Introduction to GPU delegated computing with MATLAB

Guilherme Tegoni Goedert

Fluid Dynamics Laboratory, IMPA
Rio de Janeiro, Brazil

April 7th, 2017

Table of Contents

- Introduction
  - Types of Parallelization
  - CPU vs. GPU
  - CUDA
- gpuArray
  - Check
  - Allocate
  - Operate
  - Gather
- Examples
  - FFT
  - FFT2
  - Stencil Computations
- References

Principal types of Parallelization in MATLAB

- (Intrinsic) **Multithreading**: independent instrunctions are automatically separated into separate instruction streams, working in parallel in different parts of a vectorized data array.
- **Multiprocessing**: multiple instances of MATLAB run in different processor cores (often sharing memory), executing the same instructions concurrently. Achieved by Parallel Computing Toolbox.
- **Distributed**: multiple instances of MATLAB run independent computations on separate computers, typically is a Cluster. Achieved by a Distributed Computing Server.
- **GPU delegated**: a data set and instructions over it are delegated to a GPU, which returns the result. Useful for Massively-parallel tasks and achieved by a number of means (easiest by PCT).

CPU vs. GPU

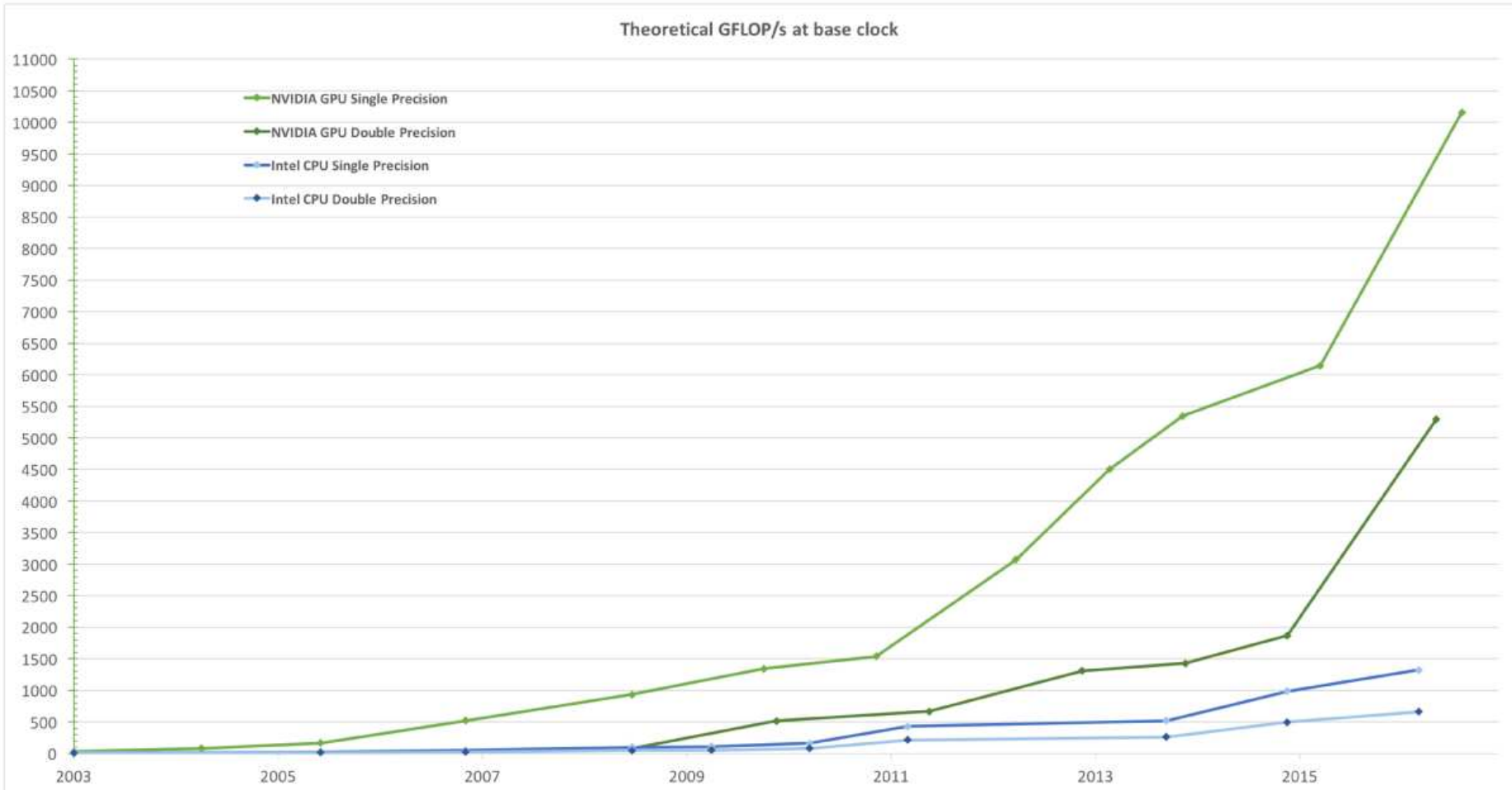
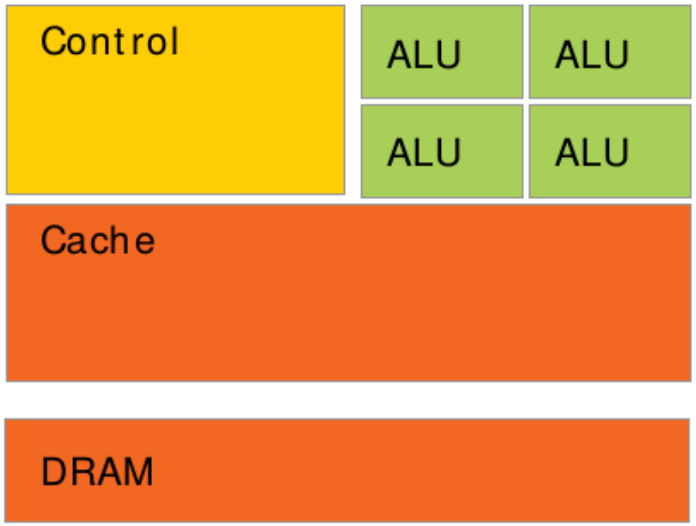
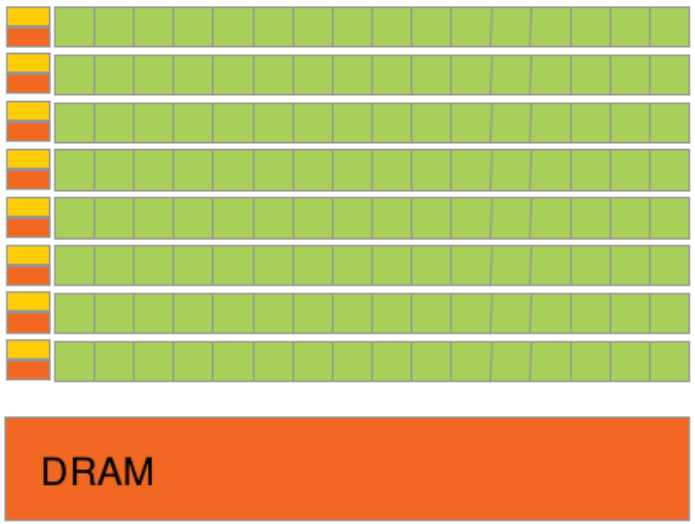


Image taken form [1].

Different Architectures





- In a CPU, more transistors are dedicated to control and cache, to decrease latency.
- In a GPU, more transistors are devoted to data processing. Control is "handled" by multithreading over a much larger number of ALU's, with larger latency but greater mean number of operations.

Image from [1].

CUDA

- Launched in November 2016 by NVIDIA;
- A general purpose parallel computing interface and programming model;
- Includes software environment for: **C/C++**, **FORTRAN**, **Python**, **Java**, etc.
- Dedicated libraries: **cuFFT**, **cuBLAS**, **cuSPARSE**, etc.
- Scalable programming model: problem is mapped into a **Grid** of **Blocks** (each given to a SM) of **Threads** (each assigned to a CUDA core).
- A CUDA Kernel is the set of instructions to be executed over the Grid.

Cuda Parallelization Model

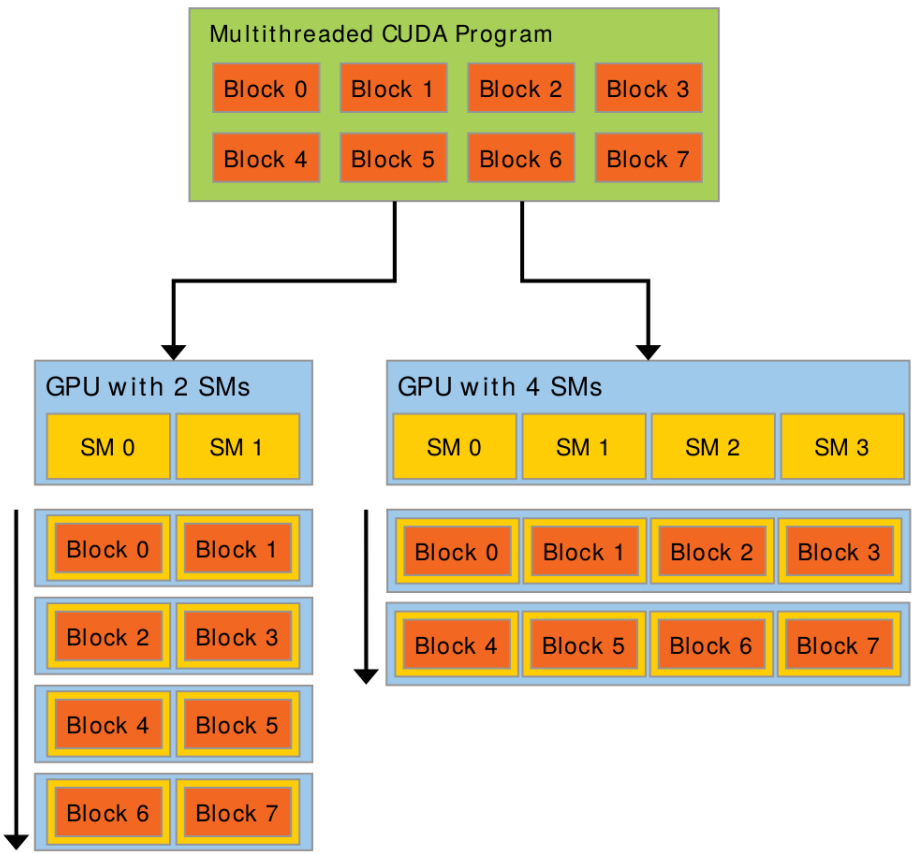


Image from [1].

- The parallelization units are the **SM**'s, or Streaming Multiprocessor.
- Some memory (allocated in L1 cache, 48KB) is shared in a SM.
- Different SM's communicated only through Global Memory.

If you had to remember ONE thing from today

GPU programming model is

Single
Program
Multiple
Data

Pursue in a GPU

GPU's run well with code which are

- SIMPLE, composed of simple arithmetic operations
- REPEATED thousands or millions of time over.

These operations need to be **element-wise** or over small matrix (enough to stay in the shared memory).
Since every core (thousands) is following the same program, you need enough threads to keep them occupied, otherwise you are wasting resources and reducing performance.

Avoid in a GPU

- AVOID BRANCHING
  Remember: GPU code is SPMD! Code that is non-uniform along your data set loses a lot of performance, as such avoid conditionals.
  As an alternative, treat your data set and separate it into the sets that should be processed differently. Then, run each processing separately.
- AVOID UNNECESSARY transfer between HOST and DEVICE
  You can easily carry out thousands of fft2's over a large matrix in a GPU in the same take it takes to copy said matrix between HOST and DEVICE.

Easiest way to use CUDA in MATLAB, gpuArray!

- What is a gpuArray?
  It's simply a numerical array allocated to the GPU (DEVICE).
- **ONLY NUMERIC or LOGICAL** data types are **ALLOWED**!

Using gpuArray in MATLAB

- **Step 0:** Check your MATLAB for a device

```
Command Window
>> D = gpuDevice

D =

  CUDADevice with properties:

                      Name: 'GeForce GTX 780'
                     Index: 1
         ComputeCapability: '3.5'
            SupportsDouble: 1
             DriverVersion: 8
            ToolkitVersion: 6.5000
        MaxThreadsPerBlock: 1024
          MaxShmemPerBlock: 49152
        MaxThreadBlockSize: [1024 1024 64]
               MaxGridSize: [2.1475e+09 65535 65535]
                 SIMDWidth: 32
               TotalMemory: 3.1639e+09
           AvailableMemory: 1.0302e+09
       MultiprocessorCount: 12
              ClockRateKHz: 901500
               ComputeMode: 'Default'
      GPUOverlapsTransfers: 1
    KernelExecutionTimeout: 1
          CanMapHostMemory: 1
           DeviceSupported: 1
            DeviceSelected: 1

fx >>
```

- **Step 1:** Allocate gpuArray
- Copy and existing array

```
d_A = gpuArray(h_A)
```

- Create it directly in the DEVICE

eye(___,'gpuArray')	rand(___,'gpuArray')
false(___,'gpuArray')	randi(___,'gpuArray')
Inf(___,'gpuArray')	randn(___,'gpuArray')
NaN(___,'gpuArray')	gpuArray.freqspace
ones(___,'gpuArray')	gpuArray.linspace
true(___,'gpuArray')	gpuArray.logspace
zeros(___,'gpuArray')	gpuArray.colon

Syntax of the constructors may easily verified with the command

```
help gpuArray/'CONSTRUCTOR'
```

- **Step 2:** Operate in DEVICE
  Note: if at least one array involved in a computation is a gpuArray, the result will be a gpuArray. The involved arrays that are in the HOST will be copied to the DEVICE temporarily on the call (not very efficient).
  - Many MATLAB functions can operate on gpuArrays. To see their list, use the command

    ```
    methods gpuArray
    ```

  - You can also write your own function to operate on gpuArrays. If `FUN` is the handle of this function, it can be run calling

    ```
    [A,B,...]  = arrayfun(FUN,C,...)
    ```

  - `FUN` MUST be composition of **element-wise** operations!
  - For operations between small matrices, `pagefun` may be used.

- **Step 3:** Send result to the HOST
  gpuArray and array in the HOST live in different memories and there is no common addressing system (yet).
  - to copy back a gpuArray `d_A` from the DEVICE, use `gather`

    $$h\_A = \text{gather}(d\_A)$$

  - note that, if you are only interested in result visualization, it is unnecessary.
    Plotting functions can operate directly on gpuArray's!

Example 1: FFT in simples signal processing

We use FFT to discover frequency components buried in noisy data.

- fft_ex1_CPU: generate and process data in HOST;
- fft_ex1_CPUtoGPU: generate data in HOST, transfer to DEVICE and process;
- fft_ex1_GPU: generate and process in DEVICE

```
>> fft_ex1_CPU
Total time taken with GPU 1.000s
>> fft_ex1_CPUtoGPU
Total time taken with GPU when copying from CPU 0.920s
>> fft_ex1_GPU
Total time taken with GPU 0.200s
>> fft_ex1_CPU
Total time taken with GPU 1.151s
>> fft_ex1_CPUtoGPU
Total time taken with GPU when copying from CPU 0.707s
>> fft_ex1_GPU
Total time taken with GPU 0.182s
```

Example from [3]

Example 2: FFT2 in far field diffraction patterns

We use FFT to discover frequency components buried in noisy data.

- Double_Slit_CPU.m
- Double_Slit_CPU.m

```
>> Double_Slit_CPU
Time taken with CPU computation 0.485715s
>> Double_Slit_CPU
Time taken with CPU computation 0.450600s
>> Double_Slit_CPU
Time taken with CPU computation 0.422389s
>>
>>
>> Double_Slit_GPU
Time taken with CPU computation 0.386173s
>> Double_Slit_GPU
Time taken with CPU computation 0.276201s
>> Double_Slit_GPU
Time taken with CPU computation 0.290598s
```

Change in ONE LINE made it 20% faster! Example from [3]

Example 3: Game of Life

We use this game as an example of Stencil Computations.
At each generation, count the number of live neighbours of each cell to determine the state of these cells in the next generation.

- GameOfLife_GPU.m: generate and process data in HOST;

```
>> GameOfLife_GPU
CPU:          7.453ms per generation.
Simple GPU:   0.949ms per generation (7.9x faster).
Arrayfun GPU: 1.032ms per generation (7.2x faster).
>> GameOfLife_GPU
CPU:          7.600ms per generation.
Simple GPU:   0.954ms per generation (8.0x faster).
Arrayfun GPU: 1.035ms per generation (7.3x faster).
>> GameOfLife_GPU
CPU:          7.523ms per generation.
Simple GPU:   0.952ms per generation (7.9x faster).
Arrayfun GPU: 1.033ms per generation (7.3x faster).
```

Example from [3]

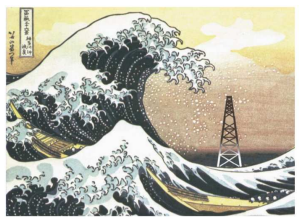
References and Code

[1] NVIDIA, *Cuda C Programming Guide*, 2017.
[2] Y. Altman, *Accelerating MATLAB Performance*, CRC Press, 2015.
[3] MathWorks, *MATLAB 2017a Documentation*, 2017.

Presentation examples's code may be found at my Git.

Institutional support

- FAPERJ grant E-26/202.574/2016;



IMPA PRH/ANP-32

FAPERJ
Fundação Carlos Chagas Filho de Amparo à Pesquisa do Estado do Rio de Janeiro