



Applied Parallel Computing LLC

<http://parallel-computing.pro>

# Введение в CUDA

к.т.н. Алексей Ивахненко

25 Февраля, 2017



## ■ **CUDA = Compute-Unified Device Architecture**

- Набор аппаратных средств и концепций потокового исполнения для работы с массивным параллелизмом

## ■ **CUDA C (или точнее C++)**

- Один из языков программирования, использующих специальные расширения для обеспечения поддержки CUDA
- Существуют и другие языки с поддержкой CUDA: CUDA Fortran, PyCUDA, ...

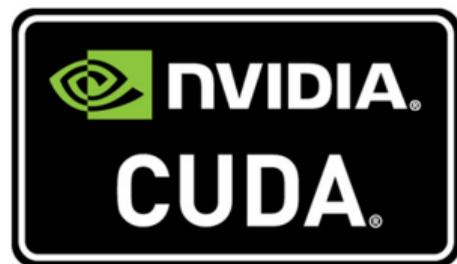


- **CUDA = Compute-Unified Device Architecture**

- Набор аппаратных средств и концепций потокового исполнения для работы с массивным параллелизмом

- **CUDA C (или точнее C++)**

- Один из языков программирования, использующих специальные расширения для обеспечения поддержки CUDA
- Существуют и другие языки с поддержкой CUDA: CUDA Fortran, PyCUDA, ...



- **CUDA = Compute-Unified Device Architecture**

- Набор аппаратных средств и концепций потокового исполнения для работы с массивным параллелизмом

- **CUDA C (или точнее C++)**

- Один из языков программирования, использующих специальные расширения для обеспечения поддержки CUDA
- Существуют и другие языки с поддержкой CUDA: CUDA Fortran, PyCUDA, ...

## ■ Hardware:

- Массивный параллелизм:  
тысячи вычислительных ядер

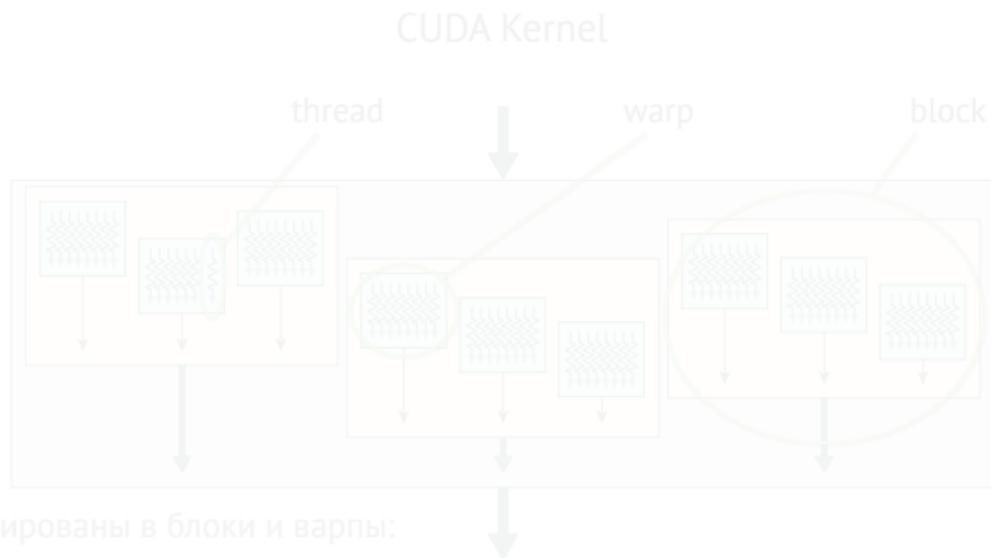
## ■ Потокное исполнение:

- Тысячи параллельных нитей сгруппированы в блоки и варпы:

- Нити каждого блока могут использовать быструю ( $\approx$ на уровне регистров) память, называемую разделяемая/общая (shared) память
- Варпы – это подгруппы нитей внутри блоков, исполняемые синхронно

## ■ SIMT: Simultaneous Instruction – Multiple Threads

- Все нити одного варпа исполняют одни и те же инструкции над разными данными  
(в то время как в традиционном SIMD – одна инструкция применяется ко множеству данных)
- Исполнение различных варпов происходит несинхронно/неодновременно



## ■ Hardware:

- Массивный параллелизм:  
тысячи вычислительных ядер

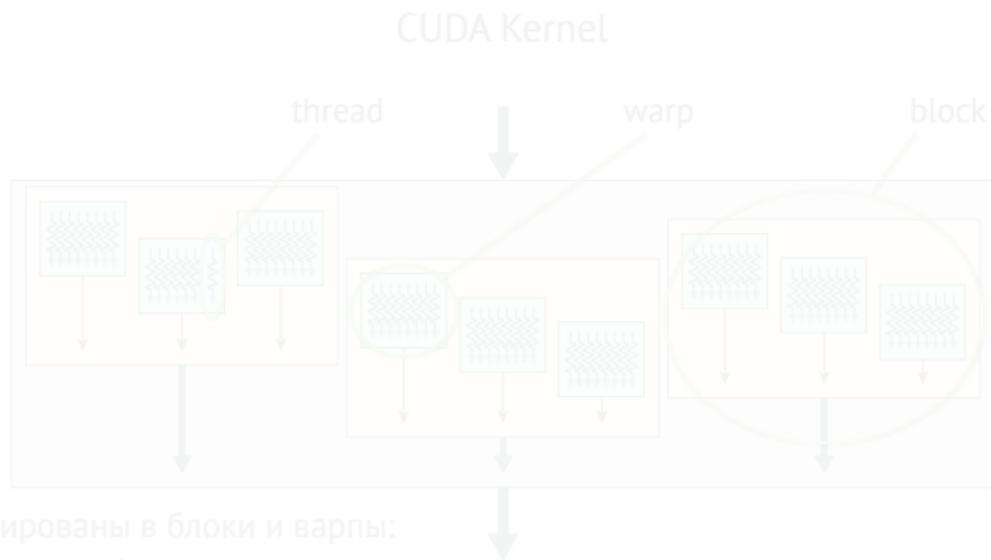
## ■ Потокное исполнение:

- Тысячи параллельных нитей сгруппированы в блоки и варпы:

- Нити каждого блока могут использовать быструю ( $\approx$ на уровне регистров) память, называемую разделяемая/общая (shared) память
- Варпы – это подгруппы нитей внутри блоков, исполняемые синхронно

## ■ SIMT: Simultaneous Instruction – Multiple Threads

- Все нити одного варпа исполняют одни и те же инструкции над разными данными  
(в то время как в традиционном SIMD – одна инструкция применяется ко множеству данных)
- Исполнение различных варпов происходит несинхронно/неодновременно

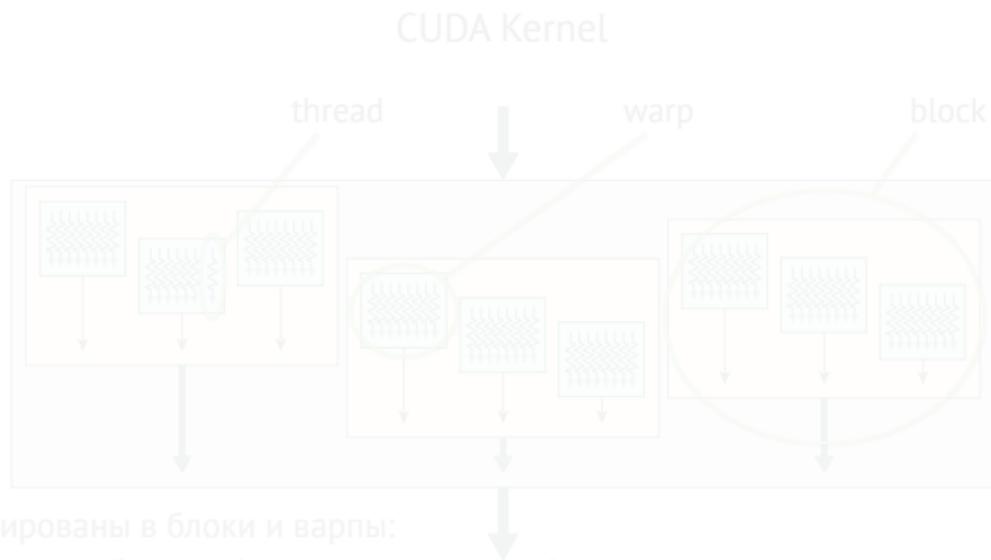


## ■ Hardware:

- Массивный параллелизм:  
тысячи вычислительных ядер

## ■ Потокное исполнение:

- Тысячи параллельных нитей сгруппированы в блоки и варпы:
  - Нити каждого блока могут использовать быструю ( $\approx$ на уровне регистров) память, называемую разделяемая/общая (shared) память
  - Варпы – это подгруппы нитей внутри блоков, исполняемые синхронно
- **SIMT: Simultaneous Instruction – Multiple Threads**
  - Все нити одного варпа исполняют одни и те же инструкции над разными данными  
(в то время как в традиционном SIMD – одна инструкция применяется ко множеству данных)
  - Исполнение различных варпов происходит несинхронно/неодновременно



## ■ Hardware:

- Массивный параллелизм:  
тысячи вычислительных ядер

## ■ Потокное исполнение:

- Тысячи параллельных нитей сгруппированы в блоки и варпы:

- Нити каждого блока могут использовать быструю ( $\approx$ на уровне регистров) память, называемую разделяемая/общая (shared) память

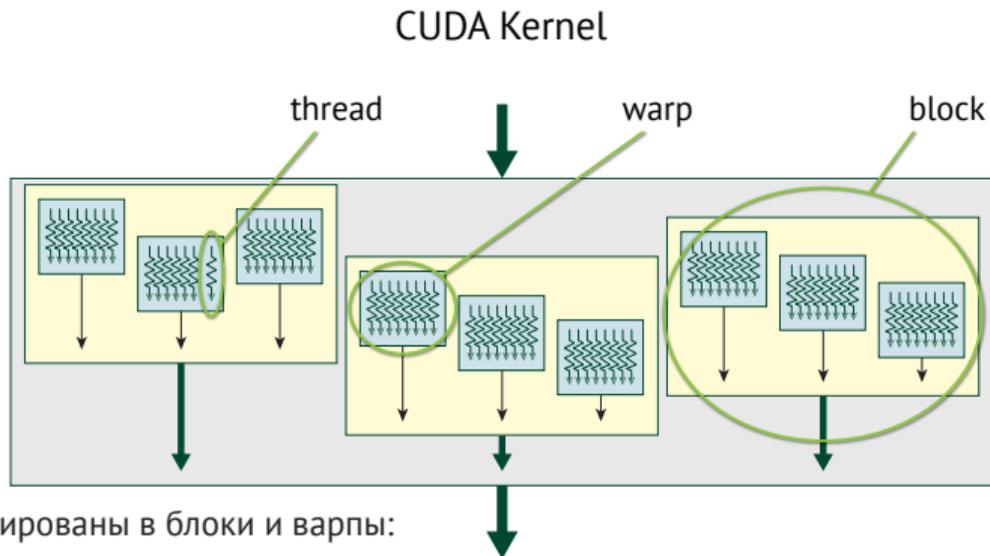
- Варпы – это подгруппы нитей внутри блоков, исполняемые синхронно

## ■ SIMT: Simultaneous Instruction – Multiple Threads

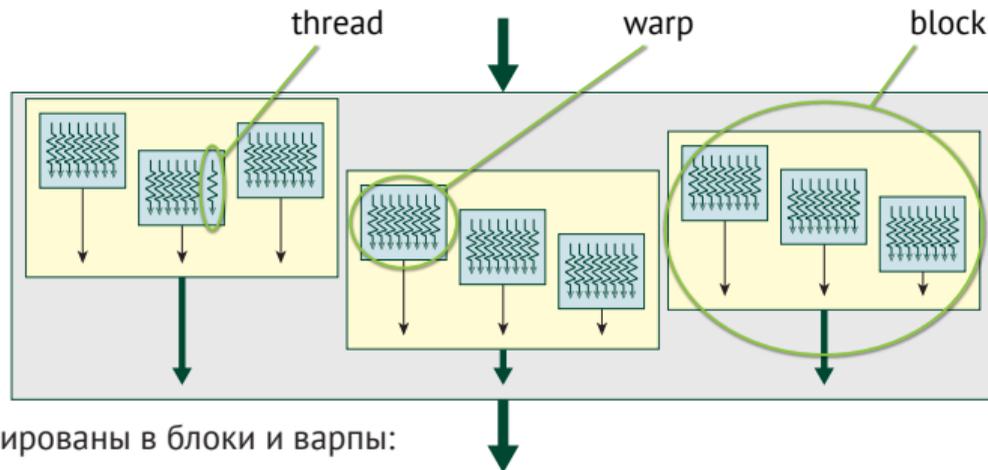
- Все нити одного варпа исполняют одни и те же инструкции над разными данными

(в то время как в традиционном SIMD – одна инструкция применяется ко множеству данных)

- Исполнение различных варпов происходит несинхронно/неодновременно



## CUDA Kernel



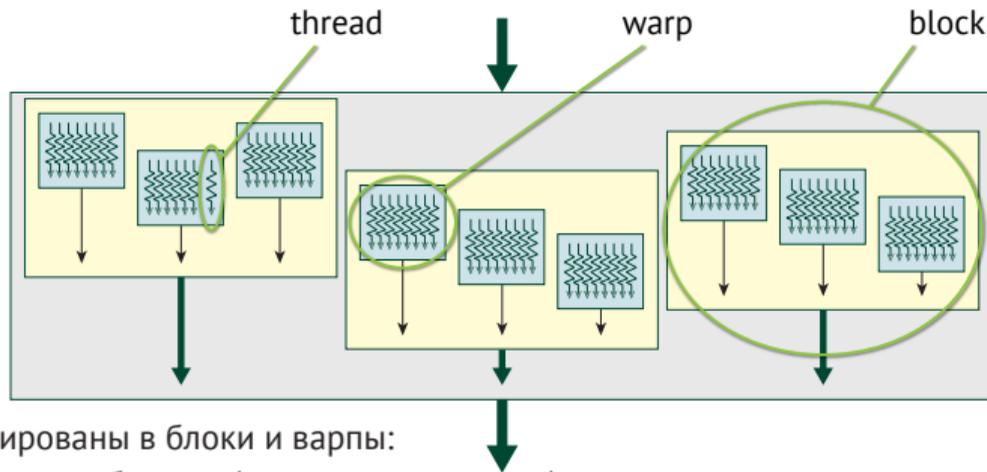
### ■ Hardware:

- Массивный параллелизм:  
тысячи вычислительных ядер

### ■ Потокное исполнение:

- Тысячи параллельных нитей сгруппированы в блоки и варпы:
  - Нити каждого блока могут использовать быструю ( $\approx$  на уровне регистров) память, называемую разделяемая/общая (shared) память
  - Варпы – это подгруппы нитей внутри блоков, исполняемые синхронно
- SIMT: Simultaneous Instruction – Multiple Threads
  - Все нити одного варпа исполняют одни и те же инструкции над разными данными  
(в то время как в традиционном SIMD – одна инструкция применяется ко множеству данных)
  - Исполнение различных варпов происходит несинхронно/неодновременно

## CUDA Kernel



### ■ Hardware:

- Массивный параллелизм:  
тысячи вычислительных ядер

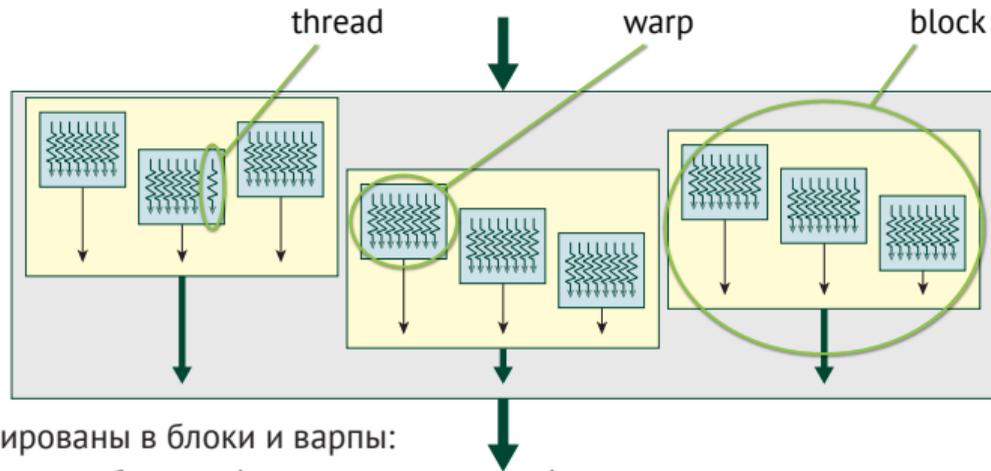
### ■ Потокное исполнение:

- Тысячи параллельных нитей сгруппированы в блоки и варпы:
  - Нити каждого блока могут использовать быструю ( $\approx$ на уровне регистров) память, называемую разделяемая/общая (shared) память
  - Варпы – это подгруппы нитей внутри блоков, исполняемые синхронно

### ■ SIMT: Simultaneous Instruction – Multiple Threads

- Все нити одного варпа исполняют одни и те же инструкции над разными данными  
(в то время как в традиционном SIMD – одна инструкция применяется ко множеству данных)
- Исполнение различных варпов происходит несинхронно/неодновременно

## CUDA Kernel



### ■ Hardware:

- Массивный параллелизм:  
тысячи вычислительных ядер

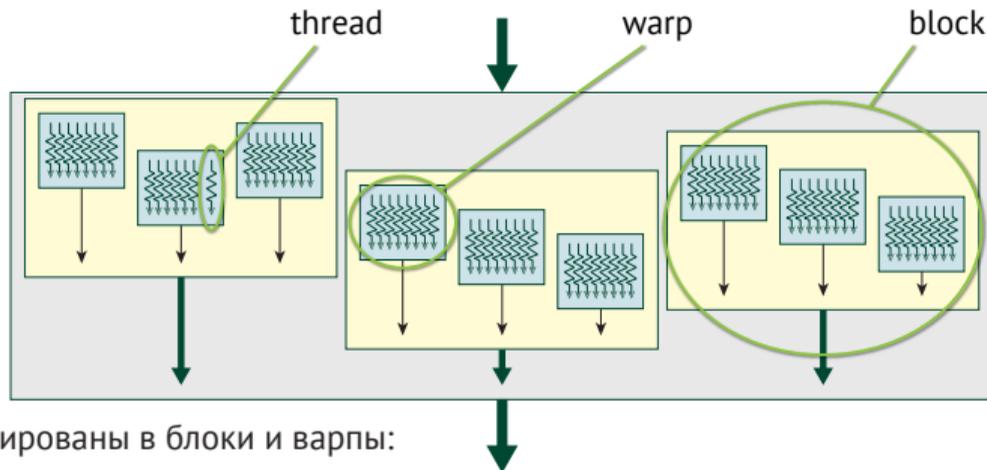
### ■ Потокное исполнение:

- Тысячи параллельных нитей сгруппированы в блоки и варпы:
  - Нити каждого блока могут использовать быструю ( $\approx$ на уровне регистров) память, называемую разделяемая/общая (shared) память
  - Варпы – это подгруппы нитей внутри блоков, исполняемые синхронно

### ■ **SIMT: Simultaneous Instruction – Multiple Threads**

- Все нити одного варпа исполняют одни и те же инструкции над разными данными  
(в то время как в традиционном SIMD – одна инструкция применяется ко множеству данных)
- Исполнение различных варпов происходит несинхронно/неодновременно

## CUDA Kernel



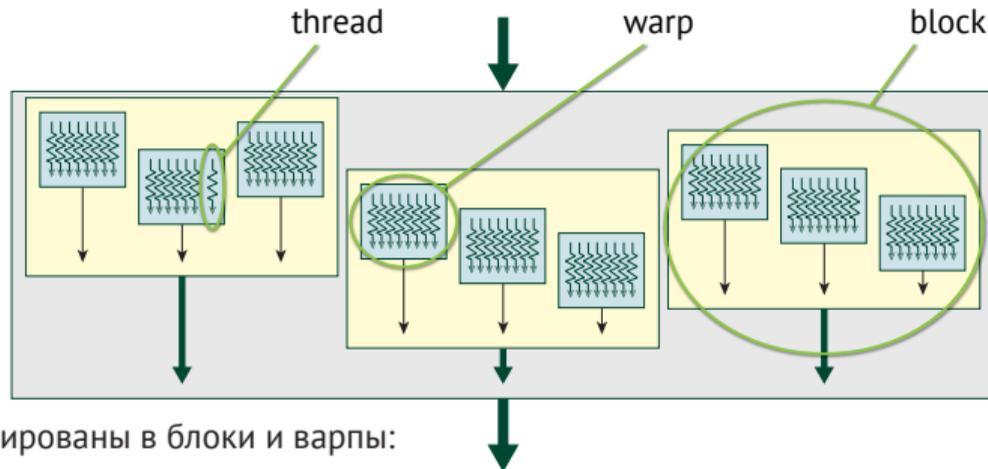
### ■ Hardware:

- Массивный параллелизм:  
тысячи вычислительных ядер

### ■ Потокное исполнение:

- Тысячи параллельных нитей сгруппированы в блоки и варпы:
  - Нити каждого блока могут использовать быструю ( $\approx$ на уровне регистров) память, называемую разделяемая/общая (shared) память
  - Варпы – это подгруппы нитей внутри блоков, исполняемые синхронно
- **SIMT: Simultaneous Instruction – Multiple Threads**
  - Все нити одного варпа исполняют одни и те же инструкции над разными данными  
(в то время как в традиционном SIMD – одна инструкция применяется ко множеству данных)
  - Исполнение различных варпов происходит несинхронно/неодновременно

## CUDA Kernel



### ■ Hardware:

- Массивный параллелизм:  
тысячи вычислительных ядер

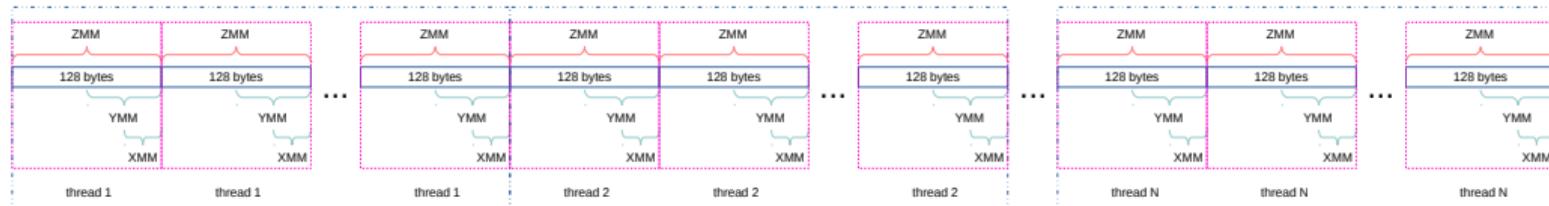
### ■ Потокное исполнение:

- Тысячи параллельных нитей сгруппированы в блоки и варпы:
  - Нити каждого блока могут использовать быструю ( $\approx$ на уровне регистров) память, называемую разделяемая/общая (shared) память
  - Варпы – это подгруппы нитей внутри блоков, исполняемые синхронно
- **SIMT: Simultaneous Instruction – Multiple Threads**
  - Все нити одного варпа исполняют одни и те же инструкции над разными данными  
(в то время как в традиционном SIMD – одна инструкция применяется ко множеству данных)
  - Исполнение различных варпов происходит несинхронно/неодновременно

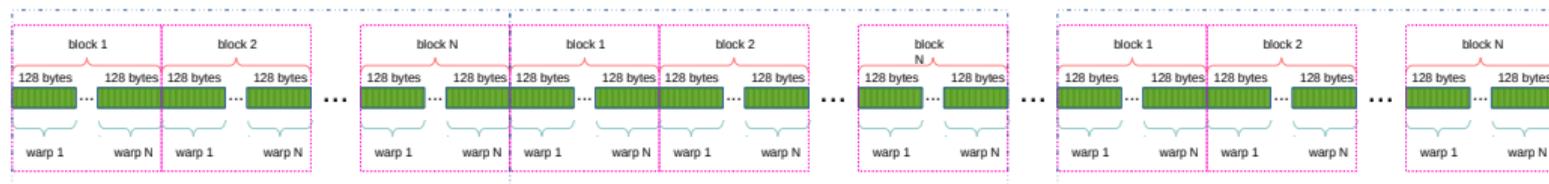
Рассмотрим, как вычисления будут распределены между потоками/нитеями в следующем цикле ( $N \gg$  количество нитей):

```
float *A, *B, *C = ...; for (int i = 0; i < N; i++) A[i] = B[i] + C[i];
```

- SIMD-схема для CPU с поддержкой AVX-512 (512-битные векторные регистры – Xeon Phi и 2015' CPU):



- SIMT-схема для CUDA/GPU с 32 нитями в варпе:

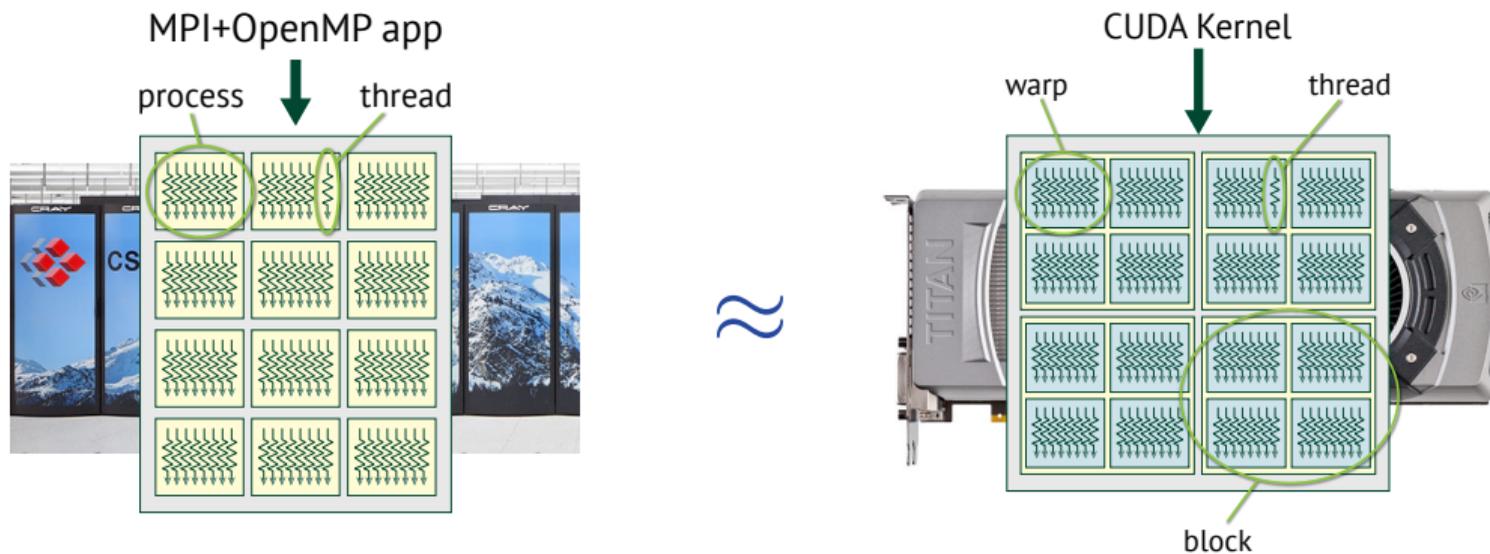


█ - thread (lightweight GPU-thread)

⇒ SIMT позволяет CUDA GPU производить “векторные” вычисления с помощью скалярных ядер, что гораздо проще, чем заставить компилятор автоматически векторизовать код для CPU и уж точно проще, чем векторизовать его вручную.

# CUDA in a nutshell

Имея представление об MPI и OpenMP, очень легко понять устройство CUDA:



⇒ CUDA GPU – это  $\approx$  MPI+OpenMP кластер в микромасштабе, помещенный в 11-дюймовую коробочку!

# CUDA вычислительная сетка (compute grid)

MPI	OpenMP	CUDA
MPI_Comm_rank() MPI_Comm_size()	omp_get_thread_num() omp_get_num_threads()	blockIdx gridDim threadIdx blockDim

mpi\_openmp\_test.c

```
int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int block_idx, grid_dim;
    MPI_Comm_rank(MPI_COMM_WORLD, &block_idx);
    MPI_Comm_size(MPI_COMM_WORLD, &grid_dim);

    #pragma omp parallel
    {
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            block_idx, grid_dim,
            omp_get_thread_num(), omp_get_num_threads());
    }
    MPI_Finalize();
    return 0;
}
```

cuda\_gpu\_test.cu

```
__global__ void gpu_kernel() {

    int block_idx, grid_dim;
    block_idx = blockIdx.x;
    grid_dim = gridDim.x;

    printf("Hello from b %#d of %d, t %#d of %d!\n",
        block_idx, grid_dim,
        threadIdx.x, blockDim.x);
}
```

# CUDA вычислительная сетка (compute grid)

MPI	OpenMP	CUDA
MPI_Comm_rank() MPI_Comm_size()	omp_get_thread_num() omp_get_num_threads()	blockIdx gridDim threadIdx blockDim

mpi\_openmp\_test.c

```
int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int block_idx, grid_dim;
    MPI_Comm_rank(MPI_COMM_WORLD, &block_idx);
    MPI_Comm_size(MPI_COMM_WORLD, &grid_dim);

    #pragma omp parallel
    {
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            block_idx, grid_dim,
            omp_get_thread_num(), omp_get_num_threads());
    }
    MPI_Finalize();
    return 0;
}
```

cuda\_gpu\_test.cu

```
__global__ void gpu_kernel() {

    int block_idx, grid_dim;
    block_idx = blockIdx.x;
    grid_dim = gridDim.x;

    printf("Hello from b %#d of %d, t %#d of %d!\n",
        block_idx, grid_dim,
        threadIdx.x, blockDim.x);
}
```

# CUDA вычислительная сетка (compute grid)

MPI	OpenMP	CUDA
MPI_Comm_rank() MPI_Comm_size()	omp_get_thread_num() omp_get_num_threads()	blockIdx gridDim threadIdx blockDim

mpi\_openmp\_test.c

```
int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int block_idx, grid_dim;
    MPI_Comm_rank(MPI_COMM_WORLD, &block_idx);
    MPI_Comm_size(MPI_COMM_WORLD, &grid_dim);

    #pragma omp parallel
    {
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            block_idx, grid_dim,
            omp_get_thread_num(), omp_get_num_threads());
    }
    MPI_Finalize();
    return 0;
}
```

cuda\_gpu\_test.cu

```
__global__ void gpu_kernel() {

    int block_idx, grid_dim;
    block_idx = blockIdx.x;
    grid_dim = gridDim.x;

    printf("Hello from b %#d of %d, t %#d of %d!\n",
        block_idx, grid_dim,
        threadIdx.x, blockDim.x);
}
```

# CUDA вычислительная сетка (compute grid)

MPI	OpenMP	CUDA
MPI_Comm_rank() MPI_Comm_size()	omp_get_thread_num() omp_get_num_threads()	blockIdx gridDim threadIdx blockDim

mpi\_openmp\_test.c

```
int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int block_idx, grid_dim;
    MPI_Comm_rank(MPI_COMM_WORLD, &block_idx);
    MPI_Comm_size(MPI_COMM_WORLD, &grid_dim);

    #pragma omp parallel
    {
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            block_idx, grid_dim,
            omp_get_thread_num(), omp_get_num_threads());
    }
    MPI_Finalize();
    return 0;
}
```

cuda\_gpu\_test.cu

```
__global__ void gpu_kernel() {

    int block_idx, grid_dim;
    block_idx = blockIdx.x;
    grid_dim = gridDim.x;

    printf("Hello from b %#d of %d, t %#d of %d!\n",
        block_idx, grid_dim,
        threadIdx.x, blockDim.x);
}
```

# CUDA вычислительная сетка (compute grid)

MPI	OpenMP	CUDA
MPI_Comm_rank() MPI_Comm_size()	omp_get_thread_num() omp_get_num_threads()	blockIdx gridDim threadIdx blockDim

mpi\_openmp\_test.c

```
int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int block_idx, grid_dim;
    MPI_Comm_rank(MPI_COMM_WORLD, &block_idx);
    MPI_Comm_size(MPI_COMM_WORLD, &grid_dim);

    #pragma omp parallel
    {
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            block_idx, grid_dim,
            omp_get_thread_num(), omp_get_num_threads());
    }
    MPI_Finalize();
    return 0;
}
```

cuda\_gpu\_test.cu

```
__global__ void gpu_kernel() {

    int block_idx, grid_dim;
    block_idx = blockIdx.x;
    grid_dim = gridDim.x;

    printf("Hello from b %#d of %d, t %#d of %d!\n",
        block_idx, grid_dim,
        threadIdx.x, blockDim.x);
}
```

# CUDA вычислительная сетка (compute grid)

MPI	OpenMP	CUDA
MPI_Comm_rank() MPI_Comm_size()	omp_get_thread_num() omp_get_num_threads()	blockIdx gridDim threadIdx blockDim

mpi\_openmp\_test.c

```
int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int block_idx, grid_dim;
    MPI_Comm_rank(MPI_COMM_WORLD, &block_idx);
    MPI_Comm_size(MPI_COMM_WORLD, &grid_dim);

    #pragma omp parallel
    {
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            block_idx, grid_dim,
            omp_get_thread_num(), omp_get_num_threads());
    }
    MPI_Finalize();
    return 0;
}
```

cuda\_gpu\_test.cu

```
__global__ void gpu_kernel() {

    int block_idx, grid_dim;
    block_idx = blockIdx.x;
    grid_dim = gridDim.x;

    printf("Hello from b %#d of %d, t %#d of %d!\n",
        block_idx, grid_dim,
        threadIdx.x, blockDim.x);

}
```

# CUDA вычислительная сетка (compute grid)

MPI	OpenMP	CUDA
MPI_Comm_rank() MPI_Comm_size()	omp_get_thread_num() omp_get_num_threads()	blockIdx gridDim threadIdx blockDim

mpi\_openmp\_test.c

```
int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int block_idx, grid_dim;
    MPI_Comm_rank(MPI_COMM_WORLD, &block_idx);
    MPI_Comm_size(MPI_COMM_WORLD, &grid_dim);

    #pragma omp parallel
    {
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            block_idx, grid_dim,
            omp_get_thread_num(), omp_get_num_threads());
    }
    MPI_Finalize();
    return 0;
}
```

cuda\_gpu\_test.cu

```
__global__ void gpu_kernel() {

    int block_idx, grid_dim;
    block_idx = blockIdx.x;
    grid_dim = gridDim.x;

    printf("Hello from b %#d of %d, t %#d of %d!\n",
        block_idx, grid_dim,
        threadIdx.x, blockDim.x);
}
```

# CUDA вычислительная сетка (compute grid)

MPI	OpenMP	CUDA
MPI_Comm_rank() MPI_Comm_size()	omp_get_thread_num() omp_get_num_threads()	blockIdx gridDim threadIdx blockDim

mpi\_openmp\_test.c

```
int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int block_idx, grid_dim;
    MPI_Comm_rank(MPI_COMM_WORLD, &block_idx);
    MPI_Comm_size(MPI_COMM_WORLD, &grid_dim);

    #pragma omp parallel
    {
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            block_idx, grid_dim,
            omp_get_thread_num(), omp_get_num_threads());
    }
    MPI_Finalize();
    return 0;
}
```

cuda\_gpu\_test.cu

```
__global__ void gpu_kernel() {

    int block_idx, grid_dim;
    block_idx = blockIdx.x;
    grid_dim = gridDim.x;

    printf("Hello from b %#d of %d, t %#d of %d!\n",
        block_idx, grid_dim,
        threadIdx.x, blockDim.x);

}
```

# CUDA вычислительная сетка (compute grid)

MPI	OpenMP	CUDA
MPI_Comm_rank() MPI_Comm_size()	omp_get_thread_num() omp_get_num_threads()	blockIdx gridDim threadIdx blockDim

mpi\_openmp\_test.c

```
int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int block_idx, grid_dim;
    MPI_Comm_rank(MPI_COMM_WORLD, &block_idx);
    MPI_Comm_size(MPI_COMM_WORLD, &grid_dim);

    #pragma omp parallel
    {
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            block_idx, grid_dim,
            omp_get_thread_num(), omp_get_num_threads());
    }
    MPI_Finalize();
    return 0;
}
```

cuda\_gpu\_test.cu

```
__global__ void gpu_kernel() {

    int block_idx, grid_dim;
    block_idx = blockIdx.x;
    grid_dim = gridDim.x;

    printf("Hello from b %#d of %d, t %#d of %d!\n",
        block_idx, grid_dim,
        threadIdx.x, blockDim.x);

}
```

# CUDA вычислительная сетка (compute grid)

MPI	OpenMP	CUDA
MPI_Comm_rank() MPI_Comm_size()	omp_get_thread_num() omp_get_num_threads()	blockIdx gridDim threadIdx blockDim

mpi\_openmp\_test.c

```
int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int block_idx, grid_dim;
    MPI_Comm_rank(MPI_COMM_WORLD, &block_idx);
    MPI_Comm_size(MPI_COMM_WORLD, &grid_dim);

    #pragma omp parallel
    {
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            block_idx, grid_dim,
            omp_get_thread_num(), omp_get_num_threads());
    }
    MPI_Finalize();
    return 0;
}
```

cuda\_gpu\_test.cu

```
__global__ void gpu_kernel() {

    int block_idx, grid_dim;
    block_idx = blockIdx.x;
    grid_dim = gridDim.x;

    printf("Hello from b %#d of %d, t %#d of %d!\n",
        block_idx, grid_dim,
        threadIdx.x, blockDim.x);
}
```

# CUDA вычислительная сетка (compute grid)

MPI	OpenMP	CUDA
MPI_Comm_rank() MPI_Comm_size()	omp_get_thread_num() omp_get_num_threads()	blockIdx gridDim threadIdx blockDim

mpi\_openmp\_test.c

```
int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int block_idx, grid_dim;
    MPI_Comm_rank(MPI_COMM_WORLD, &block_idx);
    MPI_Comm_size(MPI_COMM_WORLD, &grid_dim);

    #pragma omp parallel
    {
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            block_idx, grid_dim,
            omp_get_thread_num(), omp_get_num_threads());
    }
    MPI_Finalize();
    return 0;
}
```

cuda\_gpu\_test.cu

```
__global__ void gpu_kernel() {

    int block_idx, grid_dim;
    block_idx = blockIdx.x;
    grid_dim = gridDim.x;

    printf("Hello from b %#d of %d, t %#d of %d!\n",
        block_idx, grid_dim,
        threadIdx.x, blockDim.x);
}
```

## mpi\_omp\_test.c

```
int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int block_idx, grid_dim;
    MPI_Comm_rank(MPI_COMM_WORLD, &block_idx);
    MPI_Comm_size(MPI_COMM_WORLD, &grid_dim);

    #pragma omp parallel
    {
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            block_idx, grid_dim,
            omp_get_thread_num(), omp_get_num_threads());
    }
    MPI_Finalize();
    return 0;
}
```

```
$ OMP_NUM_THREADS=2 mpirun -np 2 ./mpi_omp_test
```

```
Hello from b #0 of 2, t #0 of 2!
Hello from b #0 of 2, t #1 of 2!
Hello from b #1 of 2, t #0 of 2!
Hello from b #1 of 2, t #1 of 2!
```

## cuda\_gpu\_test.cu

```
__global__ void gpu_kernel() {

    int block_idx, grid_dim;
    block_idx = blockIdx.x;
    grid_dim = gridDim.x;

    printf("Hello from b %#d of %d, t %#d of %d!\n",
        block_idx, grid_dim,
        threadIdx.x, blockDim.x);
}
```

```
int main() {
    gpu_kernel<<<2, 2>>>();
    CUDA_ERR_CHECK( cudaGetLastError() );
    CUDA_ERR_CHECK( cudaDeviceSynchronize() );
    return 0;
}
```

```
Hello from b #0 of 2, t #0 of 2!
Hello from b #0 of 2, t #1 of 2!
Hello from b #1 of 2, t #0 of 2!
Hello from b #1 of 2, t #1 of 2!
```

## mpi\_omp\_test.c

```
int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int block_idx, grid_dim;
    MPI_Comm_rank(MPI_COMM_WORLD, &block_idx);
    MPI_Comm_size(MPI_COMM_WORLD, &grid_dim);

    #pragma omp parallel
    {
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            block_idx, grid_dim,
            omp_get_thread_num(), omp_get_num_threads());
    }
    MPI_Finalize();
    return 0;
}
```

```
$ OMP_NUM_THREADS=2 mpirun -np 2 ./mpi_omp_test
```

```
Hello from b #0 of 2, t #0 of 2!
Hello from b #0 of 2, t #1 of 2!
Hello from b #1 of 2, t #0 of 2!
Hello from b #1 of 2, t #1 of 2!
```

## cuda\_gpu\_test.cu

```
__global__ void gpu_kernel() {

    int block_idx, grid_dim;
    block_idx = blockIdx.x;
    grid_dim = gridDim.x;

    printf("Hello from b %#d of %d, t %#d of %d!\n",
        block_idx, grid_dim,
        threadIdx.x, blockDim.x);
}
```

```
int main() {
    gpu_kernel<<<2, 2>>>();
    CUDA_ERR_CHECK( cudaGetLastError() );
    CUDA_ERR_CHECK( cudaDeviceSynchronize() );
    return 0;
}
```

```
Hello from b #0 of 2, t #0 of 2!
Hello from b #0 of 2, t #1 of 2!
Hello from b #1 of 2, t #0 of 2!
Hello from b #1 of 2, t #1 of 2!
```

## mpi\_omp\_test.c

```
int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int block_idx, grid_dim;
    MPI_Comm_rank(MPI_COMM_WORLD, &block_idx);
    MPI_Comm_size(MPI_COMM_WORLD, &grid_dim);

    #pragma omp parallel
    {
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            block_idx, grid_dim,
            omp_get_thread_num(), omp_get_num_threads());
    }
    MPI_Finalize();
    return 0;
}
```

```
$ OMP_NUM_THREADS=2 mpirun -np 2 ./mpi_omp_test
```

```
Hello from b #0 of 2, t #0 of 2!
Hello from b #0 of 2, t #1 of 2!
Hello from b #1 of 2, t #0 of 2!
Hello from b #1 of 2, t #1 of 2!
```

## cuda\_gpu\_test.cu

```
__global__ void gpu_kernel() {

    int block_idx, grid_dim;
    block_idx = blockIdx.x;
    grid_dim = gridDim.x;

    printf("Hello from b %#d of %d, t %#d of %d!\n",
        block_idx, grid_dim,
        threadIdx.x, blockDim.x);
}
```

```
int main() {
    gpu_kernel<<<2, 2>>>();
    CUDA_ERR_CHECK( cudaGetLastError() );
    CUDA_ERR_CHECK( cudaDeviceSynchronize() );
    return 0;
}
```

```
Hello from b #0 of 2, t #0 of 2!
Hello from b #0 of 2, t #1 of 2!
Hello from b #1 of 2, t #0 of 2!
Hello from b #1 of 2, t #1 of 2!
```

## mpi\_omp\_test.c

```
int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int block_idx, grid_dim;
    MPI_Comm_rank(MPI_COMM_WORLD, &block_idx);
    MPI_Comm_size(MPI_COMM_WORLD, &grid_dim);

    #pragma omp parallel
    {
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            block_idx, grid_dim,
            omp_get_thread_num(), omp_get_num_threads());
    }
    MPI_Finalize();
    return 0;
}
```

```
$ OMP_NUM_THREADS=2 mpirun -np 2 ./mpi_omp_test
```

```
Hello from b #0 of 2, t #0 of 2!
Hello from b #0 of 2, t #1 of 2!
Hello from b #1 of 2, t #0 of 2!
Hello from b #1 of 2, t #1 of 2!
```

## cuda\_gpu\_test.cu

```
__global__ void gpu_kernel() {

    int block_idx, grid_dim;
    block_idx = blockIdx.x;
    grid_dim = gridDim.x;

    printf("Hello from b %#d of %d, t %#d of %d!\n",
        block_idx, grid_dim,
        threadIdx.x, blockDim.x);
}
```

```
int main() {
    gpu_kernel<<<2, 2>>>();
    CUDA_ERR_CHECK( cudaGetLastError() );
    CUDA_ERR_CHECK( cudaDeviceSynchronize() );
    return 0;
}
```

```
Hello from b #0 of 2, t #0 of 2!
Hello from b #0 of 2, t #1 of 2!
Hello from b #1 of 2, t #0 of 2!
Hello from b #1 of 2, t #1 of 2!
```

## mpi\_omp\_test.c

```
int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int block_idx, grid_dim;
    MPI_Comm_rank(MPI_COMM_WORLD, &block_idx);
    MPI_Comm_size(MPI_COMM_WORLD, &grid_dim);

    #pragma omp parallel
    {
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            block_idx, grid_dim,
            omp_get_thread_num(), omp_get_num_threads());
    }
    MPI_Finalize();
    return 0;
}
```

```
$ OMP_NUM_THREADS=2 mpirun -np 2 ./mpi_omp_test
```

```
Hello from b #0 of 2, t #0 of 2!
Hello from b #0 of 2, t #1 of 2!
Hello from b #1 of 2, t #0 of 2!
Hello from b #1 of 2, t #1 of 2!
```

## cuda\_gpu\_test.cu

```
__global__ void gpu_kernel() {

    int block_idx, grid_dim;
    block_idx = blockIdx.x;
    grid_dim = gridDim.x;

    printf("Hello from b %#d of %d, t %#d of %d!\n",
        block_idx, grid_dim,
        threadIdx.x, blockDim.x);
}
```

```
int main() {
    gpu_kernel<<<2, 2>>>();
    CUDA_ERR_CHECK( cudaGetLastError() );
    CUDA_ERR_CHECK( cudaDeviceSynchronize() );
    return 0;
}
```

```
Hello from b #0 of 2, t #0 of 2!
Hello from b #0 of 2, t #1 of 2!
Hello from b #1 of 2, t #0 of 2!
Hello from b #1 of 2, t #1 of 2!
```

## mpi\_omp\_test.c

```
int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int block_idx, grid_dim;
    MPI_Comm_rank(MPI_COMM_WORLD, &block_idx);
    MPI_Comm_size(MPI_COMM_WORLD, &grid_dim);

    #pragma omp parallel
    {
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            block_idx, grid_dim,
            omp_get_thread_num(), omp_get_num_threads());
    }
    MPI_Finalize();
    return 0;
}
```

```
$ OMP_NUM_THREADS=2 mpirun -np 2 ./mpi_omp_test
```

```
Hello from b #0 of 2, t #0 of 2!
Hello from b #0 of 2, t #1 of 2!
Hello from b #1 of 2, t #0 of 2!
Hello from b #1 of 2, t #1 of 2!
```

## cuda\_gpu\_test.cu

```
__global__ void gpu_kernel() {

    int block_idx, grid_dim;
    block_idx = blockIdx.x;
    grid_dim = gridDim.x;

    printf("Hello from b %#d of %d, t %#d of %d!\n",
        block_idx, grid_dim,
        threadIdx.x, blockDim.x);
}
```

```
int main() {
    gpu_kernel<<<2, 2>>>();
    CUDA_ERR_CHECK( cudaGetLastError() );
    CUDA_ERR_CHECK( cudaDeviceSynchronize() );
    return 0;
}
```

```
Hello from b #0 of 2, t #0 of 2!
Hello from b #0 of 2, t #1 of 2!
Hello from b #1 of 2, t #0 of 2!
Hello from b #1 of 2, t #1 of 2!
```

## mpi\_omp\_test.c

```
int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int block_idx, grid_dim;
    MPI_Comm_rank(MPI_COMM_WORLD, &block_idx);
    MPI_Comm_size(MPI_COMM_WORLD, &grid_dim);

    #pragma omp parallel
    {
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            block_idx, grid_dim,
            omp_get_thread_num(), omp_get_num_threads());
    }
    MPI_Finalize();
    return 0;
}
```

```
$ OMP_NUM_THREADS=2 mpirun -np 2 ./mpi_omp_test
```

```
Hello from b #0 of 2, t #0 of 2!
Hello from b #0 of 2, t #1 of 2!
Hello from b #1 of 2, t #0 of 2!
Hello from b #1 of 2, t #1 of 2!
```

## cuda\_gpu\_test.cu

```
__global__ void gpu_kernel() {

    int block_idx, grid_dim;
    block_idx = blockIdx.x;
    grid_dim = gridDim.x;

    printf("Hello from b %#d of %d, t %#d of %d!\n",
        block_idx, grid_dim,
        threadIdx.x, blockDim.x);
}
```

```
int main() {
    gpu_kernel<<<2, 2>>>();
    CUDA_ERR_CHECK( cudaGetLastError() );
    CUDA_ERR_CHECK( cudaDeviceSynchronize() );
    return 0;
}
```

```
Hello from b #0 of 2, t #0 of 2!
Hello from b #0 of 2, t #1 of 2!
Hello from b #1 of 2, t #0 of 2!
Hello from b #1 of 2, t #1 of 2!
```

## mpi\_omp\_test.c

```
int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int block_idx, grid_dim;
    MPI_Comm_rank(MPI_COMM_WORLD, &block_idx);
    MPI_Comm_size(MPI_COMM_WORLD, &grid_dim);

    #pragma omp parallel
    {
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            block_idx, grid_dim,
            omp_get_thread_num(), omp_get_num_threads());
    }
    MPI_Finalize();
    return 0;
}
```

```
$ OMP_NUM_THREADS=2 mpirun -np 2 ./mpi_omp_test
```

```
Hello from b #0 of 2, t #0 of 2!
Hello from b #0 of 2, t #1 of 2!
Hello from b #1 of 2, t #0 of 2!
Hello from b #1 of 2, t #1 of 2!
```

## cuda\_gpu\_test.cu

```
__global__ void gpu_kernel() {

    int block_idx, grid_dim;
    block_idx = blockIdx.x;
    grid_dim = gridDim.x;

    printf("Hello from b %#d of %d, t %#d of %d!\n",
        block_idx, grid_dim,
        threadIdx.x, blockDim.x);
}
```

```
int main() {
    gpu_kernel<<<2, 2>>>();
    CUDA_ERR_CHECK( cudaGetLastError() );
    CUDA_ERR_CHECK( cudaDeviceSynchronize() );
    return 0;
}
```

```
Hello from b #0 of 2, t #0 of 2!
Hello from b #0 of 2, t #1 of 2!
Hello from b #1 of 2, t #0 of 2!
Hello from b #1 of 2, t #1 of 2!
```

## mpi\_omp\_test.c

```
int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int block_idx, grid_dim;
    MPI_Comm_rank(MPI_COMM_WORLD, &block_idx);
    MPI_Comm_size(MPI_COMM_WORLD, &grid_dim);

    #pragma omp parallel
    {
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            block_idx, grid_dim,
            omp_get_thread_num(), omp_get_num_threads());
    }
    MPI_Finalize();
    return 0;
}
```

```
$ OMP_NUM_THREADS=2 mpirun -np 2 ./mpi_omp_test
```

```
Hello from b #0 of 2, t #0 of 2!
Hello from b #0 of 2, t #1 of 2!
Hello from b #1 of 2, t #0 of 2!
Hello from b #1 of 2, t #1 of 2!
```

## cuda\_gpu\_test.cu

```
__global__ void gpu_kernel() {

    int block_idx, grid_dim;
    block_idx = blockIdx.x;
    grid_dim = gridDim.x;

    printf("Hello from b %#d of %d, t %#d of %d!\n",
        block_idx, grid_dim,
        threadIdx.x, blockDim.x);
}
```

```
int main() {
    gpu_kernel<<<2, 2>>>();
    CUDA_ERR_CHECK( cudaGetLastError() );
    CUDA_ERR_CHECK( cudaDeviceSynchronize() );
    return 0;
}
```

```
Hello from b #0 of 2, t #0 of 2!
Hello from b #0 of 2, t #1 of 2!
Hello from b #1 of 2, t #0 of 2!
Hello from b #1 of 2, t #1 of 2!
```

## mpi\_omp\_test.c

```
int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int block_idx, grid_dim;
    MPI_Comm_rank(MPI_COMM_WORLD, &block_idx);
    MPI_Comm_size(MPI_COMM_WORLD, &grid_dim);

    #pragma omp parallel
    {
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            block_idx, grid_dim,
            omp_get_thread_num(), omp_get_num_threads());
    }
    MPI_Finalize();
    return 0;
}
```

```
$ OMP_NUM_THREADS=2 mpirun -np 2 ./mpi_omp_test
```

```
Hello from b #0 of 2, t #0 of 2!
Hello from b #0 of 2, t #1 of 2!
Hello from b #1 of 2, t #0 of 2!
Hello from b #1 of 2, t #1 of 2!
```

## cuda\_gpu\_test.cu

```
__global__ void gpu_kernel() {

    int block_idx, grid_dim;
    block_idx = blockIdx.x;
    grid_dim = gridDim.x;

    printf("Hello from b %#d of %d, t %#d of %d!\n",
        block_idx, grid_dim,
        threadIdx.x, blockDim.x);
}
```

```
int main() {
    gpu_kernel<<<2, 2>>>();
    CUDA_ERR_CHECK( cudaGetLastError() );
    CUDA_ERR_CHECK( cudaDeviceSynchronize() );
    return 0;
}
```

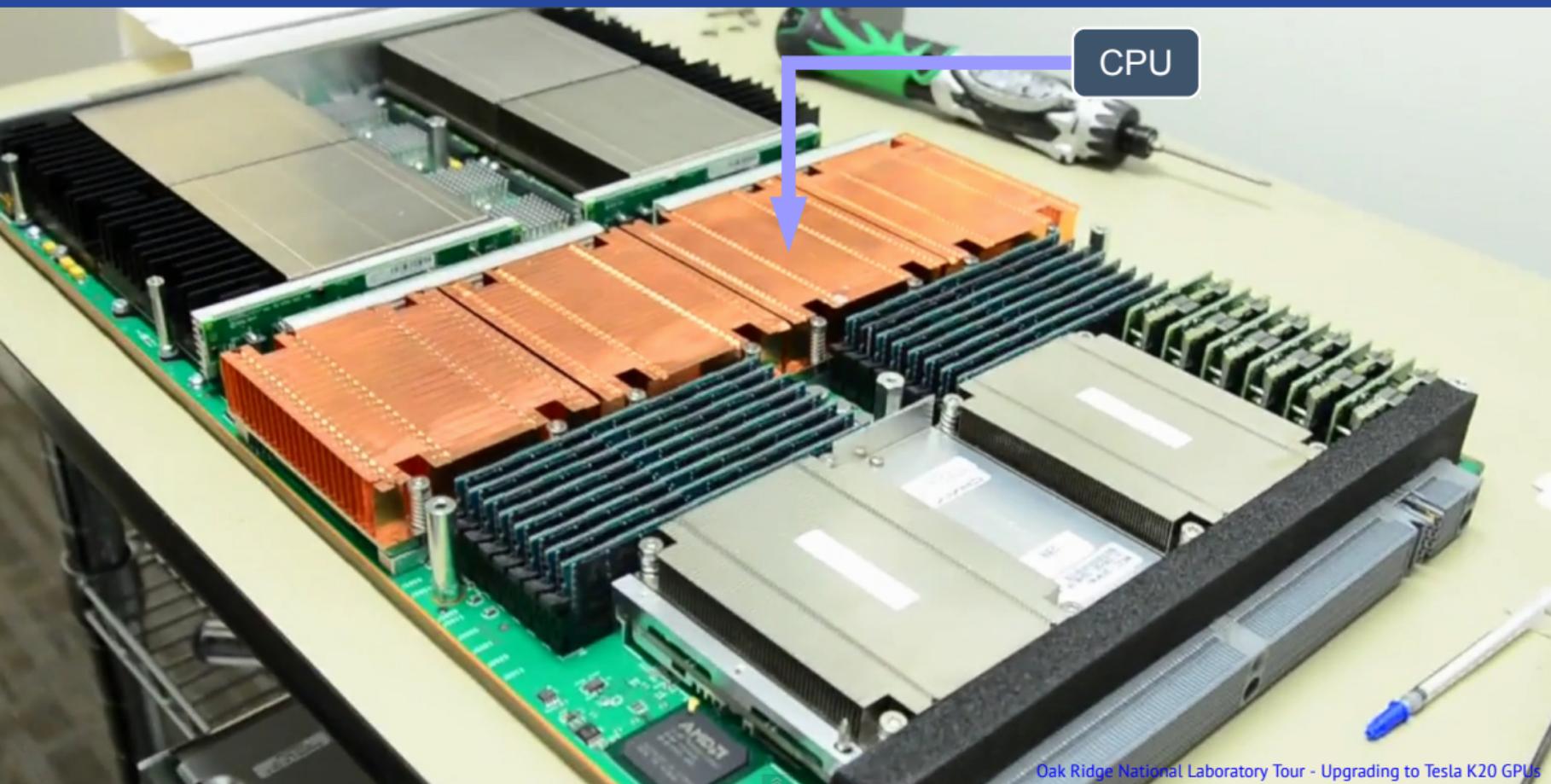
```
Hello from b #0 of 2, t #0 of 2!
Hello from b #0 of 2, t #1 of 2!
Hello from b #1 of 2, t #0 of 2!
Hello from b #1 of 2, t #1 of 2!
```

# Память GPU ↔ память хоста



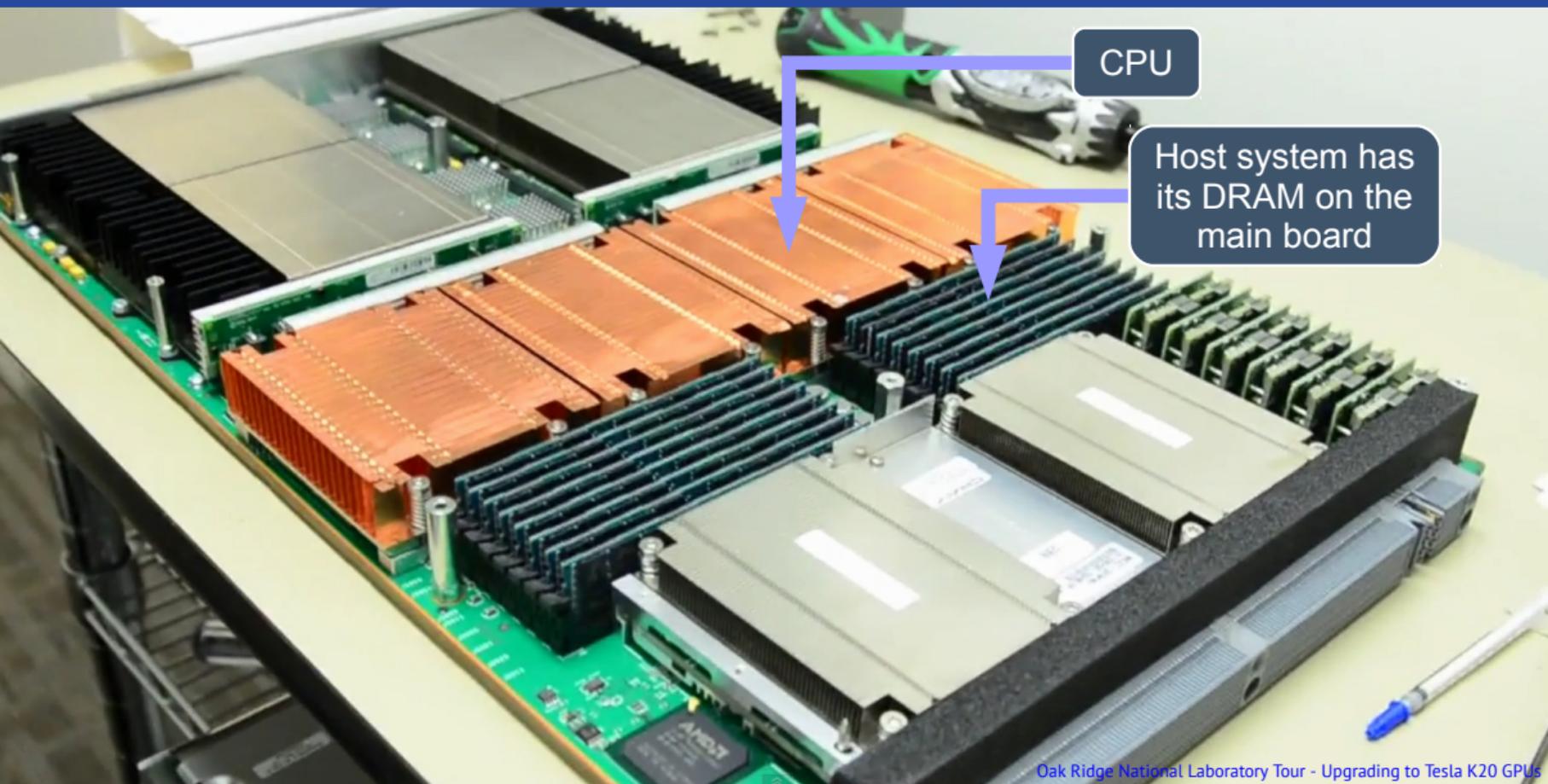
Oak Ridge National Laboratory Tour - Upgrading to Tesla K20 GPUs

# Память GPU ↔ память хоста

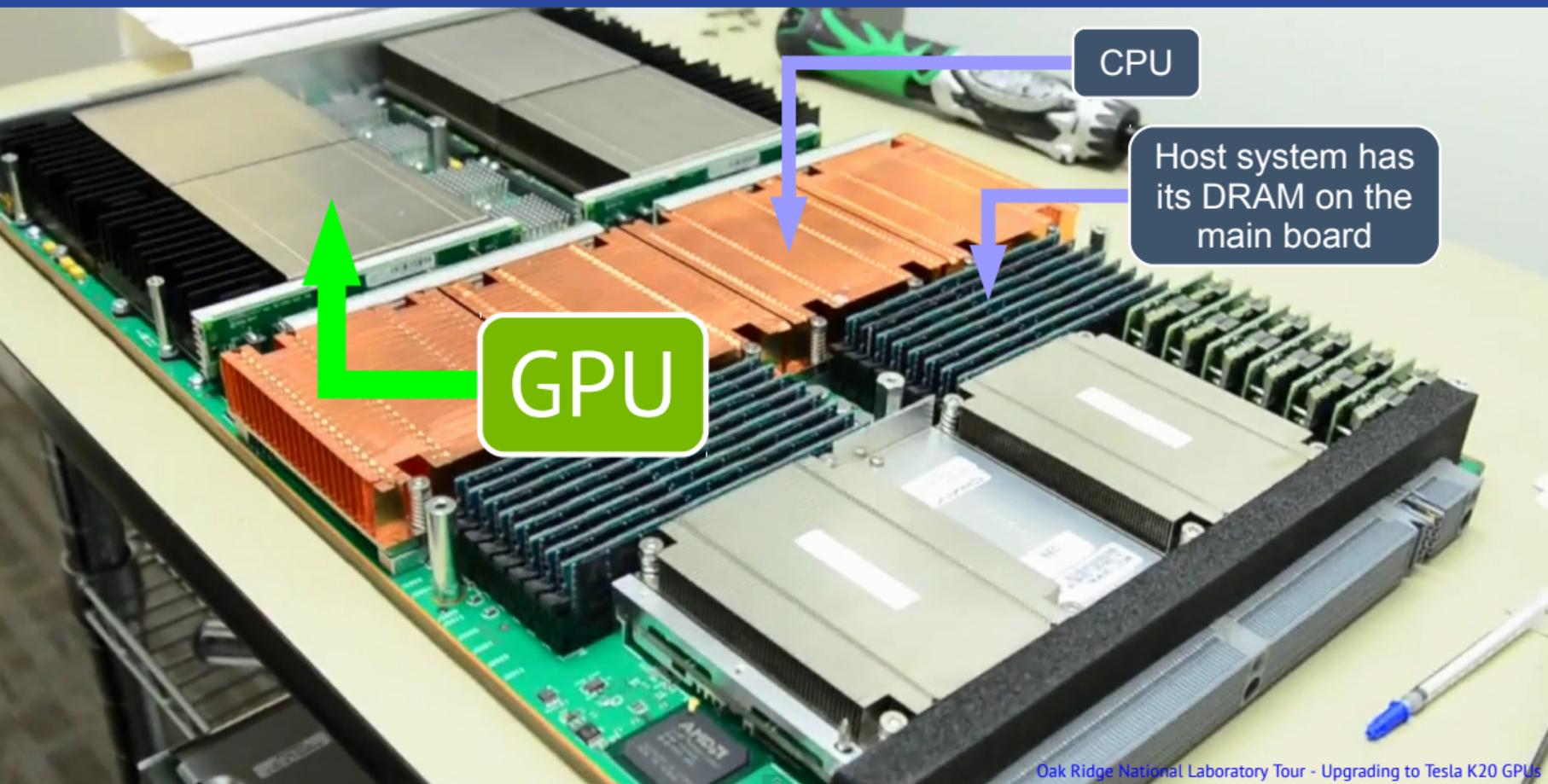


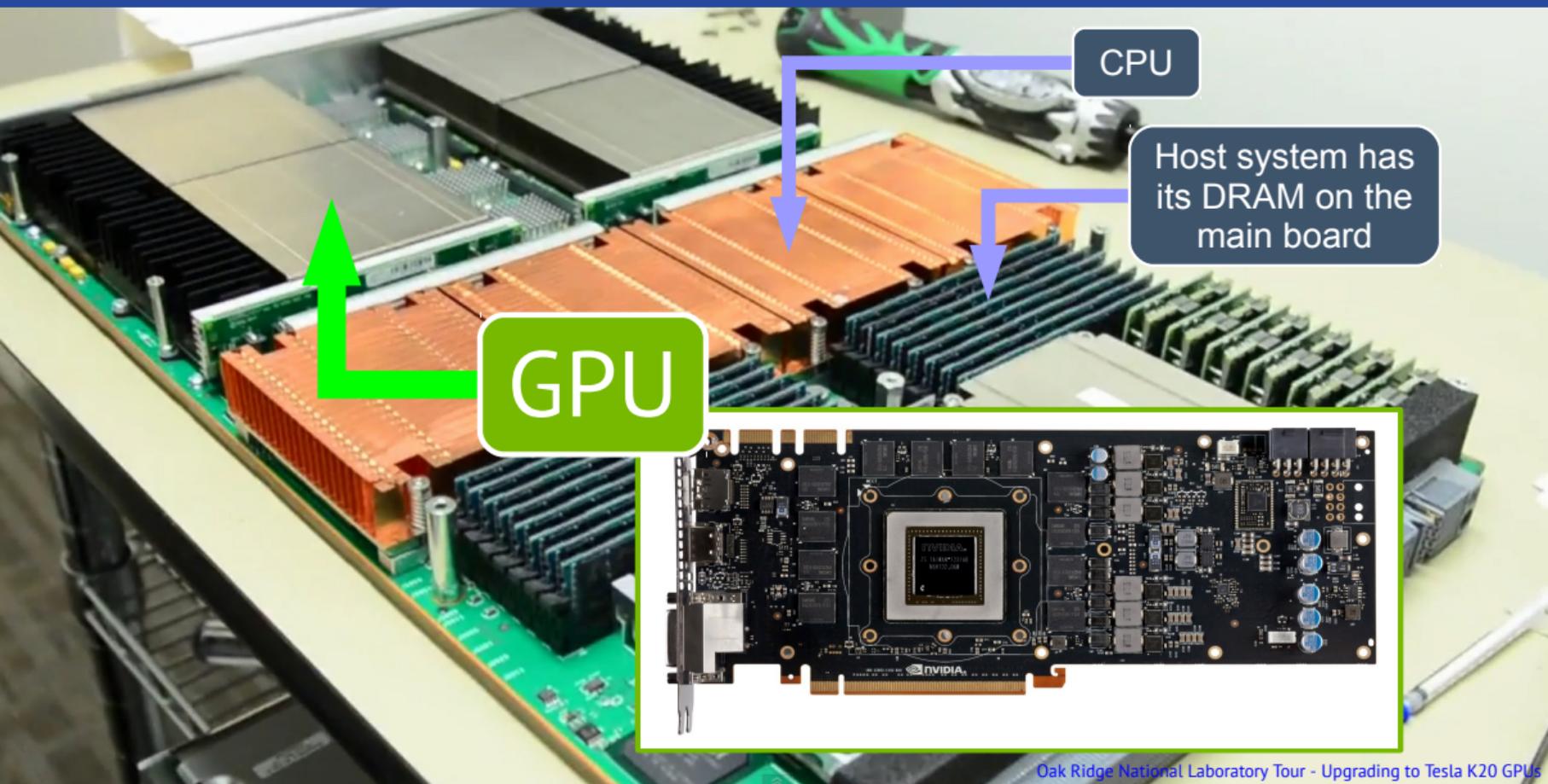
CPU

# Память GPU ↔ память хоста



# Память GPU ↔ память хоста



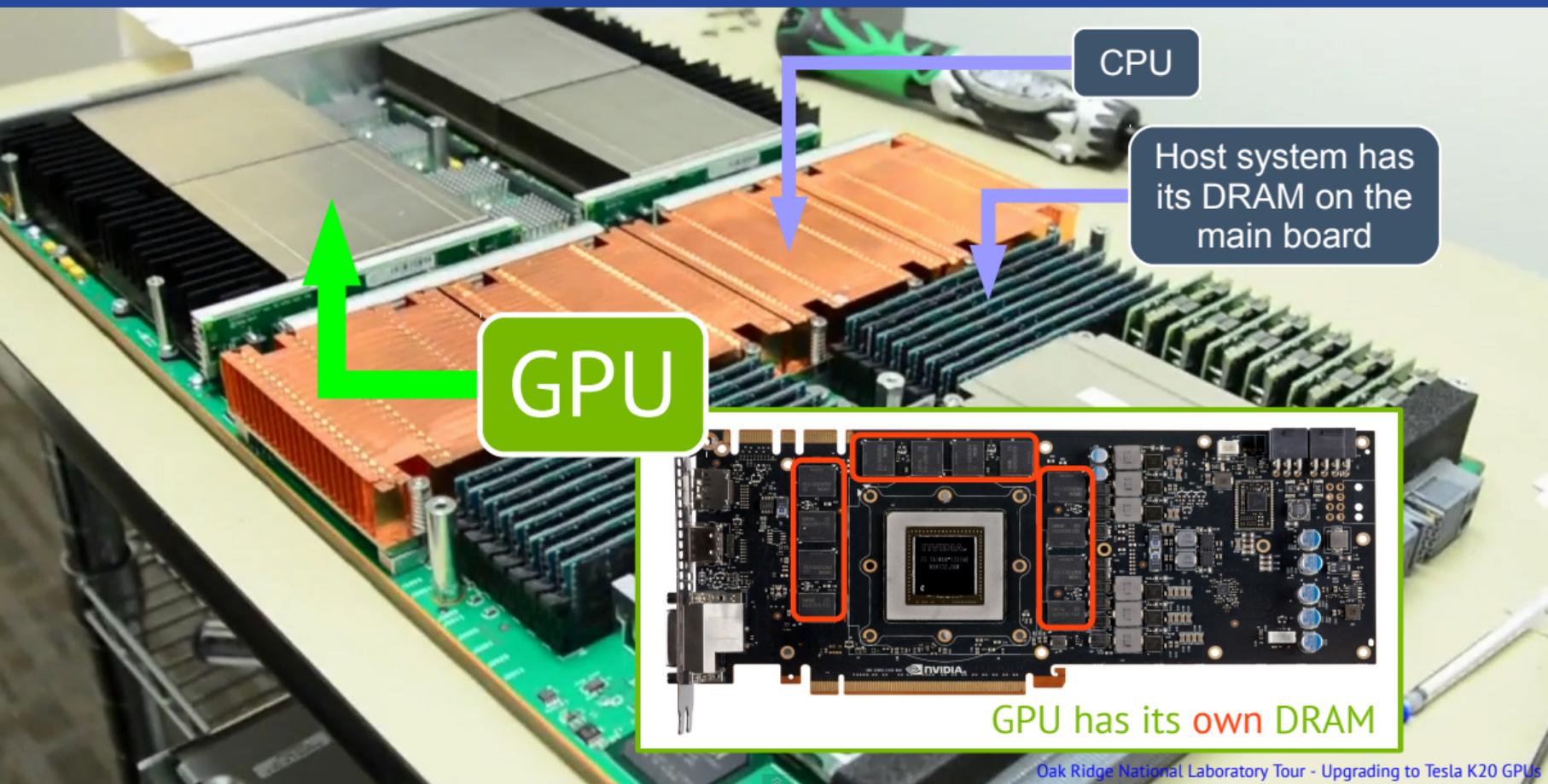


CPU

Host system has its DRAM on the main board

GPU





- Лучшая производительность достигается, когда, данные хранятся в памяти GPU DRAM
- Необходимо выделить память в области GPU DRAM и скопировать данные с хоста на GPU:

```
int main()
{
    int nthreads = 4, nblocks = 4;
    int n = nthreads * nblocks * 4;
    int* host_data = (int*)malloc(n * sizeof(int));
    int* gpu_data = NULL;
    CUDA_ERR_CHECK( cudaMalloc(&gpu_data, n * sizeof(int)) );

    gpu_kernel<<<nblocks, nthreads>>>(gpu_data);
    CUDA_ERR_CHECK( cudaGetLastError() );
    CUDA_ERR_CHECK( cudaMemcpy(host_data, gpu_data, n * sizeof(int), cudaMemcpyDeviceToHost) );
    CUDA_ERR_CHECK( cudaFree(gpu_data) );
    for (int i = 0; i < n; i += 4)
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            host_data[i], host_data[i+1], host_data[i+2], host_data[i+3]);

    free(host_data);

    return 0;
}
```

- Технология NVLink позволяет коду хоста напрямую взаимодействовать с GPU (на поддерживаемых архитектурах – на данный момент IBM Power 8+,9)

- Лучшая производительность достигается, когда, данные хранятся в памяти GPU DRAM
- Необходимо выделить память в области GPU DRAM и скопировать данные с хоста на GPU:

```
int main()
{
    int nthreads = 4, nblocks = 4;
    int n = nthreads * nblocks * 4;
    int* host_data = (int*)malloc(n * sizeof(int));
    int* gpu_data = NULL;
    CUDA_ERR_CHECK( cudaMalloc(&gpu_data, n * sizeof(int)) );

    gpu_kernel<<<nblocks, nthreads>>>(gpu_data);
    CUDA_ERR_CHECK( cudaGetLastError() );
    CUDA_ERR_CHECK( cudaMemcpy(host_data, gpu_data, n * sizeof(int), cudaMemcpyDeviceToHost) );
    CUDA_ERR_CHECK( cudaFree(gpu_data) );
    for (int i = 0; i < n; i += 4)
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            host_data[i], host_data[i+1], host_data[i+2], host_data[i+3]);

    free(host_data);

    return 0;
}
```

- Технология NVLink позволяет коду хоста напрямую взаимодействовать с GPU (на поддерживаемых архитектурах – на данный момент IBM Power 8+,9)

- Лучшая производительность достигается, когда, данные хранятся в памяти GPU DRAM
- Необходимо выделить память в области GPU DRAM и скопировать данные с хоста на GPU:

```
int main()
{
    int nthreads = 4, nblocks = 4;
    int n = nthreads * nblocks * 4;
    int* host_data = (int*)malloc(n * sizeof(int));
    int* gpu_data = NULL;
    CUDA_ERR_CHECK( cudaMalloc(&gpu_data, n * sizeof(int)) );

    gpu_kernel<<<nblocks, nthreads>>>(gpu_data);
    CUDA_ERR_CHECK( cudaGetLastError() );
    CUDA_ERR_CHECK( cudaMemcpy(host_data, gpu_data, n * sizeof(int), cudaMemcpyDeviceToHost) );
    CUDA_ERR_CHECK( cudaFree(gpu_data) );
    for (int i = 0; i < n; i += 4)
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            host_data[i], host_data[i+1], host_data[i+2], host_data[i+3]);

    free(host_data);

    return 0;
}
```

- Технология NVLink позволяет коду хоста напрямую взаимодействовать с GPU (на поддерживаемых архитектурах – на данный момент IBM Power 8+,9)

- Лучшая производительность достигается, когда, данные хранятся в памяти GPU DRAM
- Необходимо выделить память в области GPU DRAM и скопировать данные с хоста на GPU:

```
int main()
{
    int nthreads = 4, nblocks = 4;
    int n = nthreads * nblocks * 4;
    int* host_data = (int*)malloc(n * sizeof(int));
    int* gpu_data = NULL;
    CUDA_ERR_CHECK( cudaMalloc(&gpu_data, n * sizeof(int)) );

    gpu_kernel<<<nblocks, nthreads>>>(gpu_data);
    CUDA_ERR_CHECK( cudaGetLastError() );
    CUDA_ERR_CHECK( cudaMemcpy(host_data, gpu_data, n * sizeof(int), cudaMemcpyDeviceToHost) );
    CUDA_ERR_CHECK( cudaFree(gpu_data) );
    for (int i = 0; i < n; i += 4)
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            host_data[i], host_data[i+1], host_data[i+2], host_data[i+3]);

    free(host_data);

    return 0;
}
```

- Технология NVLink позволяет коду хоста напрямую взаимодействовать с GPU (на поддерживаемых архитектурах – на данный момент IBM Power 8+,9)

- Лучшая производительность достигается, когда, данные хранятся в памяти GPU DRAM
- Необходимо выделить память в области GPU DRAM и скопировать данные с хоста на GPU:

```
int main()
{
    int nthreads = 4, nblocks = 4;
    int n = nthreads * nblocks * 4;
    int* host_data = (int*)malloc(n * sizeof(int));
    int* gpu_data = NULL;
    CUDA_ERR_CHECK( cudaMalloc(&gpu_data, n * sizeof(int)) );

    gpu_kernel<<<nblocks, nthreads>>>(gpu_data);
    CUDA_ERR_CHECK( cudaGetLastError() );
    CUDA_ERR_CHECK( cudaMemcpy(host_data, gpu_data, n * sizeof(int), cudaMemcpyDeviceToHost) );
    CUDA_ERR_CHECK( cudaFree(gpu_data) );
    for (int i = 0; i < n; i += 4)
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            host_data[i], host_data[i+1], host_data[i+2], host_data[i+3]);

    free(host_data);

    return 0;
}
```

- Технология NVLink позволяет коду хоста напрямую взаимодействовать с GPU (на поддерживаемых архитектурах – на данный момент IBM Power 8+,9)

- Лучшая производительность достигается, когда, данные хранятся в памяти GPU DRAM
- Необходимо выделить память в области GPU DRAM и скопировать данные с хоста на GPU:

```
int main()
{
    int nthreads = 4, nblocks = 4;
    int n = nthreads * nblocks * 4;
    int* host_data = (int*)malloc(n * sizeof(int));
    int* gpu_data = NULL;
    CUDA_ERR_CHECK( cudaMalloc(&gpu_data, n * sizeof(int)) );

    gpu_kernel<<<nblocks, nthreads>>>(gpu_data);
    CUDA_ERR_CHECK( cudaGetLastError() );
    CUDA_ERR_CHECK( cudaMemcpy(host_data, gpu_data, n * sizeof(int), cudaMemcpyDeviceToHost) );
    CUDA_ERR_CHECK( cudaFree(gpu_data) );
    for (int i = 0; i < n; i += 4)
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            host_data[i], host_data[i+1], host_data[i+2], host_data[i+3]);

    free(host_data);

    return 0;
}
```

- Технология NVLink позволяет коду хоста напрямую взаимодействовать с GPU (на поддерживаемых архитектурах – на данный момент IBM Power 8+,9)

- Лучшая производительность достигается, когда, данные хранятся в памяти GPU DRAM
- Необходимо выделить память в области GPU DRAM и скопировать данные с хоста на GPU:

```
int main()
{
    int nthreads = 4, nblocks = 4;
    int n = nthreads * nblocks * 4;
    int* host_data = (int*)malloc(n * sizeof(int));
    int* gpu_data = NULL;
    CUDA_ERR_CHECK( cudaMalloc(&gpu_data, n * sizeof(int)) );

    gpu_kernel<<<nblocks, nthreads>>>(gpu_data);
    CUDA_ERR_CHECK( cudaGetLastError() );
    CUDA_ERR_CHECK( cudaMemcpy(host_data, gpu_data, n * sizeof(int), cudaMemcpyDeviceToHost) );
    CUDA_ERR_CHECK( cudaFree(gpu_data) );
    for (int i = 0; i < n; i += 4)
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            host_data[i], host_data[i+1], host_data[i+2], host_data[i+3]);

    free(host_data);

    return 0;
}
```

- Технология NVLink позволяет коду хоста напрямую взаимодействовать с GPU (на поддерживаемых архитектурах – на данный момент IBM Power 8+,9)

- Лучшая производительность достигается, когда, данные хранятся в памяти GPU DRAM
- Необходимо выделить память в области GPU DRAM и скопировать данные с хоста на GPU:

```
int main()
{
    int nthreads = 4, nblocks = 4;
    int n = nthreads * nblocks * 4;
    int* host_data = (int*)malloc(n * sizeof(int));
    int* gpu_data = NULL;
    CUDA_ERR_CHECK( cudaMalloc(&gpu_data, n * sizeof(int)) );

    gpu_kernel<<<nblocks, nthreads>>>(gpu_data);
    CUDA_ERR_CHECK( cudaGetLastError() );
    CUDA_ERR_CHECK( cudaMemcpy(host_data, gpu_data, n * sizeof(int), cudaMemcpyDeviceToHost) );
    CUDA_ERR_CHECK( cudaFree(gpu_data) );
    for (int i = 0; i < n; i += 4)
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            host_data[i], host_data[i+1], host_data[i+2], host_data[i+3]);

    free(host_data);

    return 0;
}
```

- Технология NVLink позволяет коду хоста напрямую взаимодействовать с GPU (на поддерживаемых архитектурах – на данный момент IBM Power 8+,9)

- Лучшая производительность достигается, когда, данные хранятся в памяти GPU DRAM
- Необходимо выделить память в области GPU DRAM и скопировать данные с хоста на GPU:

```
int main()
{
    int nthreads = 4, nblocks = 4;
    int n = nthreads * nblocks * 4;
    int* host_data = (int*)malloc(n * sizeof(int));
    int* gpu_data = NULL;
    CUDA_ERR_CHECK( cudaMalloc(&gpu_data, n * sizeof(int)) );

    gpu_kernel<<<nblocks, nthreads>>>(gpu_data);
    CUDA_ERR_CHECK( cudaGetLastError() );
    CUDA_ERR_CHECK( cudaMemcpy(host_data, gpu_data, n * sizeof(int), cudaMemcpyDeviceToHost) );
    CUDA_ERR_CHECK( cudaFree(gpu_data) );
    for (int i = 0; i < n; i += 4)
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            host_data[i], host_data[i+1], host_data[i+2], host_data[i+3]);

    free(host_data);

    return 0;
}
```

- Технология NVLink позволяет коду хоста напрямую взаимодействовать с GPU (на поддерживаемых архитектурах – на данный момент IBM Power 8+,9)

- Лучшая производительность достигается, когда, данные хранятся в памяти GPU DRAM
- Необходимо выделить память в области GPU DRAM и скопировать данные с хоста на GPU:

```
int main()
{
    int nthreads = 4, nblocks = 4;
    int n = nthreads * nblocks * 4;
    int* host_data = (int*)malloc(n * sizeof(int));
    int* gpu_data = NULL;
    CUDA_ERR_CHECK( cudaMalloc(&gpu_data, n * sizeof(int)) );

    gpu_kernel<<<nblocks, nthreads>>>(gpu_data);
    CUDA_ERR_CHECK( cudaGetLastError() );
    CUDA_ERR_CHECK( cudaMemcpy(host_data, gpu_data, n * sizeof(int), cudaMemcpyDeviceToHost) );
    CUDA_ERR_CHECK( cudaFree(gpu_data) );
    for (int i = 0; i < n; i += 4)
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            host_data[i], host_data[i+1], host_data[i+2], host_data[i+3]);

    free(host_data);

    return 0;
}
```

- Технология NVLink позволяет коду хоста напрямую взаимодействовать с GPU (на поддерживаемых архитектурах – на данный момент IBM Power 8+,9)

- Лучшая производительность достигается, когда, данные хранятся в памяти GPU DRAM
- Необходимо выделить память в области GPU DRAM и скопировать данные с хоста на GPU:

```
int main()
{
    int nthreads = 4, nblocks = 4;
    int n = nthreads * nblocks * 4;
    int* host_data = (int*)malloc(n * sizeof(int));
    int* gpu_data = NULL;
    CUDA_ERR_CHECK( cudaMalloc(&gpu_data, n * sizeof(int)) );

    gpu_kernel<<<nblocks, nthreads>>>(gpu_data);
    CUDA_ERR_CHECK( cudaGetLastError() );
    CUDA_ERR_CHECK( cudaMemcpy(host_data, gpu_data, n * sizeof(int), cudaMemcpyDeviceToHost) );
    CUDA_ERR_CHECK( cudaFree(gpu_data) );
    for (int i = 0; i < n; i += 4)
        printf("Hello from b %#d of %d, t %#d of %d!\n",
            host_data[i], host_data[i+1], host_data[i+2], host_data[i+3]);

    free(host_data);

    return 0;
}
```

- Технология **NVLink** позволяет коду хоста напрямую взаимодействовать с GPU (на поддерживаемых архитектурах – на данный момент IBM Power 8+,9)

```
#include <stdio.h>

#define CUDA_ERR_CHECK(x) \
do { cudaError_t err = x; if (err != cudaSuccess) { \
    fprintf(stderr, "Error \"%s\" at %s:%d \n", \
        cudaGetErrorString(err), \
        __FILE__, __LINE__); exit(-1); \
    } } while (0);

__global__ void gpu_kernel(int* gpu_data)
{
    int4 coords = {
        blockIdx.x, gridDim.x, threadIdx.x, blockDim.x };
    ((int4*)gpu_data)[
        blockIdx.x * blockDim.x + threadIdx.x] = coords;
}
```

```
int main()
{
    int nthreads = 4, nblocks = 4;
    int n = nthreads * nblocks * 4;
    int* host_data = (int*)malloc(n * sizeof(int));
    int* gpu_data = NULL;
    CUDA_ERR_CHECK( cudaMalloc(&gpu_data, n * sizeof(int)) );

    gpu_kernel<<<nblocks, nthreads>>>(gpu_data);
    CUDA_ERR_CHECK( cudaGetLastError() );
    CUDA_ERR_CHECK( cudaMemcpy(
        host_data, gpu_data, n * sizeof(int),
        cudaMemcpyDeviceToHost) );
    CUDA_ERR_CHECK( cudaFree(gpu_data) );
    for (int i = 0; i < n; i += 4)
        printf("Hello form b %d of %d, t %d of %d!\n",
            host_data[i], host_data[i+1],
            host_data[i+2], host_data[i+3]);

    free(host_data);

    return 0;
}
```

```
$ nvcc -O3 cuda_gpu_data.cu -o cuda_gpu_data
$ ./cuda_gpu_data
```

```
#include <stdio.h>

#define CUDA_ERR_CHECK(x) \
do { cudaError_t err = x; if (err != cudaSuccess) { \
    fprintf(stderr, "Error \"%s\" at %s:%d \n", \
        cudaGetErrorString(err), \
        __FILE__, __LINE__); exit(-1); \
    } } while (0);

__global__ void gpu_kernel(int* gpu_data)
{
    int4 coords = {
        blockIdx.x, gridDim.x, threadIdx.x, blockDim.x };
    ((int4*)gpu_data)[
        blockIdx.x * blockDim.x + threadIdx.x] = coords;
}
```

```
int main()
{
    int nthreads = 4, nblocks = 4;
    int n = nthreads * nblocks * 4;
    int* host_data = (int*)malloc(n * sizeof(int));
    int* gpu_data = NULL;
    CUDA_ERR_CHECK( cudaMalloc(&gpu_data, n * sizeof(int)) );

    gpu_kernel<<<nblocks, nthreads>>>(gpu_data);
    CUDA_ERR_CHECK( cudaGetLastError() );
    CUDA_ERR_CHECK( cudaMemcpy(
        host_data, gpu_data, n * sizeof(int),
        cudaMemcpyDeviceToHost) );
    CUDA_ERR_CHECK( cudaFree(gpu_data) );
    for (int i = 0; i < n; i += 4)
        printf("Hello form b %d of %d, t %d of %d!\n",
            host_data[i], host_data[i+1],
            host_data[i+2], host_data[i+3]);

    free(host_data);

    return 0;
}
```

```
$ nvcc -O3 cuda_gpu_data.cu -o cuda_gpu_data
$ ./cuda_gpu_data
```

```
#include <stdio.h>

#define CUDA_ERR_CHECK(x) \
do { cudaError_t err = x; if (err != cudaSuccess) { \
    fprintf(stderr, "Error \"%s\" at %s:%d \n", \
        cudaGetErrorString(err), \
        __FILE__, __LINE__); exit(-1); \
    }} while (0);

__global__ void gpu_kernel(int* gpu_data)
{
    int4 coords = {
        blockIdx.x, blockDim.x, threadIdx.x, blockDim.x };
    ((int4*)gpu_data)[
        blockIdx.x * blockDim.x + threadIdx.x] = coords;
}
```

```
int main()
{
    int nthreads = 4, nblocks = 4;
    int n = nthreads * nblocks * 4;
    int* host_data = (int*)malloc(n * sizeof(int));
    int* gpu_data = NULL;
    CUDA_ERR_CHECK( cudaMalloc(&gpu_data, n * sizeof(int)) );

    gpu_kernel<<<nblocks, nthreads>>>(gpu_data);
    CUDA_ERR_CHECK( cudaGetLastError() );
    CUDA_ERR_CHECK( cudaMemcpy(
        host_data, gpu_data, n * sizeof(int),
        cudaMemcpyDeviceToHost) );
    CUDA_ERR_CHECK( cudaFree(gpu_data) );
    for (int i = 0; i < n; i += 4)
        printf("Hello form b %d of %d, t %d of %d!\n",
            host_data[i], host_data[i+1],
            host_data[i+2], host_data[i+3]);

    free(host_data);

    return 0;
}
```

```
$ nvcc -O3 cuda_gpu_data.cu -o cuda_gpu_data
$ ./cuda_gpu_data
```

```
#include <stdio.h>

#define CUDA_ERR_CHECK(x) \
do { cudaError_t err = x; if (err != cudaSuccess) { \
    fprintf(stderr, "Error \"%s\" at %s:%d\n", \
        cudaGetErrorString(err), \
        __FILE__, __LINE__); exit(-1); \
    } } while (0);

__global__ void gpu_kernel(int* gpu_data)
{
    int4 coords = {
        blockIdx.x, gridDim.x, threadIdx.x, blockDim.x };
    ((int4*)gpu_data)[
        blockIdx.x * blockDim.x + threadIdx.x] = coords;
}
```

```
int main()
{
    int nthreads = 4, nblocks = 4;
    int n = nthreads * nblocks * 4;
    int* host_data = (int*)malloc(n * sizeof(int));
    int* gpu_data = NULL;
    CUDA_ERR_CHECK( cudaMalloc(&gpu_data, n * sizeof(int)) );

    gpu_kernel<<<nblocks, nthreads>>>(gpu_data);
    CUDA_ERR_CHECK( cudaGetLastError() );
    CUDA_ERR_CHECK( cudaMemcpy(
        host_data, gpu_data, n * sizeof(int),
        cudaMemcpyDeviceToHost) );
    CUDA_ERR_CHECK( cudaFree(gpu_data) );
    for (int i = 0; i < n; i += 4)
        printf("Hello form b %d of %d, t %d of %d!\n",
            host_data[i], host_data[i+1],
            host_data[i+2], host_data[i+3]);

    free(host_data);

    return 0;
}
```

```
$ nvcc -O3 cuda_gpu_data.cu -o cuda_gpu_data
$ ./cuda_gpu_data
```

```
#include <stdio.h>

#define CUDA_ERR_CHECK(x) \
do { cudaError_t err = x; if (err != cudaSuccess) { \
    fprintf(stderr, "Error \"%s\" at %s:%d \n", \
        cudaGetErrorString(err), \
        __FILE__, __LINE__); exit(-1); \
    } } while (0);

__global__ void gpu_kernel(int* gpu_data)
{
    int4 coords = {
        blockIdx.x, gridDim.x, threadIdx.x, blockDim.x };
    ((int4*)gpu_data)[
        blockIdx.x * blockDim.x + threadIdx.x] = coords;
}
```

```
int main()
{
    int nthreads = 4, nblocks = 4;
    int n = nthreads * nblocks * 4;
    int* host_data = (int*)malloc(n * sizeof(int));
    int* gpu_data = NULL;
    CUDA_ERR_CHECK( cudaMalloc(&gpu_data, n * sizeof(int)) );

    gpu_kernel<<<nblocks, nthreads>>>(gpu_data);
    CUDA_ERR_CHECK( cudaGetLastError() );
    CUDA_ERR_CHECK( cudaMemcpy(
        host_data, gpu_data, n * sizeof(int),
        cudaMemcpyDeviceToHost) );
    CUDA_ERR_CHECK( cudaFree(gpu_data) );
    for (int i = 0; i < n; i += 4)
        printf("Hello form b %d of %d, t %d of %d!\n",
            host_data[i], host_data[i+1],
            host_data[i+2], host_data[i+3]);

    free(host_data);

    return 0;
}
```

```
$ nvcc -O3 cuda_gpu_data.cu -o cuda_gpu_data
$ ./cuda_gpu_data
```

## Мы только что написали нашу первую CUDA программу!

- CUDA использует 2-х уровневую топологию compute grid, схожую с MPI+OpenMP
- Топология CUDA, тем не менее, имеет гибкость в выборе размерности сетки – 1D, 2D or 3D (например `blockIdx.y`, `threadIdx.z`)
- Ядро CUDA не является самостоятельной программой, его требуется вызвать из “хостового” приложения
- Вызов ядра CUDA порождает все блоки и нити, не требуется дополнительных `#pragma` как в OpenMP
- Ядра CUDA асинхронны, и могут быть синхронизированы явно (`cudaDeviceSynchronize`) или неявно (с использованием параметров ядра например в `cudaMemcpy`)
- GPU имеет встроенную память DRAM, независимую от памяти хоста; данные между ними необходимо копировать вручную
- Всегда проверяйте статус возвращаемых ошибок вызова CUDA (например, с помощью `CUDA_ERR_CHECK`)

Мы только что написали нашу первую CUDA программу!

- CUDA использует 2-х уровневую топологию compute grid, схожую с MPI+OpenMP
- Топология CUDA, тем не менее, имеет гибкость в выборе размерности сетки – 1D, 2D or 3D (например `blockIdx.y`, `threadIdx.z`)
- Ядро CUDA не является самостоятельной программой, его требуется вызвать из “хостового” приложения
- Вызов ядра CUDA порождает все блоки и нити, не требуется дополнительных `#pragma` как в OpenMP
- Ядра CUDA асинхронны, и могут быть синхронизированы явно (`cudaDeviceSynchronize`) или неявно (с использованием параметров ядра например в `cudaMemcpy`)
- GPU имеет встроенную память DRAM, независимую от памяти хоста; данные между ними необходимо копировать вручную
- Всегда проверяйте статус возвращаемых ошибок вызова CUDA (например, с помощью `CUDA_ERR_CHECK`)

Мы только что написали нашу первую CUDA программу!

- CUDA использует 2-х уровневую топологию compute grid, схожую с MPI+OpenMP
- Топология CUDA, тем не менее, имеет гибкость в выборе размерности сетки – 1D, 2D or 3D (например `blockIdx.y`, `threadIdx.z`)
- Ядро CUDA не является самостоятельной программой, его требуется вызвать из “хостового” приложения
- Вызов ядра CUDA порождает все блоки и нити, не требуется дополнительных `#pragma` как в OpenMP
- Ядра CUDA асинхронны, и могут быть синхронизированы явно (`cudaDeviceSynchronize`) или неявно (с использованием параметров ядра например в `cudaMemcpy`)
- GPU имеет встроенную память DRAM, независимую от памяти хоста; данные между ними необходимо копировать вручную
- Всегда проверяйте статус возвращаемых ошибок вызова CUDA (например, с помощью `CUDA_ERR_CHECK`)

Мы только что написали нашу первую CUDA программу!

- CUDA использует 2-х уровневую топологию compute grid, схожую с MPI+OpenMP
- Топология CUDA, тем не менее, имеет гибкость в выборе размерности сетки – 1D, 2D or 3D (например `blockIdx.y`, `threadIdx.z`)
- Ядро CUDA не является самостоятельной программой, его требуется вызвать из “хостового” приложения
- Вызов ядра CUDA порождает все блоки и нити, не требуется дополнительных `#pragma` как в OpenMP
- Ядра CUDA асинхронны, и могут быть синхронизированы явно (`cudaDeviceSynchronize`) или неявно (с использованием параметров ядра например в `cudaMemcpy`)
- GPU имеет встроенную память DRAM, независимую от памяти хоста; данные между ними необходимо копировать вручную
- Всегда проверяйте статус возвращаемых ошибок вызова CUDA (например, с помощью `CUDA_ERR_CHECK`)

Мы только что написали нашу первую CUDA программу!

- CUDA использует 2-х уровневую топологию compute grid, схожую с MPI+OpenMP
- Топология CUDA, тем не менее, имеет гибкость в выборе размерности сетки – 1D, 2D or 3D (например `blockIdx.y`, `threadIdx.z`)
- Ядро CUDA не является самостоятельной программой, его требуется вызвать из “хостового” приложения
- Вызов ядра CUDA порождает все блоки и нити, не требуется дополнительных `#pragma` как в OpenMP
- Ядра CUDA асинхронны, и могут быть синхронизированы явно (`cudaDeviceSynchronize`) или неявно (с использованием параметров ядра например в `cudaMemcpy`)
- GPU имеет встроенную память DRAM, независимую от памяти хоста; данные между ними необходимо копировать вручную
- Всегда проверяйте статус возвращаемых ошибок вызова CUDA (например, с помощью `CUDA_ERR_CHECK`)

Мы только что написали нашу первую CUDA программу!

- CUDA использует 2-х уровневую топологию compute grid, схожую с MPI+OpenMP
- Топология CUDA, тем не менее, имеет гибкость в выборе размерности сетки – 1D, 2D or 3D (например `blockIdx.y`, `threadIdx.z`)
- Ядро CUDA не является самостоятельной программой, его требуется вызвать из “хостового” приложения
- Вызов ядра CUDA порождает все блоки и нити, не требуется дополнительных `#pragma` как в OpenMP
- Ядра CUDA асинхронны, и могут быть синхронизированы явно (`cudaDeviceSynchronize`) или неявно (с использованием параметров ядра например в `cudaMemcpy`)
- GPU имеет встроенную память DRAM, независимую от памяти хоста; данные между ними необходимо копировать вручную
- Всегда проверяйте статус возвращаемых ошибок вызова CUDA (например, с помощью `CUDA_ERR_CHECK`)

Мы только что написали нашу первую CUDA программу!

- CUDA использует 2-х уровневую топологию compute grid, схожую с MPI+OpenMP
- Топология CUDA, тем не менее, имеет гибкость в выборе размерности сетки – 1D, 2D or 3D (например `blockIdx.y`, `threadIdx.z`)
- Ядро CUDA не является самостоятельной программой, его требуется вызвать из “хостового” приложения
- Вызов ядра CUDA порождает все блоки и нити, не требуется дополнительных `#pragma` как в OpenMP
- Ядра CUDA асинхронны, и могут быть синхронизированы явно (`cudaDeviceSynchronize`) или неявно (с использованием параметров ядра например в `cudaMemcpy`)
- GPU имеет встроенную память DRAM, независимую от памяти хоста; данные между ними необходимо копировать вручную
- Всегда проверяйте статус возвращаемых ошибок вызова CUDA (например, с помощью `CUDA_ERR_CHECK`)

Мы только что написали нашу первую CUDA программу!

- CUDA использует 2-х уровневую топологию compute grid, схожую с MPI+OpenMP
- Топология CUDA, тем не менее, имеет гибкость в выборе размерности сетки – 1D, 2D or 3D (например `blockIdx.y`, `threadIdx.z`)
- Ядро CUDA не является самостоятельной программой, его требуется вызвать из “хостового” приложения
- Вызов ядра CUDA порождает все блоки и нити, не требуется дополнительных `#pragma` как в OpenMP
- Ядра CUDA асинхронны, и могут быть синхронизированы явно (`cudaDeviceSynchronize`) или неявно (с использованием параметров ядра например в `cudaMemcpy`)
- GPU имеет встроенную память DRAM, независимую от памяти хоста; данные между ними необходимо копировать вручную
- Всегда проверяйте статус возвращаемых ошибок вызова CUDA (например, с помощью `CUDA_ERR_CHECK`)

CUDA поддерживает 1-3 измерения:

- `gpu_kernel<<<4, 2>>>(...);`  
– 4 блока по 2 нити в каждом
- `gpu_kernel<<<dim3(8, 4, 1), dim3(4, 2, 1)>>>(...);`  
–  $8 \times 4$  блоков по  $4 \times 2$  нити в каждом
- `gpu_kernel<<<dim3(16, 8, 4), dim3(8, 4, 2)>>>(...);`  
–  $16 \times 8 \times 4$  блоков по  $8 \times 4 \times 2$  нити в каждом



CUDA поддерживает 1-3 измерения:

**1** `gpu_kernel<<<4, 2>>>(...);`

– 4 блока по 2 нити в каждом

**2** `gpu_kernel<<<dim3(8, 4, 1), dim3(4, 2, 1)>>>(...);`

– 8×4 блоков по 4×2 нити в каждом

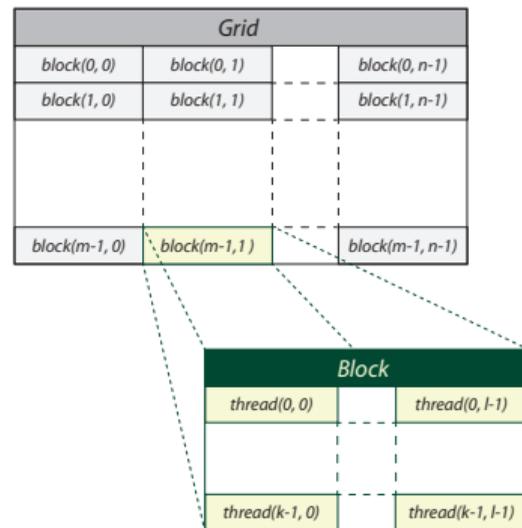
**3** `gpu_kernel<<<dim3(16, 8, 4), dim3(8, 4, 2)>>>(...);`

– 16×8×4 блоков по 8×4×2 нити в каждом



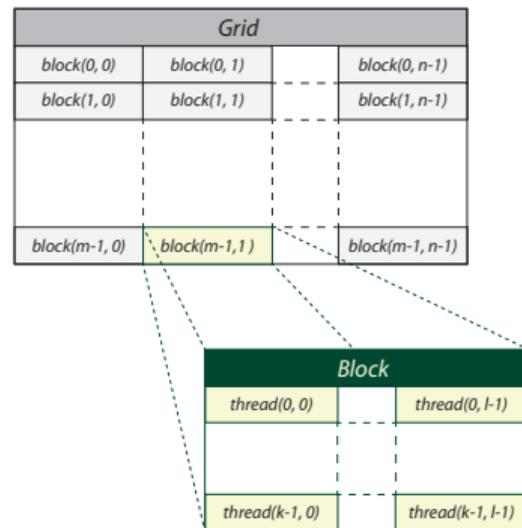
CUDA поддерживает 1-3 измерения:

- 1** `gpu_kernel<<<4, 2>>>(...);`  
– 4 блока по 2 нити в каждом
- 2** `gpu_kernel<<<dim3(8, 4, 1), dim3(4, 2, 1)>>>(...);`  
–  $8 \times 4$  блоков по  $4 \times 2$  нити в каждом
- 3** `gpu_kernel<<<dim3(16, 8, 4), dim3(8, 4, 2)>>>(...);`  
–  $16 \times 8 \times 4$  блоков по  $8 \times 4 \times 2$  нити в каждом



CUDA поддерживает 1-3 измерения:

- `gpu_kernel<<<4, 2>>>(...);`  
– 4 блока по 2 нити в каждом
- `gpu_kernel<<<dim3(8, 4, 1), dim3(4, 2, 1)>>>(...);`  
–  $8 \times 4$  блоков по  $4 \times 2$  нити в каждом
- `gpu_kernel<<<dim3(16, 8, 4), dim3(8, 4, 2)>>>(...);`  
–  $16 \times 8 \times 4$  блоков по  $8 \times 4 \times 2$  нити в каждом



# CUDA compute grid (advanced)

CUDA compute grid поддерживает 1-3 измерения  $\Rightarrow$  упрощает перенос многомерных циклов в GPU ядра:

```
for (int k = 0; k < nk; k++)
  for (int j = 0; j < nj; j++)
    for (int i = 0; i < ni; i++)
    {
      int idx = i + ni * j + nj * ni * k;
      data[idx] = idx;
    }
```

- CUDA “прячет” циклы в параметры запуска ядра
- Интервалы распределяются между нитями и блоками нитей
- Количество блоков округляется чтобы покрыть остаточные элементы
- Что если интервал больше чем максимальный размер `gridDim`  $\times$

```
__global__ void gpu_kernel(int ni, int nj, int nk, int* data)
{
  int k = blockIdx.z * blockDim.z + threadIdx.z; if (k >= nk) return;
  int j = blockIdx.y * blockDim.y + threadIdx.y; if (j >= nj) return;
  int i = blockIdx.x * blockDim.x + threadIdx.x; if (i >= ni) return;

  int idx = i + ni * j + nj * ni * k;
  data[idx] = idx;
}

...
gpu_kernel<<<dim3(max(1, roundup(ni, 16) / 16),
                 max(1, roundup(nj, 8) / 8),
                 max(1, roundup(nk, 8) / 8)), dim3(16, 8, 8)>>>(
    ni, nj, nk, gpu_data);
CUDA_ERR_CHECK( cudaGetLastError() );
```

# CUDA compute grid (advanced)

CUDA compute grid поддерживает 1-3 измерения  $\Rightarrow$  упрощает перенос многомерных циклов в GPU ядра:

```
for (int k = 0; k < nk; k++)
  for (int j = 0; j < nj; j++)
    for (int i = 0; i < ni; i++)
    {
        int idx = i + ni * j + nj * ni * k;
        data[idx] = idx;
    }
```

- CUDA “прячет” циклы в параметры запуска ядра
  - Интервалы распределяются между нитями и блоками нитей
  - Количество блоков округляется чтобы покрыть остаточные элементы
  - Что если интервал больше чем максимальный размер `gridDim` ×

```
__global__ void gpu_kernel(int ni, int nj, int nk, int* data)
{
    int k = blockIdx.z * blockDim.z + threadIdx.z; if (k >= nk) return;
    int j = blockIdx.y * blockDim.y + threadIdx.y; if (j >= nj) return;
    int i = blockIdx.x * blockDim.x + threadIdx.x; if (i >= ni) return;

    int idx = i + ni * j + nj * ni * k;
    data[idx] = idx;
}

...
gpu_kernel<<<dim3(max(1, roundup(ni, 16) / 16),
                 max(1, roundup(nj, 8) / 8),
                 max(1, roundup(nk, 8) / 8)), dim3(16, 8, 8)>>>(
    ni, nj, nk, gpu_data);
CUDA_ERR_CHECK( cudaGetLastError() );
```

# CUDA compute grid (advanced)

CUDA compute grid поддерживает 1-3 измерения  $\Rightarrow$  упрощает перенос многомерных циклов в GPU ядра:

```
for (int k = 0; k < nk; k++)
  for (int j = 0; j < nj; j++)
    for (int i = 0; i < ni; i++)
    {
        int idx = i + ni * j + nj * ni * k;
        data[idx] = idx;
    }
```

- CUDA “прячет” циклы в параметры запуска ядра
- Интервалы распределяются между нитями и блоками нитей
- Количество блоков округляется чтобы покрыть остаточные элементы
- Что если интервал больше чем максимальный размер `gridDim` ×

```
__global__ void gpu_kernel(int ni, int nj, int nk, int* data)
{
    int k = blockIdx.z * blockDim.z + threadIdx.z; if (k >= nk) return;
    int j = blockIdx.y * blockDim.y + threadIdx.y; if (j >= nj) return;
    int i = blockIdx.x * blockDim.x + threadIdx.x; if (i >= ni) return;

    int idx = i + ni * j + nj * ni * k;
    data[idx] = idx;
}

...
gpu_kernel<<<dim3(max(1, roundup(ni, 16) / 16),
                 max(1, roundup(nj, 8) / 8),
                 max(1, roundup(nk, 8) / 8)), dim3(16, 8, 8)>>>(
    ni, nj, nk, gpu_data);
CUDA_ERR_CHECK( cudaGetLastError() );
```

# CUDA compute grid (advanced)

CUDA compute grid поддерживает 1-3 измерения  $\Rightarrow$  упрощает перенос многомерных циклов в GPU ядра:

```
for (int k = 0; k < nk; k++)
  for (int j = 0; j < nj; j++)
    for (int i = 0; i < ni; i++)
    {
        int idx = i + ni * j + nj * ni * k;
        data[idx] = idx;
    }
```

- CUDA “прячет” циклы в параметры запуска ядра
- Интервалы распределяются между нитями и блоками нитей
- Количество блоков округляется чтобы покрыть остаточные элементы
- Что если интервал больше чем максимальный размер `gridDim` ×

```
__global__ void gpu_kernel(int ni, int nj, int nk, int* data)
{
    int k = blockIdx.z * blockDim.z + threadIdx.z; if (k >= nk) return;
    int j = blockIdx.y * blockDim.y + threadIdx.y; if (j >= nj) return;
    int i = blockIdx.x * blockDim.x + threadIdx.x; if (i >= ni) return;

    int idx = i + ni * j + nj * ni * k;
    data[idx] = idx;
}

...
gpu_kernel<<<dim3(max(1, roundup(ni, 16) / 16),
                 max(1, roundup(nj, 8) / 8),
                 max(1, roundup(nk, 8) / 8)), dim3(16, 8, 8)>>>(
    ni, nj, nk, gpu_data);
CUDA_ERR_CHECK( cudaGetLastError() );
```

# CUDA compute grid (advanced)

CUDA compute grid поддерживает 1-3 измерения  $\Rightarrow$  упрощает перенос многомерных циклов в GPU ядра:

```
for (int k = 0; k < nk; k++)
  for (int j = 0; j < nj; j++)
    for (int i = 0; i < ni; i++)
      {
        int idx = i + ni * j + nj * ni * k;
        data[idx] = idx;
      }
```

- CUDA “прячет” циклы в параметры запуска ядра
- Интервалы распределяются между нитями и блоками нитей
- Количество блоков округляется чтобы покрыть остаточные элементы
- Что если интервал больше чем максимальный размер  $\text{gridDim} \times \text{blockDim}$ ?

```
__global__ void gpu_kernel(int ni, int nj, int nk, int* data)
{
  int k = blockIdx.z * blockDim.z + threadIdx.z; if (k >= nk) return;
  int j = blockIdx.y * blockDim.y + threadIdx.y; if (j >= nj) return;
  int i = blockIdx.x * blockDim.x + threadIdx.x; if (i >= ni) return;

  int idx = i + ni * j + nj * ni * k;
  data[idx] = idx;
}

...
gpu_kernel<<<dim3(max(1, roundup(ni, 16) / 16),
                 max(1, roundup(nj, 8) / 8),
                 max(1, roundup(nk, 8) / 8)), dim3(16, 8, 8)>>>(
  ni, nj, nk, gpu_data);
CUDA_ERR_CHECK( cudaGetLastError() );
```

# CUDA compute grid (advanced)

CUDA compute grid поддерживает 1-3 измерения  $\Rightarrow$  упрощает перенос многомерных циклов в GPU ядра:

```
for (int k = 0; k < nk; k++)
  for (int j = 0; j < nj; j++)
    for (int i = 0; i < ni; i++)
    {
        int idx = i + ni * j + nj * ni * k;
        data[idx] = idx;
    }
```

- Можно обрабатывать множество точек в одной нити ( $\Rightarrow$  снова появляются циклы)
- Не стоит обрабатывать последовательно расположенные точки в одной нити, из-за требований коалесинга
- Лимиты compute grid можно узнать из свойств устройства

```
__global__ void gpu_kernel(int ni, int nj, int nk, int* data,
                           int i_inc, int j_inc, int k_inc)
{
    for (int k = blockIdx.z * blockDim.z + threadIdx.z; k < nk; k += k_inc)
        for (int j = blockIdx.y * blockDim.y + threadIdx.y; j < nj; j += j_inc)
            for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < ni; i += i_inc)
            {
                int idx = i + ni * j + nj * ni * k;
                data[idx] = idx;
            }
}

...
struct cudaDeviceProp props;
CUDA_ERR_CHECK( cudaGetDeviceProperties(&props, 0) );
dim3 max_grid;
max_grid.x = props.maxGridSize[0];
max_grid.y = props.maxGridSize[1];
max_grid.z = props.maxGridSize[2];

gpu_kernel<<<dim3(min(max(1, roundup(ni, 16)) / 16), roundup(max_grid.x, 16)),
                min(max(1, roundup(nj, 8)) / 8), roundup(max_grid.y, 8)),
                min(max(1, roundup(nk, 8)) / 8), roundup(max_grid.z, 8)),
                dim3(16, 8, 8)>>>(ni, nj, nk, gpu_data,
                                min(max(1, roundup(ni, 16)) / 16), roundup(max_grid.x, 16)) * 16,
                                min(max(1, roundup(nj, 8)) / 8), roundup(max_grid.y, 8)) * 8,
                                min(max(1, roundup(nk, 8)) / 8), roundup(max_grid.z, 8)) * 8);
CUDA_ERR_CHECK( cudaGetLastError() );
```

# CUDA compute grid (advanced)

CUDA compute grid поддерживает 1-3 измерения  $\Rightarrow$  упрощает перенос многомерных циклов в GPU ядра:

```
for (int k = 0; k < nk; k++)
  for (int j = 0; j < nj; j++)
    for (int i = 0; i < ni; i++)
    {
        int idx = i + ni * j + nj * ni * k;
        data[idx] = idx;
    }
```

- Можно обрабатывать множество точек в одной нити ( $\Rightarrow$  снова появляются циклы)

- Не стоит обрабатывать последовательно расположенные точки в одной нити, из-за требований коалесинга

- Лимиты compute grid можно узнать из [свойств устройства](#)

```
__global__ void gpu_kernel(int ni, int nj, int nk, int* data,
                           int i_inc, int j_inc, int k_inc)
{
    for (int k = blockIdx.z * blockDim.z + threadIdx.z; k < nk; k += k_inc)
        for (int j = blockIdx.y * blockDim.y + threadIdx.y; j < nj; j += j_inc)
            for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < ni; i += i_inc)
            {
                int idx = i + ni * j + nj * ni * k;
                data[idx] = idx;
            }
}

...
struct cudaDeviceProp props;
CUDA_ERR_CHECK( cudaGetDeviceProperties(&props, 0) );
dim3 max_grid;
max_grid.x = props.maxGridSize[0];
max_grid.y = props.maxGridSize[1];
max_grid.z = props.maxGridSize[2];

gpu_kernel<<<dim3(min(max(1, roundup(ni, 16)) / 16), roundup(max_grid.x, 16)),
                min(max(1, roundup(nj, 8)) / 8), roundup(max_grid.y, 8)),
                min(max(1, roundup(nk, 8)) / 8), roundup(max_grid.z, 8)),
                dim3(16, 8, 8)>>>(ni, nj, nk, gpu_data,
                                min(max(1, roundup(ni, 16)) / 16), roundup(max_grid.x, 16)) * 16,
                                min(max(1, roundup(nj, 8)) / 8), roundup(max_grid.y, 8)) * 8,
                                min(max(1, roundup(nk, 8)) / 8), roundup(max_grid.z, 8)) * 8);
CUDA_ERR_CHECK( cudaGetLastError() );
```

# CUDA compute grid (advanced)

CUDA compute grid поддерживает 1-3 измерения  $\Rightarrow$  упрощает перенос многомерных циклов в GPU ядра:

```
for (int k = 0; k < nk; k++)
  for (int j = 0; j < nj; j++)
    for (int i = 0; i < ni; i++)
    {
        int idx = i + ni * j + nj * ni * k;
        data[idx] = idx;
    }
```

- Можно обрабатывать множество точек в одной нити ( $\Rightarrow$  снова появляются циклы)
- Не стоит обрабатывать последовательно расположенные точки в одной нити, из-за требований коалесинга
- Лимиты compute grid можно узнать из [сайта устройства](#)

```
__global__ void gpu_kernel(int ni, int nj, int nk, int* data,
                           int i_inc, int j_inc, int k_inc)
{
    for (int k = blockIdx.z * blockDim.z + threadIdx.z; k < nk; k += k_inc)
        for (int j = blockIdx.y * blockDim.y + threadIdx.y; j < nj; j += j_inc)
            for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < ni; i += i_inc)
            {
                int idx = i + ni * j + nj * ni * k;
                data[idx] = idx;
            }
}

...
struct cudaDeviceProp props;
CUDA_ERR_CHECK( cudaGetDeviceProperties(&props, 0) );
dim3 max_grid;
max_grid.x = props.maxGridSize[0];
max_grid.y = props.maxGridSize[1];
max_grid.z = props.maxGridSize[2];

gpu_kernel<<<dim3(min(max(1, roundup(ni, 16)) / 16), roundup(max_grid.x, 16)),
                min(max(1, roundup(nj, 8)) / 8), roundup(max_grid.y, 8)),
                min(max(1, roundup(nk, 8)) / 8), roundup(max_grid.z, 8))),
           dim3(16, 8, 8)>>>(ni, nj, nk, gpu_data,
                             min(max(1, roundup(ni, 16)) / 16), roundup(max_grid.x, 16)) * 16,
                             min(max(1, roundup(nj, 8)) / 8), roundup(max_grid.y, 8)) * 8,
                             min(max(1, roundup(nk, 8)) / 8), roundup(max_grid.z, 8)) * 8);
CUDA_ERR_CHECK( cudaGetLastError() );
```

# CUDA compute grid (advanced)

CUDA compute grid поддерживает 1-3 измерения  $\Rightarrow$  упрощает перенос многомерных циклов в GPU ядра:

```
for (int k = 0; k < nk; k++)
  for (int j = 0; j < nj; j++)
    for (int i = 0; i < ni; i++)
    {
        int idx = i + ni * j + nj * ni * k;
        data[idx] = idx;
    }
```

- Можно обрабатывать множество точек в одной нити ( $\Rightarrow$  снова появляются циклы)
- Не стоит обрабатывать последовательно расположенные точки в одной нити, из-за требований коалесинга
- Лимиты compute grid можно узнать из свойств устройства

```
__global__ void gpu_kernel(int ni, int nj, int nk, int* data,
                           int i_inc, int j_inc, int k_inc)
{
    for (int k = blockIdx.z * blockDim.z + threadIdx.z; k < nk; k += k_inc)
        for (int j = blockIdx.y * blockDim.y + threadIdx.y; j < nj; j += j_inc)
            for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < ni; i += i_inc)
            {
                int idx = i + ni * j + nj * ni * k;
                data[idx] = idx;
            }
}

...
struct cudaDeviceProp props;
CUDA_ERR_CHECK( cudaGetDeviceProperties(&props, 0) );
dim3 max_grid;
max_grid.x = props.maxGridSize[0];
max_grid.y = props.maxGridSize[1];
max_grid.z = props.maxGridSize[2];

gpu_kernel<<<dim3(min(max(1, roundup(ni, 16)) / 16), roundup(nj, 8) / 8), rounddown(max_grid.y, 8)),
               min(max(1, roundup(nk, 8)) / 8), rounddown(max_grid.z, 8))),
           dim3(16, 8, 8)>>>(ni, nj, nk, gpu_data,
                           min(max(1, roundup(ni, 16)) / 16), rounddown(max_grid.x, 16)) * 16,
                           min(max(1, roundup(nj, 8)) / 8), rounddown(max_grid.y, 8)) * 8,
                           min(max(1, roundup(nk, 8)) / 8), rounddown(max_grid.z, 8)) * 8);
CUDA_ERR_CHECK( cudaGetLastError() );
```

# Production example: stencil in CUDA

```
#define _A(array, is, iy, ix) (array[(ix) + nx * (iy) + nx * ny * (is)])

extern "C" __global__ void wave13pt(const int nx, const int ny, const int ns,
kernel_config_t config, const real m0, const real m1, const real m2,
const real* const __restrict__ w0, const real* const __restrict__ w1,
real* const __restrict__ w2)
{
#define k_offset (blockIdx.z * blockDim.z + threadIdx.z)
#define j_offset (blockIdx.y * blockDim.y + threadIdx.y)
#define i_offset (blockIdx.x * blockDim.x + threadIdx.x)
for (int k = 2 + k_offset; k < ns - 2; k += config.strideDim.z) {
for (int j = 2 + j_offset; j < ny - 2; j += config.strideDim.y) {
for (int i = i_offset; i < nx; i += config.strideDim.x) {
if ((i < 2) || (i >= nx - 2)) continue;

_A(w2, k, j, i) = m0 * _A(w1, k, j, i) - _A(w0, k, j, i) +
m1 * (
_A(w1, k, j, i+1) + _A(w1, k, j, i-1) +
_A(w1, k, j+1, i) + _A(w1, k, j-1, i) +
_A(w1, k+1, j, i) + _A(w1, k-1, j, i)) +
m2 * (
_A(w1, k, j, i+2) + _A(w1, k, j, i-2) +
_A(w1, k, j+2, i) + _A(w1, k, j-2, i) +
_A(w1, k+2, j, i) + _A(w1, k-2, j, i));
}
}
}
}
```

## ■ Dual Intel Xeon E5-2670 (OpenMP version):

```
cpu> ./wave13pt 512 256 256 10
m0 = 0.680375, m1 = -0.035206, m2 = 0.094366
initial mean = 0.000024
compute time = 0.279701 sec
final mean = 0.000173
```

## ■ NVIDIA GTX 680M (cheap laptop graphics):

```
cuda> ./wave13pt 512 256 256 10
m0 = 0.680375, m1 = -0.035206, m2 = 0.094366
initial mean = 0.000024
init time = 0.202504 sec
device buffer alloc time = 0.017779 sec
data load time = 0.066286 sec (5.657294 GB/sec)
compute time = 0.122417 sec
data save time = 0.020925 sec (5.973645 GB/sec)
device buffer free time = 0.000459 sec
final mean = 0.000173
```

# Production example: stencil in CUDA

```
#define _A(array, is, iy, ix) (array[(ix) + nx * (iy) + nx * ny * (is)])

extern "C" __global__ void wave13pt(const int nx, const int ny, const int ns,
kernel_config_t config, const real m0, const real m1, const real m2,
const real* const __restrict__ w0, const real* const __restrict__ w1,
real* const __restrict__ w2)
{
#define k_offset (blockIdx.z * blockDim.z + threadIdx.z)
#define j_offset (blockIdx.y * blockDim.y + threadIdx.y)
#define i_offset (blockIdx.x * blockDim.x + threadIdx.x)
for (int k = 2 + k_offset; k < ns - 2; k += config.strideDim.z) {
for (int j = 2 + j_offset; j < ny - 2; j += config.strideDim.y) {
for (int i = i_offset; i < nx; i += config.strideDim.x) {
if ((i < 2) || (i >= nx - 2)) continue;

_A(w2, k, j, i) = m0 * _A(w1, k, j, i) - _A(w0, k, j, i) +
m1 * (
_A(w1, k, j, i+1) + _A(w1, k, j, i-1) +
_A(w1, k, j+1, i) + _A(w1, k, j-1, i) +
_A(w1, k+1, j, i) + _A(w1, k-1, j, i)) +
m2 * (
_A(w1, k, j, i+2) + _A(w1, k, j, i-2) +
_A(w1, k, j+2, i) + _A(w1, k, j-2, i) +
_A(w1, k+2, j, i) + _A(w1, k-2, j, i));
}
}
}
}
```

## ■ Dual Intel Xeon E5-2670 (OpenMP version):

```
cpu> ./wave13pt 512 256 256 10
m0 = 0.680375, m1 = -0.035206, m2 = 0.094366
initial mean = 0.000024
compute time = 0.279701 sec
final mean = 0.000173
```

## ■ NVIDIA GTX 680M (cheap laptop graphics):

```
cuda> ./wave13pt 512 256 256 10
m0 = 0.680375, m1 = -0.035206, m2 = 0.094366
initial mean = 0.000024
init time = 0.202504 sec
device buffer alloc time = 0.017779 sec
data load time = 0.066286 sec (5.657294 GB/sec)
compute time = 0.122417 sec
data save time = 0.020925 sec (5.973645 GB/sec)
device buffer free time = 0.000459 sec
final mean = 0.000173
```

# Production example: stencil in CUDA

```
#define _A(array, is, iy, ix) (array[(ix) + nx * (iy) + nx * ny * (is)])

extern "C" __global__ void wave13pt(const int nx, const int ny, const int ns,
kernel_config_t config, const real m0, const real m1, const real m2,
const real* const __restrict__ w0, const real* const __restrict__ w1,
real* const __restrict__ w2)
{
#define k_offset (blockIdx.z * blockDim.z + threadIdx.z)
#define j_offset (blockIdx.y * blockDim.y + threadIdx.y)
#define i_offset (blockIdx.x * blockDim.x + threadIdx.x)
for (int k = 2 + k_offset; k < ns - 2; k += config.strideDim.z) {
for (int j = 2 + j_offset; j < ny - 2; j += config.strideDim.y) {
for (int i = i_offset; i < nx; i += config.strideDim.x) {
if ((i < 2) || (i >= nx - 2)) continue;

_A(w2, k, j, i) = m0 * _A(w1, k, j, i) - _A(w0, k, j, i) +
m1 * (
_A(w1, k, j, i+1) + _A(w1, k, j, i-1) +
_A(w1, k, j+1, i) + _A(w1, k, j-1, i) +
_A(w1, k+1, j, i) + _A(w1, k-1, j, i)) +
m2 * (
_A(w1, k, j, i+2) + _A(w1, k, j, i-2) +
_A(w1, k, j+2, i) + _A(w1, k, j-2, i) +
_A(w1, k+2, j, i) + _A(w1, k-2, j, i));
}
}
}
}
```

## ■ Dual Intel Xeon E5-2670 (OpenMP version):

```
cpu> ./wave13pt 512 256 256 10
m0 = 0.680375, m1 = -0.035206, m2 = 0.094366
initial mean = 0.000024
compute time = 0.279701 sec
final mean = 0.000173
```

## ■ NVIDIA GTX 680M (cheap laptop graphics):

```
cuda> ./wave13pt 512 256 256 10
m0 = 0.680375, m1 = -0.035206, m2 = 0.094366
initial mean = 0.000024
init time = 0.202504 sec
device buffer alloc time = 0.017779 sec
data load time = 0.066286 sec (5.657294 GB/sec)
compute time = 0.122417 sec
data save time = 0.020925 sec (5.973645 GB/sec)
device buffer free time = 0.000459 sec
final mean = 0.000173
```

# Production example: stencil in CUDA

```
#define _A(array, is, iy, ix) (array[(ix) + nx * (iy) + nx * ny * (is)])

extern "C" __global__ void wave13pt(const int nx, const int ny, const int ns,
kernel_config_t config, const real m0, const real m1, const real m2,
const real* const __restrict__ w0, const real* const __restrict__ w1,
real* const __restrict__ w2)
{
#define k_offset (blockIdx.z * blockDim.z + threadIdx.z)
#define j_offset (blockIdx.y * blockDim.y + threadIdx.y)
#define i_offset (blockIdx.x * blockDim.x + threadIdx.x)
for (int k = 2 + k_offset; k < ns - 2; k += config.strideDim.z) {
for (int j = 2 + j_offset; j < ny - 2; j += config.strideDim.y) {
for (int i = i_offset; i < nx; i += config.strideDim.x) {
if ((i < 2) || (i >= nx - 2)) continue;

_A(w2, k, j, i) = m0 * _A(w1, k, j, i) - _A(w0, k, j, i) +
m1 * (
_A(w1, k, j, i+1) + _A(w1, k, j, i-1) +
_A(w1, k, j+1, i) + _A(w1, k, j-1, i) +
_A(w1, k+1, j, i) + _A(w1, k-1, j, i)) +
m2 * (
_A(w1, k, j, i+2) + _A(w1, k, j, i-2) +
_A(w1, k, j+2, i) + _A(w1, k, j-2, i) +
_A(w1, k+2, j, i) + _A(w1, k-2, j, i));
}
}
}
}
```

## ■ Dual Intel Xeon E5-2670 (OpenMP version):

```
cpu> ./wave13pt 512 256 256 10
m0 = 0.680375, m1 = -0.035206, m2 = 0.094366
initial mean = 0.000024
compute time = 0.279701 sec
final mean = 0.000173
```

## ■ NVIDIA GTX 680M (cheap laptop graphics):

```
cuda> ./wave13pt 512 256 256 10
m0 = 0.680375, m1 = -0.035206, m2 = 0.094366
initial mean = 0.000024
init time = 0.202504 sec
device buffer alloc time = 0.017779 sec
data load time = 0.066286 sec (5.657294 GB/sec)
compute time = 0.122417 sec
data save time = 0.020925 sec (5.973645 GB/sec)
device buffer free time = 0.000459 sec
final mean = 0.000173
```

# Production example: stencil in CUDA

```
#define _A(array, is, iy, ix) (array[(ix) + nx * (iy) + nx * ny * (is)])

extern "C" __global__ void wave13pt(const int nx, const int ny, const int ns,
kernel_config_t config, const real m0, const real m1, const real m2,
const real* const __restrict__ w0, const real* const __restrict__ w1,
real* const __restrict__ w2)
{
#define k_offset (blockIdx.z * blockDim.z + threadIdx.z)
#define j_offset (blockIdx.y * blockDim.y + threadIdx.y)
#define i_offset (blockIdx.x * blockDim.x + threadIdx.x)
for (int k = 2 + k_offset; k < ns - 2; k += config.strideDim.z) {
for (int j = 2 + j_offset; j < ny - 2; j += config.strideDim.y) {
for (int i = i_offset; i < nx; i += config.strideDim.x) {
if ((i < 2) || (i >= nx - 2)) continue;

_A(w2, k, j, i) = m0 * _A(w1, k, j, i) - _A(w0, k, j, i) +
m1 * (
_A(w1, k, j, i+1) + _A(w1, k, j, i-1) +
_A(w1, k, j+1, i) + _A(w1, k, j-1, i) +
_A(w1, k+1, j, i) + _A(w1, k-1, j, i)) +
m2 * (
_A(w1, k, j, i+2) + _A(w1, k, j, i-2) +
_A(w1, k, j+2, i) + _A(w1, k, j-2, i) +
_A(w1, k+2, j, i) + _A(w1, k-2, j, i));
}
}
}
}
```

## ■ Dual Intel Xeon E5-2670 (OpenMP version):

```
cpu> ./wave13pt 512 256 256 10
m0 = 0.680375, m1 = -0.035206, m2 = 0.094366
initial mean = 0.000024
compute time = 0.279701 sec
final mean = 0.000173
```

## ■ NVIDIA GTX 680M (cheap laptop graphics):

```
cuda> ./wave13pt 512 256 256 10
m0 = 0.680375, m1 = -0.035206, m2 = 0.094366
initial mean = 0.000024
init time = 0.202504 sec
device buffer alloc time = 0.017779 sec
data load time = 0.066286 sec (5.657294 GB/sec)
compute time = 0.122417 sec
data save time = 0.020925 sec (5.973645 GB/sec)
device buffer free time = 0.000459 sec
final mean = 0.000173
```

# Production example: stencil in CUDA

```
#define _A(array, is, iy, ix) (array[(ix) + nx * (iy) + nx * ny * (is)])

extern "C" __global__ void wave13pt(const int nx, const int ny, const int ns,
kernel_config_t config, const real m0, const real m1, const real m2,
const real* const __restrict__ w0, const real* const __restrict__ w1,
real* const __restrict__ w2)
{
#define k_offset (blockIdx.z * blockDim.z + threadIdx.z)
#define j_offset (blockIdx.y * blockDim.y + threadIdx.y)
#define i_offset (blockIdx.x * blockDim.x + threadIdx.x)
for (int k = 2 + k_offset; k < ns - 2; k += config.strideDim.z) {
for (int j = 2 + j_offset; j < ny - 2; j += config.strideDim.y) {
for (int i = i_offset; i < nx; i += config.strideDim.x) {
if ((i < 2) || (i >= nx - 2)) continue;

_A(w2, k, j, i) = m0 * _A(w1, k, j, i) - _A(w0, k, j, i) +
m1 * (
_A(w1, k, j, i+1) + _A(w1, k, j, i-1) +
_A(w1, k, j+1, i) + _A(w1, k, j-1, i) +
_A(w1, k+1, j, i) + _A(w1, k-1, j, i)) +
m2 * (
_A(w1, k, j, i+2) + _A(w1, k, j, i-2) +
_A(w1, k, j+2, i) + _A(w1, k, j-2, i) +
_A(w1, k+2, j, i) + _A(w1, k-2, j, i));
}
}
}
}
```

## ■ Dual Intel Xeon E5-2670 (OpenMP version):

```
cpu> ./wave13pt 512 256 256 10
m0 = 0.680375, m1 = -0.035206, m2 = 0.094366
initial mean = 0.000024
compute time = 0.279701 sec
final mean = 0.000173
```

## ■ NVIDIA GTX 680M (cheap laptop graphics):

```
cuda> ./wave13pt 512 256 256 10
m0 = 0.680375, m1 = -0.035206, m2 = 0.094366
initial mean = 0.000024
init time = 0.202504 sec
device buffer alloc time = 0.017779 sec
data load time = 0.066286 sec (5.657294 GB/sec)
compute time = 0.122417 sec
data save time = 0.020925 sec (5.973645 GB/sec)
device buffer free time = 0.000459 sec
final mean = 0.000173
```

- Всегда старайтесь использовать выровненный доступ к памяти (coalescing)
- Размеры блоков  $\{128, 1, 1\}$  будут приблизительно оптимальными ждя большинства задач и современных GPU
- Компилятор может оптимизировать код для конкретной архитектуры GPU, определяемой Compute Capability (CC):

`nvcc -arch=sm_35` для запуска на процессорах Tesla K20

```
$ nvcc -arch=sm_35 -O3 cuda_gpu_data.cu -o cuda_gpu_data
$ ./cuda_gpu_data
```

- Всегда старайтесь использовать выровненный доступ к памяти (coalescing)
- Размеры блоков  $\{128, 1, 1\}$  будут приблизительно оптимальными ждя большинства задач и современных GPU
- Компилятор может оптимизировать код для конкретной архитектуры GPU, определяемой Compute Capability (CC):

`nvcc -arch=sm_35` для устройств поколения Tesla K20x

```
$ nvcc -arch=sm_35 -O3 cuda_gpu_data.cu -o cuda_gpu_data
$ ./cuda_gpu_data
```

- Всегда старайтесь использовать выровненный доступ к памяти (coalescing)
- Размеры блоков `{128, 1, 1}` будут приблизительно оптимальными для большинства задач и современных GPU
- Компилятор может оптимизировать код для конкретной архитектуры GPU, определяемой Compute Capability (CC):

`nvcc -arch=sm_35 -O3 cuda_gpu_data.cu -o cuda_gpu_data`

```
$ nvcc -arch=sm_35 -O3 cuda_gpu_data.cu -o cuda_gpu_data
$ ./cuda_gpu_data
```

- Всегда старайтесь использовать выровненный доступ к памяти (coalescing)
- Размеры блоков `{128, 1, 1}` будут приблизительно оптимальными для большинства задач и современных GPU
- Компилятор может оптимизировать код для конкретной архитектуры GPU, определяемой Compute Capability (CC):

add option `-arch=sm_35` для лучших результатов на Tesla K20:

```
$ nvcc -arch=sm_35 -O3 cuda_gpu_data.cu -o cuda_gpu_data
$ ./cuda_gpu_data
```

- CUDA 8 поддерживает интеграцию с Microsoft Visual Studio 2015
- Скачать и установить Visual Studio (Community Edition будет достаточно):  
<https://www.visualstudio.com/downloads/>
- Скачать и установить CUDA 8 Toolkit:  
<https://developer.nvidia.com/cuda-toolkit>

# Microsoft Visual Studio 2015 для CUDA

## Visual Studio Downloads

### Visual Studio Community

Free, fully-featured IDE for students, open-source and individual developers

Free download 



### Visual Studio Professional

Professional developer tools, services, and subscription benefits for small teams

Free trial 



### Visual Studio Enterprise

End-to-end solution to meet demanding quality and scale needs of teams of all sizes

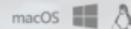
Free trial 



### Visual Studio Code

Code editing, redefined. Free, open source, and runs everywhere.

Free download 



## NVIDIA DGX-1 THE WORLD'S FIRST DEEP LEARNING SUPERCOMPUTER IN A BOX

[LEARN MORE](#)



[Home](#) > [ComputeWorks](#) > [CUDA ZONE](#) > [Tools & Ecosystem](#) > [CUDA Toolkit](#) > [CUDA 8.0 Downloads](#)

Learn more about CUDA Toolkit 8.0:

- Read the **CUDA 8 Features Revealed** Parallel Forall Blog Post.
- Sign up for a live walkthrough: **What's New in CUDA 8** webinar on Thursday, October 13th.
- Review the **CUDA 8 Overview** slides presented at GPU Technology Conference (GTC) 2016

For Linux users upgrading from previous versions of the CUDA Toolkit, click to see instructions in this section before proceeding.

For developers on systems using Tesla P100 GPUs, click to see instructions before installation..

For developers on Mac systems, click here to read a known limitation.

### Select Target Platform

Click on the green buttons that describe your target platform. Only supported platforms will be shown.

Operating System

[Windows](#)

[Linux](#)

[Mac OSX](#)

### Related Links

[CUDA Quick Start Guide](#)

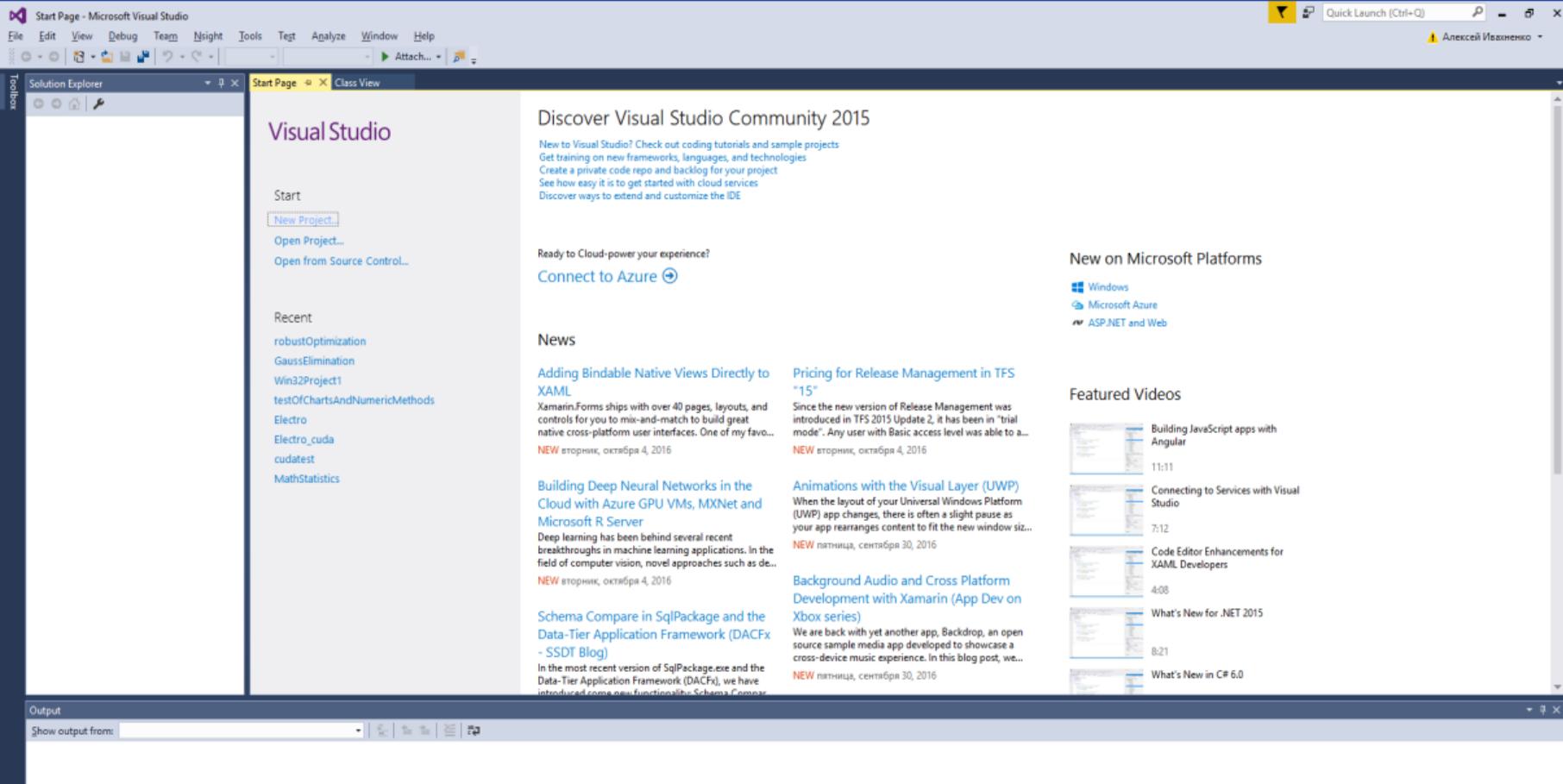
[Release Notes](#)

[EULA](#)

[Online Documentation](#)

[CUDA Toolkit 8.0](#)

# Microsoft Visual Studio 2015 для CUDA



The screenshot shows the Microsoft Visual Studio 2015 Start Page. The interface includes a menu bar (File, Edit, View, Debug, Team, Insight, Tools, Test, Analyze, Window, Help), a toolbar, and a Solution Explorer on the left. The main content area is titled "Visual Studio" and features several sections:

- Start:** Includes buttons for "New Project...", "Open Project...", and "Open from Source Control...".
- Recent:** Lists recent projects such as "robustOptimization", "GaussElimination", "Win32Project1", "testOfChartsAndNumericMethods", "Electro", "Electro\_cuda", "cudatest", and "MathStatistics".
- Discover Visual Studio Community 2015:** A section with introductory text and links for new users.
- Ready to Cloud-power your experience? Connect to Azure:** A section with a "Connect to Azure" button.
- News:** A list of recent articles, including "Adding Bindable Native Views Directly to XAML", "Pricing for Release Management in TFS '15'", "Building Deep Neural Networks in the Cloud with Azure GPU VMs, MXNet and Microsoft R Server", "Animations with the Visual Layer (UWP)", "Background Audio and Cross Platform Development with Xamarin (App Dev on Xbox series)", and "Schema Compare in SqlPackage and the Data-Tier Application Framework (DACFx - SSDT Blog)".
- New on Microsoft Platforms:** A section with links for "Windows", "Microsoft Azure", and "ASP.NET and Web".
- Featured Videos:** A list of video thumbnails and titles, including "Building JavaScript apps with Angular", "Connecting to Services with Visual Studio", "Code Editor Enhancements for XAML Developers", "What's New for .NET 2015", and "What's New in C# 6.0".

At the bottom, there is an "Output" window with a "Show output from:" dropdown and various icons for output control.

# Microsoft Visual Studio 2015 для CUDA

New Project

Recent .NET Framework 4.5.2 Sort by: Default Search Installed Templates (Ctrl+E)

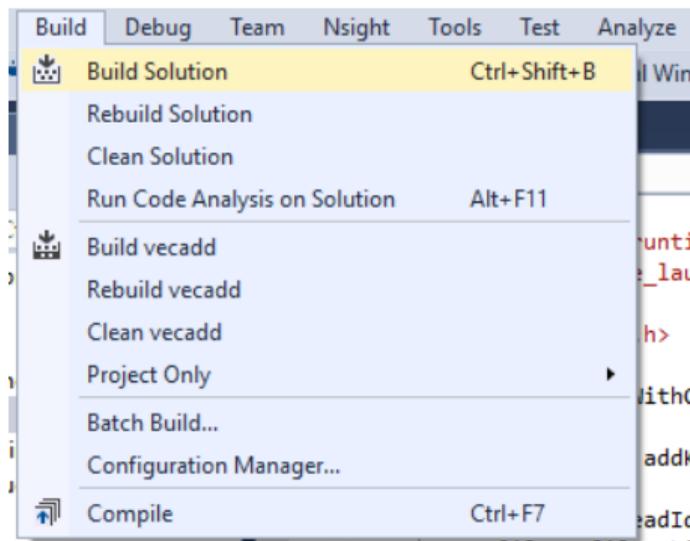
- Installed
  - Visual Basic
  - Visual F#
  - Visual C++
    - Windows
    - ATL
    - CLR
    - General
    - MFC
    - Test
    - Win32
    - Cross Platform
    - Extensibility
  - SQL Server
  - Python
  - JavaScript
  - Game
  - Build Accelerator
  - Other Project Types
  - NVIDIA
    - CUDA 8.0
  - Samples
- Online

Click here to go online and find templates.

Type: CUDA 8.0  
A project that uses the CUDA 8.0 runtime

Name: vecadd

Location: C:\Users\ПК\Documents\Visual Studio 2015\Projects Browse...



# Microsoft Visual Studio 2015 для CUDA

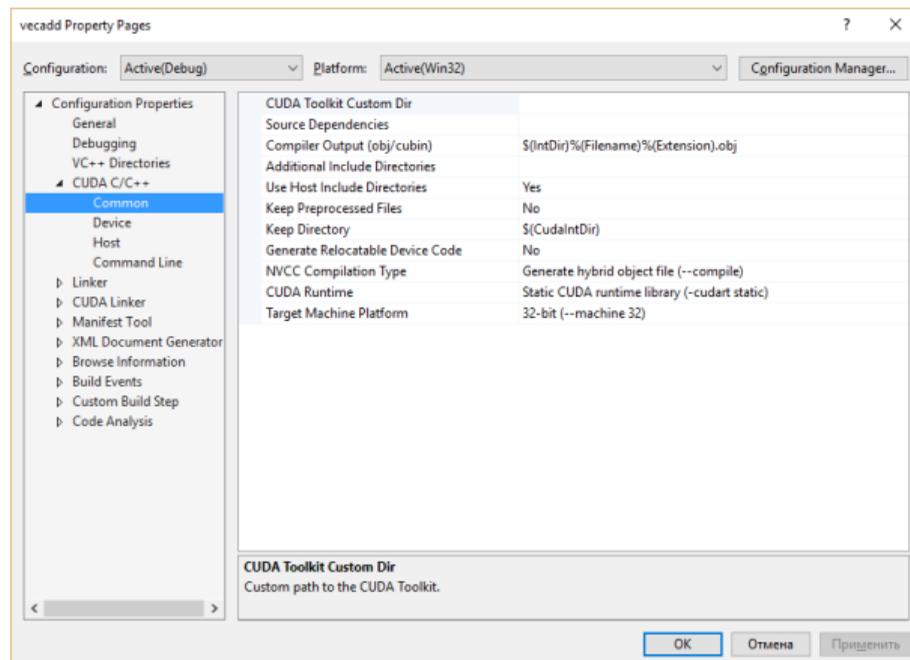
The screenshot displays the Microsoft Visual Studio 2015 IDE with a CUDA project named 'vecadd'. The main editor window shows the source code for 'kernel.cu'. The code includes headers for 'cuda\_runtime.h', 'device\_launch\_parameters.h', and 'stdio.h'. It defines a global function 'addKernel' and a 'main' function. The 'main' function sets up an array of size 5, calls 'addKernel' to perform a parallel addition, and prints the results. The Output window at the bottom shows the execution output, indicating that all threads exited successfully with code 0.

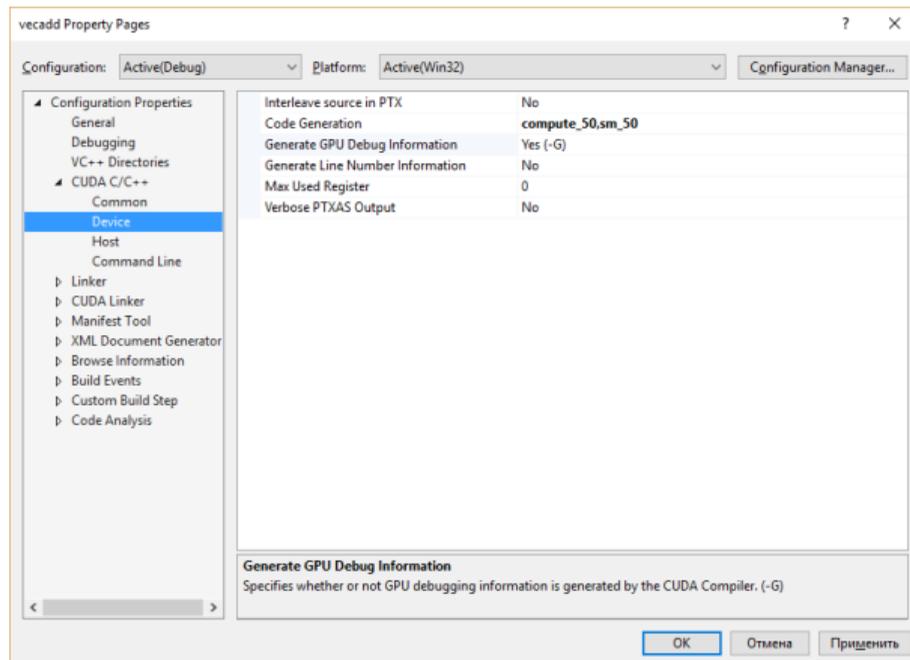
```
1
2 #include "cuda_runtime.h"
3 #include "device_launch_parameters.h"
4
5 #include <stdio.h>
6
7 cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int size);
8
9 #global__ void addKernel(int *c, const int *a, const int *b)
10 {
11     int i = threadIdx.x;
12     c[i] = a[i] + b[i];
13 }
14
15 int main()
16 {
17     const int arraySize = 5;
18     const int a[arraySize] = { 1, 2, 3, 4, 5 };
19     const int b[arraySize] = { 10, 20, 30, 40, 50 };
20     int c[arraySize] = { 0 };
21
22     // Add vectors in parallel.
23     cudaError_t cudaStatus = addWithCuda(c, a, b, arraySize);
24     if (cudaStatus != cudaSuccess) {
25         fprintf(stderr, "addWithCuda failed!");
26         return 1;
27     }
28
29     printf("{1,2,3,4,5} + {10,20,30,40,50} = {%d,%d,%d,%d,%d}\n",
30           c[0], c[1], c[2], c[3], c[4]);
31
32     // cudaDeviceReset must be called before exiting in order for profiling and
33     // tracing tools such as Nsight and Visual Profiler to show complete traces.
34     cudaStatus = cudaDeviceReset();
35     if (cudaStatus != cudaSuccess) {
36         fprintf(stderr, "cudaDeviceReset failed!");
37         return 1;
38     }
39
40     return 0;
41 }
42
43
```

Output

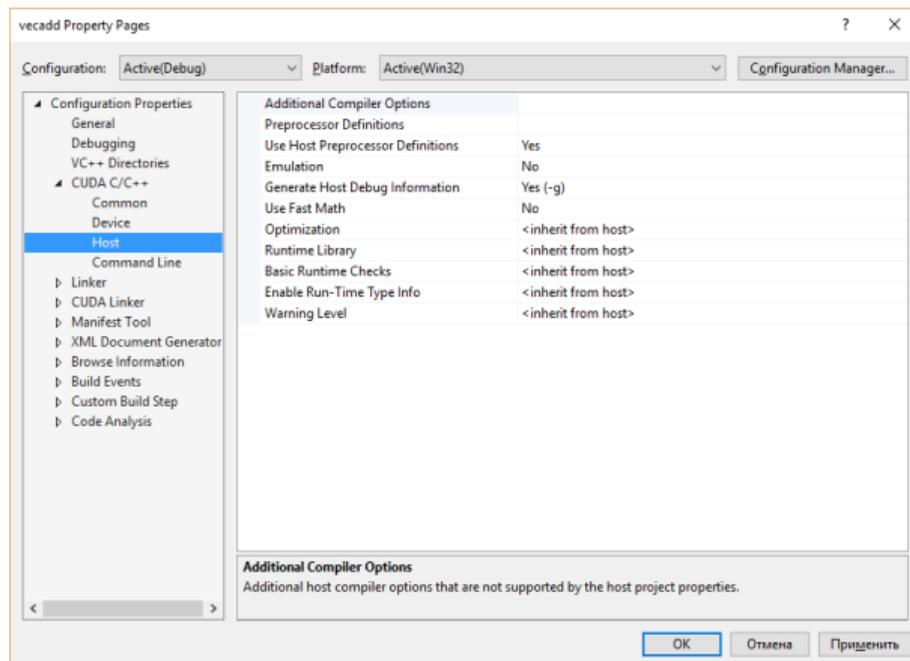
```
Show output from: Debug
The thread 0x360 has exited with code 0 (0x0).
The thread 0x10e4 has exited with code 0 (0x0).
The thread 0x1798 has exited with code 0 (0x0).
The thread 0x2338 has exited with code 0 (0x0).
```

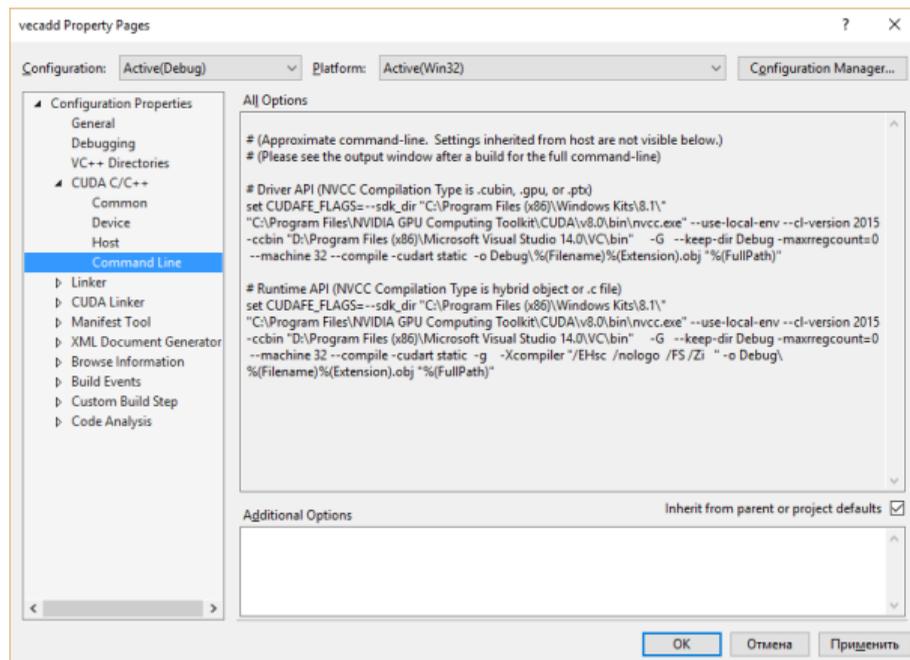
# Microsoft Visual Studio 2015 для CUDA



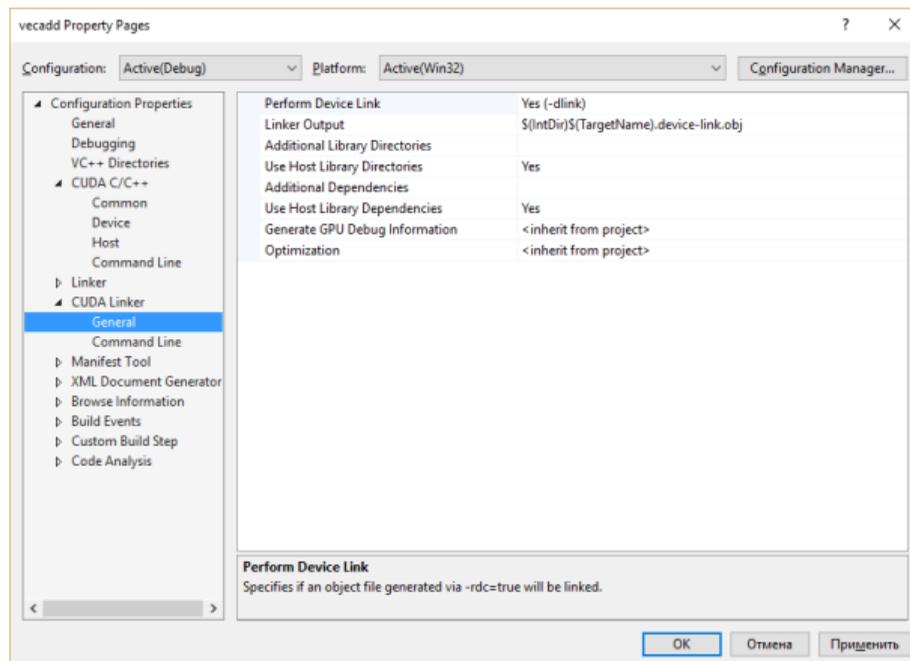


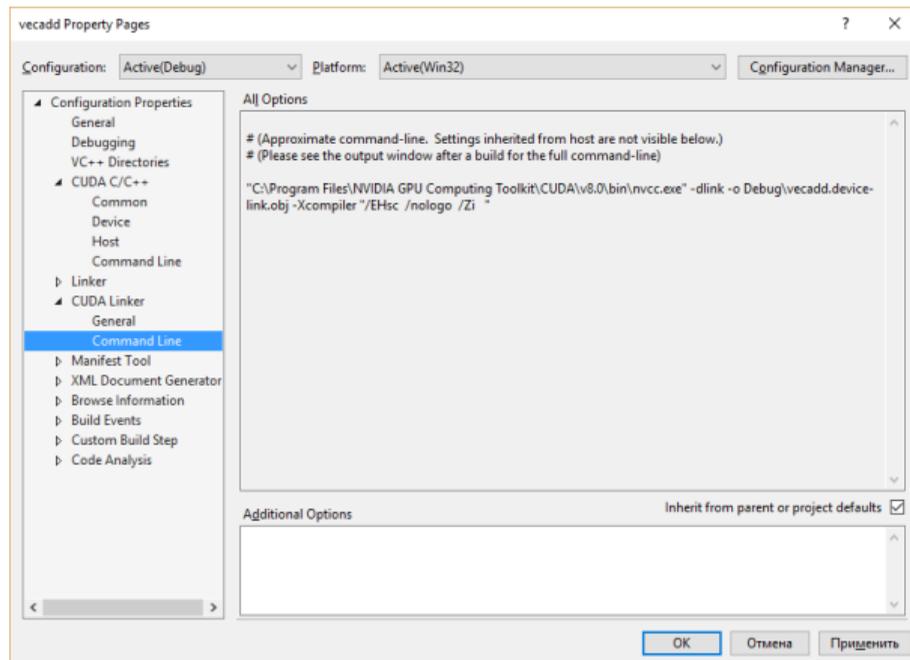
# Microsoft Visual Studio 2015 для CUDA





# Microsoft Visual Studio 2015 для CUDA





- Eclipse Nsight Edition, часть **CUDA toolkit**, поставляется со встроенной поддержкой CUDA
- Если у вас нет локальной GPU NVIDIA, можно разрабатывать приложения в Eclipse Nsight Edition и запускать их на **удаленном GPU**

```
C:/C++-bilinear/src/bilinear.cu - Nsight
File Edit Source Refactor Navigate Search Project Run Window Help

Project Explorer
  bilinear
  Binaries
  Includes
  src
  [.] bilinear.cu
  Debug

bilinear.cu
36 int main(void) {
37     void *d = MLL;
38     int i;
39     unsigned int idata[WORK_SIZE], odata[WORK_SIZE];
40
41     for (i = 0; i < WORK_SIZE; i++)
42         idata[i] = (unsigned int) i;
43
44     CUDA_CHECK_RETURN(cudaMalloc((void**) &d, sizeof(int) * WORK_SIZE));
45     CUDA_CHECK_RETURN(cudaMemcpy(odata, d, sizeof(int) * WORK_SIZE, cudaMemcpyHostToDevice));
46
47     bitreverse<<=1, WORK_SIZE, WORK_SIZE * sizeof(int)>>>(d);
48
49     CUDA_CHECK_RETURN(cudaDeviceSynchronize()); // Wait for the GPU launched work to complete
50     CUDA_CHECK_RETURN(cudaGetLastError());
51     CUDA_CHECK_RETURN(cudaMemcpy(odata, d, sizeof(int) * WORK_SIZE, cudaMemcpyDeviceToHost));
52
53     for (i = 0; i < WORK_SIZE; i++)
54         printf("Input value: %u\n", idata[i], odata[i]);
55
56
57     CUDA_CHECK_RETURN(cudaFree((void*) d));
58     CUDA_CHECK_RETURN(cudaDeviceReset());
59
60     return 0;
61 }
62

Problems Tasks Console Properties
<terminated-tesla-cmc [C/C++ Remote Application] Remote Shell
Last login: Sun Mar 1 18:30:06 2015 from 213.188.52.207
echo $PWD>^
/bin/sh -c "cd \"/tmp/nsight-debug"/:"/tmp/nsight-debug/bilinear";exit
deikashin@tesla-cmc:~$ echo $PWD>^
~/nsight-deikashin=
deikashin@tesla-cmc:~$ /bin/sh -c "cd \"/tmp/nsight-debug"/:"/tmp/nsight-debug/ bilinear";exit
Input value: 0, device output: 0
Input value: 1, device output: 128
Input value: 2, device output: 64
Input value: 3, device output: 192
Input value: 4, device output: 32
```

- Eclipse Nsight Edition, часть [CUDA toolkit](#), поставляется со встроенной поддержкой CUDA
- Если у вас нет локальной GPU NVIDIA, можно разрабатывать приложения в Eclipse Nsight Edition и запускать их на **удаленном** GPU

Как настроить проект CUDA с нуля:

- File → New → CUDA C/C++ project
- CUDA Runtime Project “hello” → Так же укажите соответствующее CC для удаленного GPU
- CPU architecture: x86 (32-bit) → Finish
- Будет сгенерирован код примера CUDA
- Project → Build All
- Run → Run → если у вас имеется локальная NVIDIA GPU → Local C/C++ Application, иначе – use GPU from the remote server → Remote C/C++ Application:
  - Remote connection → Manage → specify your SSH account data
  - Toolkit path: /opt/cuda/bin
  - После нажатия “Finish”, приложение должно исполняться удаленно

- CUDA легко использовать, если вы уже работали с MPI+OpenMP
- Основная стратегия портирования кода:
  - Перенесите тело цикла в `__global__` функцию – GPU ядро
  - Замените оригинальный цикл на вызов ядра, преобразуйте интервалы циклов в `<<< ... >>>` параметры вызова ядра
  - Скопируйте данные на GPU до, скопируйте результаты с GPU после выполнения ядра
  - Замените компилятор с C/C++ на nvcc и перекомпилируйте
- Существуют IDE для CUDA
- Некоторые вещи, не рассмотренные в этой лекции:
  - Разделение памяти
  - Тренировка и оптимизация CUDA кода
  - Работа с потоками

- CUDA легко использовать, если вы уже работали с MPI+OpenMP
- Основная стратегия портирования кода:
  - Перенесите тело цикла в `__global__` функцию – GPU ядро
  - Замените оригинальный цикл на вызов ядра, преобразуйте интервалы циклов в `<<< ... >>>` параметры вызова ядра
  - Скопируйте данные на GPU до, скопируйте результаты с GPU после выполнения ядра
  - Замените компилятор с C/C++ на nvcc и перекомпилируйте
- Существуют IDE для CUDA
- Некоторые вещи, не рассмотренные в этой лекции:
  - Разделение памяти
  - Тренировка и использование графического ядра
  - Работа с потоками

- CUDA легко использовать, если вы уже работали с MPI+OpenMP
- Основная стратегия портирования кода:
  - Перенесите тело цикла в `__global__` функцию – GPU ядро
    - Замените оригинальный цикл на вызов ядра, преобразуйте интервалы циклов в `<<< ... >>>` параметры вызова ядра
    - Скопируйте данные на GPU до, скопируйте результаты с GPU после выполнения ядра
    - Замените компилятор с C/C++ на nvcc и перекомпилируйте
- Существуют IDE для CUDA
- Некоторые вещи, не рассмотренные в этой лекции:
  - Разделение памяти
  - Тренировка и использование графического ядра
  - Перенос на другие архитектуры

- CUDA легко использовать, если вы уже работали с MPI+OpenMP
- Основная стратегия портирования кода:
  - Перенесите тело цикла в `__global__` функцию – GPU ядро
  - Замените оригинальный цикл на вызов ядра, преобразуйте интервалы циклов в `<<< ... >>>` параметры вызова ядра
    - Скопируйте данные на GPU до, скопируйте результаты с GPU после выполнения ядра
    - Замените компилятор с C/C++ на nvcc и перекомпилируйте
- Существуют IDE для CUDA
- Некоторые вещи, не рассмотренные в этой лекции:

1. [Введение в CUDA](#)

2. [Портинг и оптимизация CUDA кода](#)

3. [Портинг OpenMP кода](#)

- CUDA легко использовать, если вы уже работали с MPI+OpenMP
- Основная стратегия портирования кода:
  - Перенесите тело цикла в `__global__` функцию – GPU ядро
  - Замените оригинальный цикл на вызов ядра, преобразуйте интервалы циклов в `<<< ... >>>` параметры вызова ядра
  - Скопируйте данные на GPU до, скопируйте результаты с GPU после выполнения ядра
    - Замените компилятор с C/C++ на nvcc и перекомпилируйте
- Существуют IDE для CUDA
- Некоторые вещи, не рассмотренные в этой лекции:

- CUDA легко использовать, если вы уже работали с MPI+OpenMP
- Основная стратегия портирования кода:
  - Перенесите тело цикла в `__global__` функцию – GPU ядро
  - Замените оригинальный цикл на вызов ядра, преобразуйте интервалы циклов в `<<< ... >>>` параметры вызова ядра
  - Скопируйте данные на GPU до, скопируйте результаты с GPU после выполнения ядра
  - Замените компилятор с C/C++ на nvcc и перекомпилируйте
- Существуют IDE для CUDA
- Некоторые вещи, не рассмотренные в этой лекции:

- CUDA легко использовать, если вы уже работали с MPI+OpenMP
- Основная стратегия портирования кода:
  - Перенесите тело цикла в `__global__` функцию – GPU ядро
  - Замените оригинальный цикл на вызов ядра, преобразуйте интервалы циклов в `<<< ... >>>` параметры вызова ядра
  - Скопируйте данные на GPU до, скопируйте результаты с GPU после выполнения ядра
  - Замените компилятор с C/C++ на nvcc и перекомпилируйте
- Существуют IDE для CUDA
- Некоторые вещи, не рассмотренные в этой лекции:

- CUDA легко использовать, если вы уже работали с MPI+OpenMP
- Основная стратегия портирования кода:
  - Перенесите тело цикла в `__global__` функцию – GPU ядро
  - Замените оригинальный цикл на вызов ядра, преобразуйте интервалы циклов в `<<< ... >>>` параметры вызова ядра
  - Скопируйте данные на GPU до, скопируйте результаты с GPU после выполнения ядра
  - Замените компилятор с C/C++ на nvcc и перекомпилируйте
- Существуют IDE для CUDA
- Некоторые вещи, не рассмотренные в этой лекции:
  - Разделяемая память
  - Потoki и асинхронная передача данных
  - Отладка и профилирование