2 ч.)
   7.1. Совместный доступ к глобальной памяти (*coalescing*).
   7.2. Разделяемая память (*shared memory*).

8. Тема 8 (2 ч.)
   8.1. Константная память.
   8.2. Текстурная память.

9. Тема 9 (2 ч.)
   - 9.1. Уровни компиляции nvcc.
   - 9.2. *.cubin, .fatbin, .gpu* и *.ptx* файлы.
   - 9.3. *PTX* (Parallel Thread eXecution) *ISA* (Instruction Set
   - 9.4. Architecture).
   - *9.5. CUDA Driver API.*

10. Тема 10 (2 ч.)
    - 10.1. Поддержка cuda в Python.
    - 10.2. Модуль *cuda_driver*.
    - 10.3. Пакет *pycuda*.
    - 10.4. Пакет *numba.cuda.*

11. Тема 11 (2 ч.)
    11.1. Библиотека *Thrust*.
    11.2. Обобщенное программирование: контейнеры, обобщенные алгоритмы, итераторы.
    11.3. Контейнеры *host_vector* и *device_vector*.
    11.4. Алгоритмы *thrust*.
    11.5. Преобразование указателей и комбинированный код.
    11.6. Алгоритм *transform* и функторы.
    11.7. Скалярное произведение векторов с использованием thrust.
    11.8. Транспонирование матрицы с использованием thrust.

12. Тема 12 (2 ч.)
    12.1. Потоки CUDA (CUDA Stream).
    12.2. Одновременное выполнение ядер.
    12.3. Одновременное копирование и выполнение ядра.
    12.4. Использование нескольких GPU.

13. Тема 13 (2 ч.)
    13.1. Тензорные операции, произведение матриц.
    13.2. Реализация произведения матриц на основе *CUDA API*.
    13.3. Особенности использования библиотеки *cuBLAS*.
    13.4. Функции *cublas<T>gemm().*
    13.5. Тензорные процессоры.
    13.6. Вызовы *cublas<T>gemm()* с использованием тензорных процессоров.

14. Тема 14 (2 ч.)
    14.1. Программирование на уровне *warp*’ов.
    14.2. Функции *wmma*.

15. Тема 15 (4 ч.)
    15.1. Использование *GPU* при *глубоком обучении*.
    15.2. Нейросети и *глубокое обучение*.
    15.3. *Datasets для глубокого обучения*.

16. Тема 16 (2 ч.)
    16.1. Библиотеки *tensorflow* и *keras.*

Назначение GP GPU

CGA, EGA, VGA адаптеры

Отображение в ПАМЯТЬ

Память

регистры

буфер

Адаптер

Конвертер: память -> видеосигналы

ОЗУ

Таблица векторов прерываний

BIOS

Прерывание **10h**

CPU

```
  move cx,00
  move dx,00
CL:
  move ah, 0ch
  move al,03
  int 10h
  inc cx
  cmp cx,100
  jne CL
```

```
for(int i=0,j=0;i<100;i++){
  Setcolor(3);
  Setpixel(i,j)
}
```

- Увеличение памяти и пропускной способности.
- Аппаратное ускорение.
- Распараллеливание.

От адаптеров к ***GP GPU:***

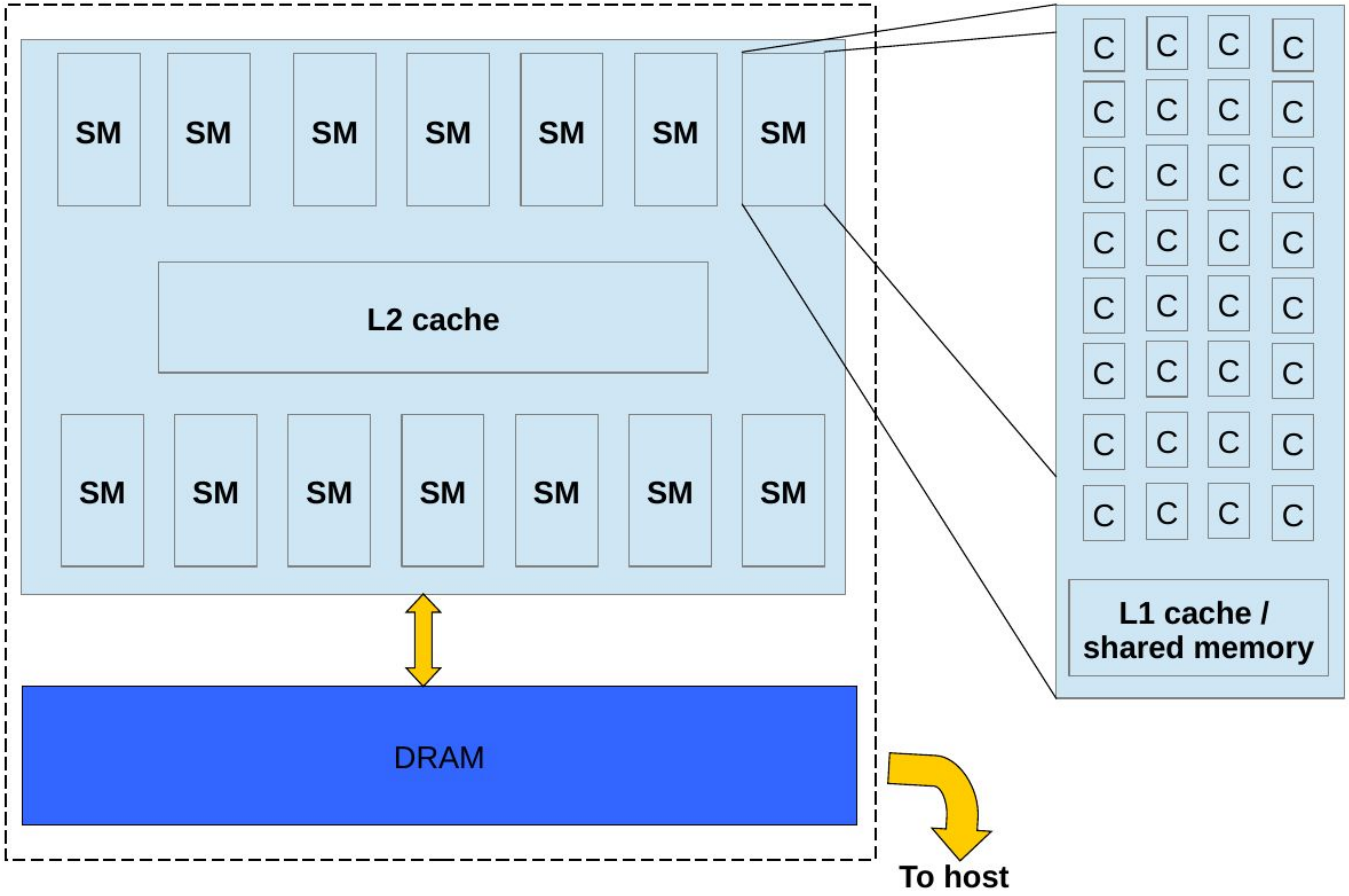
Память ≤ 256 Кб,
8-битовый интерфейс ISA,
пропускная способность - 8 Мб/с

Память ≤ 6 Гб,
32/64-битовый интерфейс PCI-E,
пропускная способность - 128 Гб/с,
процессорные ядра ≅ 10 000,
Специализированные устройства для вычислений с FP, вычислений специальных функций, **тензорные ядра**, *ray tracing* процессоры.


SM
Instruction Cache
Instruction Buffer
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File (32,768 x 32-bit)
Core
DP Unit
LD/ST
SFU
Texture / L1 Cache
Tex
64KB Shared Memory


Pascal Streaming Multiprocessor (GP104)
Instruction Cache
SM Processing Block (Partition)
Instruction Buffer
Warp Scheduler
2x Dispatch Unit
Register File
FP32 / INT32
8x LD/ST
SFU
Texture/L1 Cache
4x Texture Units
Shared Memory
Turing Streaming Multiprocessor (TU104)
L0 Instruction Cache
Warp Scheduler
Dispatch Unit
Register File
INT32
FP32
2x Tensor Cores 2nd Gen
4x LD/ST
SFU
L1 Data Cache/Shared Memory
4x Texture Units
RT Core 1st Generation
2x FP64
Ampere Streaming Multiprocessor (GA104)
FP32 / INT32
FP32
Tensor Core 3rd Gen
RT Core 2nd Generation

GPU (Fermi architecture)



Модели параллельных вычислений

Модель	Программные средства	Архитектура ВС
Общая память	POSIX (pthread), WinAPI(CreateThread), OpenMP...	MIMD, разделяемая память
Обмен сообщениями	MPI (Message Passing Interface): OpenMPI, MPICH, LAM (Local Area Multicomputer); PVM (Parallel Virtual Machine)...	MIMD, распределенная и разделяемая память
Параллелизм данных	Языки .NET, Python...	MIMD/SIMD

Архитектура фон Неймана

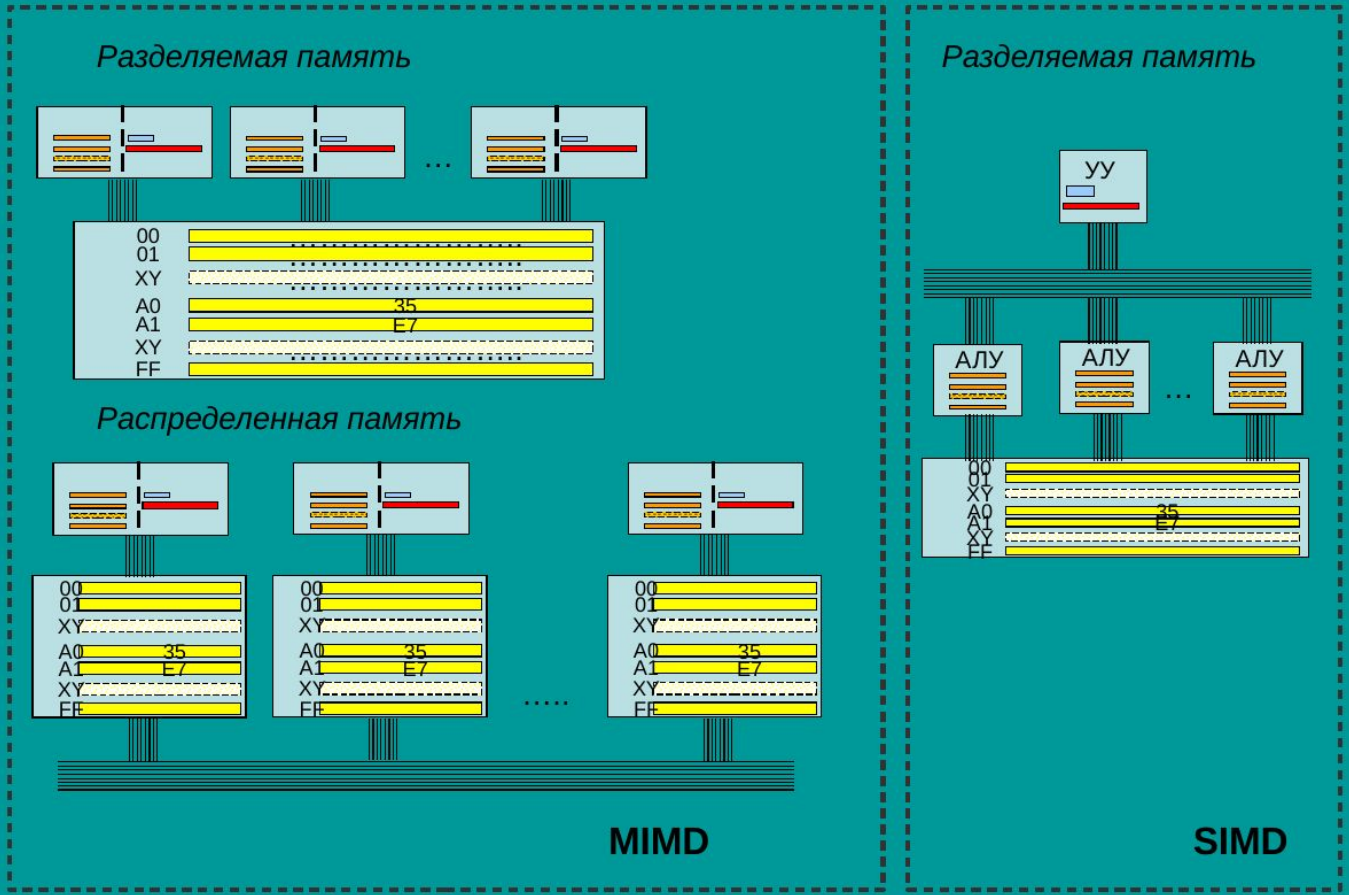
Регистры общего назначения – сумматор, регистр данных, адресный регистр и т.д.

Счетчик команд

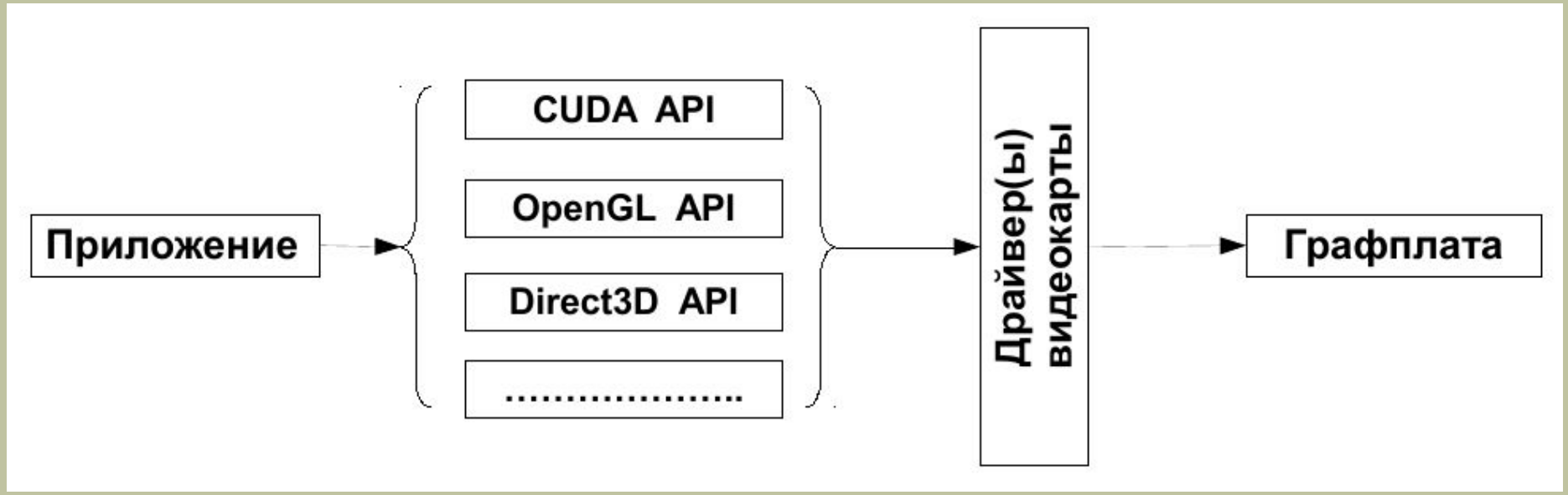
Ячейки памяти

Регистр команд

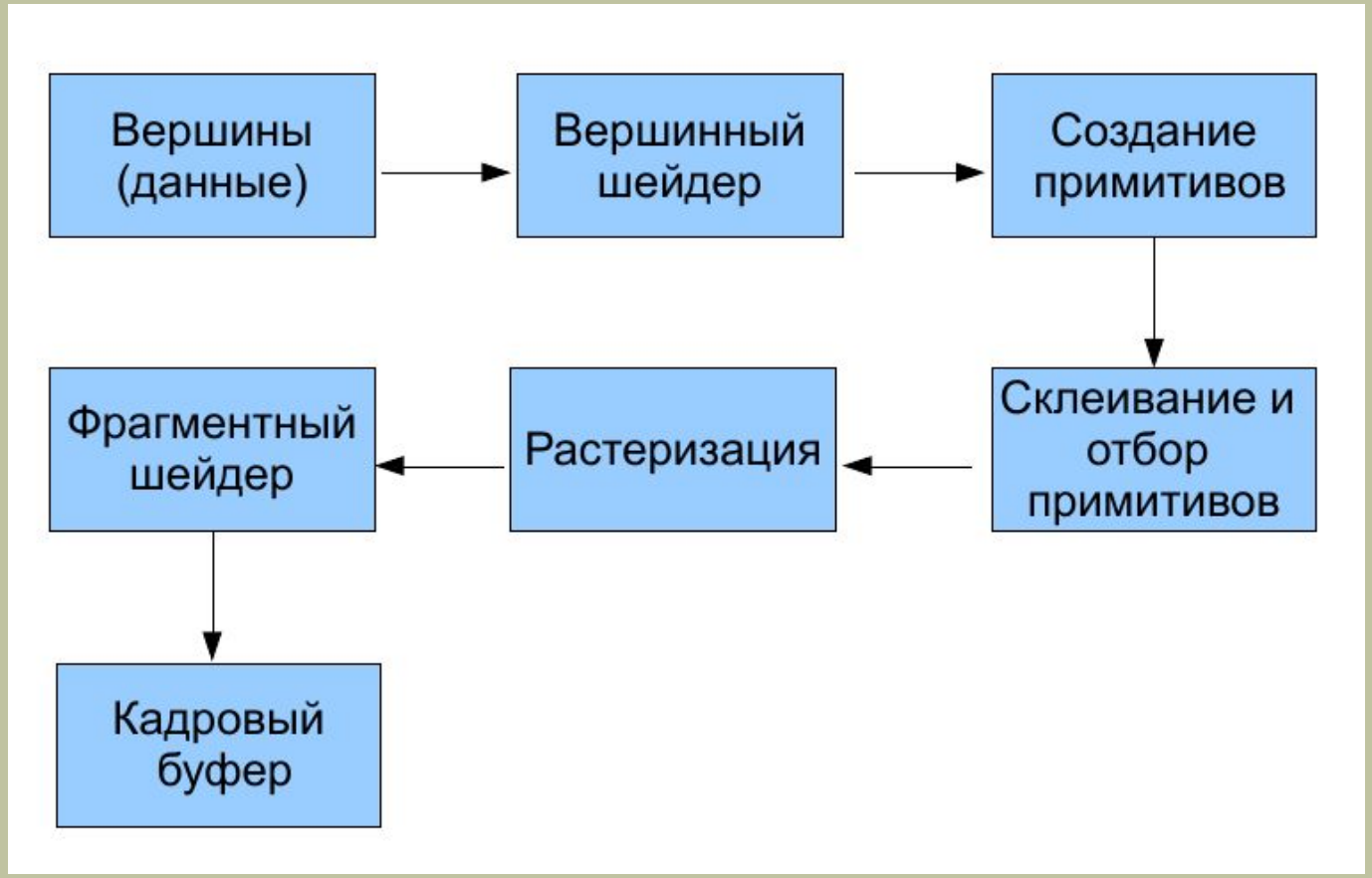
Основные архитектуры производительных ВС



Интерфейсы программирования GP GPU



Конвейер OpenGL



OpenGL 2.0: GLSL

Вершинный шейдер

```
const char *vpSrc[] = {
  "#version 430\n",
  "layout(location = 0) in vec3 pos;\
   layout(location = 1) in vec3 color;\
   out vec4 vs_color;\
   void main() {\
     gl_Position = vec4(pos,1);\
     vs_color=vec4(color,1.0);\
   }"
};
```

Фрагментный шейдер

```
const char *fpSrc[] = {
  "#version 430\n",
  "in vec4 vs_color;\
   out vec4 fcolor;\
   void main() {\
     fcolor = vs_color;\
   }"
};
```

OpenGL 4.3:
compute shaders

Вычислительные шейдеры выполняются вне конвейера.

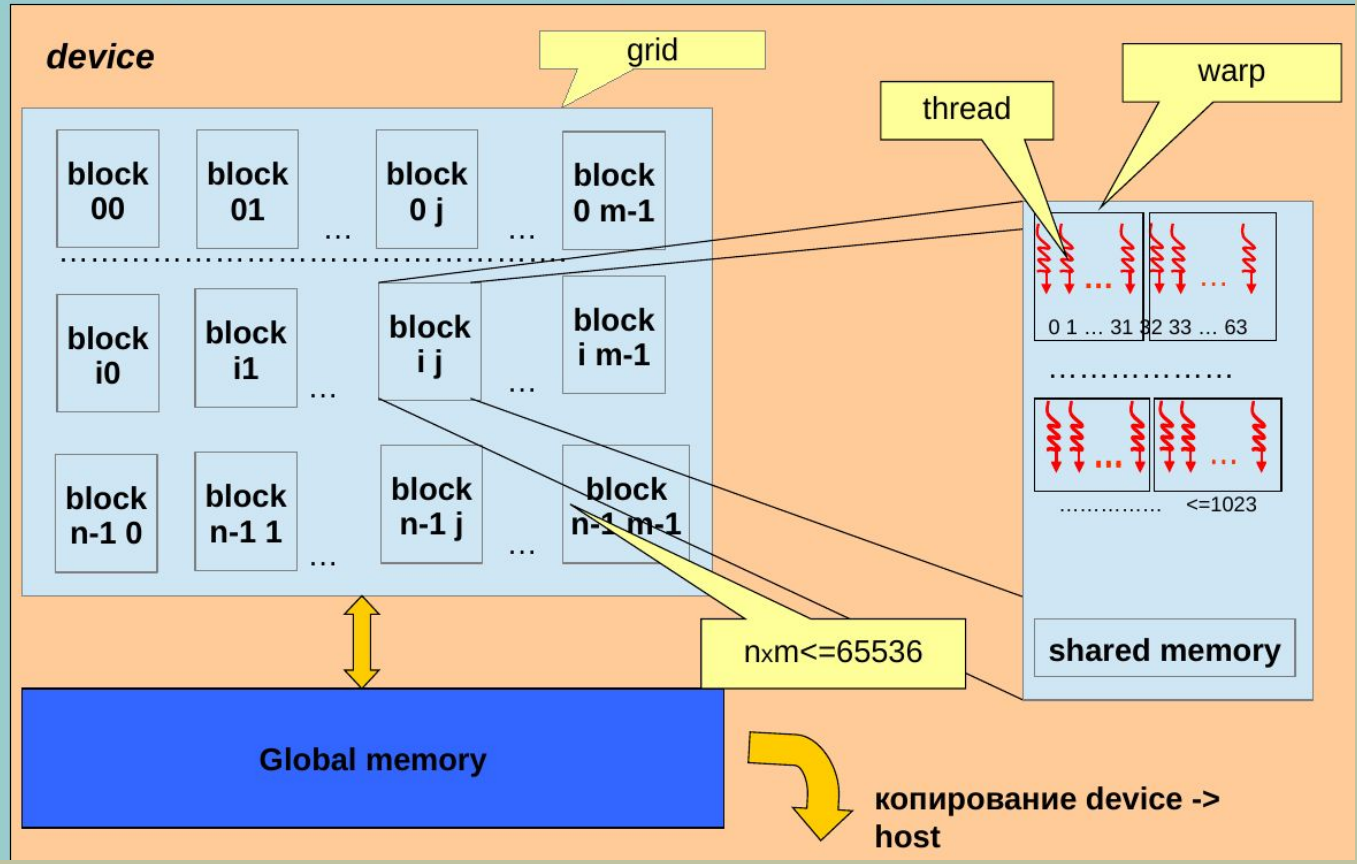
Архитектура *CUDA*

Активное использование графических процессоров (GPU) для прикладных расчетов научно-технического назначения во многом связано с предоставлением компанией NVIDIA технологии CUDA (*Compute Unified Device Architecture*). Технология CUDA предоставляет понятную для прикладного программиста абстракцию графического процессора (GPU) и простой интерфейс прикладного программирования (API – Application Programming Interface).

По терминологии CUDA вычислительный узел с CPU и main memory называется ***host***, GPU называется ***device***. Программа, выполняемая на host'е содержит код – ядро (***kernel***), который загружается на device в виде многочисленных копий. Все копии загруженного кода – нити (***threads***), объединяются в блоки (***blocks***) по 512-1024 нити в каждом. Все блоки объединяются в сеть (***grid***) с максимальным количеством блоков 65536. Все нити имеют совместный доступ на запись/чтение к памяти большого объема - ***global memory***, на чтение к кэшируемым ***constant memory*** и ***texture memory***. Нити одного блока имеют доступ к быстрой памяти небольшого объема – ***shared memory***.

CUDA (*Compute Unified Device Architecture*)

- cuda предоставляет абстракцию GPU для программистов



Расширение языка C ***CUDA C*** — спецификаторы функций и переменных, специальные директивы, встроенные переменные и новые типы данных, а так же набор функций и структур данных ***CUDA API***, предоставляют простой инструмент для программирования на GPU.

Функция-ядро (kernel)

Код, выполняемый на устройстве (ядро), определяется в виде функции типа *void* со спецификатором ***__global__***:

```
__global__ void gFunc(<params>){...}
```

Конфигурация нитей

При вызове ядра программист определяет количество нитей в блоке и количество блоков в *grid*. При этом допустима линейная, двумерная или трехмерная индексация нитей:

```
gFunc<<<dim3(bl_xdim, bl_ydim, bl_zdim),
         dim3(th_xdim, th_ydim, th_zdim)>>>(<params>);
```

test.cu

```
#include <stdio.h>
#include <cuda.h>

__global__ void gTest(float* a){
   a[threadIdx.x+blockDim.x*blockIdx.x]=
                        (float)(threadIdx.x+blockDim.x*blockIdx.x);
}
```

```
int main(){
  float *da, *ha;
  int num_of_blocks=10, threads_per_block=64;
  int N=num_of_blocks*threads_per_block;

  ha=(float*)calloc(N, sizeof(float));
  cudaMalloc((void**)&da, N*sizeof(float));
```

```
gTest<<<dim3(num_of_blocks),
          dim3(threads_per_block)>>>(da);
CudaDeviceSynchronize();

cudaMemcpy(ha,da,N*sizeof(float),
                                  cudaMemcpyDeviceToHost);
```

```
for(int i=0;i<N;i++)
  printf("%g\n", ha[i]);

free(ha);
cudaFree(da);

return0;
}
```

```
> nvcc test.cu -o test
> ./test
```