Getting started with CUDA
Part 2 - Host view of GPU computation

Edwin Carlinet, Joseph Chazalon {firstname.lastname@lrde.epita.fr}

April 2020

EPITA Research & Development Laboratory (LRDE)

Host view of GPU computation

Calling kernels, not writing them

You do not need to write kernels to run CUDA code:
you can use kernels from a library, written by someone else.

This section is about how to properly launch CUDA kernels using their API only.

Sequential and parallel sections

We use the GPU(s) as co-processor(s).

Our program is made of a series of sequential and parallel sections.

Of course, CPU code can be multi-threaded too!

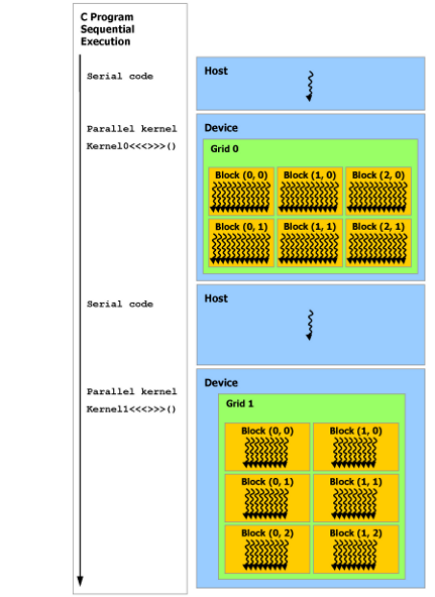

Figure 1: Heterogeneous programming

Host *vs* device: reminder

We need to transfer inputs from the host to the device and outputs the other way around.

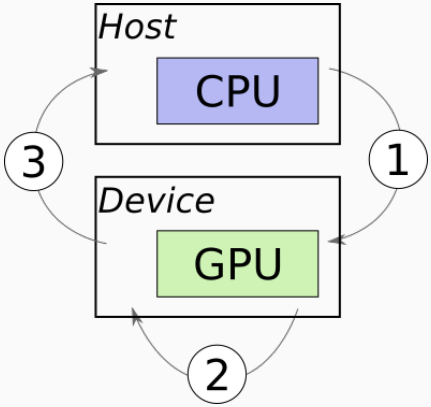

Figure 2: Computation on separate device

A proper kernel invocation

Let's fix this code!

```
#include <cuda.h>
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
  int size = n * sizeof(float);
  float *d_A, *d_B, *d_C;

  // 1.1 Allocate device memory for A, B and C
  // 1.2 Copy A and B to device memory

  // 2. Launch kernel code - computation done on device

  // 3. Copy C (result) from device memory
  // Free device vectors
}
```

CUDA memory primitives

	1D	2D	3D
Allocate	`cudaMalloc()`	`cudaMallocPitch()`	`cudaMalloc3D()`
Copy	`cudaMemcpy()`	`cudaMemcpy2D()`	`cudaMemcpy3D()`
On-device init.	`cudaMemset()`	`cudaMemset2D()`	`cudaMemset3D()`
Reclaim		`cudaFree()`	

. . . plus many others detailed in the CUDA Runtime API documentation. . .

Why 2D and 3D variants?

- Strong alignment requirements in device memory
  - Enables correct loading of memory chunks to SM caches (correct bank alignment)
- Proper striding management in automated fashion

Host ↔ Device memory transfer

We just need the three following ones for now:

```
cudaError_t cudaMalloc ( void** devPtr, size_t size_in_bytes )
```

Allocates space in the **device global** memory.

```
cudaError_t cudaMemcpy ( void* dst, const void* src, size_t size_in_bytes,
                         cudaMemcpyKind kind )
```

Asynchronous data transfer. `cudaMemcpyKind` ≈ copy direction:

- `cudaMemcpyHostToHost`
- `cudaMemcpyHostToDevice` ← **useful**
- `cudaMemcpyDeviceToHost` ← **useful**
- `cudaMemcpyDeviceToDevice`
- `cudaMemcpyDefault` ← *Direction inferred from pointer values. Requires unified virtual addressing.*

```
cudaError_t cudaFree ( void* devPtr )
```

Almost complete code

```
#include <cuda.h>
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
  int size = n * sizeof(float);
  float *d_A, *d_B, *d_C;
  // 1.1 Allocate device memory for A, B and C
  cudaMalloc((void **) &d_A, size); // TODO repeat for d_B and d_C
  // 1.2 Copy A and B to device memory
  cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
  // 2. Launch kernel code - computation done on device
  k_VecAdd<<<NB, NT>>>(d_A, d_B, d_C);  // FIXME How to compute NB and NT?
  // 3. Copy C (result) from device memory
  cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
  // Free device vectors
  cudaFree(d_A); // TODO repeat for d_B and d_C
}
```

Checking errors

In practice, we need to check for API errors

```
cudaError_t err = cudaMalloc((void **) &d_A, size);
if (err != cudaSuccess)  {
    printf("%s in %s at line %d\n",
           cudaGetErrorString(err),
           __FILE__,
           __LINE__);
    exit(EXIT_FAILURE);
}
```

Intermission: *Can I use memory management functions inside kernels?*

No: `cudaMalloc()`, `cudaMemcpy()` and `cudaFree()` shall be called from host only.

However, kernels may allocate, use and reclaim memory dynamically using regular `malloc()`, `memset()`, `memcpy()` and `free()` functions.

Note that if some device code allocates some memory, it must free it.

Fix the kernel invocation line

We want to fix this line:

```
k_VecAdd<<<NB, NT>>>(d_A, d_B, d_C);
```

Kernel invocation syntax:

```
kernel<<<blocks, threads_per_block, shmem, stream>>>(param1, param2, ...);
```

- `blocks`: number of blocks in the grid;
- `threads_per_block`: number of threads for each block;
- `shmem`: (opt.) amount of shared memory to allocate (in bytes);
- `stream`: (opt.) CUDA stream (not discussed in this course, see the documentation).

How to set gridDim and BlockDim properly?

Lvl 0: Naive trial with as many threads as possible

```
k_VecAdd<<<1, size>>>(d_A, d_B, d_C);
```

How to set gridDim and BlockDim properly?

Lvl 0: Naive trial with as many threads as possible

```
k_VecAdd<<<1, size>>>(d_A, d_B, d_C);
```

Will fail with large vectors.

Hardware limitation on the maximum number of threads per block (1024 for Compute Capability 3.0-7.5).

Will fail with vectors of size which is not a multiple of warp size.

Lvl 1: It works with just enough blocks

```
// Get max threads per block
int devId = 0; // There may be more devices!
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, devId);
printf("Maximum grid dimensions: %d x %d x %d\n",
        deviceProp.maxGridSize[0],
        deviceProp.maxGridSize[1],
        deviceProp.maxGridSize[2]);
printf("Maximum block dimensions: %d x %d x %d\n",
        deviceProp.maxThreadsDim[0],
        deviceProp.maxThreadsDim[1],
        deviceProp.maxThreadsDim[2]);
// Compute the number of blocks
int xThreads = deviceProp.maxThreadsDim[0];
dim3 DimBlock(xThreads, 1, 1); // 1D VecAdd
int xBlocks =  (int) ceil(n/xThreads);
dim3 DimGrid(xBlocks, 1, 1);
// Launch the kernel
k_VecAdd<<<DimGrid, DimBlock>>>(d_A, d_B, d_C);
```

Lvl 2: Tune block size given kernel requirements and hardware constraints
It is important to understand the difference between:

- the logical decomposition of your program:
  problem $\approx$ grid $\rightarrow$ blocks $\rightarrow$ threads
- the scheduling of the computation on the hardware:
  - assignment of each block to a *Streaming Multiprocessor* (SM)
  - groups threads into *warps*
  - run groups of *warps* concurrently

The hardware constraints are different between each *Compute Capability* version. See the CUDA C programming manual, Appendix H for details about each hardware version.
In particular, the amount of memory available on each SM may limit the number of threads one would actually want to launch...

But this depends on the kernel code!
The **CUDA Occupancy Calculator** APIs are designed to assist programmers in choosing the best number of threads per block based on register and shared memory requirements of a given kernel.

But wait...

```
#include <stdio.h>
__global__ void print_kernel() {
    printf("Hello!\n");
}

int main() {
    print_kernel<<<1, 1>>>();

}
```

This code prints nothing!

Kernel invocation is asynchronous

```
#include <stdio.h>
__global__ void print_kernel() {
    printf("Hello!\n");
}

int main() {
    print_kernel<<<1, 1>>>();
    cudaDeviceSynchronize();
}
```

Host code synchronization requires `cudaDeviceSynchronize()` because **kernel invocation is asynchronous** from host perspective.

On the device, kernel invocations are strictly sequential (unless you schedule them on different *streams*).

Intermission: *Can I call kernels inside kernels?*

Yes: This is the basis of *dynamic parallelism*.

Some restrictions over the stack size apply.
Remember that the device runtime is a functional subset of the host runtime, ie you can perform device management, kernel launching, device memcpy, etc., but with some restrictions (see the documentation for details).

The compiler may inline some of those calls, though.

Conclusion about the host-only view

A host-only view of the computation is sufficient for most of the cases:

1. upload input data to the device
2. fire a kernel
3. download output data from the device

Advanced CUDA requires to make sure we saturate the SMs, and may imply some kernel study to determine the best:

- amount of threads per blocks
- amount of blocks per grid
- work per thread (if applicable)
- ...

This depends on:

- hardware specifications: maximum `gridDim` and `blockDim`, etc.
- kernel code: amount of register and shared memory used by each thread