HP-SEE

CUDA C overview

www.hp-see.eu

Dusan Stankovic
Scientific Computer Laboratory
Institute of Physics Belgrade
dusan.stankovic@ipb.ac.rs

HP-SEE
High-Performance Computing Infrastructure
for South East Europe's Research Communities

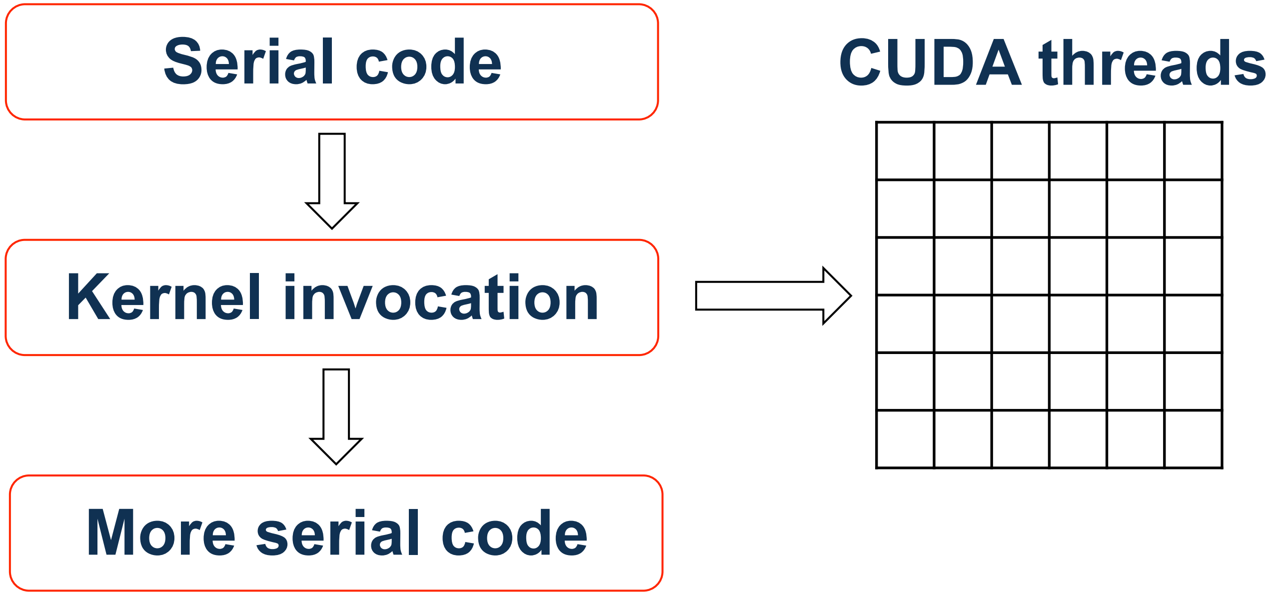
The HP-SEE initiative is co-funded by the European Commission under the FP7 Research Infrastructures contract no. 261499

Agenda

- Learn about basic features of CUDA C
  - Compilation process and compute capabilities
  - Hierarchical thread organization
  - Mapping of threads to data indices
  - Interface for GPU memory management
  - Interface for launching parallel execution
- Also some advanced features
  - Memory organization on the GPU
  - Usage of CUDA streams and asynchronous execution
  - External libraries for CUDA
  - Profiling tools and performance measuring

Heterogenous execution model

- **Host** – a CPU which executes the main program in serial
- **Device** – a GPU which executes parallel portions of code
- Memory spaces are completely separate
  - All allocations and data movement – responsibility of the programmer



Code for GPUs

HP-SEE
High-Performance Computing Infrastructure for South East Europe's Research Communities

- CUDA C program is written as follows:
  - Serial parts in host C code
  - Parallel parts in device SPMD kernel C code
- Source code is compiled separately
  - Standard C/C++ code for the CPU
  - Device code in PTX – compiled just-in-time for the exact device
- Use the **nvcc** for compilation
  - PTX is an assembly format
  - Specific binary code for the GPU devices

Device compute capability

- NVIDIA GPU devices are based on different cores
  - Each new generation changes architecture and adds some new features (Fermi, Kepler, ...)
  - All use the same programming model even when the internal organization changes a lot
- Compute capability used to show which features GPUs support
  - Major number – entirely new architecture
    - 2 for Fermi, 3 for Kepler
  - Minor number – incremental upgrades to an architecture
    - 3.5 for newest Tesla cards, includes some new features
- Sometimes new features can be significant
  - 1.3 added support for double precision arithmetic

Thread organization

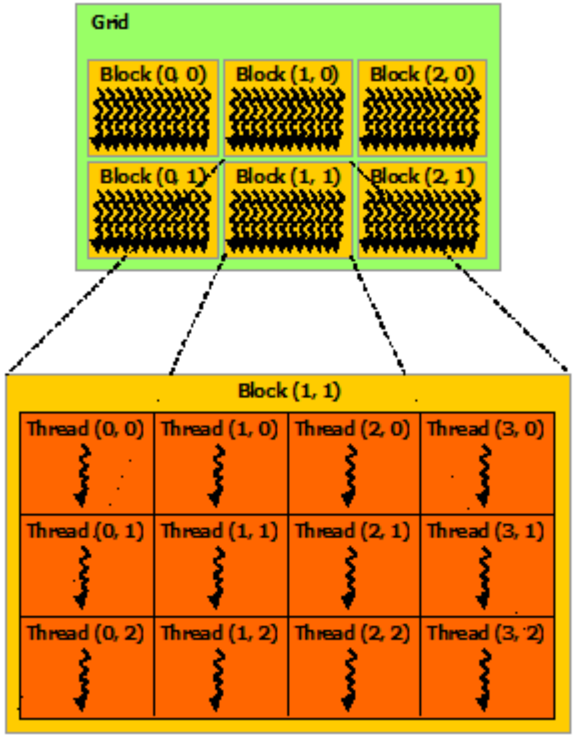


image taken from NVIDIA CUDA C Programming Guide

- Inherent variables for each thread in a kernel launch
  - `blockDim`, `blockIdx` for blocks in a grid
  - `threadIdx` for threads in a block

Thread mapping to data indices

HP-SEE
High-Performance Computing Infrastructure for South East Europe's Research Communities

- Both the grid and each thread block can be three-dimensional
  - Predefined data type dim3 to hold grid and block dimensions
  - Parameter for the kernel launch
- Example: a 2D matrix

```
float matrix[N][N];

int my_col = blockIdx.x * blockDim.x + threadIdx.x;
int my_row = blockIdx.y * blockDim.y + threadIdx.y;

matrix[my_row][my_col] = ...
```

CUDA kernels

- Kernel calls are points of parallel execution on the GPU
- Kernel is defind using `__global__` declaration specifier
  - Meaning that it can execute on the GPU
- Each kernel launch has an execution specification
  - Grid and block dimensions are necessary
  - Syntax is **`my_kernel<<< ... >>>(arg1, arg2, ...);`**
- There are some more declaration specifiers:

	Executed on:	Callable from:
`__device__ float dev_func(...)`	device	device
`__global__ void kern_func(...)`	device	host
`__host__ float host_func(...)`	host	host

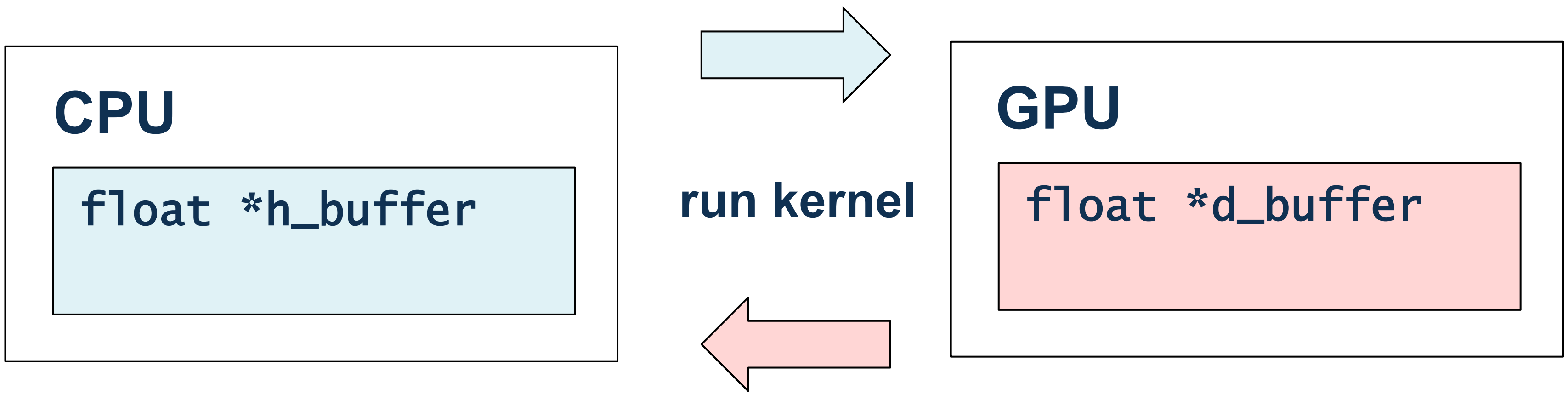
CUDA kernel example

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

GPU memory management

- CUDA GPU has its own address space
  - Necessary to allocate and free data on the GPU
  - Necessary to transfer data from the main memory into the GPU memory and in the other way



CUDA memory API - data allocation

- Memory allocation and deallocation – similar to malloc and free in C for the CPU

- `cudaMalloc(void** dev_ptr, size_t size);`
  - `dev_ptr` - address of a pointer to the device memory
  - `size` - size to allocate in bytes
  - double pointer because pointer itself will be changed
- `cudaFree(void* dev_ptr);`
  - `dev_ptr` - pointer to the device memory allocated with `cudaMalloc`

CUDA memory API - data movement

- Used to explicitly move data to the GPU and back to the CPU memory
- **`cudaMemcpy(void* dst, const void* src, size_t count, enum cudaMemcpyKind kind);`**
  - `dst` - pointer to the transfer destination address
  - `src` - pointer to the transfer source address
  - `count` – size of data to copy in bytes
  - `kind` – type of transfer
    - `cudaMemcpyHostToDevice` - from the host to the device
    - `cudaMemcpyDeviceToHost` - from the device to the host

CUDA memory API example

HP-SEE
High-Performance Computing Infrastructure for South East Europe's Research Communities

```
int main()
{
    float *host_array, *dev_array; int size =
N*sizeof(float));
    host_array = malloc(size);
    cudaMalloc(&dev_array, size);
    cudaMemcpy(dev_array, host_array, size,
                cudaMemcpyHostToDevice);
    // Kernel invocation with N threads
    process_array<<<1, N>>>(dev_array);
    cudaMemcpy(host_array, dev_array, size,
                cudaMemcpyDeviceToHost);
    free(host_array);
    cudaFree(dev_array);
}
```

Indexing of 2D structures

- Contiguous memory for multidimensional structures
  - Can be accessed with a single indexing operation
  - Good for performance, allows for easy transferring of data
- C example:
  - data is stored row by row in memory
  - **`mat[i][j]`** translates to **`mat[i*width + j];`**
- In CUDA:
  - Thread x index changes fastest (important for thread scheduling issues)
  - We should use x to select a column and y to select a row for a 2D matrix

Working with 2D arrays example

HP-SEE
High-Performance Computing Infrastructure for South East Europe's Research Communities

```
__global__ void process_matrix(float *mat, int nrows, int ncols) {
    int my_row = blockIdx.y * blockDim.y + threadIdx.y;
    int my_col = blockIdx.x * blockDim.x + threadIdx.x;
    //no need to loop through matrix elements, need to check bounds
    (if my_row < nrows && my_col < ncols) {
        mat[my_row * ncols + my_col] = some_func();
    }
}
void main() {
    ...
    cudaMemcpy(dmat, hmat, size, cudaMemcpyHostToDevice);
    dim3 block_size(NTHREADS, NTHREADS);
    dim3 grid_size((ncols-1)/NTHREADS+1, (nrows-1)/NTHREADS+1);
    process_matrix<<<grid_size, block_size>>>(dmat, nrows, ncols);
    cudaMemcpy(hmat, dmat, size, cudaMemcpyDeviceToHost);
    ...
}
```

GPU memory organization

HP-SEE
High-Performance Computing Infrastructure for South East Europe's Research Communities

- Registers are per-thread
  - **very low** latency, **very high** throughput
  - limited resource, used for automatic variables
- Shared memory (and L1 cache) is per-block
  - **low** latency, **high** throughput
  - can yield significant performance boost, depends on algorithm
  - programmer is responsible for its usage
  - shared/cache split can be controlled using the API
- Global memory is visible to all threads
  - **high** latency, **moderate** throughput
  - memory allocated with cudaMalloc is global
  - has the highest capacity

CUDA memory organization examples


```
__device__ int global_var; //global

__global__ void my_kernel(float *array, int size)
{
    int block_xsize = blockDim.x; //register
    int my_ind = blockIdx.x * blockDim.x + threadIdx.x;
    __shared__ float smem[block_xsize];  //shared
    //load into shared memory
    smem[threadIdx.x] = array[my_ind];
    ...
    //do something with shared array
}
```

Thread synchronization in CUDA

HP-SEE
High-Performance Computing Infrastructure for South East Europe's Research Communities

- Sometimes a synchronization between threads is necessary
  - happens between various computation stages
  - usually follows loading into shared memory
- Synchronization between threads in the same block
  - `__syncthreads()` function causes each thread in a block to wait untill all reach that point
  - to ensure that all needed elements are stored into shared memory
  - to ensure that all needed elements are read from shared memory before its contents are modified again
- Synchronization between threads from different blocks
  - can be done with global variables – slow, not recommended
  - best to create separate kernels and synchronize in between

Host – device synchronization in CUDA

- CUDA calls are synchronous with regard to host and device
  - example cudaMalloc, cudaMemcpy, ...
- Kernel launches are **asynchronous** on the host side
- Host can do some work while kernel is being executed on the GPU
- To synchronize after a kernel launch – use **`cudaDeviceSynchronize()`**
- Allows for partial overlap – but there is an asynchronous API for even more control
- Memory copying can be overlapped with computation on the CPU, but also with computation on the GPU

Asynchronous memory transfers

- `cudaMemcpyAsync(void* dst, const void* src, size_t count, enum cudaMemcpyKind kind, cudaStream_t stream = 0);`
  - `stream` - an additional parameter to the call, defaults to zero
  - host memory used during transfer has to be page-locked
- Page-locked host memory – prevents OS from swapping
  - allows using DMA controllers on host and device for better performance, and
  - allows to safely copy memory without OS interference, thus leaving the CPU free for other tasks
- Needs to be explicitly allocated as page-locked
  - use `cudaMallocHost()` or `cudaFreeHost()`
  - should be used carefully, too much of it can slow down the system

Introduction to streams

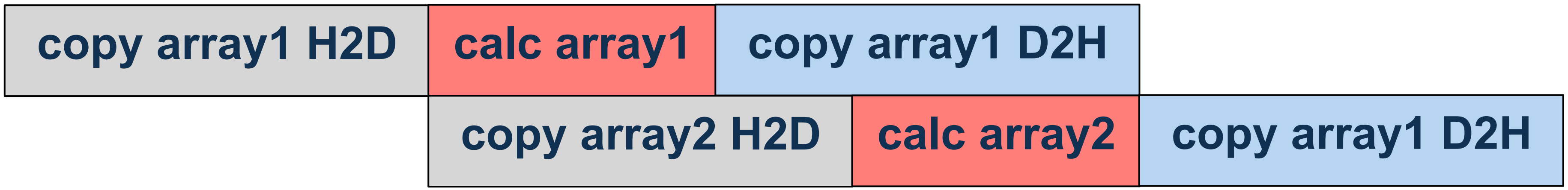
- CUDA stream is a sequence of CUDA commands always issued in order
  - even when these commands are asynchronous to the host, they are executed in sequence on the GPU

```
kernelA<<<grid, block>>>(arrayA, sizeA);
kernelB<<<grid, block>>>(arrayB, sizeB);
```

- Additional parameter in kernel configuration – stream to use
  - if none is specified, a default stream is used
- Different streams are independent, can execute their commands concurrently
- To use asynchronous copying – we need a separate stream

Using streams to overlap copying and computation

- Fermi GPUs and newer can overlap kernel execution, H2D and D2H transfers at the same time
- Create separate streams for execution and copying
- For synchronization with a specific stream use **`cudaStreamSynchronize`**



Using streams example

```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaMallocHost(&array1_h, size); cudaMalloc(&array1_d, size);
cudaMallocHost(&array2_h, size); cudaMalloc(&array2_d, size);

cudaMemcpyAsync(array1_d, array1_h, size, H2D, stream1);
kernel1<<<grid, block, 0, stream1>>>(array1_d, size);
cudaMemcpyAsync(array1_h, array1_d, size, D2H, stream1);
cudaMemcpyAsync(array2_d, array2_h, size, H2D, stream2);
kernel1<<<grid, block, 0, stream1>>>(array1_d, size);
cudaMemcpyAsync(array1_d, array1_h, size, D2H, stream1);

do_something_else(...);
//now we need the data from the first array
cudaStreamSynchronize(stream1);
process_array(array1_h);
```

Checking for errors in CUDA calls

HP-SEE
High-Performance Computing Infrastructure for South East Europe's Research Communities

- All CUDA runtime functions return an error code
- For synchronous calls (such as cudaMemcpy)
  - error is related to the call execution
  - but, can also be a result of some previous asynchronous call
- For asynchronous calls (such as kernel launches or cudaMemcpyAsync)
  - error can only be related to launching of the CUDA function (for example, wrong parameters)
  - errors that happen during execution can only be checked at subsequent synchronization points

```
cudaError_t err;
if((err=cudaMemcpy(a_d, a_h, size, H2D)) != cudaSuccess)
exit(1)
compute<<<grid, block>>>(a_d, size);
if((err=cudaDeviceSynchronize()) != cudaSuccess) exit(1);
```

Numerical libraries for CUDA GPUs

HP-SEE
High-Performance Computing Infrastructure for South East Europe's Research Communities

- NVIDIA is developing numerical libraries for its GPU cards
- CUBLAS, CUFFT, CURAND, CUSPARSE
- Thrust – a template library based on STL

- Relatively sasy to use, just swap some routine calls and link with CUDA libraries
  - memory allocation and movement is still responsibility of the programmer
  - sometimes it is more complicated – CUBLAS uses column based storage (like FORTRAN), need to swap dimensions

- They have their own error types – for example **`cublasStatus_t`** or **`cufftResult_t`**

Debugging and profiling

HP-SEE
High-Performance Computing Infrastructure for South East Europe's Research Communities

- For debugging there is an extension to gdb called CUDA-GDB
- Allows breakpoints inside kernels
- Supports switching between thread contexts and printing values of thread local variables

- Command-line profiler for CUDA is a part of the toolkit
  - very easy to use to get initial measurements – just export an environment variable
- **`export CUDA_PROFILE = 1`**
- **`export CUDA_PROFILE_LOG = path/to/log/file`**

Thank you

- Questions?