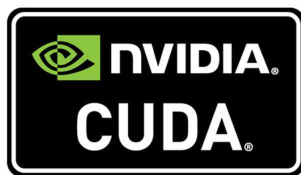
# Введение в CUDA

к.т.н. Алексей Ивахненко

25 Февраля, 2017

# Что такое CUDA




- **CUDA** = **C**ompute-**U**nified **D**evice **A**rchitecutre
  - 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, ...

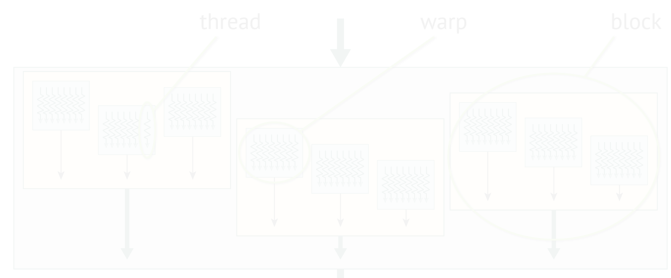
# Что такое CUDA

- **CUDA** = **C**ompute-**U**nified **D**evice **A**rchitecutre
  - 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, ...

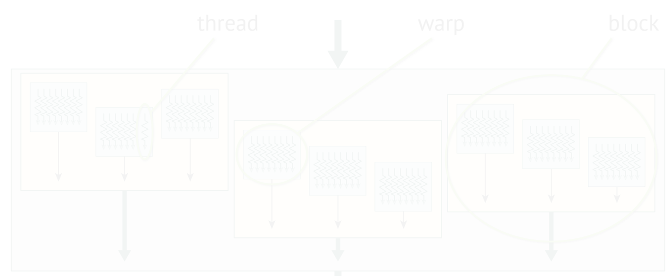
# Что такое CUDA

- **CUDA** = **C**ompute-**U**nified **D**evice **A**rchitecutre
  - 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, ...

# Основы CUDA

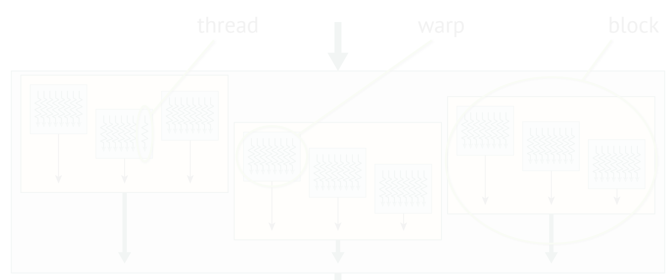



- Hardware:
  - Массивный параллелизм: тысячи вычислительных ядер
- Потоковое исполнение:
  - Тысячи параллельных нитей сгруппированы в блоки и варпы:
    - Нити каждого блока могут использовать быструю (≈на уровне регистров) память, называемую раздеяемая/общая (shared) память
    - Варпы – это подгруппы нитей внутри блоков, исполняемые синхронно
  - **SIMT**: **S**imultaneous **I**nstruction – **M**ultiple **T**hreads
    - Все нити одного варпа исполняют одни и те же инструкции над разными данными
      (в то время как в традиционном SIM**D** – одна инструкция применяется ко множеству данных)
    - Исполнение различных варпов происходит несинхронно/неодновременно




- Hardware:
  - Массивный параллелизм: тысячи вычислительных ядер
- Потоковое исполнение:
  - Тысячи параллельных нитей сгруппированы в блоки и варпы:
    - Нити каждого блока могут использовать быструю (≈на уровне регистров) память, называемую раздеяемая/общая (shared) память
    - Варпы – это подгруппы нитей внутри блоков, исполняемые синхронно
  - **SIMT**: **S**imultaneous **I**nstruction – **M**ultiple **T**hreads
    - Все нити одного варпа исполняют одни и те же инструкции над разными данными (в то время как в традиционном SIM**D** – одна инструкция применяется ко множеству данных)
    - Исполнение различных варпов происходит несинхронно/неодновременно

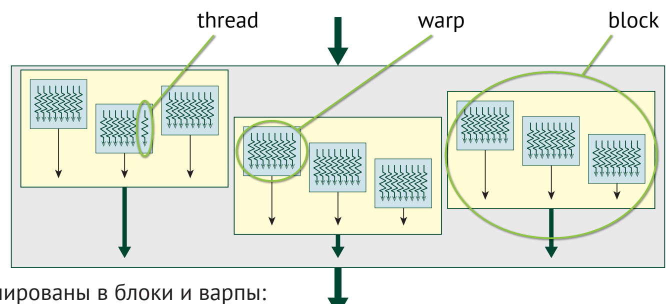
# Основы CUDA




- Hardware:
  - Массивный параллелизм: тысячи вычислительных ядер
- Потоковое исполнение:
  - Тысячи параллельных нитей сгруппированы в блоки и варпы:
    - Нити каждого блока могут использовать быструю (≈на уровне регистров) память, называемую раздеяемая/общая (shared) память
    - Варпы – это подгруппы нитей внутри блоков, исполняемые синхронно
  - **SIMT**: **S**imultaneous **I**nstruction – **M**ultiple **T**hreads
    - Все нити одного варпа исполняют одни и те же инструкции над разными данными

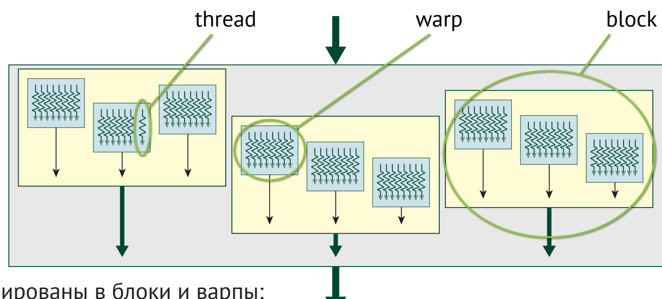
      (в то время как в традиционном SIM**D** – одна инструкция применяется ко множеству данных)
    - Исполнение различных варпов происходит несинхронно/неодновременно

# Основы CUDA




- Hardware:
  - Массивный параллелизм: тысячи вычислительных ядер
- Потоковое исполнение:
  - Тысячи параллельных нитей сгруппированы в блоки и варпы:
    - Нити каждого блока могут использовать быструю (≈на уровне регистров) память, называемую раздеяемая/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

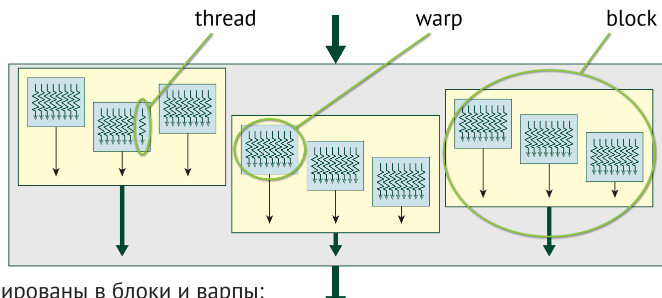
# Основы CUDA




- Hardware:
  - Массивный параллелизм: тысячи вычислительных ядер
- Потоковое исполнение:
  - Тысячи параллельных нитей сгруппированы в блоки и варпы:
    - Нити каждого блока могут использовать быструю (≈на уровне регистров) память, называемую раздеяемая/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

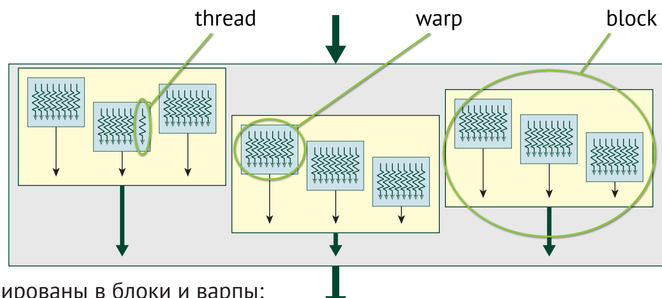
# Основы CUDA




- Hardware:
  - Массивный параллелизм: тысячи вычислительных ядер
- Потоковое исполнение:
  - Тысячи параллельных нитей сгруппированы в блоки и варпы:
    - Нити каждого блока могут использовать быструю (≈на уровне регистров) память, называемую раздеяемая/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

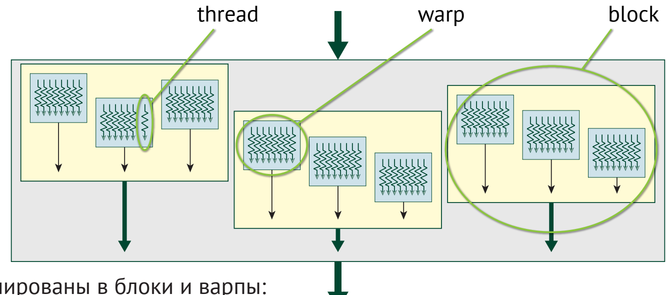
      (в то время как в традиционном SIMD – одна инструкция применяется ко множеству данных)
    - Исполнение различных варпов происходит несинхронно/неодновременно

# Основы CUDA



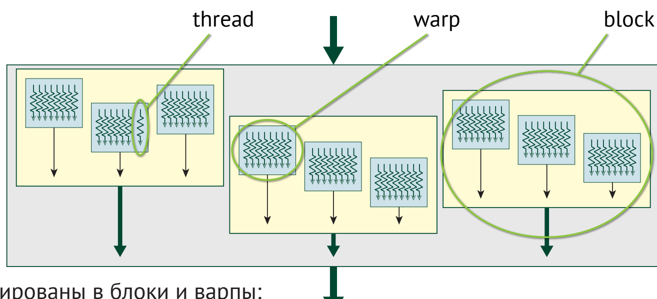

- Hardware:
  - Массивный параллелизм: тысячи вычислительных ядер
- Потоковое исполнение:
  - Тысячи параллельных нитей сгруппированы в блоки и варпы:
    - Нити каждого блока могут использовать быструю (≈на уровне регистров) память, называемую раздеяемая/общая (shared) память
    - Варпы – это подгруппы нитей внутри блоков, исполняемые синхронно
  - **SIMT**: **S**imultaneous **I**nstruction – **M**ultiple **T**hreads
    - 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

# Основы CUDA




- Hardware:
  - Массивный параллелизм: тысячи вычислительных ядер
- Потоковое исполнение:
  - Тысячи параллельных нитей сгруппированы в блоки и варпы:
    - Нити каждого блока могут использовать быструю (≈на уровне регистров) память, называемую раздеяемая/общая (shared) память
    - Варпы – это подгруппы нитей внутри блоков, исполняемые синхронно
  - **SIMT**: **S**imultaneous **I**nstruction – **M**ultiple **T**hreads
    - Все нити одного варпа исполняют одни и те же инструкции над разными данными (в то время как в традиционном SIM**D** – одна инструкция применяется ко множеству данных)
    - Исполнение различных варпов происходит несинхронно/неодновременно

# Основы CUDA



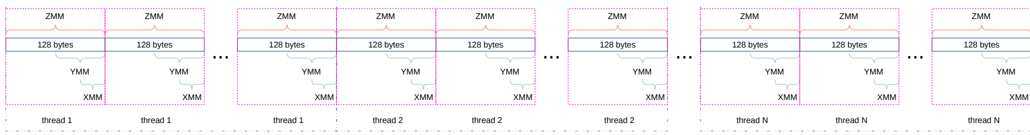

- Hardware:
  - Массивный параллелизм: тысячи вычислительных ядер
- Потоковое исполнение:
  - Тысячи параллельных нитей сгруппированы в блоки и варпы:
    - Нити каждого блока могут использовать быструю (≈на уровне регистров) память, называемую раздеяемая/общая (shared) память
    - Варпы – это подгруппы нитей внутри блоков, исполняемые синхронно
  - **SIMT**: **S**imultaneous **I**nstruction – **M**ultiple **T**hreads
    - Все нити одного варпа исполняют одни и те же инструкции над разными данными
      (в то время как в традиционном SIM**D** – одна инструкция применяется ко множеству данных)
    - Исполнение различных варпов происходит несинхронно/неодновременно

# SIMT vs SIMD

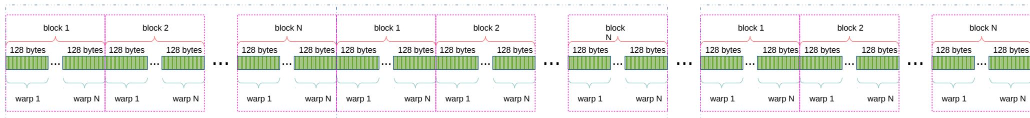
Рассмотрим, как вычисления будут распределены между потоками/нитями в следующем цикле (N $\gg$ количество нитей):

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

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




- SIM**T**-схема для CUDA/GPU с 32 нитями в варпе:

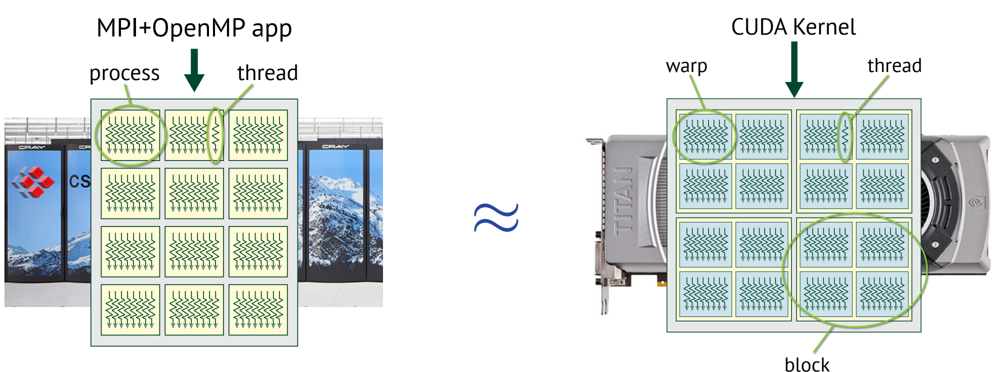



▯ - thread (lightweight GPU-thread)

$\Rightarrow$ 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 ≈ MPI+OpenMP кластер в микромасштабе, помещенный в 11-дюймовую коробочку!

# CUDA вычислительная сетка (compute grid)

| MPI | OpenMP | CUDA |
|---|---|---|
| MPI_Comm_rank() | | blockIdx |
| MPI_Comm_size() | | gridDim |
| | omp_get_thread_num() | threadIdx |
| | omp_get_num_threads() | 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() | | blockIdx |
| MPI_Comm_size() | | gridDim |
| | omp_get_thread_num() | threadIdx |
| | omp_get_num_threads() | 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() | | blockIdx |
| MPI_Comm_size() | | gridDim |
| | omp_get_thread_num() | threadIdx |
| | omp_get_num_threads() | 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() | | blockIdx |
| MPI_Comm_size() | | gridDim |
| | omp_get_thread_num() | threadIdx |
| | omp_get_num_threads() | 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() | | blockIdx |
| MPI_Comm_size() | | gridDim |
| | omp_get_thread_num() | threadIdx |
| | omp_get_num_threads() | 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() | | blockIdx |
| MPI_Comm_size() | | gridDim |
| | omp_get_thread_num() | threadIdx |
| | omp_get_num_threads() | 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() | | blockIdx |
| MPI_Comm_size() | | gridDim |
| | omp_get_thread_num() | threadIdx |
| | omp_get_num_threads() | 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() | | blockIdx |
| MPI_Comm_size() | | gridDim |
| | omp_get_thread_num() | threadIdx |
| | omp_get_num_threads() | 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() | | blockIdx |
| MPI_Comm_size() | | gridDim |
| | omp_get_thread_num() | threadIdx |
| | omp_get_num_threads() | 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() | | blockIdx |
| MPI_Comm_size() | | gridDim |
| | omp_get_thread_num() | threadIdx |
| | omp_get_num_threads() | 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() | | blockIdx |
| MPI_Comm_size() | | gridDim |
| | omp_get_thread_num() | threadIdx |
| | omp_get_num_threads() | 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 compute grid

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;
}
```

```
$ OMP_NUM_THREADS=2 mpirun -np 2 ./mpi_openmp_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!
```

# CUDA compute grid

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;
}
```

```
$ OMP_NUM_THREADS=2 mpirun —np 2 ./mpi_openmp_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!
```

# CUDA compute grid

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;
}
```

```
$ OMP_NUM_THREADS=2 mpirun —np 2 ./mpi_openmp_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!
```

# CUDA compute grid

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;
}
```

```
$ OMP_NUM_THREADS=2 mpirun —np 2 ./mpi_openmp_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!
```

# CUDA compute grid

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;
}
```

```
$ OMP_NUM_THREADS=2 mpirun —np 2 ./mpi_openmp_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!
```

# CUDA compute grid

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;
}
```

```
$ OMP_NUM_THREADS=2 mpirun —np 2 ./mpi_openmp_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!
```

# CUDA compute grid

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;
}
```

```
$ OMP_NUM_THREADS=2 mpirun —np 2 ./mpi_openmp_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!
```

# CUDA compute grid

mpi_openmp_test.c

```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_openmp_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

```c
__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);

}
```

```c
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!
```

# CUDA compute grid

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;
}
```

```
$ OMP_NUM_THREADS=2 mpirun —np 2 ./mpi_openmp_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!
```

# CUDA compute grid

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;
}
```

```
$ OMP_NUM_THREADS=2 mpirun —np 2 ./mpi_openmp_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);



}
```

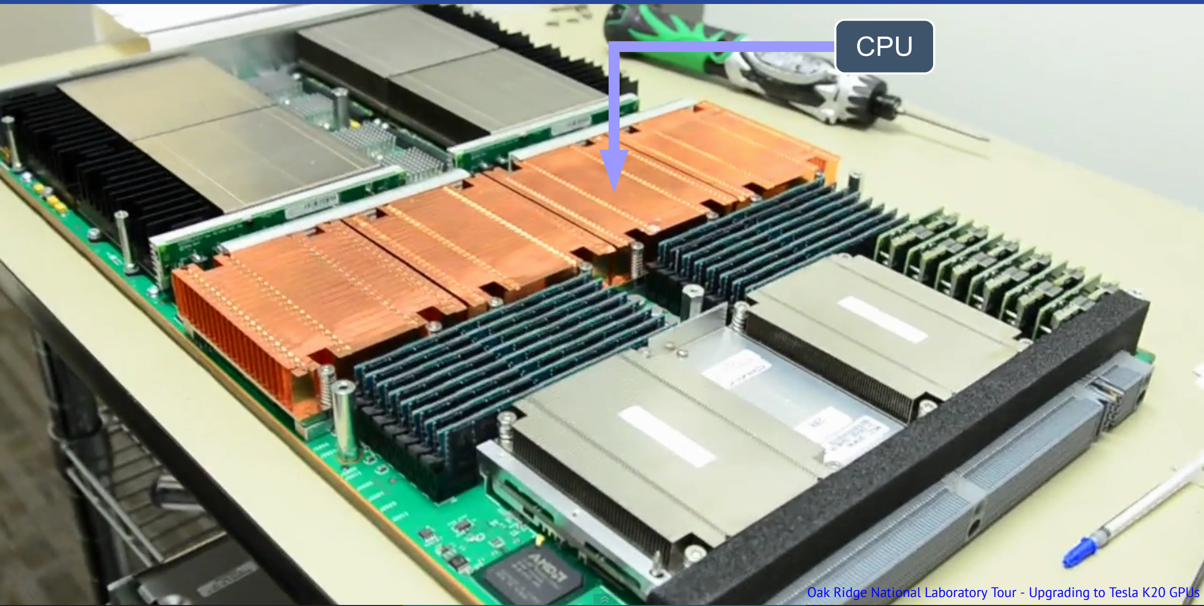
```
int main() {
    gpu_kernel<<<2, 2>>>();
    CUDA_ERR_CHECK( cudaGetLastError() );
    CUDA_ERR_CHECK( cudaDeviceSynchronize() );
    return 0;
}
```

```
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!
```

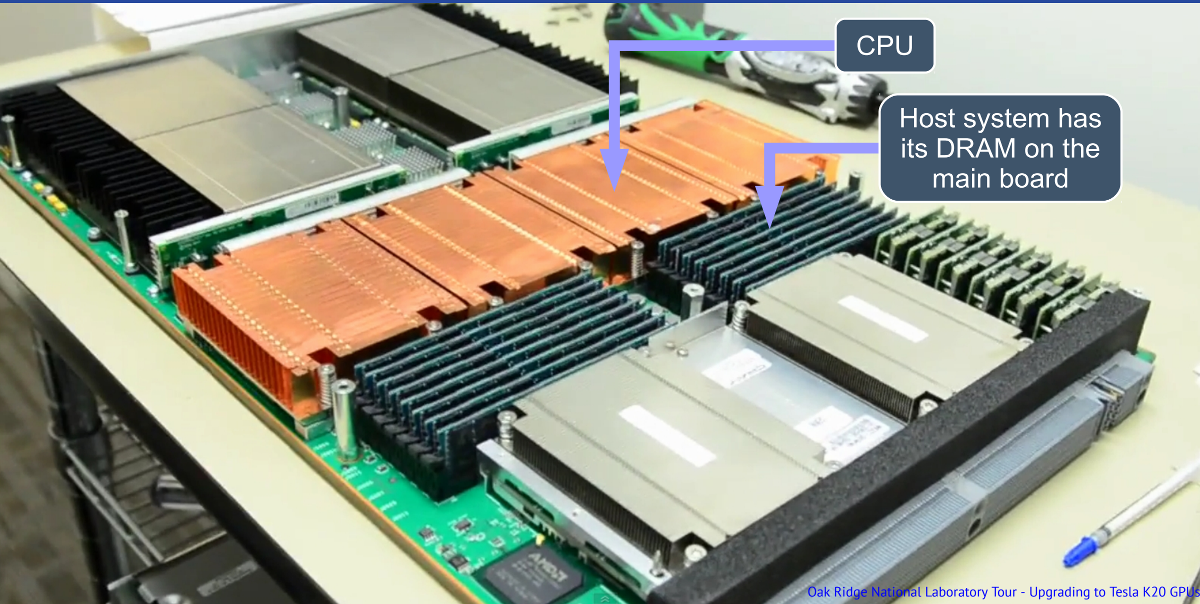
# Память GPU↔память хоста

Oak Ridge National Laboratory Tour - Upgrading to Tesla K20 GPUs

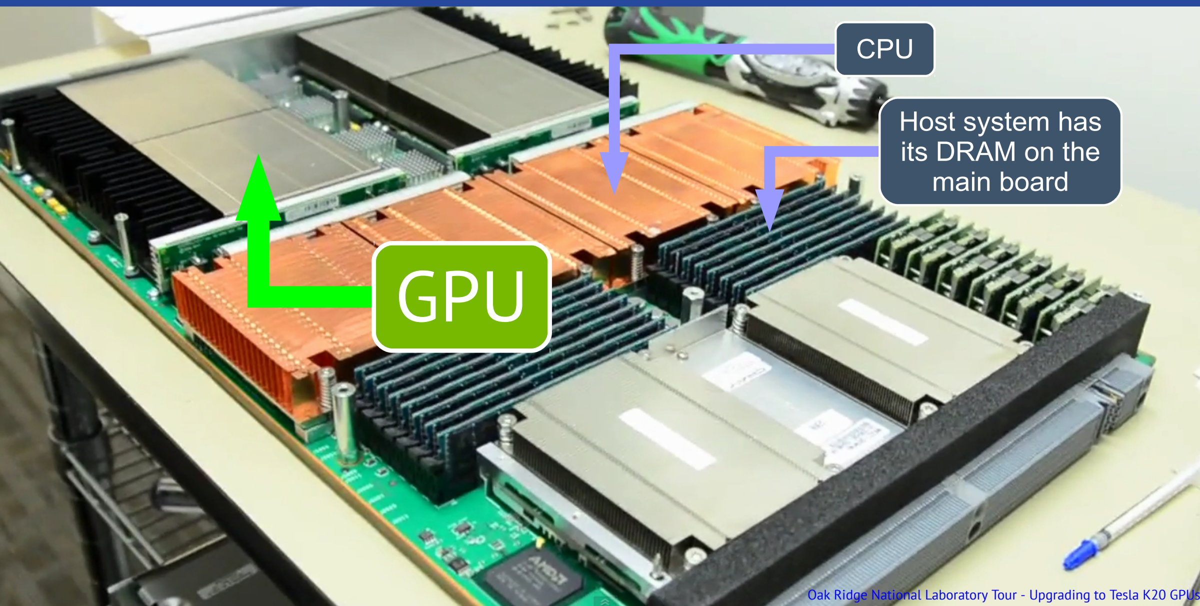
# Память GPU↔память хоста



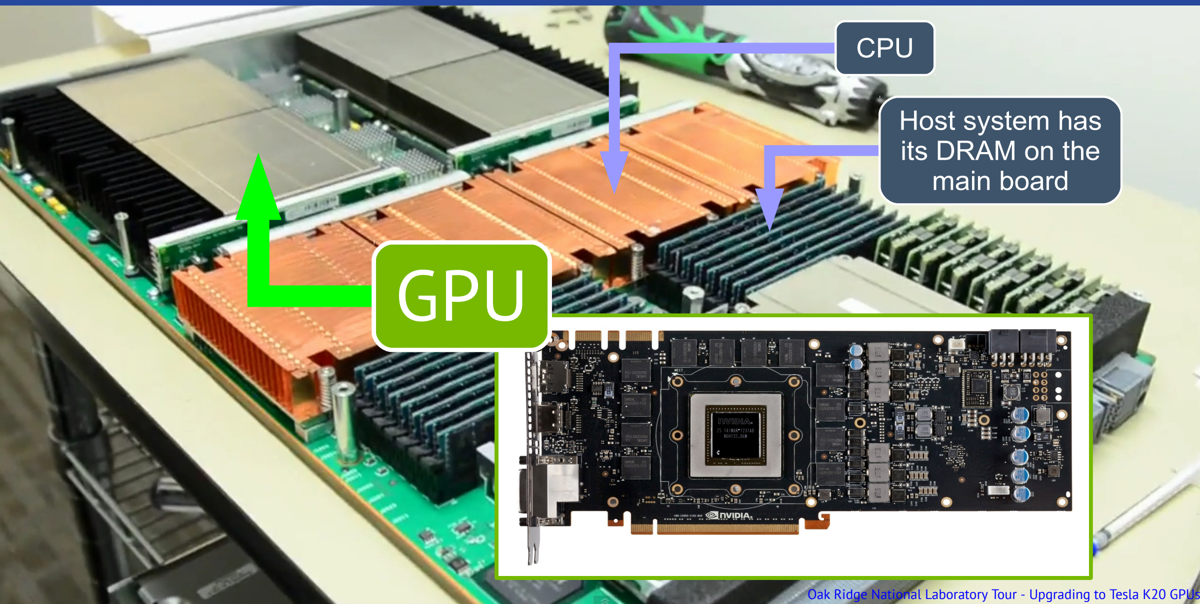
# Память GPU↔память хоста



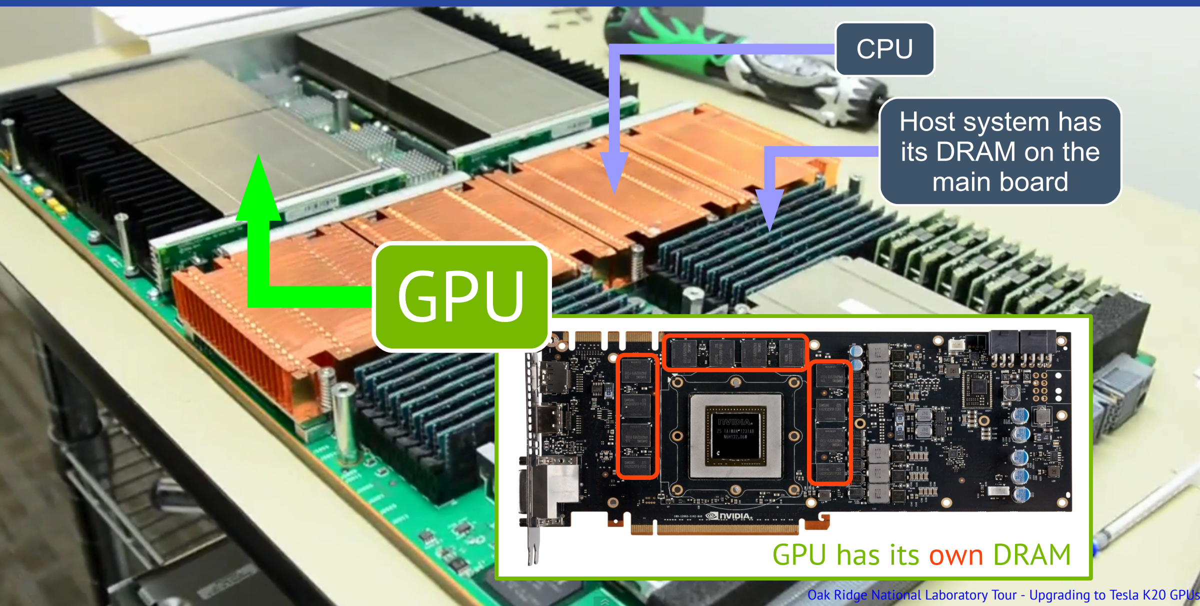
# Память GPU↔память хоста



# Память GPU↔память хоста



# Память GPU↔память хоста



# Память GPU↔память хоста

- 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)

# Память GPU↔память хоста

- 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)

# Память GPU↔память хоста

- 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)

# Память GPU↔память хоста

- 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)

# Память GPU↔память хоста

- 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)

# Память GPU↔память хоста

- 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)

# Память GPU↔память хоста

- 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)

# Память GPU↔память хоста

- 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)

# Память GPU↔память хоста

- 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)

# Память GPU↔память хоста

- 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)

# Память GPU↔память хоста

- 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)

# Память GPU↔память хоста

```
#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
```

# Память GPU↔память хоста

```
#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
```

# Память GPU↔память хоста

```
#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
```

# Память GPU↔память хоста

```
#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
```

# Память GPU↔память хоста

```
#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
```

# Итог

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`, `theadIdx.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 првоеряйте статус возвращаемых ошибок вызова CUDA (например, с помощью CUDA_ERR_CHECK)

# Итог

## 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`, `theadIdx.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 првоеряйте статус возвращаемых ошибок вызова CUDA (например, с помощью CUDA_ERR_CHECK)

# Итог

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`, `theadIdx.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 првоеряйте статус возвращаемых ошибок вызова CUDA (например, с помощью CUDA_ERR_CHECK)

# Итог

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`, `theadIdx.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 првоеряйте статус возвращаемых ошибок вызова CUDA (например, с помощью CUDA_ERR_CHECK)

# Итог

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`, `theadIdx.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 првоеряйте статус возвращаемых ошибок вызова CUDA (например, с помощью CUDA_ERR_CHECK)

# Итог

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`, `theadIdx.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 првоеряйте статус возвращаемых ошибок вызова CUDA (например, с помощью CUDA_ERR_CHECK)

# Итог

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`, `theadIdx.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 првоеряйте статус возвращаемых ошибок вызова CUDA (например, с помощью CUDA_ERR_CHECK)

# Итог

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`, `theadIdx.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 првоеряйте статус возвращаемых ошибок вызова CUDA (например, с помощью CUDA_ERR_CHECK)

# CUDA compute grid

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 compute grid

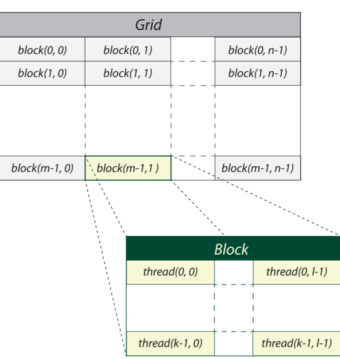
CUDA поддерживает 1-3 измерения:

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 compute grid

CUDA поддерживает 1-3 измерения:

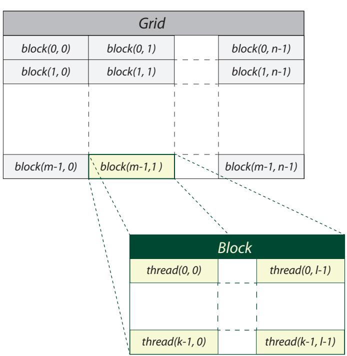
1. `gpu_kernel<<<4, 2>>>(...);`
   – 4 блока по 2 нити в каждом
2. `gpu_kernel<<<dim3(8, 4, 1), dim3(4, 2, 1)>>>(...);`
   – $8 \times 4$ блоков по $4 \times 2$ нити в каждом
3. `gpu_kernel<<<dim3(16, 8, 4), dim3(8, 4, 2)>>>(...);`
   – $16 \times 8 \times 4$ блоков по $8 \times 4 \times 2$ нити в каждом



# CUDA compute grid

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 compute grid (advanced)

CUDA compute grid поддерживает 1-3 измерения ⇒ увпрощает 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;
        }
```

```
__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 “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 × blockDim?

# CUDA compute grid (advanced)

CUDA compute grid поддерживает 1-3 измерения ⇒ увпрощает 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;
        }
```

```
__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 “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 × blockDim?

# CUDA compute grid (advanced)

CUDA compute grid поддерживает 1-3 измерения ⇒ увпрощает 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 × 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 $\Rightarrow$ увпрощает 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 × 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 ⇒ увпрощает 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` × `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 ⇒ увпрощает 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 (⇒ 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), rounddown(max_grid.x, 16)),
                  min(max(1, 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() );
```

# CUDA compute grid (advanced)

CUDA compute grid поддерживает 1-3 измерения ⇒ увпрощает 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 (⇒ 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), rounddown(max_grid.x, 16)),
                  min(max(1, 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() );
```

# CUDA compute grid (advanced)

CUDA compute grid поддерживает 1-3 измерения ⇒ увпрощает 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 (⇒ 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), rounddown(max_grid.x, 16)),
                  min(max(1, 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() );
```

# CUDA compute grid (advanced)

CUDA compute grid поддерживает 1-3 измерения ⇒ увпрощает 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 (⇒ 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), rounddown(max_grid.x, 16)),
                  min(max(1, 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
```

# Простейшие оптимизации CUDA

- 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):

  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
```

# Простейшие оптимизации CUDA

- 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):

  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
```

# Простейшие оптимизации CUDA

- 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):
  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
```

# Простейшие оптимизации CUDA

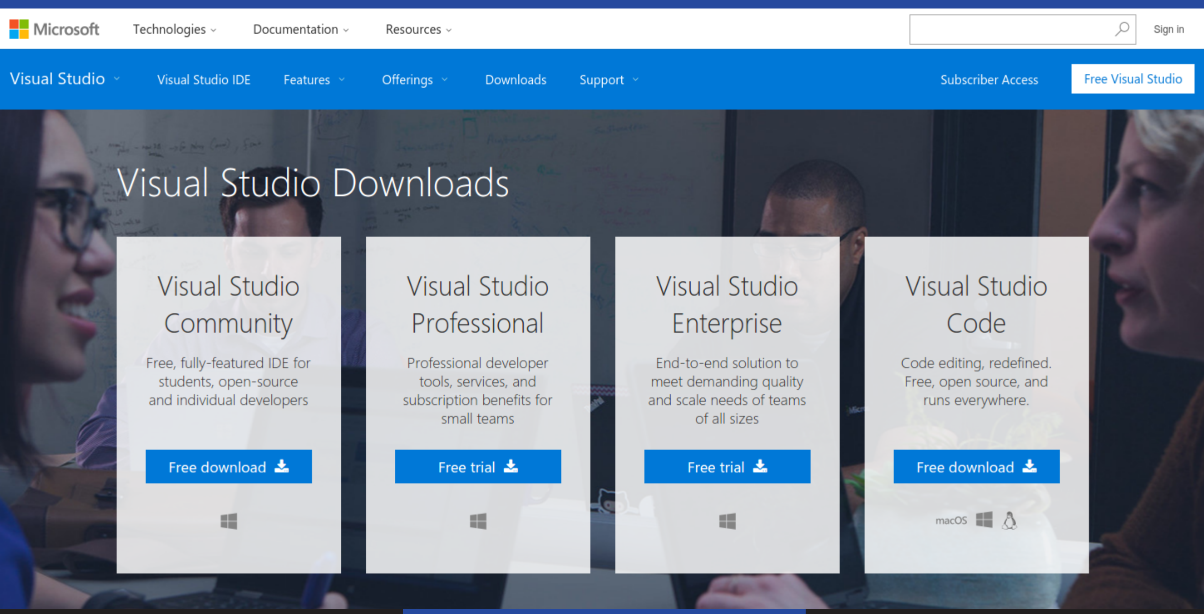
- Всегда старайтесь использовать выровненный доступ к памяти (coalescing)
- Размеры блоков `{128, 1, 1}` будут приблизительно оптимальными ждя большинства задач и современных GPU
- Компилятор может оптимизировать код для конкретной архитектуры GPU, определяемой Compute Capability (CC):
  add option `-arch=sm_35` для лучших результатов на Tesla K20:

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

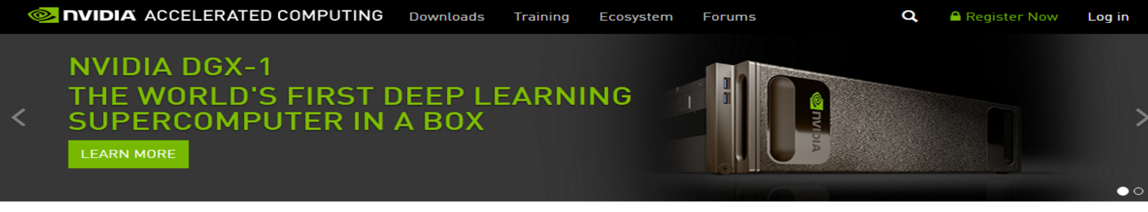
# Microsoft Visual Studio 2015 для CUDA

- 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



# Microsoft Visual Studio 2015 для CUDA




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**

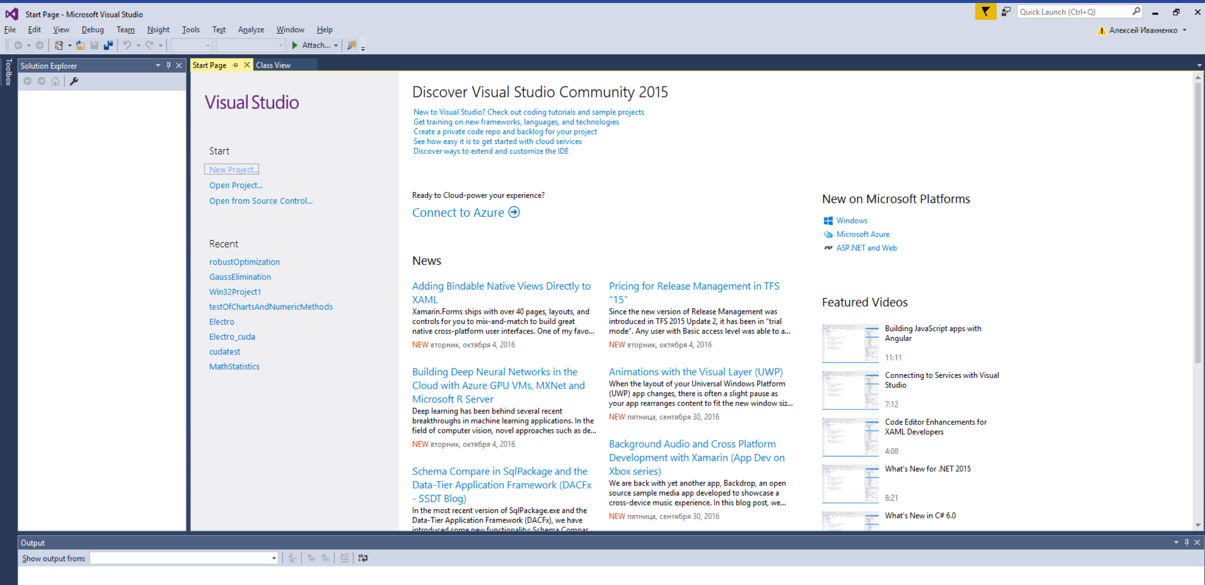
Click on the green buttons that describe your target platform. Only supported platforms will be shown.

Operating System Windows Linux Mac OSX

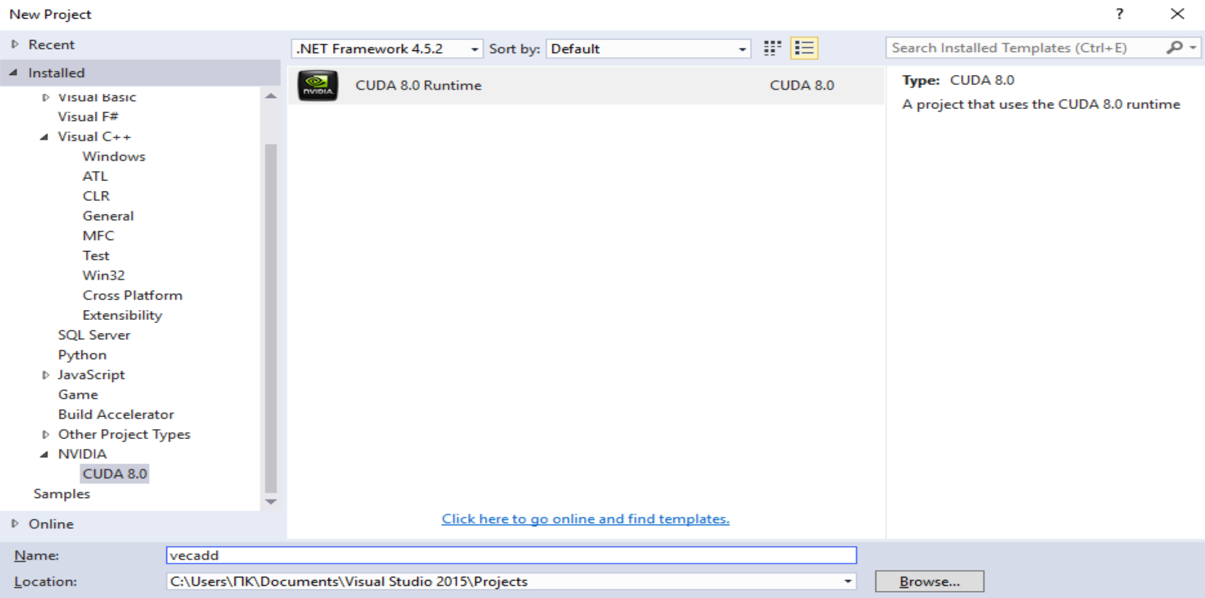
**Related Links**

CUDA Quick Start Guide
Release Notes
EULA
Online Documentation

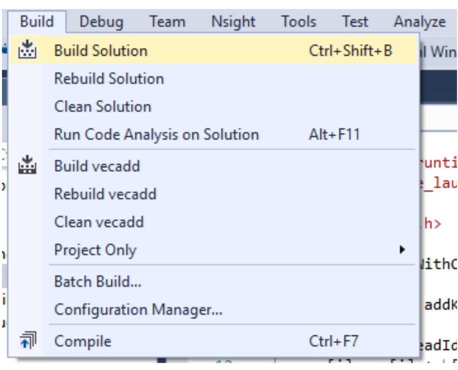
# Microsoft Visual Studio 2015 для CUDA



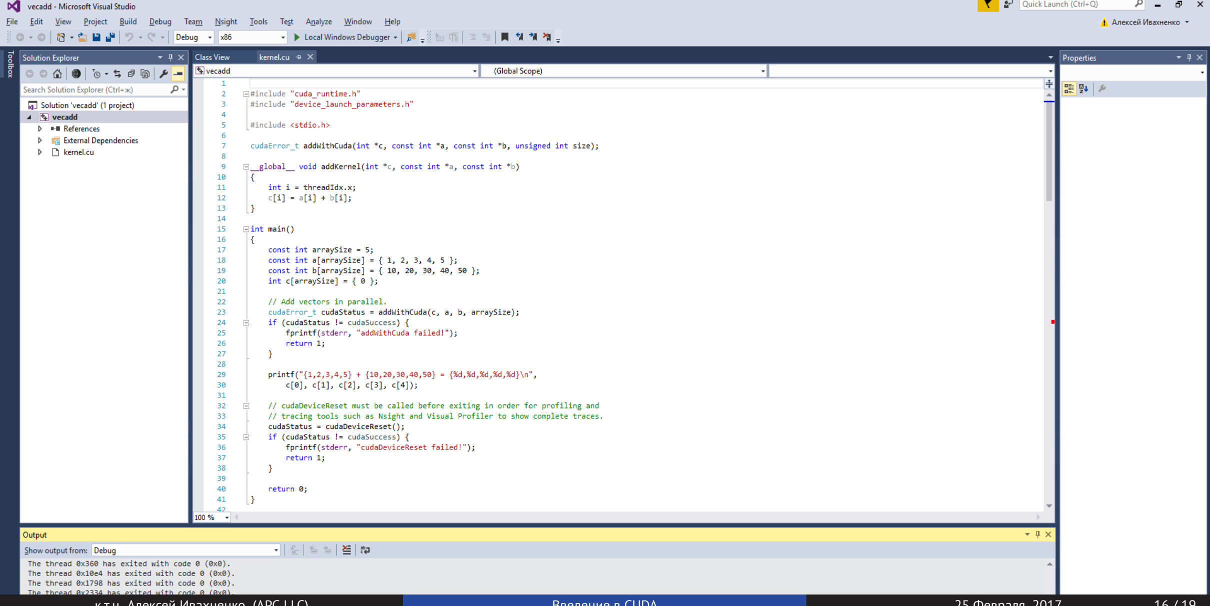
# Microsoft Visual Studio 2015 для CUDA



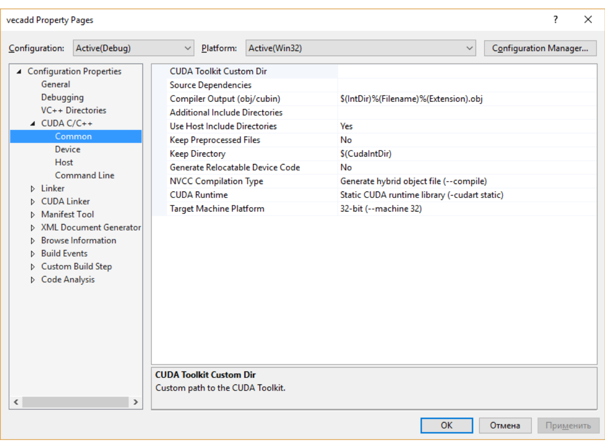
# Microsoft Visual Studio 2015 для CUDA



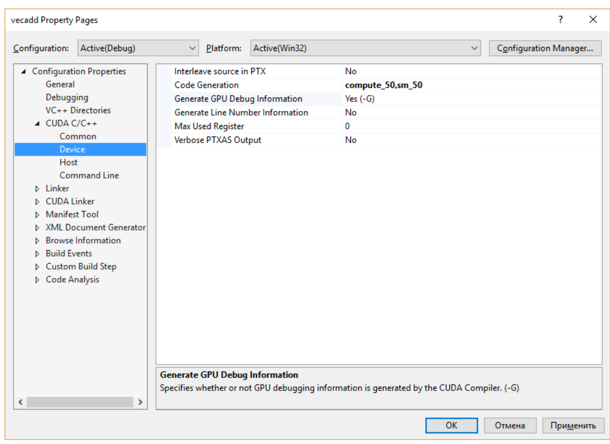
# Microsoft Visual Studio 2015 для CUDA



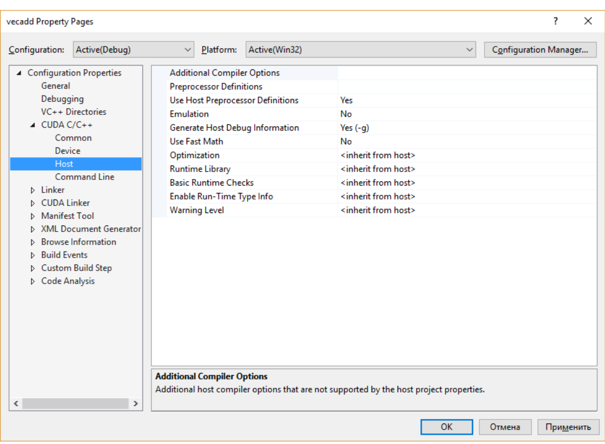
# Microsoft Visual Studio 2015 для CUDA



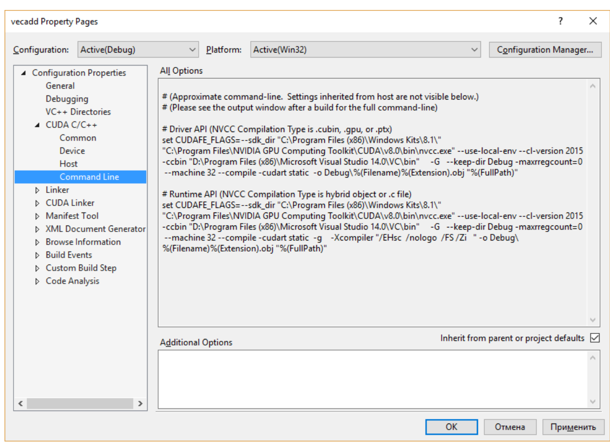
# Microsoft Visual Studio 2015 для CUDA



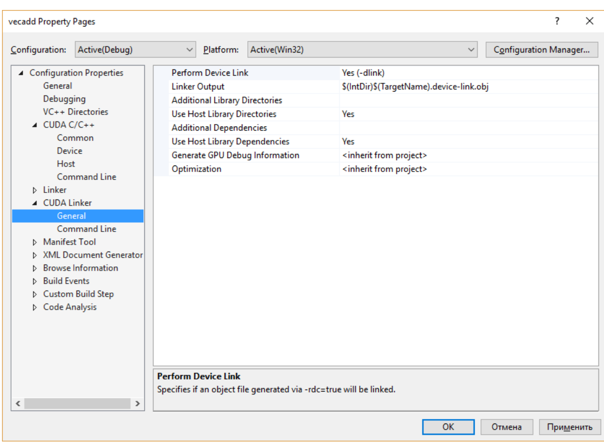
# Microsoft Visual Studio 2015 для CUDA



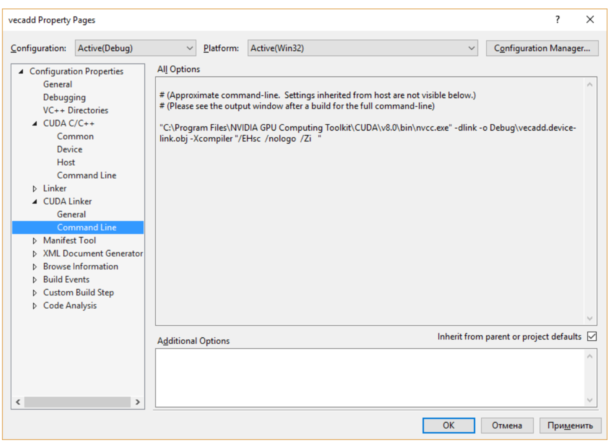
# Microsoft Visual Studio 2015 для CUDA



# Microsoft Visual Studio 2015 для CUDA

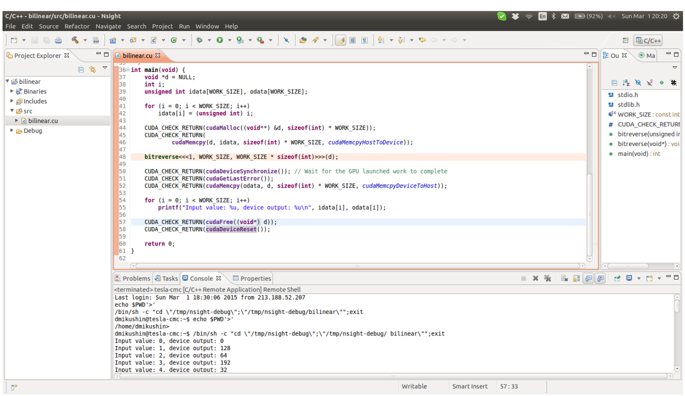


# Microsoft Visual Studio 2015 для CUDA



# Eclipse Nsight Edition для CUDA на Linux/Mac

- 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



# Eclipse Nsight Edition для CUDA на Linux/Mac

- 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 локльная 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

# Заключение

- 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
  - Потоки и асинхронная передача данных
  - Отладка и профилирование

# Заключение

- 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
  - Потоки и асинхронная передача данных
  - Отладка и профилирование

# Заключение

- 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
  - Потоки и асинхронная передача данных
  - Отладка и профилирование

# Заключение

- 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
  - Потоки и асинхронная передача данных
  - Отладка и профилирование

# Заключение

- 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
  - Потоки и асинхронная передача данных
  - Отладка и профилирование

# Заключение

- 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
  - Потоки и асинхронная передача данных
  - Отладка и профилирование

# Заключение

- 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
  - Потоки и асинхронная передача данных
  - Отладка и профилирование

# Заключение

- 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
  - Потоки и асинхронная передача данных
  - Отладка и профилирование