Basics of GPUs

Alejandro Campos

March 3, 2024

Contents

1 Introduction **1**

2 Memory management **1**

3 Kernel execution **3**

4 Memory spaces **5**

5 Hierarchical parallelism **5**

6 Software vs hardware **6**

7 Debugging **7**
7.1 Nvidia/CUDA 7
7.2 AMD/HIP 7

8 Profiling **7**
8.1 Nvidia/CUDA 7
8.2 AMD/HIP 7

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.

```
mfem::Vector rho(...);
// you can now use the vector on the CPU, e.g. you can initialize its entries, etc.

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();
```

The `rho` object can then be used on the CPU again as one would normally do. On the other hand, one can also generate explicit pointers to access data on the host, in a similar manner to the pointers for device data:

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

double* h_rho = rho.HostWrite();

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

MFEM objects (array, vector, dense matrix, dense tensor) have two member variables used to determine its state, these are `VALID_HOST` and `VALID_DEVICE`. When one first creates an object, MFEM will set `VALID_HOST` to true and `VALID_DEVICE` to false. After that, if you call any of the memory-movement functions (e.g. `Read()`, `HostWrite()`), then `VALID_HOST` and `VALID_DEVICE` would be updates as shown on the table below.

	VALID_HOST	VALID_DEVICE
Read()	does nothing	true
Write()	false	true
ReadWrite()	false	true
HostRead()	true	does nothing
HostWrite()	true	false
HostReadWrite()	true	false

Let's now consider a few specific cases:

- You create an object, then call `Write()` or `ReadWrite()` to modify its value on the device. At this point `VALID_HOST` would be false and `VALID_DEVICE` would be true. You now read

this data on the host but forget beforehand to do `HostRead()` or a `HostReadWrite()`. To catch this mistake, 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.

- You create an object, then call `Read()` to read its value on the device. At this point `VALID_HOST` is still true and `VALID_DEVICE` is also true. Let's say that on the host you now want to only read the entries of the object using the `()` operator, e.g `double new_var = 2 + rho(i)`. This is perfectly reasonable since the device has not modified the entries of the object. When using the parenthesis operator, however, MFEM doesn't automatically know that you are going to only read this entry, rather than modify it, e.g i.e. `rho(i)=new_value`. If you were to modify it, its value on the device would no longer be valid and hence the status of the flag `VALID_DEVICE`, which was left as true by the `()` operator, is no longer correct. MFEM thus provides two versions of the `()` operator, one that is used for reading entries only and another that is used for reading and modifying entries. The one used for reading entries can be used when `VALID_HOST` and `VALID_DEVICE` are both true, whereas the one for modifying entries requires the object to previously have `VALID_DEVICE` set to true and `VALID_DEVICE` to false. To use the former, you use the `mfem::AsConst()` function, as in `mfem::AsConst(rho)(i)`. To use the latter, you just use the `()` operator as usual.

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:

```
1    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

```
1    mfem::Vector rho(...);
2
3    for (int i = 0; i < rho.Size(); i++)
4    {
5        rho[i] = 1.234;
6    }
```

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 thread blocks, which are collections of 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 Memory spaces

- Global: GPU memory that is typically dynamically allocated (with malloc/free or something similar) and is accessible by all threads, from all thread blocks. This means that a thread can read and write any value in global memory.
- Local: GPU memory that is typically statically allocated from within a kernel. It is only visible, and therefore accessible, by the thread allocating it. All threads executing a kernel will have their own privately allocated local memory.
- Unified/Managed: unified memory (same as managed memory) is a single global memory buffer that can be accessed and used in multiple memory spaces (e.g. host and device). The memory will automatically migrate or stay in sync if it is not where it needs to be. Unified memory is allocated with special routines, e.g. `cudaMallocManaged()` or `hipMallocManaged()`. It can also be a lower type of memory because the memory copy won't happen until one of the spaces "faults" on the memory address.
- Temporary
- Shared

5 Hierarchical parallelism

Let's say one has loops within loops, and wants to execute all of them on the GPU. One way to do so is as following

```
    Vector rho;
    rho.UseDevice(true);

    const int size_1 = ...;
    const int size_2 = ...;
    const int size_3 = ...;
    const int size_4 = ...;

    auto d_rho = Reshape(rho.Write(), size_1, size_2, size_3, size_4);

    mfem::forall(size_4, [=] MFEM_HOST_DEVICE (int l)
    {
        for (int k = 0; k < size_3; k++)
```

```
            for (int j = 0; j < size_2; j++)
               for (int i = 0; i < size_1; i++)
                  d_rho(i,j,k,l) = ...;
   });
```

For this case, the GPU will launch `size_4` threads, and each will execute all of the `i`, `j`, and `k` instances for a single value of `l`. One can expose a higher level of parallelism by launching more threads, namely `size_1 x size_2 x size_ 3 x size_4`, so that each thread has to execute only a single combination of the `i`, `j`, `k`, and `l` indices. One can do this as follows

```
   Vector rho;
   rho.UseDevice(true);

   const int size_1 = ...;
   const int size_2 = ...;
   const int size_3 = ...;
   const int size_4 = ...;

   auto d_rho = Reshape(rho.Write(), size_1, size_2, size_3, size_4);

   mfem::forall_3D(size_4, size_1, size_2, size_3, [=] MFEM_HOST_DEVICE (int l
   )
   {
       MFEM_FOREACH_THREAD(k, thread_id_3, size_3)
          MFEM_FOREACH_THREAD(j, thread_id_2, size_2)
             MFEM_FOREACH_THREAD(i, thread_id_1, size_1)
                d_rho(i,j,k,l) = ...;
   });
```

In the first example above the GPU launches `size_4 / MFEM_CUDA_BLOCKS` thread blocks, each with `MFEM_CUDA_BLOCKS` threads, for a total of `size_4` threads. On the second example, the GPU launches `size_4` thread blocks, each with `size_1 x size_2 x size_ 3` threads, for a total of `size_1 x size_2 x size_ 3 x size_4`.

6 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

7 Debugging

7.1 Nvidia/CUDA

`cuda-memcheck` and `compute-sanitizer` are two debugging tools for CUDA, which are essentially the same, with the latter being a newer version of the former. Each tool is essentially four tools, which can be chosen at runtime. The four options are

- `memcheck`
- `syncheck`
- `racecheck`
- `initcheck`

7.2 AMD/HIP

8 Profiling

NOTE : For both NVIDIA/CUDA and AMD/HIP, don't forget to limit the number of cycles in the simulations that you want to profile (e.g. by setting the number of time cycles to a sufficiently small number, such as ten or twenty). The more cycles you run, the more cumbersome profilers for both architectures can become, and might not even be able to display the needed results.

8.1 Nvidia/CUDA

We recommend you use `nsys` (CLI tool) for profiling and `nsys-ui` (GUI tool) to look at the results but you can use the deprecated nvprof and nvvp tools if you experience issues with `nsys`. There are two main types of profiling when it comes to CUDA, the timeline profile and the performance counters profile.

`nsys-ui` supports kernel renaming and can be toggled from the GUI: Tools/Options.../Report Behavior/Rename CUDA Kernels by NVTX.

To use `nsys` make sure the executable is in your path. If it is not, you can add it by doing `module load nsight-systems/<some version of nsight-systems>`. An example on how to launch your simulation with `nsys` is shown below.

```
lrun -n 1 \texttt{nsys} profile --output=profile.%q{OMPI_COMM_WORLD_RANK} \
    my_executable my_input_file --caliper nvtx
```

To visualize the output, launch `nsys-ui`, click on File in the top left corner, and open up the output file generated from the sample launch command above.

8.2 AMD/HIP

For El Capitan you can use `rocprof`. Like `nsys`, `rocprof` can generate timeline profiles and performance counter profiles.

`rocprof` supports kernel renaming for timeline profiles with `ROCP_RENAME_KERNEL=1`—either export it or prepend it to your profiling command line.

To use `rocprof` make sure the executable is in your path. If it is not, you can add it by doing `module load rocm/<version of rocm used with your code>`. An example on how to launch your simulation with `rocprof` is shown below.

```
ROCP_RENAME_KERNEL=1 flux run -n 1 rocprof --hip-trace --roctx-trace -o profile.csv \
    my_executable my_input_file --caliper roctx
```

Unlike `nsys` there is no vendor-provided time viewer so you must use either chrome://tracing in Google Chrome or Perfetto to visualize the output of rocprof. To launch Google Chrome from a terminal simply type `google-chrome`. Then, type chrome://tracing in the address bar, click on Load in the top left corner, and open up the file profile.json generated from the sample launch command above.