Лекция 2

- Характеристики GPU.
- Вычислительные возможности и версии CUDA.
- Обработка ошибок.
- Анализ производительности.

~/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)
……
```

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);
```

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
```

https://docs.nvidia.com/cuda/archive/11.5.1/

https://docs.nvidia.com/cuda/archive/11.5.1/cuda-c-programming-guide/index.html#compute-capabilities

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
.............................................................................
 }
```

Задача:

1. Определить при какой длине векторов имеет смысл распараллеливать операцию сложения.
2. Определить оптимальное количество потоков POSIX для распараллеливания.
3. Определить зависимость времени выполнения операции сложения на GPU от длины векторов (выбирать количество нитей равным длине вектора).

```
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

void hTest(int N, int* a, int* b){
 for(int i=0; i<N;i++)
    a[i]+=b[i];
}

int main(int argc, char** argv){
 if(argc<2){
   fprintf(stderr, "USAGE: lab2 <N>\n");
   return -1;
 }
```

```
struct timeval t;
double Start, Finish;
double ElapsedTime;

int N=atoi(argv[1]);
if(N==0)
  N=1<<30;

int* a=(int*)calloc(N, sizeof(int));
int* b=(int*)calloc(N, sizeof(int));

for(int i=0; i<N;i++){
  a[i]=2*i;
  b[i]=2*i+1;
}
```

```
gettimeofday(&t, NULL);
Start =(double)t.tv_sec*1000000.0 + (double)t.tv_usec;
hTest(N,a,b);
gettimeofday(&t, NULL);
Finish =(double)t.tv_sec*1000000.0 + (double)t.tv_usec;
ElapsedTime = (double)(Finish-Start)/1000.0;
fprintf(stderr, "Elapsed time: %g ms \n", ElapsedTime);

for(int i=0; i<N;i+=N/16)
  fprintf(stdout, "%d\t%d\t%d\n", i, a[i], b[i]);

return 0;
}
```

```
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <math.h>
#include <pthread.h>

int *a, *b;

struct targ{
  int num_thread;
  int num_threads;
  int length;
};
```

```
void* hTest(void* arg){
 struct targ* s_arg=(struct targ*)arg;
 int length=s_arg->length;
 int offset=s_arg->num_thread*length;
 int i;
 for(i=0;i<length;i++)
   a[i+offset]+=/*1000*sin((double)*/ b[i+offset];

 return NULL;
}
```

```
int main(int argc, char** argv){
 if(argc<3){
  fprintf(stderr, "USAGE: <program name> <num_of_threads>
<vector_size>\n");
  return -1;
 }

  struct timeval t;
  double Start, Finish;
  double ElapsedTime;

  int i;
  int th_n=atoi(argv[1]);
  int N=atoi(argv[2]);
```

```
struct targ* Targs=(struct targ*)calloc(th_n, sizeof(struct targ));
 pthread_t* th_id=(pthread_t*)calloc(th_n, sizeof(pthread_t));

 a=(int*)calloc(N, sizeof(int));
 b=(int*)calloc(N, sizeof(int));

 for(i=0;i<N;i++){
   a[i]=2*i;
   b[i]=2*i+1;
 }
```

```
for(i=0;i<th_n; i++){
 Targs[i].num_threads=th_n;
 Targs[i].num_thread=i;
 Targs[i].length=N/th_n;
}
gettimeofday(&t, NULL);
Start =(double)t.tv_sec*1000000.0 + (double)t.tv_usec;
for(i=0;i<th_n; i++){
  pthread_create(&th_id[i], NULL, &hTest, &Targs[i]);
}
for(i=0;i<th_n; i++)
  pthread_join(th_id[i], NULL);

gettimeofday(&t, NULL);
Finish =(double)t.tv_sec*1000000.0 + (double)t.tv_usec;
```

```
 ElapsedTime = (double)(Finish-Start)/1000.0;
  fprintf(stderr, "Elapsed time: %g ms \n", ElapsedTime);

  free(Targs);
  free(th_id);

  for(i=0;i<N;i++)
    fprintf(stdout, "%d\t%d\t%d\n", i, b[i], a[i]);

  return 0;
}
```

```
/Lab2b> gcc lab2b-3.c -lm  -lpthread  -o lab2b-3
```

```
/Lab2b> taskset -c 0-5   ./lab2b-3 8 65536 > tmp
```