Applied Parallel Computing LLC
http://parallel-computing.pro

USI

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

Introduction to CUDA

Dmitry Mikushin

July 8, 2014

What is CUDA

- **CUDA** = **C**ompute-**U**nified **D**evice **A**rchitecutre
  - A set of hardware architecture and threading runtime concepts to work with massive parallelism
- **CUDA C** (or actually **C++**)
  - One of the *programming languages*, that were extended with specific constructs to provide CUDA support
  - There are other CUDA-aware languages: CUDA Fortran, PyCUDA, ...

What is CUDA

- **CUDA** = **C**ompute-**U**nified **D**evice **A**rchitecutre
  - A set of hardware architecture and threading runtime concepts to work with massive parallelism
- **CUDA C** (or actually **C++**)
  - One of the *programming languages*, that were extended with specific constructs to provide CUDA support
  - There are other CUDA-aware languages: CUDA Fortran, PyCUDA, ...

What is CUDA

- **CUDA** = **C**ompute-**U**nified **D**evice **A**rchitecutre
  - A set of hardware architecture and threading runtime concepts to work with massive parallelism
- **CUDA C** (or actually **C++**)
  - One of the *programming languages*, that were extended with specific constructs to provide CUDA support
  - There are other CUDA-aware languages: CUDA Fortran, PyCUDA, ...

CUDA concepts

- Hardware:
  - Massive parallelism:
    thousands of compute cores
- Threading runtime:
  - Thousands of parallel threads grouped into *blocks* and *warps*:
    - Threads of each block can share some fast (≈registers speed) memory called *shared memory*
    - Warps are subgroups of threads in block that are executed *synchronously*
  - **SIMT**: **S**ingle **I**nstruction – **M**ultiple **T**hreads
    - All threads of the same warp handles the same instruction on different data items
      (whereas in traditional SIM**D** – the same thread handles multiple data items)
    - Warps execution may be not simultaneous/synchronous

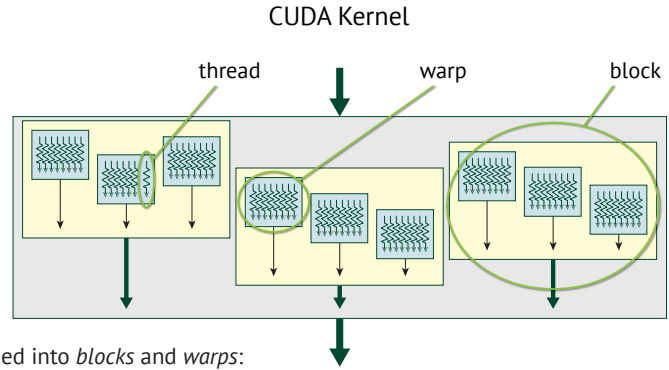
CUDA concepts

- Hardware:
  - Massive parallelism: thousands of compute cores
- Threading runtime:
  - Thousands of parallel threads grouped into *blocks* and *warps*:
    - Threads of each block can share some fast (≈registers speed) memory called *shared memory*
    - Warps are subgroups of threads in block that are executed *synchronously*
  - **SIMT**: **S**ingle **I**nstruction – **M**ultiple **T**hreads
    - All threads of the same warp handles the same instruction on different data items (whereas in traditional SIM**D** – the same thread handles multiple data items)
    - Warps execution may be not simultaneous/synchronous

CUDA concepts

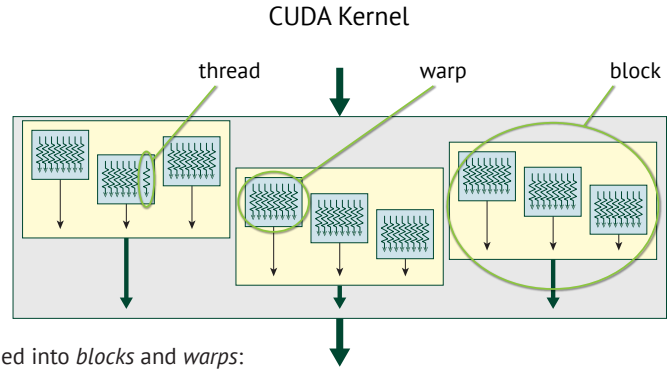
- Hardware:
  - Massive parallelism:
    thousands of compute cores
- Threading runtime:
  - Thousands of parallel threads grouped into *blocks* and *warps*:
    - Threads of each block can share some fast (≈registers speed) memory called *shared memory*
    - Warps are subgroups of threads in block that are executed *synchronously*
  - **SIMT**: **S**ingle **I**nstruction – **M**ultiple **T**hreads
    - All threads of the same warp handles the same instruction on different data items
      (whereas in traditional SIM**D** – the same thread handles multiple data items)
    - Warps execution may be not simultaneous/synchronous

CUDA concepts



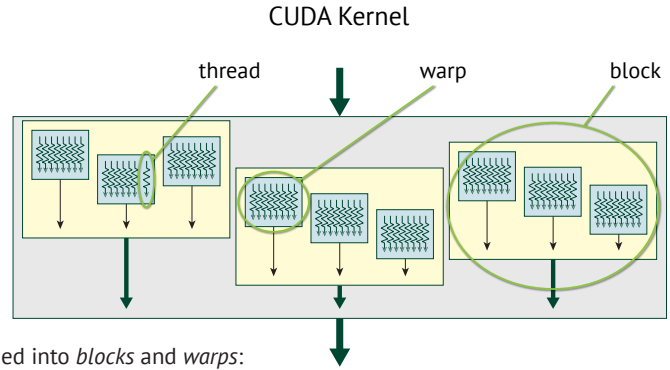
- Hardware:
  - Massive parallelism:
    thousands of compute cores
- Threading runtime:
  - Thousands of parallel threads grouped into *blocks* and *warps*:
    - Threads of each block can share some fast (≈registers speed) memory called *shared memory*
    - Warps are subgroups of threads in block that are executed *synchronously*
  - **SIMT**: **S**ingle **I**nstruction – **M**ultiple **T**hreads
    - All threads of the same warp handles the same instruction on different data items
      (whereas in traditional SIM**D** – the same thread handles multiple data items)
    - Warps execution may be not simultaneous/synchronous

CUDA concepts



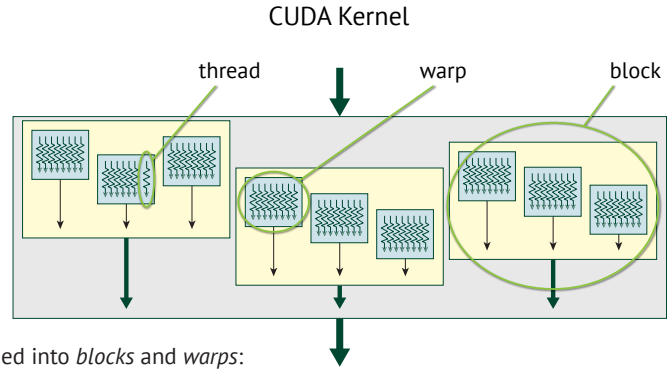
- Hardware:
  - Massive parallelism:
    thousands of compute cores
- Threading runtime:
  - Thousands of parallel threads grouped into *blocks* and *warps*:
    - Threads of each block can share some fast (≈registers speed) memory called *shared memory*
    - Warps are subgroups of threads in block that are executed *synchronously*
  - **SIMT**: **S**ingle **I**nstruction – **M**ultiple **T**hreads
    - All threads of the same warp handles the same instruction on different data items
      (whereas in traditional SIM**D** – the same thread handles multiple data items)
    - Warps execution may be not simultaneous/synchronous

CUDA concepts



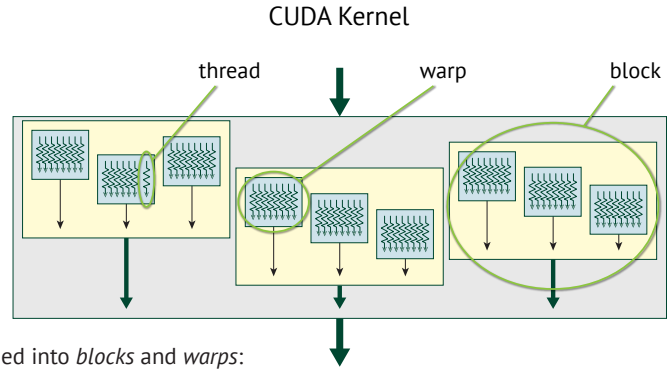
- Hardware:
  - Massive parallelism:
    thousands of compute cores
- Threading runtime:
  - Thousands of parallel threads grouped into *blocks* and *warps*:
    - Threads of each block can share some fast (≈registers speed) memory called *shared memory*
    - Warps are subgroups of threads in block that are executed *synchronously*
  - **SIMT**: **S**ingle **I**nstruction – **M**ultiple **T**hreads
    - All threads of the same warp handles the same instruction on different data items
      (whereas in traditional SIM**D** – the same thread handles multiple data items)
    - Warps execution may be not simultaneous/synchronous

CUDA concepts



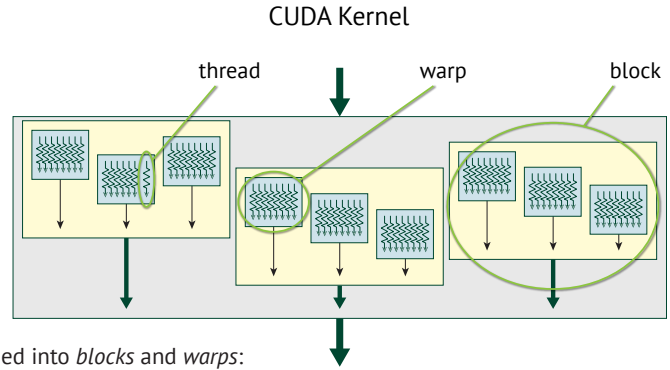
- Hardware:
  - Massive parallelism:
    thousands of compute cores
- Threading runtime:
  - Thousands of parallel threads grouped into *blocks* and *warps*:
    - Threads of each block can share some fast (≈registers speed) memory called *shared memory*
    - Warps are subgroups of threads in block that are executed *synchronously*
  - **SIMT**: **S**ingle **I**nstruction – **M**ultiple **T**hreads
    - All threads of the same warp handles the same instruction on different data items
      (whereas in traditional SIM**D** – the same thread handles multiple data items)
    - Warps execution may be not simultaneous/synchronous

CUDA concepts



- Hardware:
  - Massive parallelism:
    thousands of compute cores
- Threading runtime:
  - Thousands of parallel threads grouped into *blocks* and *warps*:
    - Threads of each block can share some fast (≈registers speed) memory called *shared memory*
    - Warps are subgroups of threads in block that are executed *synchronously*
  - **SIMT**: **S**ingle **I**nstruction – **M**ultiple **T**hreads
    - All threads of the same warp handles the same instruction on different data items
      (whereas in traditional SIM**D** – the same thread handles multiple data items)
    - Warps execution may be not simultaneous/synchronous

CUDA concepts



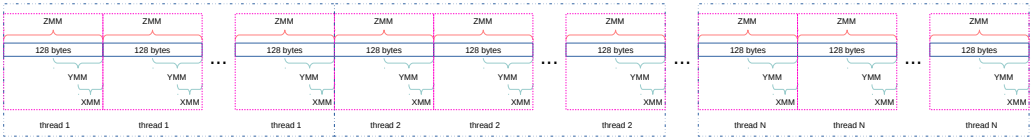
- Hardware:
  - Massive parallelism:
    thousands of compute cores
- Threading runtime:
  - Thousands of parallel threads grouped into *blocks* and *warps*:
    - Threads of each block can share some fast (≈registers speed) memory called *shared memory*
    - Warps are subgroups of threads in block that are executed *synchronously*
  - **SIMT**: **S**ingle **I**nstruction – **M**ultiple **T**hreads
    - All threads of the same warp handles the same instruction on different data items
      (whereas in traditional SIM**D** – the same thread handles multiple data items)
    - Warps execution may be not simultaneous/synchronous

SIM**T** vs SIM**D**

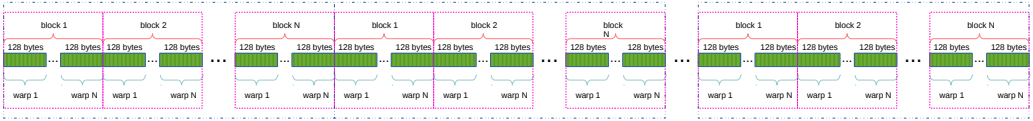
Consider how computations will be distributed between threads for the following loop (N $\gg$ threads count):

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

- SIM**D**-scheme for CPU with AVX-512 support (512-bit vector registers – Xeon Phi and 2015' CPU):



- SIM**T**-scheme for CUDA/GPU with 32 threads per warp:

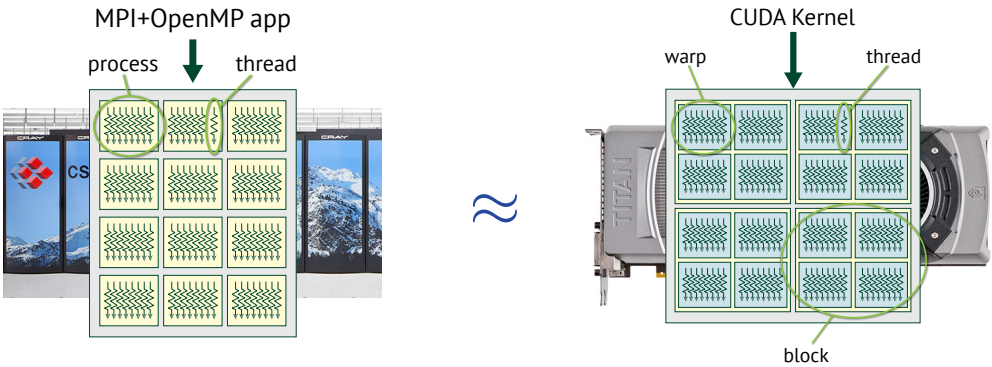


▯ - thread (lightweight GPU-thread)

$\Rightarrow$ SIMT allows CUDA GPU to perform "vector" computations on scalar cores, which is much easier, than getting compiler to autovectorize on CPU and much easier than to vectorize the code maually.

CUDA in a nutshell

Having an idea of MPI and OpenMP, it's very easy to understand CUDA:



⇒ CUDA GPU is ≈ MPI+OpenMP cluster in microscale, packed into 11-inch box!

CUDA 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 form 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 form b #%d of %d, t #%d of %d!\n",
           block_idx, grid_dim,
           threadIdx.x, blockDim.x);

}
```

CUDA 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

```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 form 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

```c
__global__ void gpu_kernel() {


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



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



}
```

CUDA 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 form 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 form b #%d of %d, t #%d of %d!\n",
            block_idx, grid_dim,
            threadIdx.x, blockDim.x);



}
```

CUDA 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 form 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 form b #%d of %d, t #%d of %d!\n",
            block_idx, grid_dim,
            threadIdx.x, blockDim.x);



}
```

CUDA 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

```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 form 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

```c
__global__ void gpu_kernel() {


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



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



}
```

CUDA 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 form 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 form b #%d of %d, t #%d of %d!\n",
            block_idx, grid_dim,
            threadIdx.x, blockDim.x);



}
```

CUDA 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 form 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 form b #%d of %d, t #%d of %d!\n",
            block_idx, grid_dim,
            threadIdx.x, blockDim.x);



}
```

CUDA 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 form 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 form b #%d of %d, t #%d of %d!\n",
            block_idx, grid_dim,
            threadIdx.x, blockDim.x);



}
```

CUDA 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

```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 form 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

```c
__global__ void gpu_kernel() {


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



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



}
```

CUDA 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

```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 form 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

```c
__global__ void gpu_kernel() {


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



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



}
```

CUDA 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

```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 form 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

```c
__global__ void gpu_kernel() {


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



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



}
```

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 form 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 form b #0 of 2, t #0 of 2!
Hello form b #0 of 2, t #1 of 2!
Hello form b #1 of 2, t #0 of 2!
Hello form 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 form 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 form b #0 of 2, t #0 of 2!
Hello form b #0 of 2, t #1 of 2!
Hello form b #1 of 2, t #0 of 2!
Hello form 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 form 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 form b #0 of 2, t #0 of 2!
Hello form b #0 of 2, t #1 of 2!
Hello form b #1 of 2, t #0 of 2!
Hello form 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 form 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 form b #0 of 2, t #0 of 2!
Hello form b #0 of 2, t #1 of 2!
Hello form b #1 of 2, t #0 of 2!
Hello form 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 form 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 form b #0 of 2, t #0 of 2!
Hello form b #0 of 2, t #1 of 2!
Hello form b #1 of 2, t #0 of 2!
Hello form 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 form b #%d of %d, t #%d of %d!\n",
            block_idx, grid_dim,
            threadIdx.x, blockDim.x);



}
```

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 form 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 form b #0 of 2, t #0 of 2!
Hello form b #0 of 2, t #1 of 2!
Hello form b #1 of 2, t #0 of 2!
Hello form 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 form 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 form b #0 of 2, t #0 of 2!
Hello form b #0 of 2, t #1 of 2!
Hello form b #1 of 2, t #0 of 2!
Hello form 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 form 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 form b #0 of 2, t #0 of 2!
Hello form b #0 of 2, t #1 of 2!
Hello form b #1 of 2, t #0 of 2!
Hello form 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 form 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 form b #0 of 2, t #0 of 2!
Hello form b #0 of 2, t #1 of 2!
Hello form b #1 of 2, t #0 of 2!
Hello form 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 form 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 form b #0 of 2, t #0 of 2!
Hello form b #0 of 2, t #1 of 2!
Hello form b #1 of 2, t #0 of 2!
Hello form 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 form 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 form b #0 of 2, t #0 of 2!
Hello form b #0 of 2, t #1 of 2!
Hello form b #1 of 2, t #0 of 2!
Hello form 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 form 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 form b #0 of 2, t #0 of 2!
Hello form b #0 of 2, t #1 of 2!
Hello form b #1 of 2, t #0 of 2!
Hello form 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 form 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 form b #0 of 2, t #0 of 2!
Hello form b #0 of 2, t #1 of 2!
Hello form b #1 of 2, t #0 of 2!
Hello form 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 form 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 form b #0 of 2, t #0 of 2!
Hello form b #0 of 2, t #1 of 2!
Hello form b #1 of 2, t #0 of 2!
Hello form 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 form 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 form b #0 of 2, t #0 of 2!
Hello form b #0 of 2, t #1 of 2!
Hello form b #1 of 2, t #0 of 2!
Hello form 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 form 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 form b #0 of 2, t #0 of 2!
Hello form b #0 of 2, t #1 of 2!
Hello form b #1 of 2, t #0 of 2!
Hello form 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 form 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 form b #0 of 2, t #0 of 2!
Hello form b #0 of 2, t #1 of 2!
Hello form b #1 of 2, t #0 of 2!
Hello form 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 form 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 form b #0 of 2, t #0 of 2!
Hello form b #0 of 2, t #1 of 2!
Hello form b #1 of 2, t #0 of 2!
Hello form 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 form b #%d of %d, t #%d of %d!\n",
            block_idx, grid_dim,
            threadIdx.x, blockDim.x);



}
```

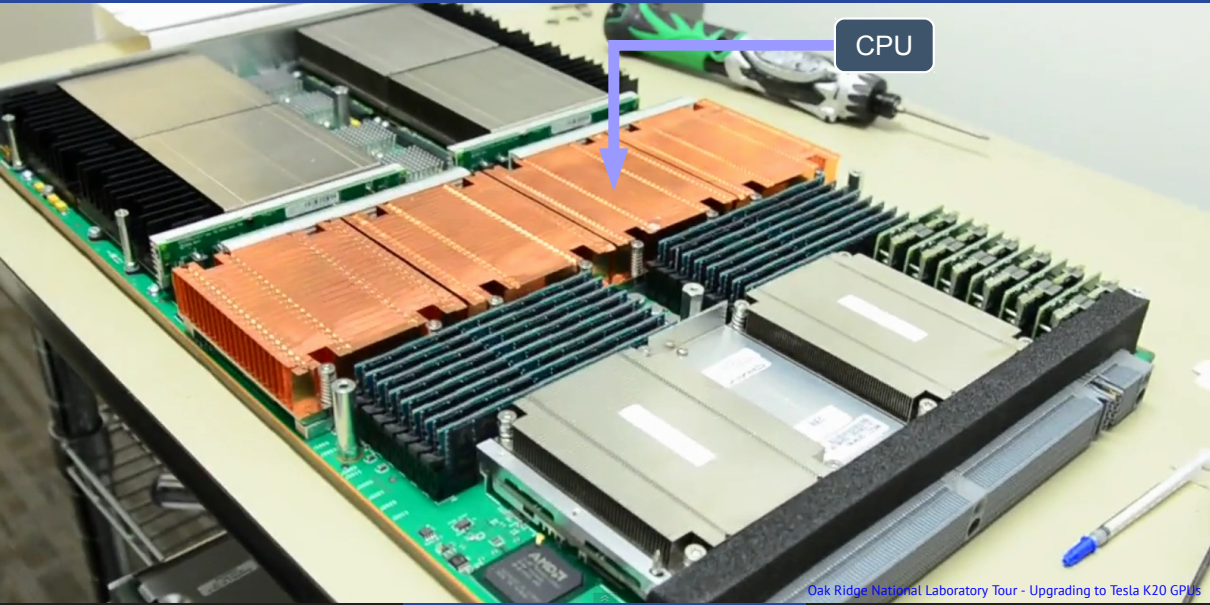
```c
int main() {
    gpu_kernel<<<2, 2>>>();
    CUDA_ERR_CHECK( cudaGetLastError() );
    CUDA_ERR_CHECK( cudaDeviceSynchronize() );
    return 0;
}
```

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

GPU↔host memory

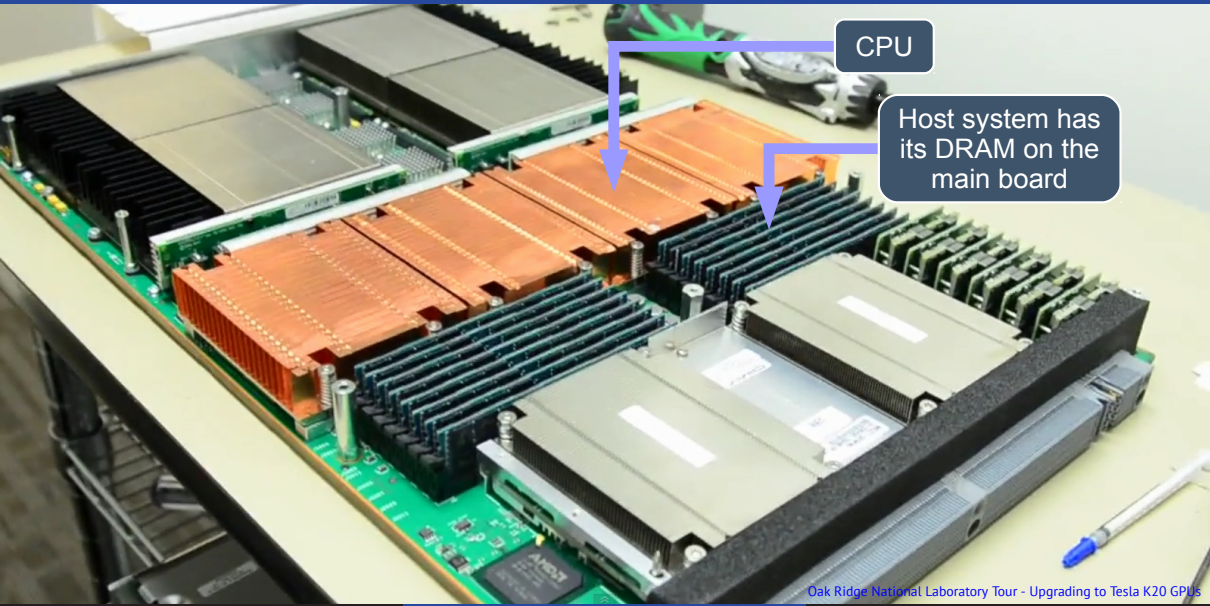
Oak Ridge National Laboratory Tour - Upgrading to Tesla K20 GPUs

GPU↔host memory



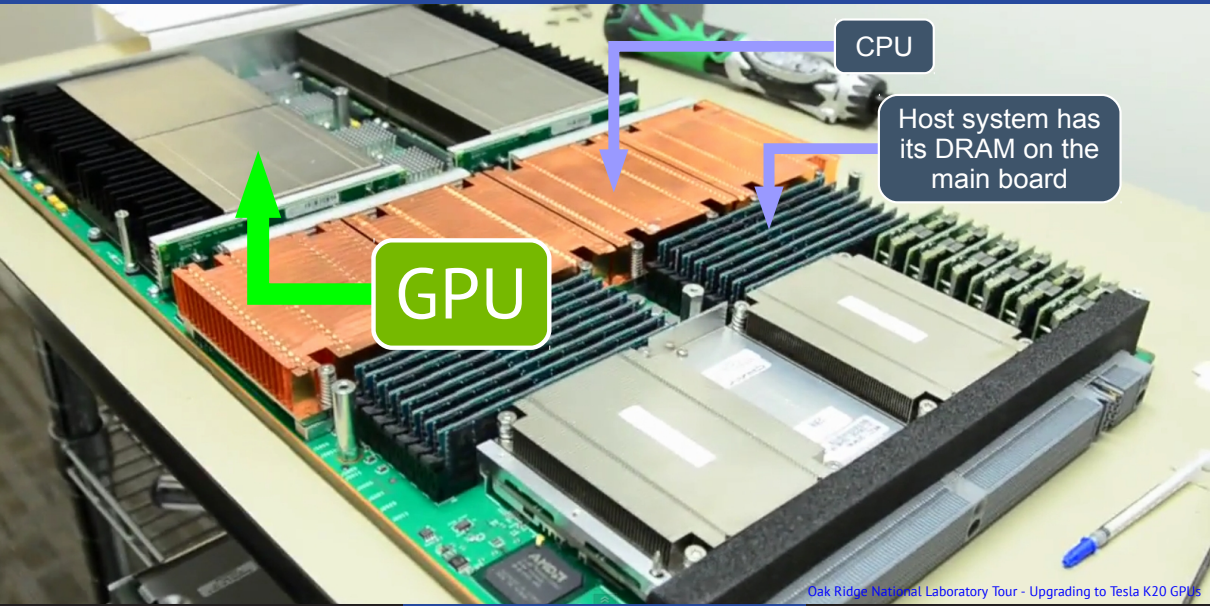
Oak Ridge National Laboratory Tour - Upgrading to Tesla K20 GPUs

GPU↔host memory



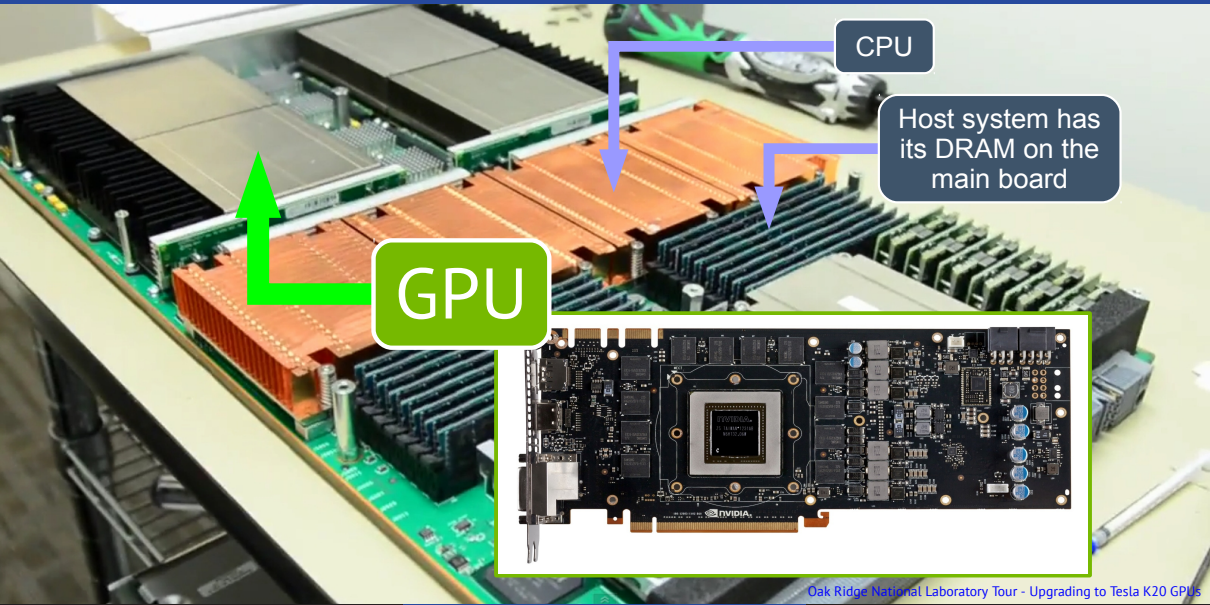
Oak Ridge National Laboratory Tour - Upgrading to Tesla K20 GPUs

GPU↔host memory



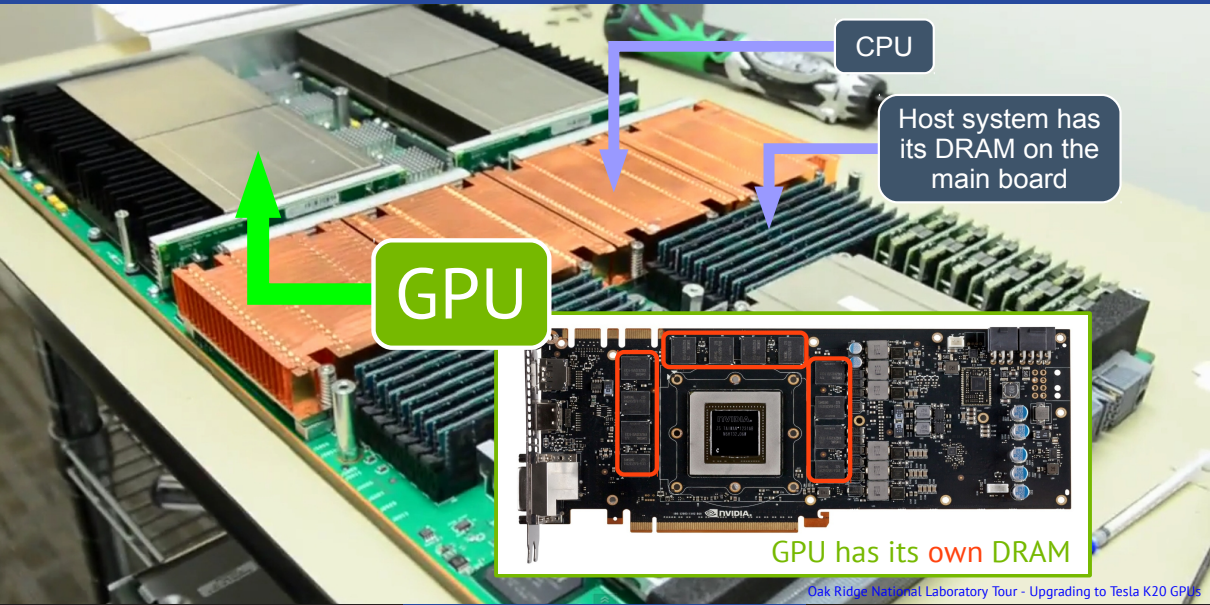
Oak Ridge National Laboratory Tour - Upgrading to Tesla K20 GPUs

GPU↔host memory



Oak Ridge National Laboratory Tour - Upgrading to Tesla K20 GPUs

GPU↔host memory



GPU↔host memory

- Best GPU efficiency is usually achieved, when data being handled is located in GPU DRAM
- Currently CUDA developer has to allocate GPU DRAM explicitly and copy data between host and 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;
}
```

- New NVLink technology will allow fast direct access from GPU to host memory in 2016-2017

GPU↔host memory

- Best GPU efficiency is usually achieved, when data being handled is located in GPU DRAM
- Currently CUDA developer has to allocate GPU DRAM explicitly and copy data between host and 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 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;
}
```

- New NVLink technology will allow fast direct access from GPU to host memory in 2016-2017

GPU↔host memory

- Best GPU efficiency is usually achieved, when data being handled is located in GPU DRAM
- Currently CUDA developer has to allocate GPU DRAM explicitly and copy data between host and 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 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;
}
```

- New NVLink technology will allow fast direct access from GPU to host memory in 2016-2017

GPU↔host memory

- Best GPU efficiency is usually achieved, when data being handled is located in GPU DRAM
- Currently CUDA developer has to allocate GPU DRAM explicitly and copy data between host and 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 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;
}
```

- New NVLink technology will allow fast direct access from GPU to host memory in 2016-2017

GPU↔host memory

- Best GPU efficiency is usually achieved, when data being handled is located in GPU DRAM
- Currently CUDA developer has to allocate GPU DRAM explicitly and copy data between host and 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 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;
}
```

- New NVLink technology will allow fast direct access from GPU to host memory in 2016-2017

GPU↔host memory

- Best GPU efficiency is usually achieved, when data being handled is located in GPU DRAM
- Currently CUDA developer has to allocate GPU DRAM explicitly and copy data between host and 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 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;
}
```

- New NVLink technology will allow fast direct access from GPU to host memory in 2016-2017

GPU↔host memory

- Best GPU efficiency is usually achieved, when data being handled is located in GPU DRAM
- Currently CUDA developer has to allocate GPU DRAM explicitly and copy data between host and 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 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;
}
```

- New NVLink technology will allow fast direct access from GPU to host memory in 2016-2017

GPU↔host memory

- Best GPU efficiency is usually achieved, when data being handled is located in GPU DRAM
- Currently CUDA developer has to allocate GPU DRAM explicitly and copy data between host and 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 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;
}
```

- New NVLink technology will allow fast direct access from GPU to host memory in 2016-2017

GPU↔host memory

- Best GPU efficiency is usually achieved, when data being handled is located in GPU DRAM
- Currently CUDA developer has to allocate GPU DRAM explicitly and copy data between host and 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 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;
}
```

- New NVLink technology will allow fast direct access from GPU to host memory in 2016-2017

GPU↔host memory

- Best GPU efficiency is usually achieved, when data being handled is located in GPU DRAM
- Currently CUDA developer has to allocate GPU DRAM explicitly and copy data between host and 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 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;
}
```

- New NVLink technology will allow fast direct access from GPU to host memory in 2016-2017

GPU↔host memory

- Best GPU efficiency is usually achieved, when data being handled is located in GPU DRAM
- Currently CUDA developer has to allocate GPU DRAM explicitly and copy data between host and 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 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;
}
```

- New NVLink technology will allow fast direct access from GPU to host memory in 2016-2017

GPU↔host memory

```
#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
```

GPU↔host memory

```
#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
```

GPU↔host memory

```
#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
```

GPU↔host memory

```
#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
```

GPU↔host memory

```
#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
```

Summary

We just wrote out first CUDA program!

- CUDA uses 2-level compute grid topology, similar to MPI+OpenMP
- CUDA topology, however has flexibility to be 1D, 2D or 3D (e.g. `blockIdx.y`, `theadIdx.z`)
- CUDA kernel can't be a standalone program and needs to be called from "host" application
- CUDA kernel launch spawns all blocks and threads, no OpenMP-like pragmas are needed in kernel code
- CUDA kernel launch is asynchronous, and could be synchronized explicitly (`cudaDeviceSynchronize`) or implicitly (by using kernel parameter in e.g. `cudaMemcpy`)
- GPU has on-board DRAM, which is independent from host's DRAM; data between these memories has to be transfered explicitly
- Always check the error status of every CUDA call (e.g. with CUDA_ERR_CHECK macro)

Summary

We just wrote out first CUDA program!

- CUDA uses 2-level compute grid topology, similar to MPI+OpenMP
- CUDA topology, however has flexibility to be 1D, 2D or 3D (e.g. `blockIdx.y`, `theadIdx.z`)
- CUDA kernel can't be a standalone program and needs to be called from "host" application
- CUDA kernel launch spawns all blocks and threads, no OpenMP-like pragmas are needed in kernel code
- CUDA kernel launch is asynchronous, and could be synchronized explicitly (`cudaDeviceSynchronize`) or implicitly (by using kernel parameter in e.g. `cudaMemcpy`)
- GPU has on-board DRAM, which is independent from host's DRAM; data between these memories has to be transfered explicitly
- Always check the error status of every CUDA call (e.g. with CUDA_ERR_CHECK macro)

Summary

We just wrote out first CUDA program!

- CUDA uses 2-level compute grid topology, similar to MPI+OpenMP
- CUDA topology, however has flexibility to be 1D, 2D or 3D (e.g. `blockIdx.y`, `theadIdx.z`)
- CUDA kernel can't be a standalone program and needs to be called from "host" application
- CUDA kernel launch spawns all blocks and threads, no OpenMP-like pragmas are needed in kernel code
- CUDA kernel launch is asynchronous, and could be synchronized explicitly (`cudaDeviceSynchronize`) or implicitly (by using kernel parameter in e.g. `cudaMemcpy`)
- GPU has on-board DRAM, which is independent from host's DRAM; data between these memories has to be transfered explicitly
- Always check the error status of every CUDA call (e.g. with CUDA_ERR_CHECK macro)

Summary

We just wrote out first CUDA program!

- CUDA uses 2-level compute grid topology, similar to MPI+OpenMP
- CUDA topology, however has flexibility to be 1D, 2D or 3D (e.g. `blockIdx.y`, `theadIdx.z`)
- CUDA kernel can't be a standalone program and needs to be called from "host" application
- CUDA kernel launch spawns all blocks and threads, no OpenMP-like pragmas are needed in kernel code
- CUDA kernel launch is asynchronous, and could be synchronized explicitly (`cudaDeviceSynchronize`) or implicitly (by using kernel parameter in e.g. `cudaMemcpy`)
- GPU has on-board DRAM, which is independent from host's DRAM; data between these memories has to be transfered explicitly
- Always check the error status of every CUDA call (e.g. with CUDA_ERR_CHECK macro)

Summary

We just wrote out first CUDA program!

- CUDA uses 2-level compute grid topology, similar to MPI+OpenMP
- CUDA topology, however has flexibility to be 1D, 2D or 3D (e.g. `blockIdx.y`, `theadIdx.z`)
- CUDA kernel can't be a standalone program and needs to be called from "host" application
- CUDA kernel launch spawns all blocks and threads, no OpenMP-like pragmas are needed in kernel code
- CUDA kernel launch is asynchronous, and could be synchronized explicitly (`cudaDeviceSynchronize`) or implicitly (by using kernel parameter in e.g. `cudaMemcpy`)
- GPU has on-board DRAM, which is independent from host's DRAM; data between these memories has to be transfered explicitly
- Always check the error status of every CUDA call (e.g. with CUDA_ERR_CHECK macro)

Summary

We just wrote out first CUDA program!

- CUDA uses 2-level compute grid topology, similar to MPI+OpenMP
- CUDA topology, however has flexibility to be 1D, 2D or 3D (e.g. `blockIdx.y`, `theadIdx.z`)
- CUDA kernel can't be a standalone program and needs to be called from "host" application
- CUDA kernel launch spawns all blocks and threads, no OpenMP-like pragmas are needed in kernel code
- CUDA kernel launch is asynchronous, and could be synchronized explicitly (`cudaDeviceSynchronize`) or implicitly (by using kernel parameter in e.g. `cudaMemcpy`)
- GPU has on-board DRAM, which is independent from host's DRAM; data between these memories has to be transfered explicitly
- Always check the error status of every CUDA call (e.g. with CUDA_ERR_CHECK macro)

Summary

We just wrote out first CUDA program!

- CUDA uses 2-level compute grid topology, similar to MPI+OpenMP
- CUDA topology, however has flexibility to be 1D, 2D or 3D (e.g. `blockIdx.y`, `theadIdx.z`)
- CUDA kernel can't be a standalone program and needs to be called from "host" application
- CUDA kernel launch spawns all blocks and threads, no OpenMP-like pragmas are needed in kernel code
- CUDA kernel launch is asynchronous, and could be synchronized explicitly (`cudaDeviceSynchronize`) or implicitly (by using kernel parameter in e.g. `cudaMemcpy`)
- GPU has on-board DRAM, which is independent from host's DRAM; data between these memories has to be transfered explicitly
- Always check the error status of every CUDA call (e.g. with CUDA_ERR_CHECK macro)

Summary

We just wrote out first CUDA program!

- CUDA uses 2-level compute grid topology, similar to MPI+OpenMP
- CUDA topology, however has flexibility to be 1D, 2D or 3D (e.g. `blockIdx.y`, `theadIdx.z`)
- CUDA kernel can't be a standalone program and needs to be called from "host" application
- CUDA kernel launch spawns all blocks and threads, no OpenMP-like pragmas are needed in kernel code
- CUDA kernel launch is asynchronous, and could be synchronized explicitly (`cudaDeviceSynchronize`) or implicitly (by using kernel parameter in e.g. `cudaMemcpy`)
- GPU has on-board DRAM, which is independent from host's DRAM; data between these memories has to be transfered explicitly
- Always check the error status of every CUDA call (e.g. with `CUDA_ERR_CHECK` macro)

CUDA compute grid

CUDA compute grid supports 1-3 dimensions:

1. `gpu_kernel<<<4, 2>>>(...);`
   — 4 blocks with 2 threads each
2. `gpu_kernel<<<dim3(8, 4, 1), dim3(4, 2, 1)>>>(...);`
   — 8×4 blocks with 4×2 threads each
3. `gpu_kernel<<<dim3(16, 8, 4), dim3(8, 4, 2)>>>(...);`
   — 16×8×4 blocks with 8×4×2 threads each

CUDA compute grid

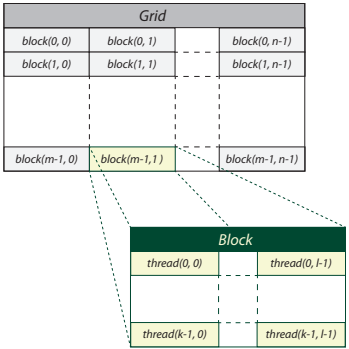
CUDA compute grid supports 1-3 dimensions:

1. `gpu_kernel<<<4, 2>>>(...);`
   — 4 blocks with 2 threads each
2. `gpu_kernel<<<dim3(8, 4, 1), dim3(4, 2, 1)>>>(...);`
   — 8×4 blocks with 4×2 threads each
3. `gpu_kernel<<<dim3(16, 8, 4), dim3(8, 4, 2)>>>(...);`
   — 16×8×4 blocks with 8×4×2 threads each

CUDA compute grid

CUDA compute grid supports 1-3 dimensions:

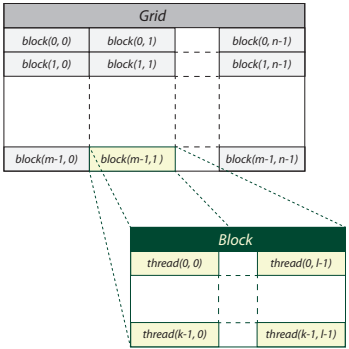
1. `gpu_kernel<<<4, 2>>>(...);`
   - 4 blocks with 2 threads each
2. `gpu_kernel<<<dim3(8, 4, 1), dim3(4, 2, 1)>>>(...);`
   - 8×4 blocks with 4×2 threads each
3. `gpu_kernel<<<dim3(16, 8, 4), dim3(8, 4, 2)>>>(...);`
   - $16 \times 8 \times 4$ blocks with $8 \times 4 \times 2$ threads each



CUDA compute grid

CUDA compute grid supports 1-3 dimensions:

1. `gpu_kernel<<<4, 2>>>(...);`
   - 4 blocks with 2 threads each
2. `gpu_kernel<<<dim3(8, 4, 1), dim3(4, 2, 1)>>>(...);`
   - 8×4 blocks with 4×2 threads each
3. `gpu_kernel<<<dim3(16, 8, 4), dim3(8, 4, 2)>>>(...);`
   - $16 \times 8 \times 4$ blocks with $8 \times 4 \times 2$ threads each



CUDA compute grid (advanced)

CUDA compute grid supports 1-3 dimensions $\Rightarrow$ eases moving multidimensional loops into GPU kernels:

```
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 "hides" loop headers into kernel launch parameters
- Ranges are distributed between threads and blocks of threads
- Blocks number is rounded up to handle the remainder
- What if range is larger than maximum `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 supports 1-3 dimensions $\Rightarrow$ eases moving multidimensional loops into GPU kernels:

```
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 "hides" loop headers into kernel launch parameters
- Ranges are distributed between threads and blocks of threads
- Blocks number is rounded up to handle the remainder
- What if range is larger than maximum `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 supports 1-3 dimensions $\Rightarrow$ eases moving multidimensional loops into GPU kernels:

```
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 "hides" loop headers into kernel launch parameters
- Ranges are distributed between threads and blocks of threads
- Blocks number is rounded up to handle the remainder
- What if range is larger than maximum `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 supports 1-3 dimensions $\Rightarrow$ eases moving multidimensional loops into GPU kernels:

```
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 "hides" loop headers into kernel launch parameters
- Ranges are distributed between threads and blocks of threads
- Blocks number is rounded up to handle the remainder
- What if range is larger than maximum `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 supports 1-3 dimensions $\Rightarrow$ eases moving multidimensional loops into GPU kernels:

```
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 "hides" loop headers into kernel launch parameters
- Ranges are distributed between threads and blocks of threads
- Blocks number is rounded up to handle the remainder
- What if range is larger than maximum `gridDim` $\times$ `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 supports 1-3 dimensions $\Rightarrow$ eases moving multidimensional loops into GPU kernels:

```
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;
        }
```

- We can handle multiple points in single thread ($\Rightarrow$ the loop returns)
- We can't handle consecutive points in one thread due to coalescing requirements
- Compute grid limits are known from device properties

```
__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 supports 1-3 dimensions $\Rightarrow$ eases moving multidimensional loops into GPU kernels:

```
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;
        }
```

- We can handle multiple points in single thread ($\Rightarrow$ the loop returns)
- We can't handle consecutive points in one thread due to coalescing requirements
- Compute grid limits are known from device properties

```
__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 supports 1-3 dimensions $\Rightarrow$ eases moving multidimensional loops into GPU kernels:

```
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;
        }
```

- We can handle multiple points in single thread ($\Rightarrow$ the loop returns)
- We can't handle consecutive points in one thread due to coalescing requirements
- Compute grid limits are known from device properties

```
__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 supports 1-3 dimensions $\Rightarrow$ eases moving multidimensional loops into GPU kernels:

```
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;
        }
```

- We can handle multiple points in single thread ($\Rightarrow$ the loop returns)
- We can't handle consecutive points in one thread due to coalescing requirements
- Compute grid limits are known from device properties

```
__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
```

Simple CUDA optimizations

- Always try to perform coalesced memory accesses
- Block size of `{128, 1, 1}` should be nearly optimal for most of the cases on modern GPUs
- Compiler can tune code for particular GPU identified by *Compute Capability (CC)*:
  add option `-arch=sm_35` for best results on Tesla K20:

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

Simple CUDA optimizations

- Always try to perform coalesced memory accesses
- Block size of `{128, 1, 1}` should be nearly optimal for most of the cases on modern GPUs
- Compiler can tune code for particular GPU identified by *Compute Capability (CC)*:
  add option `-arch=sm_35` for best results on Tesla K20:

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

Simple CUDA optimizations

- Always try to perform coalesced memory accesses
- Block size of `{128, 1, 1}` should be nearly optimal for most of the cases on modern GPUs
- Compiler can tune code for particular GPU identified by *Compute Capability (CC)*:
  add option `-arch=sm_35` for best results on Tesla K20:

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

Simple CUDA optimizations

- Always try to perform coalesced memory accesses
- Block size of `{128, 1, 1}` should be nearly optimal for most of the cases on modern GPUs
- Compiler can tune code for particular GPU identified by *Compute Capability (CC)*:
  add option `-arch=sm_35` for best results on Tesla K20:

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

CUDA development environment

File Edit Source Refactor Navigate Search Project Run Window Help

Project Explorer

- wave13pt
  - Binaries
  - Includes
  - cuda
  - cuda_shfl
  - cuda_shmem1d
  - cuda_shmem1dreg1d
  - cuda_shmem1dreg1dvec2
  - cuda_shmem1dvec2
  - cuda_shmem2d
  - cuda_shmem2dreg1d
  - cuda_shmem2dreg1dvec2
  - cuda_shmem2dvec2
  - cuda_vec2
  - cuda_vec2shfl
  - gcc
  - kernelgen
  - pgi
  - ppcg
  - timing.c
  - timing.h
  - wave13pt.c
  - wave13pt.cu

wave13pt.cu

```
900         nocopy(w2:length(szarray) alloc_if(0) free_if(0))
901 #endif
902         {
903 #if !defined(__CUDACC__) || defined(_PPCG)
904         real *w0p = w0, *w1p = w1, *w2p = w2;
905 #else
906         real *w0p = w0_dev, *w1p = w1_dev, *w2p = w2_dev;
907 #endif
908         for (int it = 0; it < nt; it++)
909         {
910 #if !defined(__CUDACC__) || defined(_PPCG)
911             wave13pt(nx, ny, ns, m0, m1, m2, w0p, w1p, w2p);
912 #else
913             wave13pt<<<config.gridDim, config.blockDim, config.szshmem>>>(
914                 nx, ny, ns,
915                 config,
916                 m0, m1, m2, w0p, w1p, w2p);
917             CUDA_SAFE_CALL(cudaGetLastError());
918             CUDA_SAFE_CALL(cudaDeviceSynchronize());
919 #endif
920             real* w = w0p; w0p = w1p; w1p = w2p; w2p = w;
921             int idx = idxs[0]; idxs[0] = idxs[1]; idxs[1] = idxs[2]; idxs[2] = idx;
922         }
923     }
924     get_time(&compute_f);
925     double compute_t = get_time_diff((struct timespec*)&compute_s, (struct timespec*)&compute_f);
926     if (!no_timing) printf("compute time = %f sec\n", compute_t);
```

O M

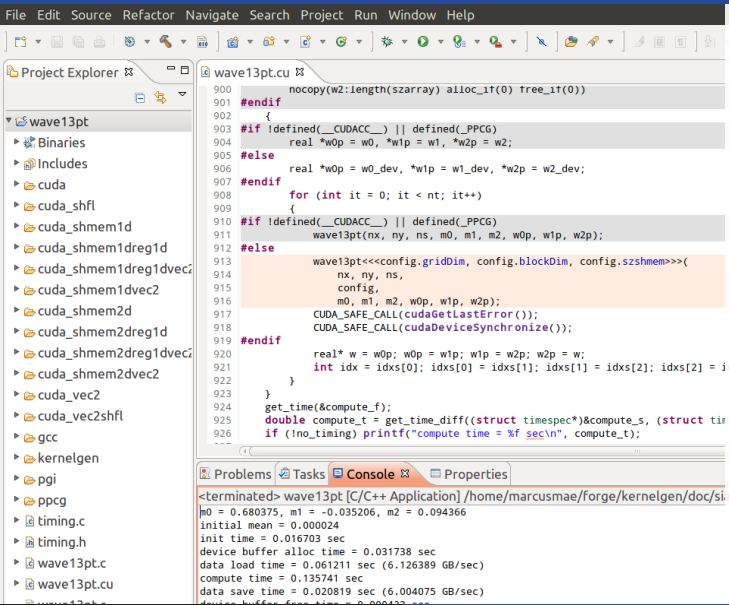
- assert.h
- malloc.h
- stdio.h
- stdlib.h
- timing.h
- openacc.h
- openacc_profilin
- cuda_profiling.h
- MEMALIGN
- STENCIL_BOUNI
- STENCIL_BOUNI
- STENCIL_BOUNI
- STENCIL_BOUNI
- STENCIL_BOUNI
- STENCIL_BOUNI

Problems Tasks Console Properties

<terminated> wave13pt [C/C++ Application] /home/marcusmae/forge/kernelgen/doc/siam_pp14/src/tests/wave13pt/cuda_shmem2d/wave13pt (3/27/14 2:30 AM)

```
m0 = 0.680375, m1 = -0.035206, m2 = 0.094366
initial mean = 0.000024
init time = 0.016703 sec
device buffer alloc time = 0.031738 sec
data load time = 0.061211 sec (6.126389 GB/sec)
compute time = 0.135741 sec
data save time = 0.020819 sec (6.004075 GB/sec)
```

CUDA development environment



Eclipse Nsight Edition

- CUDA/C++ code highlighting
- Embedded CUDA Profiler
- Embedded CUDA Debugger

– for GPU workstation/laptop
– for large codebase

http://docs.nvidia.com/cuda/nsight-eclipse-edition-getting-started-guide/index.html

CUDA development environment

Open Save Undo

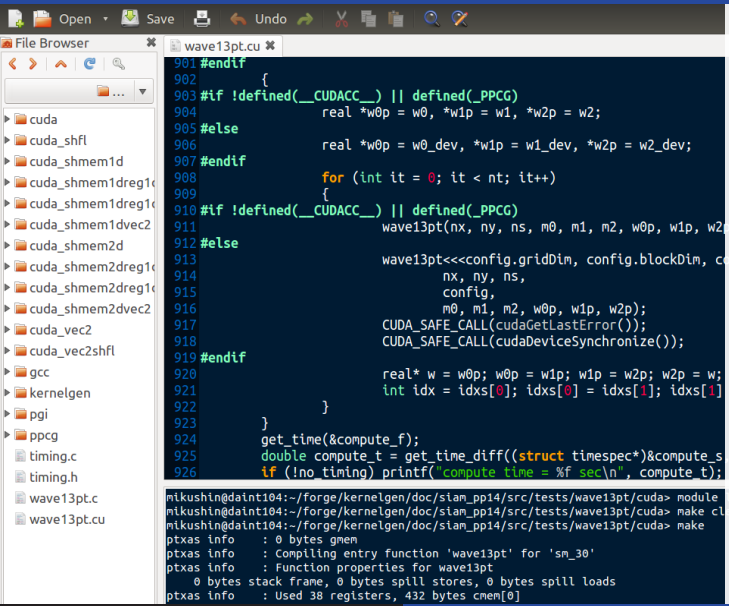
File Browser

wave13pt.cu

```
901 #endif
902         {
903 #if !defined(__CUDACC__) || defined(_PPCG)
904                 real *w0p = w0, *w1p = w1, *w2p = w2;
905 #else
906                 real *w0p = w0_dev, *w1p = w1_dev, *w2p = w2_dev;
907 #endif
908                 for (int it = 0; it < nt; it++)
909                 {
910 #if !defined(__CUDACC__) || defined(_PPCG)
911                         wave13pt(nx, ny, ns, m0, m1, m2, w0p, w1p, w2p);
912 #else
913                         wave13pt<<<config.gridDim, config.blockDim, config.szshmem>>>(
914                                 nx, ny, ns,
915                                 config,
916                                 m0, m1, m2, w0p, w1p, w2p);
917                         CUDA_SAFE_CALL(cudaGetLastError());
918                         CUDA_SAFE_CALL(cudaDeviceSynchronize());
919 #endif
920                         real* w = w0p; w0p = w1p; w1p = w2p; w2p = w;
921                         int idx = idxs[0]; idxs[0] = idxs[1]; idxs[1] = idxs[2]; idxs[2] = idx;
922                 }
923         }
924         get_time(&compute_f);
925         double compute_t = get_time_diff((struct timespec*)&compute_s, (struct timespec*)&compute_f);
926         if (!no_timing) printf("compute time = %f sec\n", compute_t);
```

```
mikushin@daint104:~/forge/kernelgen/doc/siam_pp14/src/tests/wave13pt/cuda> module load cudatoolkit
mikushin@daint104:~/forge/kernelgen/doc/siam_pp14/src/tests/wave13pt/cuda> make clean
mikushin@daint104:~/forge/kernelgen/doc/siam_pp14/src/tests/wave13pt/cuda> make
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function 'wave13pt' for 'sm_30'
ptxas info    : Function properties for wave13pt
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 38 registers, 432 bytes cmem[0]
```

CUDA development environment



GEdit

- Remote editing over SFTP
- Remote shell console

– for remote GPU server/cluster

Conclusion

- CUDA is very easy, if you already worked with MPI+OpenMP
- General exiting code porting strategy:
  - Extract loop body into `__global__` function – GPU kernel
  - Replace original loop with kernel launch, transform loop ranges into `<<< ... >>>` – launch bounds
  - Transfer input data to GPU before, transfer output data from GPU after kernel launch
  - Switch compiler from C/C++ to `nvcc` and rebuild
- Development environments are available for CUDA
- Several things beyond this tutorial:
  - Shared memory
  - Streams and asynchronous data transfers
  - Debugging & profiling
  - And the most important one: the power of CUDA is concentrated in libraries which we will continue with right after the break!

Conclusion

- CUDA is very easy, if you already worked with MPI+OpenMP
- General exiting code porting strategy:
  - Extract loop body into `__global__` function – GPU kernel
  - Replace original loop with kernel launch, transform loop ranges into `<<< ... >>>` – launch bounds
  - Transfer input data to GPU before, transfer output data from GPU after kernel launch
  - Switch compiler from C/C++ to `nvcc` and rebuild
- Development environments are available for CUDA
- Several things beyond this tutorial:
  - Shared memory
  - Streams and asynchronous data transfers
  - Debugging & profiling
  - And the most important one: the great CUDA programming documentation, which we will continue with right after the break!

Conclusion

- CUDA is very easy, if you already worked with MPI+OpenMP
- General exiting code porting strategy:
  - Extract loop body into `__global__` function – GPU kernel
  - Replace original loop with kernel launch, transform loop ranges into `<<< ... >>>` – launch bounds
  - Transfer input data to GPU before, transfer output data from GPU after kernel launch
  - Switch compiler from C/C++ to nvcc and rebuild
- Development environments are available for CUDA
- Several things beyond this tutorial:
  - Shared memory
  - Streams and asynchronous data transfers
  - Debugging & profiling
  - And the most important one: the power of CUDA is concentrated in libraries, which we will continue with right after the break!

Conclusion

- CUDA is very easy, if you already worked with MPI+OpenMP
- General exiting code porting strategy:
  - Extract loop body into `__global__` function – GPU kernel
  - Replace original loop with kernel launch, transform loop ranges into `<<<` ... `>>>` – launch bounds
  - Transfer input data to GPU before, transfer output data from GPU after kernel launch
  - Switch compiler from C/C++ to nvcc and rebuild
- Development environments are available for CUDA
- Several things beyond this tutorial:
  - Shared memory
  - Streams and asynchronous data transfers
  - Debugging & profiling
  - And the most important one: the power of CUDA is concentrated in [illegible] which we will continue with right after the break!

Conclusion

- CUDA is very easy, if you already worked with MPI+OpenMP
- General exiting code porting strategy:
  - Extract loop body into `__global__` function – GPU kernel
  - Replace original loop with kernel launch, transform loop ranges into `<<<` ... `>>>` – launch bounds
  - Transfer input data to GPU before, transfer output data from GPU after kernel launch
  - Switch compiler from C/C++ to nvcc and rebuild
- Development environments are available for CUDA
- Several things beyond this tutorial:
  - Shared memory
  - Streams and asynchronous data transfers
  - Debugging & profiling
  - And the most important one: the power of CUDA is concentrated in libraries, which we will continue with right after the break!

Conclusion

- CUDA is very easy, if you already worked with MPI+OpenMP
- General exiting code porting strategy:
  - Extract loop body into `__global__` function – GPU kernel
  - Replace original loop with kernel launch, transform loop ranges into `<<<` ... `>>>` – launch bounds
  - Transfer input data to GPU before, transfer output data from GPU after kernel launch
  - Switch compiler from C/C++ to `nvcc` and rebuild
- Development environments are available for CUDA
- Several things beyond this tutorial:
  - Shared memory
  - Streams and asynchronous data transfers
  - Debugging & profiling
  - ...
  - And the most important one: the power of CUDA is concentrated in libraries which we will continue with right after the break!

Conclusion

- CUDA is very easy, if you already worked with MPI+OpenMP
- General exiting code porting strategy:
  - Extract loop body into `__global__` function – GPU kernel
  - Replace original loop with kernel launch, transform loop ranges into `<<<` ... `>>>` – launch bounds
  - Transfer input data to GPU before, transfer output data from GPU after kernel launch
  - Switch compiler from C/C++ to `nvcc` and rebuild
- Development environments are available for CUDA
- Several things beyond this tutorial:
  - Shared memory
  - Streams and asynchronous data transfers
  - Debugging & profiling
  - And the most important one: the power of CUDA is concentrated in libraries which we will continue with right after the break!

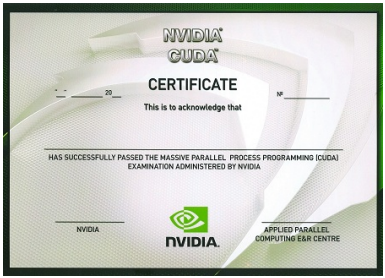
Conclusion

- CUDA is very easy, if you already worked with MPI+OpenMP
- General exiting code porting strategy:
  - Extract loop body into `__global__` function – GPU kernel
  - Replace original loop with kernel launch, transform loop ranges into `<<<` ... `>>>` – launch bounds
  - Transfer input data to GPU before, transfer output data from GPU after kernel launch
  - Switch compiler from C/C++ to `nvcc` and rebuild
- Development environments are available for CUDA
- Several things beyond this tutorial:
  - Shared memory
  - Streams and asynchronous data transfers
  - Debugging & profiling
  - And the most important one: the practical CUDA programming experience, which we will continue with right after the break!

Conclusion

- CUDA is very easy, if you already worked with MPI+OpenMP
- General exiting code porting strategy:
  - Extract loop body into `__global__` function – GPU kernel
  - Replace original loop with kernel launch, transform loop ranges into `<<<` ... `>>>` – launch bounds
  - Transfer input data to GPU before, transfer output data from GPU after kernel launch
  - Switch compiler from C/C++ to `nvcc` and rebuild
- Development environments are available for CUDA
- Several things beyond this tutorial:
  - Shared memory
  - Streams and asynchronous data transfers
  - Debugging & profiling
  - And the most important one: the practical CUDA programming experience, which we will continue with right after the break!

GPU / CUDA Training & Certification



Academic: Parallel & Distributed Computing LAB

- Institute of Computational Science, USI Lugano
- Prof. Dr. Olaf Schenk
- 1 semester (master course), assignments on GPU cluster
- GPU, profiling, optimization, scientific applications case studies
- Contact: dmitry.mikushin@usi.ch

Commercial: Advanced GPU computing course

- Applied Parallel Computing LLC & NVIDIA
- 3-5 days, on-site, hands-ons on GPU cluster
- GPU, profiling, optimization, customer-oriented hands-ons
- Register: http://www.nvidia.de/object/gpu-computing-workshop-registration-form.html

Certificate: granted upon successful pass of comprehensive practical tests