Содержание курса

- Введение: GPGPU, архитектура CUDA, CUDA API (1 л, 2 пз).
- Отладка и профилирование кода (2л, 2 пз).
- PTX ISA и CUDA Driver API (2 л, 2 пз).
- Программирование тензорных ядер; wmma, cuBlas, cuTensor (3 л, 3 пз)
- PyCUDA и Numba (1 л, 2 пз)

Лекция 1

- модели параллельных вычислений;
- аппаратные особенности графических процессоров;
- архитектура CUDA – основные свойства и принципы;
- программная модель: хост, устройства, ядра, иерархия нитей (threads);
- программный интерфейс CUDA;
- установка CUDA Toolkit.

https://developer.nvidia.com/cuda-zone

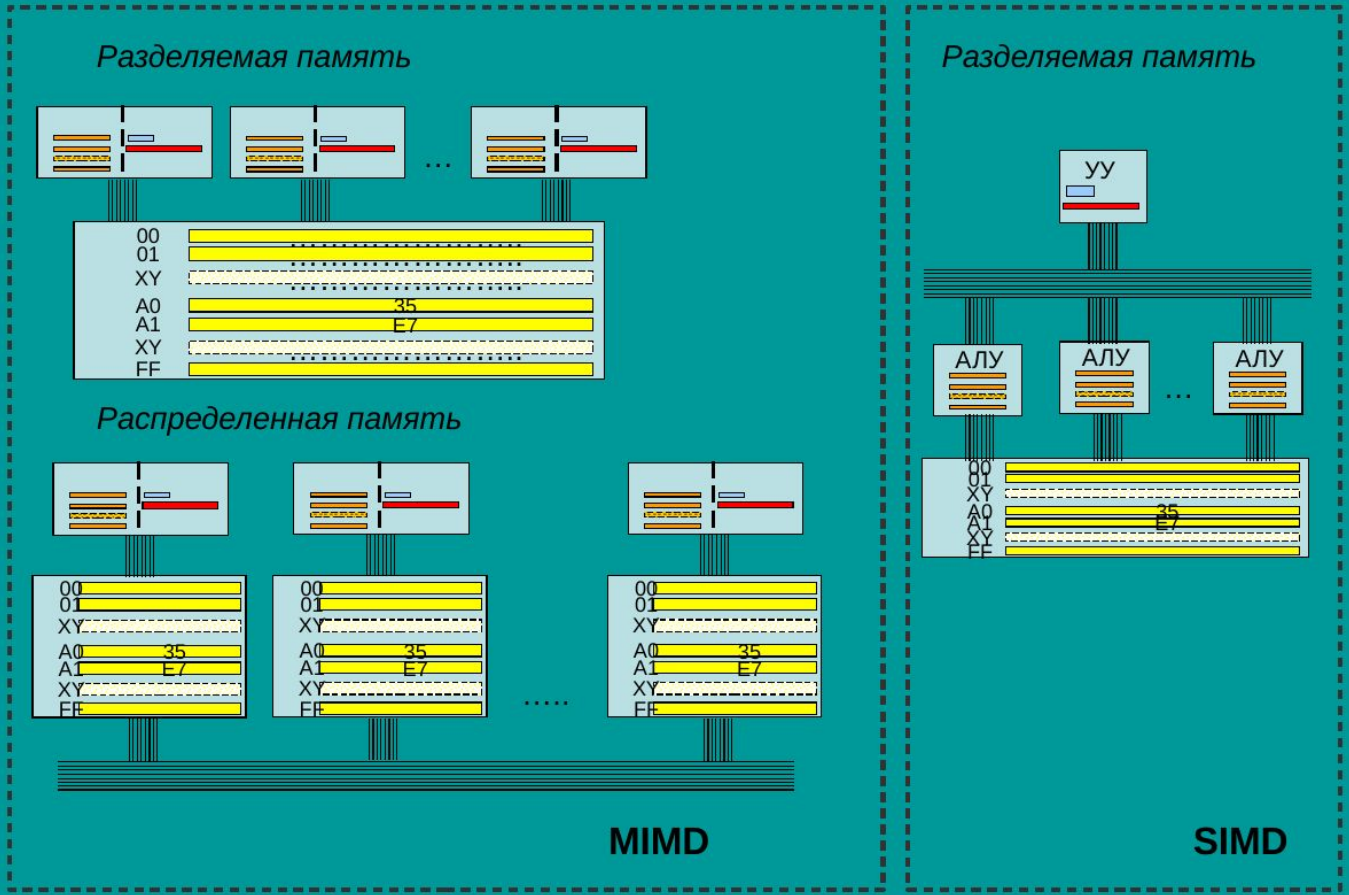
https://developer.nvidia.com/cuda-downloads

https://github.com/mlkv52git/sibsutis_cuda-2024

Модели параллельных вычислений

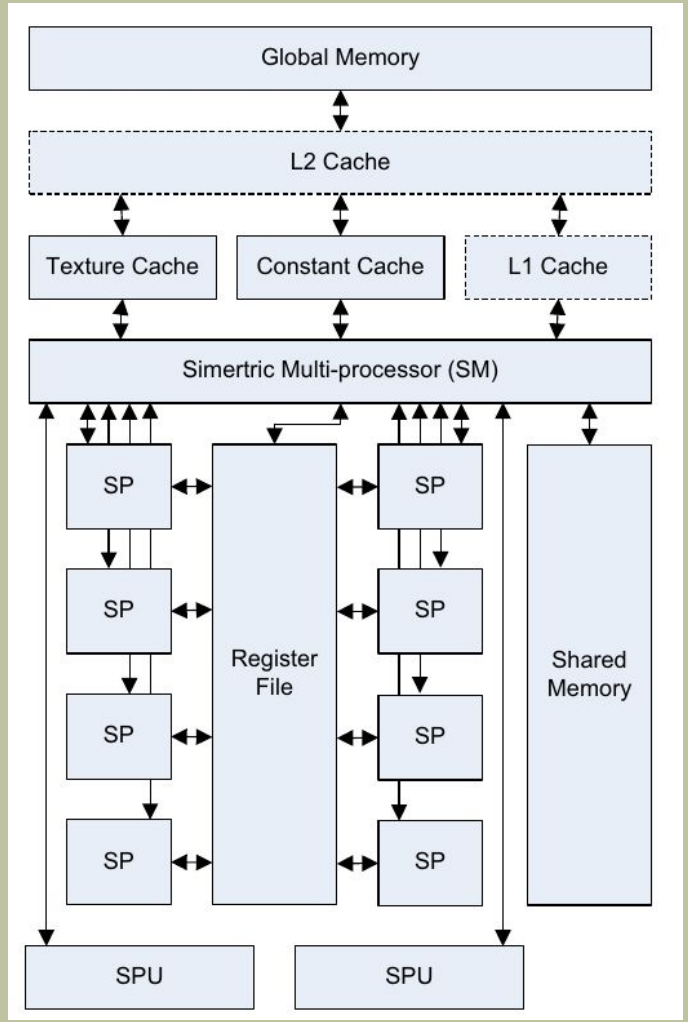
Модель	Программные средства	Архитектура ВС
Общая память	POSIX (pthread), WinAPI(CreateThread), OpenMP...	MIMD, разделяемая память
Обмен сообщениями	MPI (Message Passing Interface): OpenMPI, MPICH, LAM (Local Area Multicomputer); PVM (Parallel Virtual Machine)...	MIMD, распределенная и разделяемая память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

Основные архитектуры производительных ВС

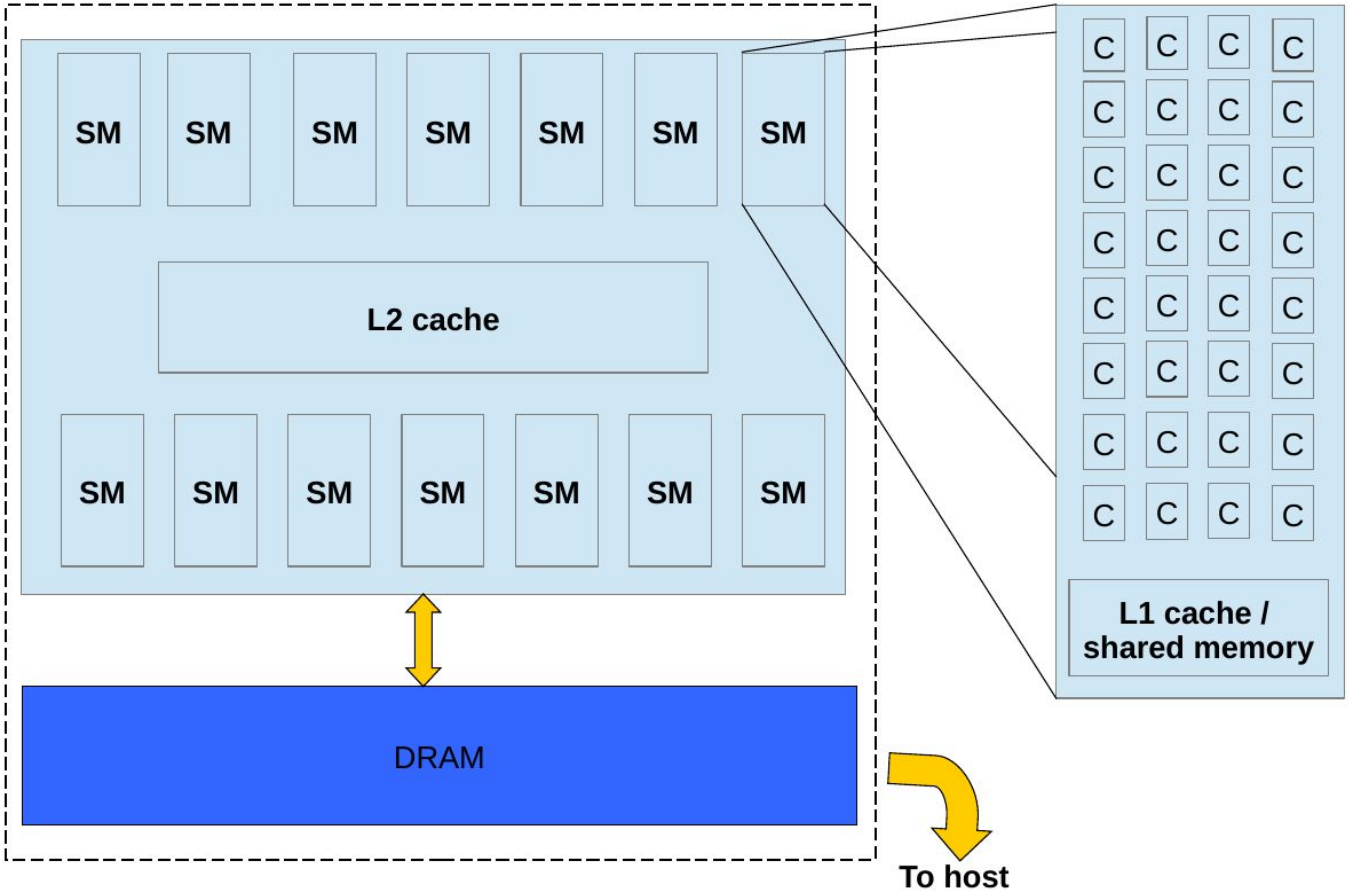



SM
Instruction Cache
Instruction Buffer
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File (32,768 x 32-bit)
Core
DP Unit
LD/ST
SFU
Texture / L1 Cache
Tex
64KB Shared Memory


L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
INT
FP32
FP32
TENSOR CORE
TENSOR CORE
LD/ ST
SFU


Global Memory
L2 Cache
Texture Cache
Constant Cache
L1 Cache
Simertric Multi-processor (SM)
SP
SP
SP
SP
SP
SP
SP
SP
Register File
Shared Memory
SPU
SPU

GPU (Fermi architecture)



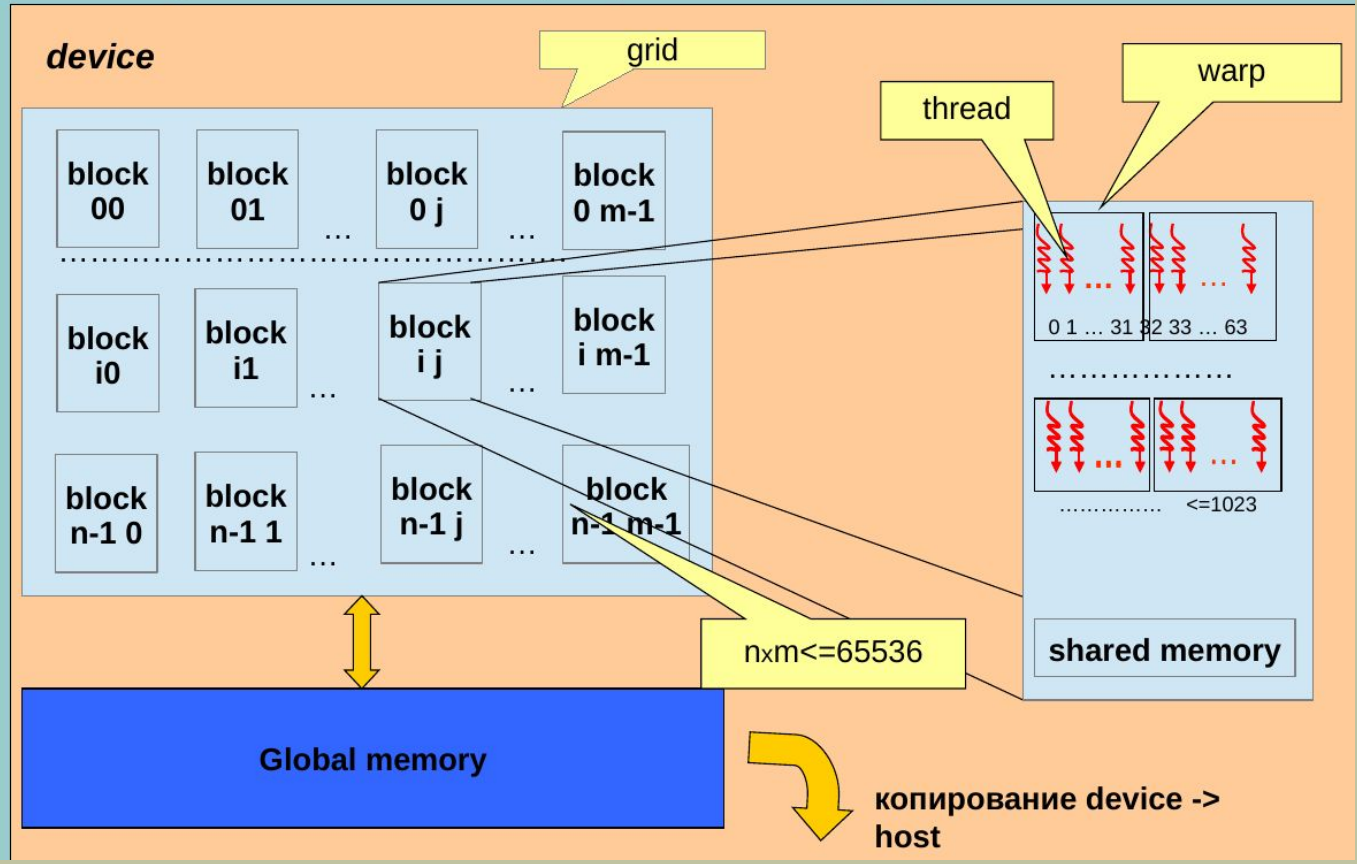
Логическое представление GPU

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
```

- Характеристики GPU.
- Вычислительные возможности и версии CUDA.

Получение сведений об устройстве.

```
cudaSetDevice(dev);
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, dev);
printf("  Total amount of constant memory:  %lu bytes\n",
                                                deviceProp.totalConstMem);
printf("  Total amount of shared memory per block: %lu bytes\n",

deviceProp.sharedMemPerBlock);
printf("  Total number of registers available per block: %d\n",
                                                  deviceProp.regsPerBlock);
printf("  Warp size: %d\n", deviceProp.warpSize);
printf("  Maximum number of threads per multiprocessor:  %d\n",
                                          deviceProp.maxThreadsPerMultiProcessor);
printf("  Maximum number of threads per block:  %d\n",

deviceProp.maxThreadsPerBlock);
```

~/NVIDIA_CUDA-11.1_Samples/1_Utilities/deviceQuery>./deviceQuery

```
Device 0: "NVIDIA GeForce RTX 2060"
 CUDA Driver Version / Runtime Version          12.0 / 11.1
 CUDA Capability Major/Minor version number:    7.5
 Total amount of global memory:                 5919 MBytes
(6206324736 bytes)
 (30) Multiprocessors, ( 64) CUDA Cores/MP:     1920 CUDA Cores
 GPU Max Clock rate:                            1695 MHz (1.70 GHz)
 Memory Clock rate:                             7001 Mhz
 Memory Bus Width:                              192-bit
……
 Maximum number of threads per multiprocessor:  1024
 Maximum number of threads per block:           1024
 Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
 Max dimension size of a grid size    (x,y,z): (2147483647,
65535, 65535)
……
```

Архитектура GPU	Вычислительные возможности	Версия CUDA
Tesla	1.*	CUDA 2.*-3.*
Fermi	2.*	CUDA 4.*-5.*
Kepler	3.*	CUDA 5.*
Maxwell	5.*	CUDA 6.*-7.*
Pascal	6.*	CUDA 8.*
Volta	7.*	CUDA 9.*
Turing	7.5	CUDA 10.*
Ampere	8.*-9.*	CUDA 11.*

```
~>nvcc -arch=sm_60 file_name.cu -o file_name
```

- Обработка ошибок.
- Анализ производительности.

Макрос для определения ошибки

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

#define CUDA_CHECK_RETURN(value) {\
      cudaError_t    _m_cudaStat = value;\
      if (_m_cudaStat != cudaSuccess) {\
        fprintf(stderr, "Error %s at line %d in file %s\n",\
        cudaGetErrorString(_m_cudaStat), __LINE__, __FILE__);\
        exit(1);\
      } }
```

const char* **cudaGetErrorString** (cudaError_t error) - *возвращает сообщение с кодом ошибки error.*

__FILE__ и __LINE__ - *предопределенные макросы препроцессора для определения местоположения в коде программы - имени файла и номера строки.*

Диагностика синхронных вызовов

```
__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= 1025;
int N=num_of_blocks*threads_per_block;

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

Диагностика асинхронных вызовов

```
    gTest<<<dim3(num_of_blocks), dim3(threads_per_block)>>>(da);
    CUDA_CHECK_RETURN(cudaDeviceSynchronize(););
    CUDA_CHECK_RETURN(cudaGetLastError());
    CUDA_CHECK_RETURN(cudaMemcpy(ha,da,N*sizeof(float),
                                                cudaMemcpyDeviceToHost));

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

    free(ha);
    cudaFree(da);

    return 0;
}
```

Профилирование программ с помощью объектов событий

```
……..
 int main(){
……..

 float elapsedTime;
 cudaEvent_t    start,stop; // встроенный тип данных – структура, для
                            // фиксации контрольных точек
 cudaEventCreate(&start); // инициализация
 cudaEventCreate(&stop); // событий
```

Синхронизация по событию

```
  cudaEventRecord(start,0); // привязка (регистрация) события start
  gTest<<<dim3(num_of_blocks), dim3(threads_per_block)>>>(da);
  cudaEventRecord(stop,0); // привязка события stop
  cudaEventSynchronize(stop); // синхронизация по событию
//CUDA_CHECK_RETURN(cudaDeviceSynchronize());
  CUDA_CHECK_RETURN(cudaGetLastError());
  cudaEventElapsedTime(&elapsedTime,start,stop); // вычисление
                                                  // затраченного времени

  fprintf(stderr,"gTest took %g\n", elapsedTime);

  cudaEventDestroy(start); // освобождение
  cudaEventDestroy(stop); // памяти
.................................................................
  }
```

https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html