Basics of GPUs

Alejandro Campos

February 21, 2024

Contents

1 Introduction

There are two key terms used when dealing with heterogeneous architectures, namely, the "host" and the "device". The host refers to the CPU available on the system, and the device to the GPU.

In these notes we'll rely on MFEM to handle algorithms on the GPU. For example, MFEM will be used to move data between host and device (Sec. 2) and to execute for-loops on the GPU (Sec. 3).

2 Memory management

Host memory resides in the CPU, and device memory in the GPU. In many systems, memory allocated on the host can only be accessed by the CPU, and memory allocated on the device can only be accessed by the GPU. Thus, if you have an array that lives in host memory and want to alter its entries on the GPU, you first need to transfer it to device memory.

This can be performed using MFEM's memory manager. Below are three ways we can transfer data from the host to the device.

```
1    mfem::Vector rho(...);
2    // you can now use the vector on the CPU, e.g. you can initialize its
     entries, etc.
3
4    double* d_rho = rho.Read();
```

```
    // the pointer d_rho can now be used on the GPU, only to read the entries
    of the rho vector
    // E.g.
    // double val = d_rho[nth_entry];

    double* d_rho = rho.Write();
    // the pointer d_rho can now be used on the GPU, only to write to the
    entries of the rho vector
    // E.g.
    // d_rho[nth_entry] = 1.234;
    // note: if you try to read one of the entries on the GPU, you might get an
     unexpected value

    double* d_rho = rho.ReadWrite();
    // the pointer d_rho can now be used on the GPU, to either read or write to
     the entries of the rho vector
```

If the data is currently residing on device memory, one can bring it back to host memory by using the analogues of the above. That is,

```
    rho.HostRead();

    rho.HostWrite();

    rho.HostReadWrite();
```

You can then continue to use the `rho` object on the CPU as you would normally do.

Let's say you had data on the device, and altered its value. You now want to access this data on the host but forget to do a `HostRead()` or a `HostReadWrite()`. How would you catch this faux pas? You can compile MFEM in debug mode and then run the code. This will lead to a runtime error and a stack trace since the host is now possibly out-of-sync with the device. On the other hand, there are no checks for the reverse, that is, to ensure the device memory is valid after modified on the host. This is because you are using raw pointers on the device and not getting the pointer indirectly though a trackable object as is done on the host. That being said, as long as you use a "fresh" device pointer every single time you want to modify data on the GPU, then you should be good. In other words, always use a `Read()`, `Write()`, or `ReadWrite()` if you aren't sure the device pointer you'll be working with is valid.

Finally, you can also explicitly grab pointers to access data on the host, in a similar manner to the pointers for device data:

```
    double* h_rho = density.HostRead();

    double* h_rho = density.HostWrite();

    double* h_rho = density.HostReadWrite();
```

3 Kernel execution

A kernel is a basic unit of code to be executed on the GPU (e.g. a loop, function, or program). In this section we'll focus on kernels that consist of for-loops. To execute a for-loop on the GPU, one invokes one of the various macros available in MFEM. The standard for-loop macro is shown below:

```
    mfem::forall(N, [=] MFEM_HOST_DEVICE (int i) \{\dots \} )
```

The second argument above, i.e. `[=] MFEM_HOST_DEVICE (int i) {...}`, is a lambda expression, which is essentially a small function without a name. The body of the lambda expression,

i.e. the code to be executed, is depicted by the dots .... The first argument of the lambda expression, i.e. `[=]`, is the capture clause. It is used to indicate how variables in the enclosing scope should be used, or "captured", within the for loop. For example

- `[=]` capture variables by value
- `[&]` capture variables by reference
- `[]` do not access any variables in the enclosing scope

The second argument, `MFEM_HOST_DEVICE`, allows the lambda expression to be called from both the host and the device. The third argument in the lambda expression is the parameter list. These are essentially inputs to the lambda expression that can be used within its body.

An specific example on how to use `mfem::forall` is as follows. Imagine you have the following standard for-loop code

```
mfem::Vector rho(...);

for (int i = 0; i < rho.Size(); i++)
{
    rho[i] = 1.234;
}
```

Re-writing the above using the mfem macro would look like this

```
mfem::Vector rho(...);

double *d_rho = rho.Write()

mfem::forall(rho.Size(), [=] MFEM_HOST_DEVICE (int i)
{
    d_rho[i] = 1.234;
});
```

For the above, the GPU will launch multiple blocks, each consisting of various threads. Each thread in a block takes care of a single instance of the for loop. That is, one thread will execute the contents within the for-loop for value `i=0`, a second thread will execute the same code for `i=1` and so on till one last thread takes care of `i=rho.Size()-1`.

Since lambda expressions can capture variables from the enclosing scope only, a lambda expression defined within a member function of a class cannot capture a member variable of that same class. For this scenario, one needs to use a local version of that variable, as shown below

```
mfem::Vector rho(...);

const double factor = 10.0;
const double rho_min = rho_min_; // here rho_min_ is a member variable of a
 class

double *d_rho = rho.Write()

mfem::forall(rho.Size(), [=] MFEM_HOST_DEVICE (int i)
{
    d_rho[i] = factor * density_min;
});
```

In the example above, if we had tried to use `rho_min_` directly the device lambda would have captured and tried to dereference the `this` pointer (`this->rho_min_`), which is non-portable (or would result in extra memory movement if it was portable). We note that with nvcc (the

NVIDIA compiler) you can compile a device kernel that uses `rho_min_` directly, but if you run with `--atsdisable` it will segfault. With ROCm+hip (one of the AMD compilers) you will get a compilation error.

Another key difference between standard for-loops and `mfem::forall` is the way in which one exits the loop. For example, consider the following standard for-loop in which specific iterations are skipped if a condition is met

```
for (int i = 0; i < N; ++i)
{
    // skip to next iteration if condition is met
    if (condition to be met)
    {
        continue;
    }
    ...
}
```

The equivalent for a GPU kernel would be

```
mfem::forall(N, [=] MFEM_HOST_DEVICE (int i)
{
    // skip to next iteration if condition is met
    if (condition to be met)
    {
        return;
    }
    ...
});
```

That is, we use `return` rather than `continue`. The reason for this is that the GPU version of code is essentially a for-loop that calls lambda expressions, i.e.

```
// mfem::forall(N, lambda ) :=
for (int iter = 0; iter < N: iter++)
{
    lambda(iter);
}
```

Thus, what you want to do is "return" from `lambda(iter)` so that the `iter` iterator can go on to its next value.

4 Software vs hardware

First we'll focus on the software side. A thread executes an instance of the kernel. By this we mean the thread can execute an instance of a for-loop (note that this for-loop can have sub for-loops or not). A thread block is a set of concurrently executing threads that can cooperate among themselves through barrier synchronization and shared memory. A thread has a thread ID within its thread block. A grid is an array of thread blocks that execute the same kernel, read inputs from global memory, write results to global memory, and synchronize between dependent kernel calls. A thread block has a block ID within its grid.

On the hardware side, the GPU is formed by many streaming multiprocessors (Nvidia) or compute units (AMD), and each streaming multiprocessor/compute unit contains multiple GPU cores. From the NVIDIA Fermi Compute Architecture Whitepaper: "a GPU executes one or more kernel grids; a streaming multiprocessor (SM) executes one or more thread blocks; and CUDA cores and other execution units in the SM execute threads." A summary of these terms is shown in the following table

Software	Hardware
thread	core
thread block	SM/CP
grid	GPU

5 Memory spaces

- Global
- Local
- Unified/Managed
- Temporary
- Shared

6 Hierarchical parallelism

7 Debugging

8 Profiling