CUDA
Lecture 2

Manfred Liebmann
Technische Universität München
Chair of Optimal Control
Center for Mathematical Sciences, M17
manfred.liebmann@tum.de

December 15, 2015

CUDA Programming Fundamentals

CUDA kernels are routines that run on the graphics processing unit (GPU).

```
#include<iostream>
#include<cuda.h>
using namespace std;

__global__ void vector_set(float *x, float a, int n)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if(i < n) x[i] = a;
}
int main(int argc, char** argv)
{
    int n = 256 * 1024, BLOCK = 256, GRID = (n + BLOCK - 1) / BLOCK;
    float *dev_z = 0, *hst_z = new float[n]();

    cudaMalloc((void**)&dev_z, n * sizeof(float));
    if (dev_z != 0) {
        vector_set<<<GRID, BLOCK>>>(dev_z, 2.0, n);
        cudaMemcpy(hst_z, dev_z, n * sizeof(float), cudaMemcpyDeviceToHost);
        cout << "z[0]: " << hst_z[0] << endl;
    }
    cudaFree(dev_z);
    return 0;
}
```

Compiling CUDA Programs

CUDA programs use an extension of the C/C++ programming language and require the Nvidia `nvcc` compiler for building programs. Historically source files with CUDA code required the file extension `.cu` to be recognized by the Nvidia compiler. The compiler flag `-x=cu` overrides this behavior and allows standard `.cpp` files to be compiled.

```
nvcc -O3 -x=cu -arch=sm_20 -o basic basic.cpp
```

Furthermore the device hardware architecture of the GPU has to be specified with the flag `-arch=sm_11` to produce correct code. Architectures: Tesla `sm_1x`; Fermi `sm_2x`; Kepler `sm_3x`; Maxwell `sm_5x`.

CUDA programs are executed as standard programs. The `CUDA_PROFILE` environment variable enables a driver level profiler which collects kernel and data transfer timings in a profile log file: `cuda_profile_0.log`. Timings are given in microseconds μs.

```
export CUDA_PROFILE=1
./basic
```

Developing, Debugging and Profiling CUDA Programs

Nvidia Nsight IDE: `nsight`

Debugging: `cuda-gdb`

Memory checking: `cuda-memcheck`

Basic profiling: `nvprof`

Nvidia visual profiler: `nvvp`

The Nvidia visual profiler `nvvp` allows a detailed analysis of CUDA programs and is highly recommended for CUDA kernel analysis and optimization.

Profiling Tools for MPI+OpenMP+CUDA

- CUDA aware parallel profiling tools
  - **Score-P** (`http://www.vi-hps.org/projects/score-p/`)
  - Vampir
  - Tau

Score-P is a new initiative to unify several parallel performance profiling tools: Vampir, Scalasca, TAU, Periscope. These tools are good for discovering MPI issues as well as basic CUDA performance inhibitors.

Reading List

Nvidia Corporation: *CUDA C Programming Guide*
(`http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`)

D. B. Kirk, W. W. Hwu: *Programming Massively Parallel Processors*, MK 2010

GPU Hardware: Nvidia Kepler GK110 Die Map

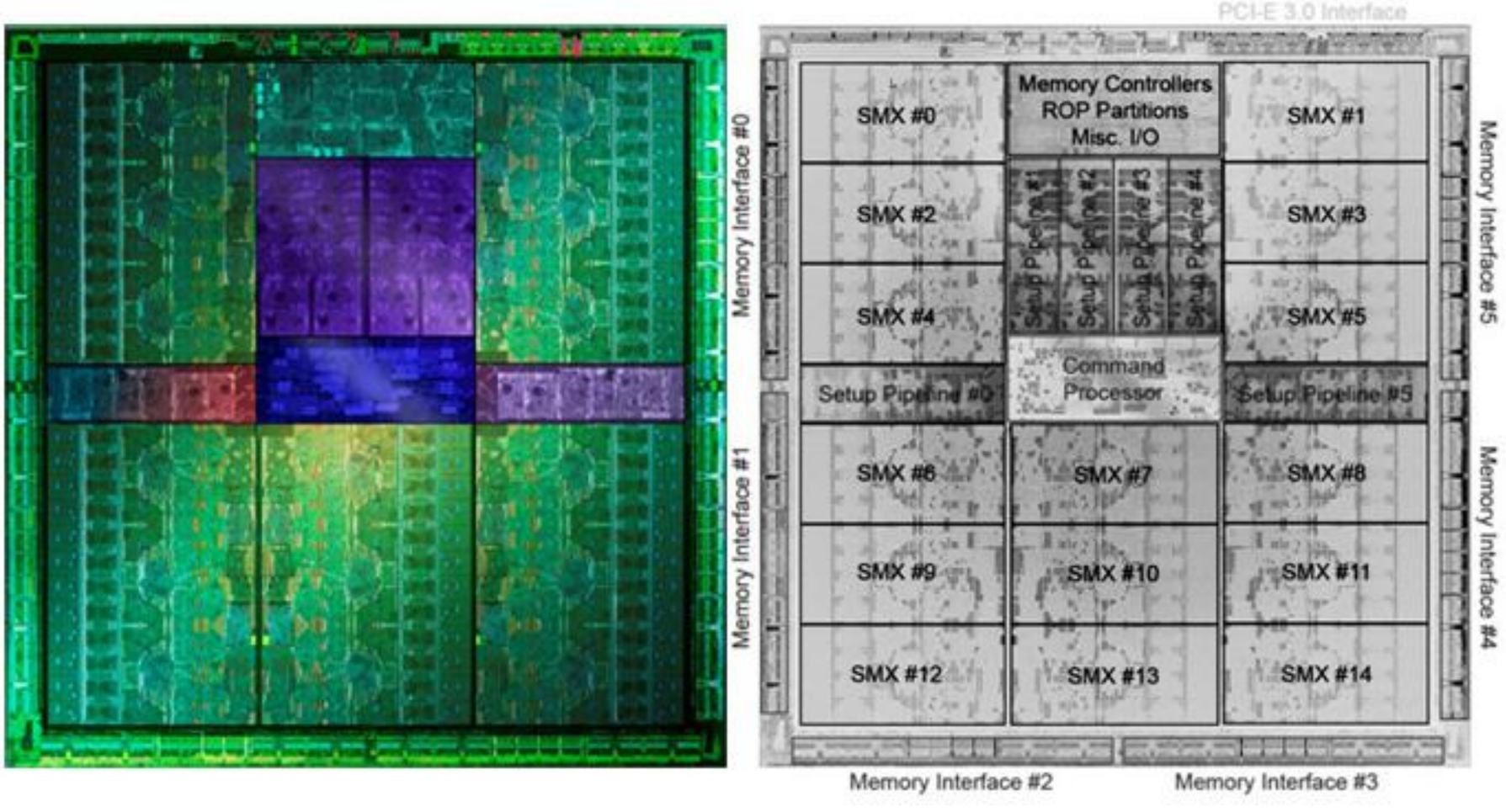

Figure 1: Nvidia Kepler GK110 Processor: 2880 Cores

Nvidia Kepler Hardware Architecture

Figure 2: Nvidia GK110 Processor Diagramm

Nvidia Kepler Streaming Multiprocessor SMX

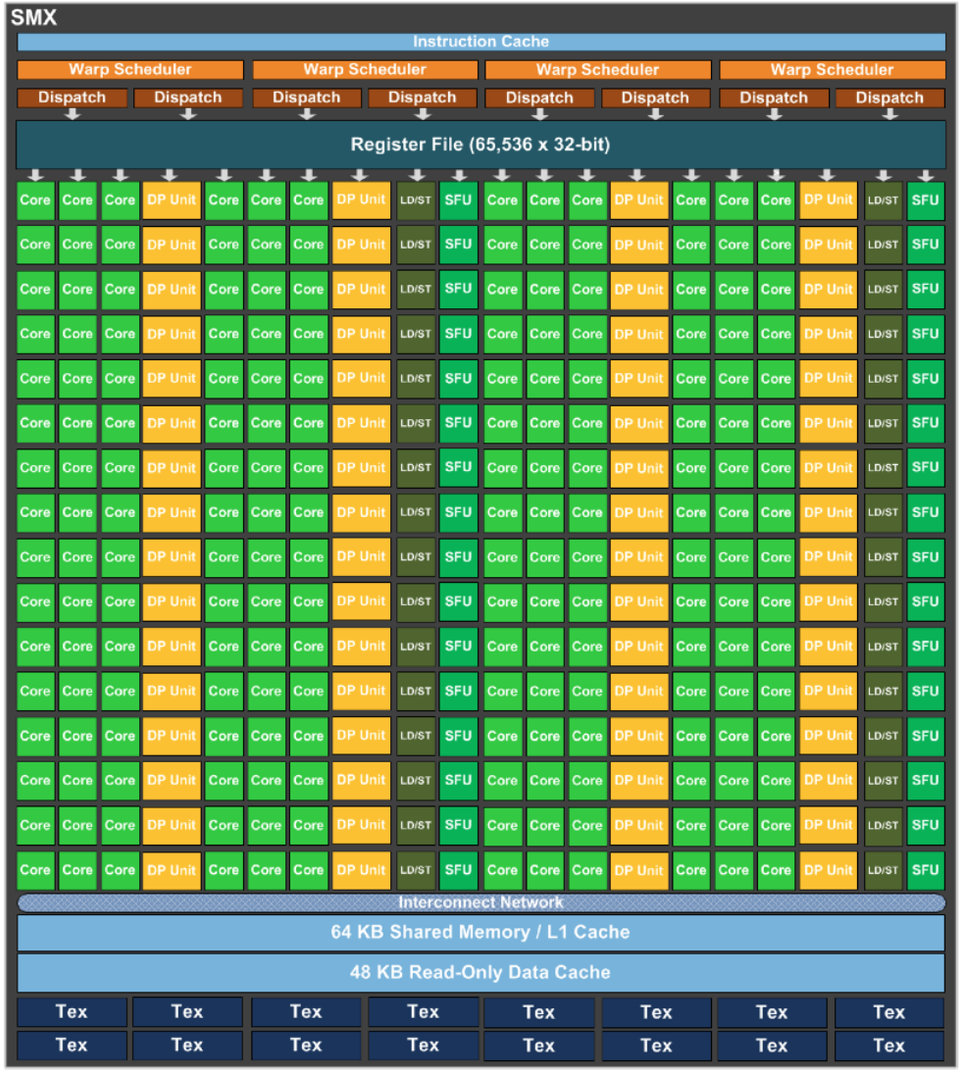

Figure 3: Kepler GK110 SMX architecture

Graphics Processor Hardware Architecture

- Building block: Streaming multiprocessor (SMX)
  - 192 cores, 64 DP units, 32 load / store units, 32 SFUs, 16 tex units
  - 65536 x 32 bit register file
  - 64 KB L1 cache / shared memory
  - 48 KB Read-only data cache (texture cache)
  - 4 warp scheduler
  - 8 dispatch units

The GPU hardware architecture details vary between the different generations: Tesla, Fermi, Kepler, Maxwell.

Parallelization Concepts on GPUs

- Single instruction multiple thread (SIMT) architecture
  - Implemented in the streaming multiprocessor (SMX)
  - Smallest execution unit is a **warp**: 32 threads
  - **32 threads within a warp execute a single instruction stream in lockstep**
  - Warps are combined into **thread blocks**: 32 - 2048 threads
  - Warp schedulers and dispatch units distribute instructions to free compute resources
  - Threads within a thread block are organized in 1D / 2D / 3D layouts
  - Thread blocks are furthermore organized in 1D / 2D / 3D **grids**
  - The threads within a thread block execute on the same SMX
  - Different thread blocks within a grid can execute on different SMX

Schedule hundreds of threads on graphics processors to hide memory access latencies and instruction execution latencies!

Grids and Thread Blocks on GPUs

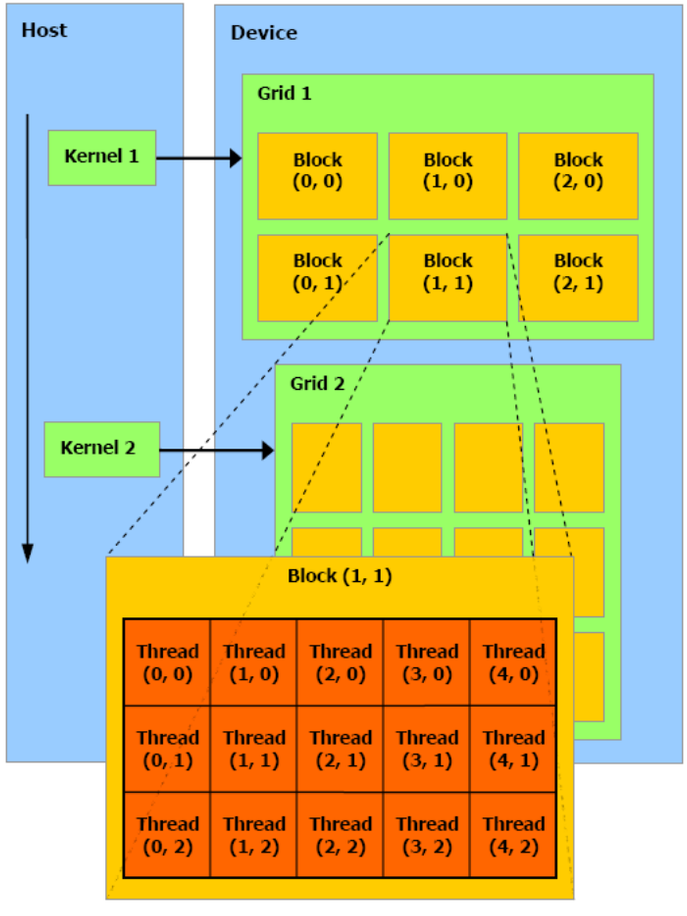

Figure 4: Kernels, threads, blocks, and grids

Memory Access and Data Caching

- **Streaming multiprocessor and memory access**
  - Streaming multiprocessors can completely hide memory access latencies
  - L1/L2/read-only caches and shared memory can improve effective memory bandwidth
  - Configurable L1 cache and on-chip shared memory (16-48 KB) and L2 cache (1.5 MB)

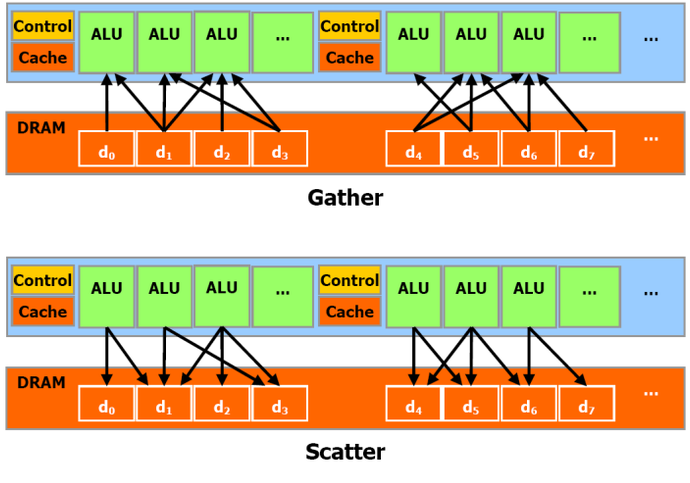

Figure 5: Memory access patterns

On-Chip High Bandwidth Shared Memory

- **Streaming multiprocessors and shared memory**
  - Shared memory is accessible only to the threads within a thread block
  - Shared memory is a banked memory, access penalties apply
  - Access to the same memory location by different threads is serialized

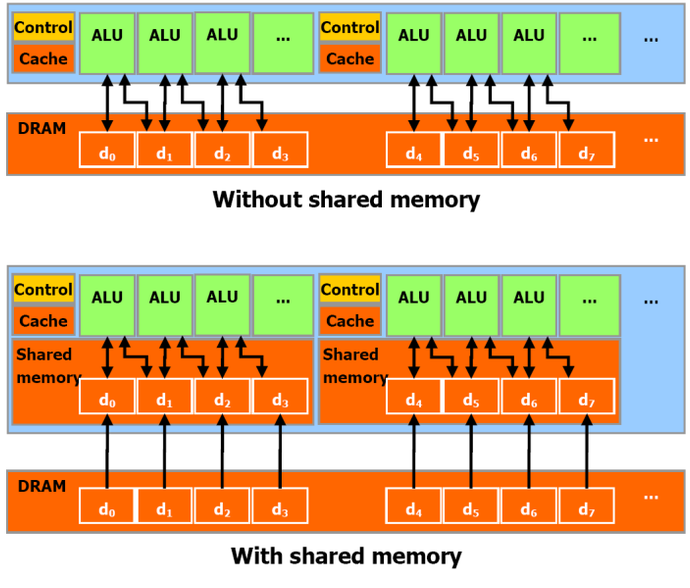

Figure 6: Shared memory is an on-chip high bandwidth memory

Coalesced Memory Access

- **Coalesced memory access is important for optimal memory bandwidth usage!**
  - A half-warp, i.e. 16 threads, must access an aligned block of memory
  - Performance penalty for non-coalesced memory access
  - Relaxed rules: Permutations within an aligned memory block are allowed

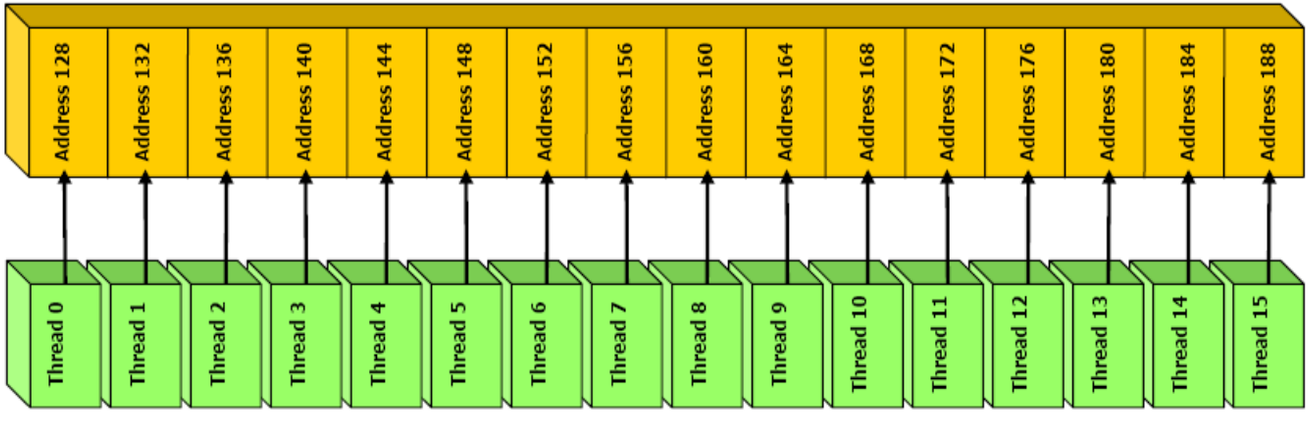

Figure 7: Coalesced memory access

Computing in Lock-Step: General Considerations

Implications of the single instruction multiple thread (SIMT) model of the streaming multiprocessor (SMX).

- **SIMT performance optimizations**
  - Coalesced memory access: read/write memory in aligned blocks of 32 elements
  - Thread divergence and code serialization within control structures: for, while, if–else
  - Optimize code within control structures for *conditional instruction*

C Language Extensions: Function Type Qualifiers

The `__global__` qualifier declares a function as being a kernel. Such a function is executed on the device, callable from the host, and callable from the device for devices of compute capability 3.x.

The `__device__` qualifier declares a function that is executed on the device and callable from the device only.

The `__host__` qualifier declares a function that is executed on the host and callable from the host only.

C Language Extensions: Variable Type Qualifiers

The `__device__` qualifier declares a variable that resides on the device. The variable resides in global memory space and has the lifetime of an application. Is accessible from all the threads within the grid and from the host through the runtime library.

The `__constant__` qualifier, optionally used together with `__device__`, declares a variable that resides in constant memory space, has the lifetime of an application, and is accessible from all the threads within the grid and from the host through the runtime library.

The `__shared__` qualifier, optionally used together with `__device__`, declares a variable that resides in the shared memory space of a thread block, has the lifetime of the block, and is only accessible from all the threads within the block.

The `__managed__` qualifier, optionally used together with `__device__`, declares a variable that can be referenced from both device and host code, e.g., its address can be taken or it can be read or written directly from a device or host function, and has the lifetime of an application.

The `nvcc` compiler supports restricted pointers via the `__restrict__` keyword.

C Language Extensions: Built-in Vector Types

- Standard data types
  - `char, short, int, long, longlong, float, double`
- Vector data types
  - `char1, uchar1, char2, uchar2, char3, uchar3, char4, uchar4`
  - `short1, ushort1, short2, ushort2, short3, ushort3, short4, ushort4`
  - `int1, uint1, int2, uint2, int3, uint3, int4, uint4`
  - `long1, ulong1, long2, ulong2, long3, ulong3, long4, ulong4`
  - `longlong1, ulonglong1, longlong2, ulonglong2`
  - `float1, float2, float3, float4`
  - `double1, double2`

C Language Extensions: Built-in Variables

Built-in variables specify the grid and block dimensions and the block and thread indices. They are only valid within functions that are executed on the device.

`dim3` is an integer vector type based on `uint3` that is used to specify dimensions. When defining a variable of type `dim3`, any component left unspecified is initialized to 1.

- Built-in Variables
  - `gridDim` is of type `dim3` and contains the dimensions of the grid.
  - `blockIdx` is of type `uint3` and contains the block index within the grid.
  - `blockDim` is of type `dim3` and contains the dimensions of the block.
  - `threadIdx` is of type `uint3` and contains the thread index within the block.
  - `warpSize` is of type `int` and contains the warp size in threads.

C Language Extensions: Block and Grid Dimensions

- **Block dimensions**
  - 1 - 1024 threads arranged in 1D, 2D, 3D layout
  - Maximum size of x, y, z component: 1024, 1024, 64
  - Maximum of 8 - 32 resident blocks per multiprocessor
  - Maximum of 48 - 64 resident warps per multiprocessor
  - Maximum of 1536 - 2048 resident threads per multiprocessor
- **Grid dimensions**
  - Organizational structure to group thread blocks together
  - 1D, 2D, 3D layout
  - Maximum grid size of x component: 65535 - 2147483647
  - Maximum grid size of y, z component: 65535, 65535

C Language Extensions: Execution Configuration

Any call to a `__global__` function must specify the execution configuration for that call. The execution configuration defines the dimension of the grid and blocks that will be used to execute the function on the device, as well as the associated stream.

The execution configuration is specified by inserting `<<< Dg, Db, Ns, S >>>` between the function name and the parenthesized argument list, where `Dg` is of type `dim3` and specifies the dimension and size of the grid, such that `Dg.x * Dg.y * Dg.z` equals the number of blocks being launched.

`Db` is of type `dim3` and specifies the dimension and size of each block, such that `Db.x * Db.y * Db.z` equals the number of threads per block.

`Ns` is of type `size_t` and specifies the number of bytes in shared memory that is dynamically allocated per block for this call in addition to the statically allocated memory. This dynamically allocated memory is used by any of the variables declared as an external array as mentioned in `__shared__`. `Ns` is an optional argument which defaults to 0.

`S` is of type `cudaStream_t` and specifies the associated stream. `S` is an optional argument which defaults to 0.

Cuda C Runtime: Memory Allocation

- **Device memory**
  - `cudaMalloc`
  - `cudaMemcpy`
  - `cudaFree`
- **Page-Locked Host Memory**
  - `cudaHostAlloc`
  - `cudaHostRegister`
  - `cudaFreeHost`