Introduction to CUDA Programming

Pierre Talbot
pierre.talbot@uni.lu

7th June 2023

University of Luxembourg

Parallel Programming on GPUs

Vendor-specific API

- Nvidia GPUs are the most widespread, programmed in CUDA, an extension of C++.

Parallel Programming on GPUs

Vendor-specific API

- Nvidia GPUs are the most widespread, programmed in CUDA, an extension of C++.

Abstractions for heterogeneous computing

- Next session on OpenACC.

Parallel Programming on GPUs

Vendor-specific API

- Nvidia GPUs are the most widespread, programmed in CUDA, an extension of C++.

Abstractions for heterogeneous computing

- Next session on OpenACC.

Programming languages

Parallel Programming on GPUs

Vendor-speci

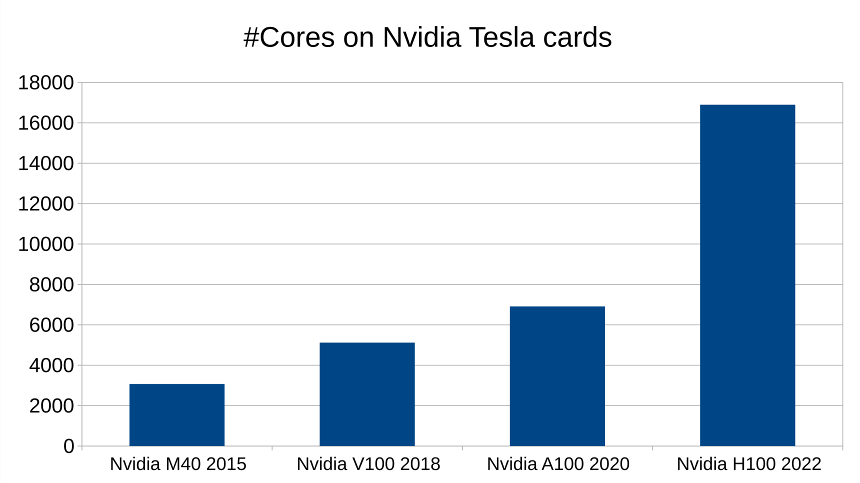
- Nvidia GPUs a

Abstractions

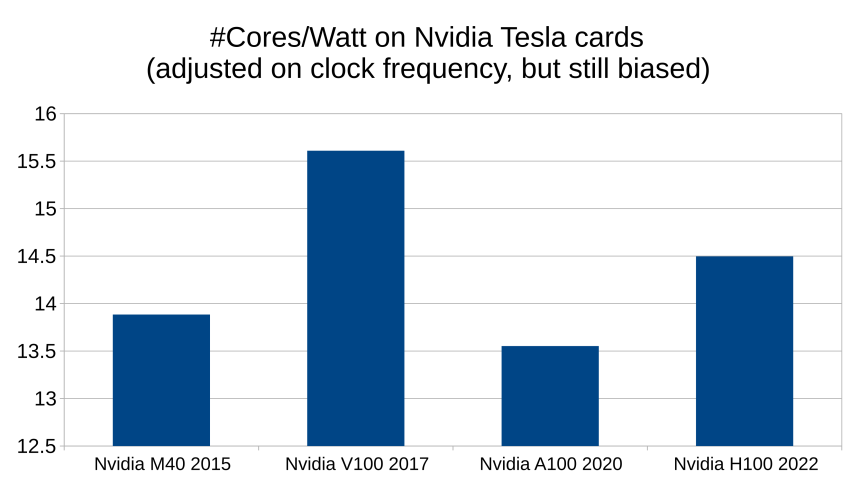
- Next session o

Programmin

Short Evolution of NVIDIA GPUs



Towards Multi-Grid Design



Getting Started

Getting Started

```
ssh -p 8022 ptalbot@access-iris.uni.lu
git clone git@github.com:ULHPC/tutorials.git
cd tutorials/gpu/cuda2023
si-gpu --reservation=hpcschool-gpu
module load system/CUDA
nvcc demo/hello_world.cu -arch=sm_70 -std=c++17 -O3 -o hello_world
./hello_world
```

Find out the driver and CUDA version

Type `nvidia-smi` for the driver
`nvcc --version` for the compiler version (CUDA 11.1 on the HPC).

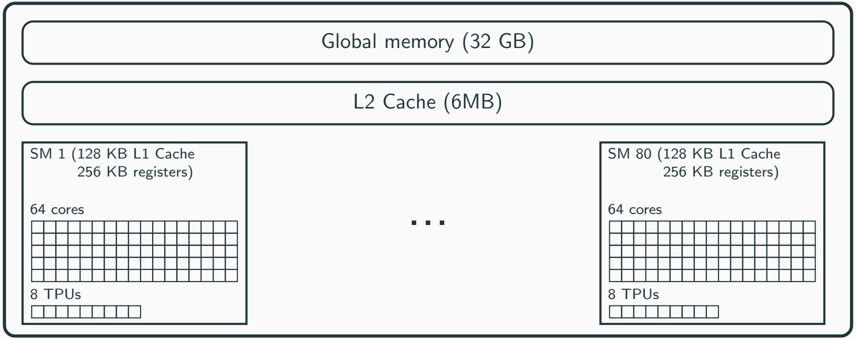
Find out the architecture of your GPU

`https://developer.nvidia.com/cuda-gpus`
For the GPU Nvidia V100 (installed on the HPC), the compute capabilities is 7.0, thus we compile towards this architecture (`-arch=sm_70`).

Hello World (demo/hello_world2.cu)

```
#include <cstdio>

#define CUDIE(result) { \
  cudaError_t e = (result); \
  if (e != cudaSuccess) { \
    printf("%s:%d CUDA runtime error %s\n", __FILE__, __LINE__, cudaGetErrorString(e)); \
  }}

__host__ __device__ void print(const char* msg) {
  printf("%s\n", msg);
}

__global__ void hello_world() {
  print("world");
}

int main(int argc, char** argv) {
  print("hello");
  hello_world<<<1, 1>>>();
  CUDIE(cudaDeviceSynchronize())
  return 0;
}
```

(Simplified) Architecture of the GPU Nvidia V100



5120 cores on a single V100 GPU @ 1290MHz

640 Tensor Processing Units (TPUs)

Whitepaper: `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`

This Session in a Nutshell

This session follows the hierarchical architecture of GPUs.
From programming a single core to programming the full grid.

1. Execute a program on a single thread.
2. Execute a program on a streaming multiprocessor (data parallelism).
3. Execute a program on a full GPU grid (task parallelism).
4. Going further (shared memory, common mistakes and idioms).
5. Tools and documentation.

Single Threaded Program

Running Example: All-Pairs Shortest Path Problem

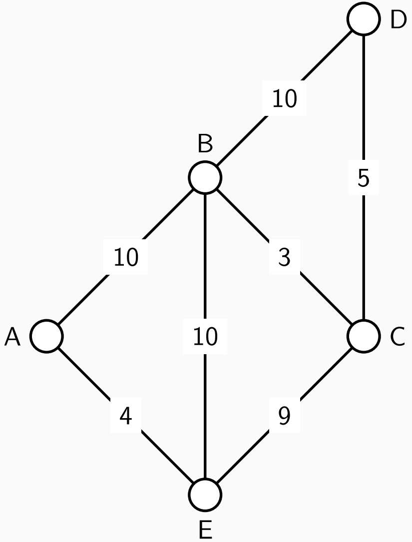
Floyd-Warshall algorithm computes all shortest paths between any pair of nodes ($\mathcal{O}(n^3)$). It takes a node k, then for each pair of nodes (i, j), compute $min(d[i][j], d[i][k] + d[k][j])$, and repeats for all nodes k.

```
void floyd_warshall(vector<vector<int>>& d) {
  size_t n = d.size();
  for(int k = 0; k < n; ++k)
    for(int i = 0; i < n; ++i)
      for(int j = 0; j < n; ++j)
        if(d[i][j] > d[i][k] + d[k][j])
          d[i][j] = d[i][k] + d[k][j];
}
```

Very useful algorithm beyond shortest paths

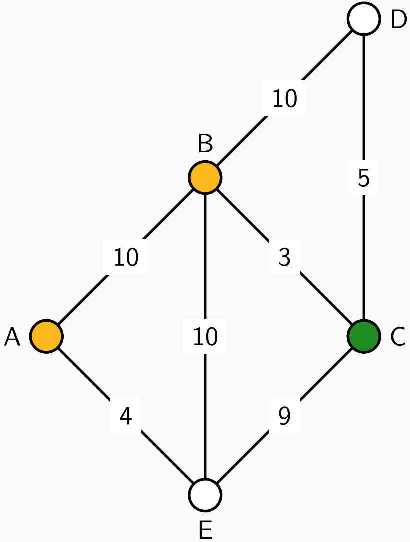
- Solving procedure for system of difference constraints ($\pm x_i \pm y_i \leq k_i$).
- Inversion of real matrices (Gauss-Jordan algorithm).
- Find a regular expression from a finite automaton (Kleene's algorithm).
- ...

All-Pairs Shortest Path Problem Illustrated



	A	B	C	D	E
A	0	10	∞	∞	4
B	10	0	3	10	10
C	∞	3	0	5	9
D	∞	10	5	0	∞
E	4	10	9	∞	0

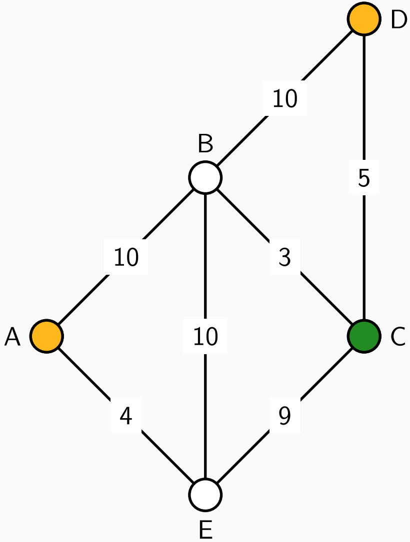
All-Pairs Shortest Path Problem Illustrated



	A	B	C	D	E
A	0	10	∞	∞	4
B	10	0	3	10	10
C	∞	3	0	5	9
D	∞	10	5	0	∞
E	4	10	9	∞	0

k	i	j	$d[i,j]$
C	A	B	$min(10, \infty + 3)$

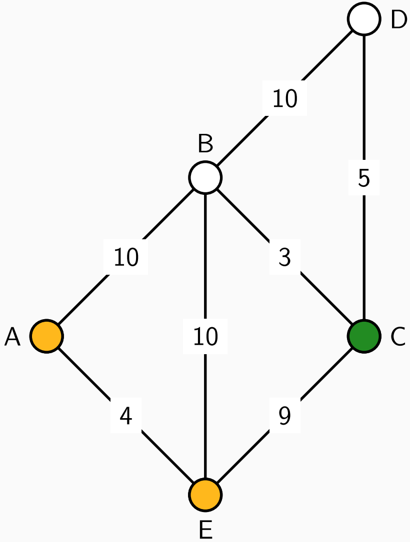
All-Pairs Shortest Path Problem Illustrated



	A	B	C	D	E
A	0	10	∞	∞	4
B	10	0	3	10	10
C	∞	3	0	5	9
D	∞	10	5	0	∞
E	4	10	9	∞	0

k	i	j	$d[i,j]$
C	A	B	$min(10, \infty + 3)$
C	A	D	$min(\infty, \infty + 5)$

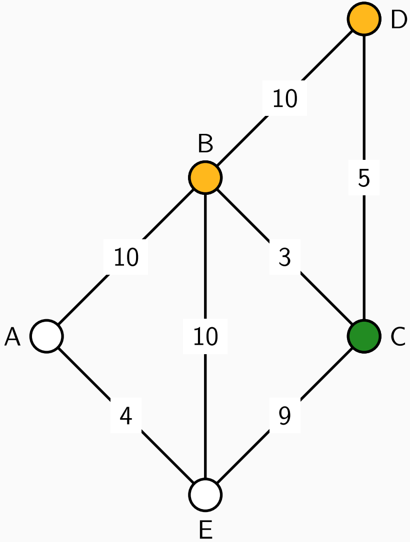
All-Pairs Shortest Path Problem Illustrated



	A	B	C	D	E
A	0	10	∞	∞	4
B	10	0	3	10	10
C	∞	3	0	5	9
D	∞	10	5	0	∞
E	4	10	9	∞	0

k	i	j	$d[i,j]$
C	A	B	$min(10, \infty + 3)$
C	A	D	$min(\infty, \infty + 5)$
C	A	E	$min(\infty, \infty + 9)$

All-Pairs Shortest Path Problem Illustrated



	A	B	C	D	E
A	0	10	∞	∞	4
B	10	0	3	8	10
C	∞	3	0	5	9
D	∞	10	5	0	∞
E	4	10	9	∞	0

k	i	j	$d[i,j]$
C	A	B	$min(10, \infty + 3)$
C	A	D	$min(\infty, \infty + 5)$
C	A	E	$min(\infty, \infty + 9)$
C	B	D	$min(10, 3 + 5)$

...

Running Floyd-Warshall on GPU (`thread_floyd.cu`)

Managed Memory

```
int* array;
CUDIE(cudaMallocManaged(&array, sizeof(int) * 10));
// ...
cudaFree(array);
```

`array` can be used in **both host and device code** (memory transfer will be automatic between CPU and GPU).

Exercise (`thread_floyd.cu`)

- Create a CUDA kernel executing the Floyd-Warshall algorithm.
- Call this kernel.
- Compile and run with different size of the matrix.

Single Block, Many Threads

Block Parallelism

Block

- A block is the "software abstraction" of a streaming multiprocessor, e.g., a thread VS a core.
- Several blocks can be executed on the same SM, and cannot migrate to another SM during execution.
- **Single Instruction Multiple Threads** (SIMT) inside a block: "the threads execute the same instructions when possible".

Special Variables

- `threadIdx.x`: Index of the current thread inside a block.
- `blockDim.x`: Number of threads per block.
- `blockIdx.x`: Index of the current block inside the grid (useful later).
- `gridDim.x`: Number of blocks in the grid (useful later).

Launching a kernel:

```
my_kernel<<<number_of_blocks, threads_per_block>>>();
```

Block Parallelism: Find the Minimum in an Array

Each thread computes its local min (*map*), then we compute the min of all local min (*reduce*).

- Map:

$$\underbrace{\boxed{3}\boxed{22}\boxed{10}\boxed{23}}_{\text{Thread 0, } m_0 = 3}\underbrace{\boxed{21}\boxed{7}\boxed{91}\boxed{1}}_{\text{Thread 1, } m_1 = 1}\underbrace{\boxed{3}\boxed{10}\boxed{42}\boxed{11}}_{\text{Thread 2, } m_2 = 3}\underbrace{\boxed{8}\boxed{7}\boxed{32}}_{\text{Thread 3, } m_3 = 7}$$

- Reduce: $min([3, 1, 3, 7]) = 1$.

Block Parallelism: Find the Minimum in an Array

Intuitive implementation `demo/block_min.cu`

```
__global__ void parallel_min(int* v, size_t n, int* local_min) {
  local_min[threadIdx.x] = INT_MAX;
  size_t m = n / blockDim.x + (n % blockDim.x != 0);
  size_t from = threadIdx.x * m;
  size_t to = min(n, from + m);
  for(size_t i = from; i < to; ++i) {
    local_min[threadIdx.x] = min(local_min[threadIdx.x], v[i]);
  }
}
```

Iteration 1:

- Map:

$$\underbrace{\boxed{3}\ 22\ 10\ 23}_{\text{Thread 0, } m_0 = 3}\ \underbrace{\boxed{21}\ 7\ 91\ 1}_{\text{Thread 1, } m_1 = 1}\ \underbrace{\boxed{3}\ 10\ 42\ 11}_{\text{Thread 2, } m_2 = 3}\ \underbrace{\boxed{8}\ 7\ 32}_{\text{Thread 3, } m_3 = 7}$$

- Reduce: $min([3, 1, 3, 7]) = 1$.

Block Parallelism: Find the Minimum in an Array

Intuitive implementation `demo/block_min.cu`

```
__global__ void parallel_min(int* v, size_t n, int* local_min) {
  local_min[threadIdx.x] = INT_MAX;
  size_t m = n / blockDim.x + (n % blockDim.x != 0);
  size_t from = threadIdx.x * m;
  size_t to = min(n, from + m);
  for(size_t i = from; i < to; ++i) {
    local_min[threadIdx.x] = min(local_min[threadIdx.x], v[i]);
  }
}
```

Iteration 2:

- Map:

3	22	10	23	21	7	91	1	3	10	42	11	8	7	32
Thread 0, $m_0 = 3$				Thread 1, $m_1 = 1$				Thread 2, $m_2 = 3$				Thread 3, $m_3 = 7$		

- Reduce: $min([3, 1, 3, 7]) = 1$.

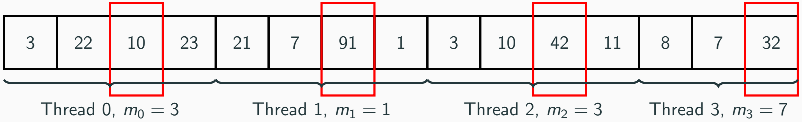
Block Parallelism: Find the Minimum in an Array

Intuitive implementation `demo/block_min.cu`

```
__global__ void parallel_min(int* v, size_t n, int* local_min) {
  local_min[threadIdx.x] = INT_MAX;
  size_t m = n / blockDim.x + (n % blockDim.x != 0);
  size_t from = threadIdx.x * m;
  size_t to = min(n, from + m);
  for(size_t i = from; i < to; ++i) {
    local_min[threadIdx.x] = min(local_min[threadIdx.x], v[i]);
  }
}
```

Iteration 3:

- Map:



- Reduce: $min([3, 1, 3, 7]) = 1$.

Block Parallelism: Find the Minimum in an Array

Intuitive implementation `demo/block_min.cu`

```
__global__ void parallel_min(int* v, size_t n, int* local_min) {
  local_min[threadIdx.x] = INT_MAX;
  size_t m = n / blockDim.x + (n % blockDim.x != 0);
  size_t from = threadIdx.x * m;
  size_t to = min(n, from + m);
  for(size_t i = from; i < to; ++i) {
    local_min[threadIdx.x] = min(local_min[threadIdx.x], v[i]);
  }
}
```

Iteration 4:

- Map:

3	22	10	23	21	7	91	1	3	10	42	11	8	7	32

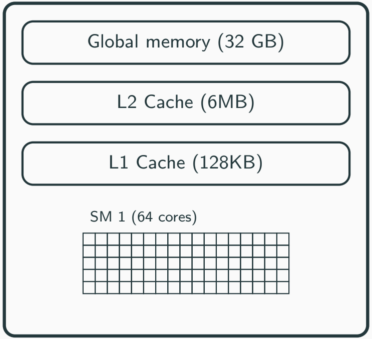
Thread 0, $m_0 = 3$ Thread 1, $m_1 = 1$ Thread 2, $m_2 = 3$ Thread 3, $m_3 = 7$

- Reduce: $min([3, 1, 3, 7]) = 1$.

Optimization: Contiguous Memory Accesses

Knowing about the hardware is crucial for efficiency.

- Previous implementation can work well on CPU since each core has its own cache.
- On GPU, it is better to access the memory contiguously—it allows the GPU to move data from global memory to cache faster.



Optimization: Contiguous Memory Accesses

Strided implementation `demo/block_min.cu`

```
__global__ void parallel_min_stride(int* v, size_t n, int* local_min) {
  local_min[threadIdx.x] = INT_MAX;
  for(size_t i = threadIdx.x; i < n; i += blockDim.x) {
    local_min[threadIdx.x] = min(local_min[threadIdx.x], v[i]);
  }
}
```

3	22	10	23	21	7	91	1	3	10	42	11	8	7	32
T_0	T_1	T_2	T_3	T_0	T_1	T_2	T_3	T_0	T_1	T_2	T_3	T_0	T_1	T_2

Very important: up to an order of magnitude faster (10x).

Optimization: Contiguous Memory Accesses

Strided implementation `demo/block_min.cu`

```
__global__ void parallel_min_stride(int* v, size_t n, int* local_min) {
  local_min[threadIdx.x] = INT_MAX;
  for(size_t i = threadIdx.x; i < n; i += blockDim.x) {
    local_min[threadIdx.x] = min(local_min[threadIdx.x], v[i]);
  }
}
```

3	22	10	23	21	7	91	1	3	10	42	11	8	7	32
T_0	T_1	T_2	T_3	T_0	T_1	T_2	T_3	T_0	T_1	T_2	T_3	T_0	T_1	T_2

Very important: up to an order of magnitude faster (10x).

Optimization: Contiguous Memory Accesses

Strided implementation `demo/block_min.cu`

```
__global__ void parallel_min_stride(int* v, size_t n, int* local_min) {
  local_min[threadIdx.x] = INT_MAX;
  for(size_t i = threadIdx.x; i < n; i += blockDim.x) {
    local_min[threadIdx.x] = min(local_min[threadIdx.x], v[i]);
  }
}
```

3	22	10	23	21	7	91	1	3	10	42	11	8	7	32
T_0	T_1	T_2	T_3	T_0	T_1	T_2	T_3	T_0	T_1	T_2	T_3	T_0	T_1	T_2

Very important: up to an order of magnitude faster (10x).

Optimization: Contiguous Memory Accesses

Strided implementation `demo/block_min.cu`

```
__global__ void parallel_min_stride(int* v, size_t n, int* local_min) {
  local_min[threadIdx.x] = INT_MAX;
  for(size_t i = threadIdx.x; i < n; i += blockDim.x) {
    local_min[threadIdx.x] = min(local_min[threadIdx.x], v[i]);
  }
}
```

3	22	10	23	21	7	91	1	3	10	42	11	8	7	32
T_0	T_1	T_2	T_3	T_0	T_1	T_2	T_3	T_0	T_1	T_2	T_3	T_0	T_1	T_2

Very important: up to an order of magnitude faster (10x).

Running Floyd-Warshall on GPU (`block_floyd.cu`)

Exercise (`block_floyd.cu`)

- Parallelize the most inner loop of the algorithm (`for(int j = ...)`) using contiguous memory accesses.
- Compile and run with different size of the matrix and threads-per-block numbers.

Correctness of the algorithm

For a fixed k:

- **No write conflict**: each iteration modifies a different memory cell, namely `d[i][j]`.
- **No read conflict**: each iteration reads cell that are not written in, namely `d[i][k]` and `d[k][j]`.
- **No enforced order**: within the two nested loops, the order of the operations does not matter.

Synchronizing Threads in a Block

Threads are not necessarily always synchronized, actually every thread has its own instruction counter (*independent thread scheduling*).

- **Block barrier**: a thread reaching `__syncthreads()` is blocked until all threads of the current block reach this barrier.
- **Be careful**: it is possible to deadlock if all threads do not reach this barrier.

Exercise

Add the instruction `__syncthreads();` to avoid a thread to start the $k+1$ iteration before all threads have finished the k^{th} iteration.

Single Grid, Many Blocks, Many Threads

Grid Parallelism: Find the Minimum in an Array

We divide the array into as many slices as blocks, and then solve each slice on each block.

demo/grid_min.cu

```
__global__ void grid_min(int* v, size_t n, int* local_min) {
  size_t m = n / gridDim.x + (n % gridDim.x != 0);
  size_t begin = blockIdx.x * m;
  size_t end = min(n, begin + m);
  block_min_stride(v, begin, end, local_min);
}
```

Parallel Execution of Many Blocks

Exercise (`grid_floyd.cu`)

- Parallelize the middle loop of the algorithm (`for(int i = ...)`) using several blocks.

Parallel Execution of Many Blocks

Exercise (`grid_floyd.cu`)

- Parallelize the middle loop of the algorithm (`for(int i = ...)`) using several blocks.

▶ `__syncthreads()` **does not synchronize across blocks**.

▶ We need a **barrier across blocks**: simply wait for the kernel to terminate, `cudaDeviceSynchronize()` acts as a barrier in host code.

Exercise continued (`grid_floyd.cu`)

- Create a CUDA kernel executing the k^{th} iteration of the Floyd-Warshall algorithm.
- Call this kernel in a loop from $k = 0$ to $k = n - 1$.
- Compile and run with different size of the matrix, threads-per-block and numbers of block.

Going Further

Shared Memory

A fast but small memory allocated per block.

Additional parameter to the kernel launch to say how much you want:

```
kernel<<<1, 1, 10 * sizeof(int)>>>(10);
```

Then you can use it in the kernel as:

```
__global__ void kernel(int n) {
  extern __shared__ int shared_mem[]; // block of memory of size ‘n‘.
  // ...
}
```

See `demo/block_shared_floyd.cu`.

Kernel Parameter Passing Semantics (14.5.10.3)

Very awkward semantics (several copies of objects, destructor might be called before kernel termination, **pass-by-reference does not work**, etc.).

- You cannot rely on it to pass objects.

Pass only primitive types or pointers to arrays or objects **by copy** allocated in global memory.

Polymorphism

You cannot initialize a hierarchy of classes (using virtual methods) on the host side (even in managed memory), and then transfer it to the device.

- The *vtable* is not copied, and thus stay initialized in host (segfault if used on device).

Idiom: Initialization Within Kernel

How to declare and initialize data within the kernel that is:

- **Shared among threads in a block?** Use *shared memory* (see `demo/block_shared_min.cu`).
- **Shared among threads in the grid?** Not possible, you must declare it beforehand:

```
struct SharedData {
  // Data shared among all threads in the grid.
};
__global__ void init_data(SharedData* data) {
  // Run this kernel with 1 thread / 1 block to initialize the data.
}
__global__ void kernel(SharedData* data) {
  // Run the kernel with the data initialized.
}
```

Tools and Documentation

Your CUDA Friends

- **CUDA docs**: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`
- **GTC conference**: `https://www.nvidia.com/en-us/on-demand`
- **CUDA debugger**: Directly in VS Code, or using gdb-like command line.
- **CUDA memory analyzer** (like Valgrind): `compute-sanitizer ./a.out`
- **Nsight**: Profiler, really nice interface in Windows Visual Studio.

See more tools at `https://developer.nvidia.com/tools-overview`.

Advice: spend time learning the tools!

CUDA Battery library

Get your `vector`, `shared_ptr`, etc. with various allocators working on the GPU!

`https://github.com/lattice-land/cuda-battery`

GTC Selected Videos

GTC website: `https://www.nvidia.com/en-us/on-demand`

Beginners and Tools

- How CUDA Programming Works [a41101]
- From the Macro to the Micro: CUDA Developer Tools Find and Fix Problems at Any Scale [s51205]
- Debugging CUDA: An Overview of CUDA Correctness Tools [s51772]
- Measure Right! Best Practices when Benchmarking CUDA Applications [s51334]

New and Advanced Features

- CUDA: New Features and Beyond [s51225]
- CUDA Graphs 101 [s51211]

Standard C++

- C++ Standard Parallelism [s51755]
- Accelerating HPC applications with ISO C++ on Grace Hopper [s51054]