Second Year Internship

Jetson TX2 and CUDA Programming

Tutor
M. Alexandru STANCU
M. Eduard CODRES
M. Mario Martinez Guerrero

Contents

List of Figures

Chapter 1

Background information

1.1 CPU and GPU

In recent years, the computing power of GPUs (Graphic Processor Units) has evolved exponentially, much faster than that of CPUs (Central Processing Units). As these two components tackle very different problems, they also move in different directions.

CPUs are designed to get the maximum performance from an instruction flow, so to execute a task as quickly as possible; whereas GPUs are designed to process the maximum number of tasks in a reduced time so parallel computing is using different units.

Figure 1.1: Representation of a CPU and a GPU

A CPU is composed of:

- ALU (Arithmetic and Logical Unit), which support arithmetic calculations and tests, and allow several instructions to be processed at the same time.
- a control unit, which synchronises the various processor elements (register initialisation during machine startup and interrupt management).
- registers, small memories (a few bytes), fast enough for the ALU to manipulate their contents at each clock cycle.

- a clock that synchronises all the CPU's actions.
- an input-output unit, supporting communication with the computer's memory, and allowing the processor to access the computer's peripherals.

Today's processors also incorporate more complex elements, such as memory cache that speeds up processing by reducing memory access times. These Buffer memories are much faster than RAM and slow down the CPU less. The cache instructions receives the next instructions to be executed, the data cache manipulates the data.

A graphics processor forms the core of the graphics card and processes the images according to the selected resolution and coding depth. The GPU is a specialised processor with advanced image processing instructions, including 3D.

By definition, texture is a surface containing data (most of the time, these data represent colors). It is composed of a set of 2D pixels (Picture Element). But by applying the texture on a 3D surface, these pixels are then The two products are shaped and constitute a texel (TEXture ELementS). Texel represents the smallest 2D graphic unit applied to a surface and can occupy the space of several pixels or on the contrary, be lower to the size of a pixel.

1.2 Graphics Card Architecture

The architecture of graphics cards has undergone certain developments enabling programmers to make better use of their computing power. The graphics card has non-specialised computing units that can process all types of data.

A graphics card processor is actually a multiprocessor, composed of several dozen processors on which calculations are performed in parallel.

1.2.1 Memory

Graphics cards have a RAM called DRAM (Device Random Access Memory). They also have different memories more or less close to GPU processors:

- A global memory.
- A set of 32-bit local registers per processor.
- Shared memory: cache shared by all processors.
- A constant read-only memory.
- A read-only texture memory.

Constant and texture memories are faster to access than global memory because they have a cache.

1.2.2 Programming with streams

The architecture of the new GPUs is designed around the principles of stream-based programming, which involves performing multiple calculations in parallel on one or more data streams. We define:

- a stream: set of elements on which similar treatments will be applied.
- a kernel: treatment applied to each element of a stream.
- a thread: treatment executed by a programmable unit applied to an element of the stream.

1.2.3 Gather and Scatter

When a kernel is applied to a stream, it applies all its instructions to each element of the stream and writes to a defined element on an output stream. In a GPU, memory writes are usually made at well-defined positions (the x and y coordinates of the pixel). The scatter is a direct writing, i.e. writing a value to a given index in the stream. It can be represented by the operation a[i]=n.

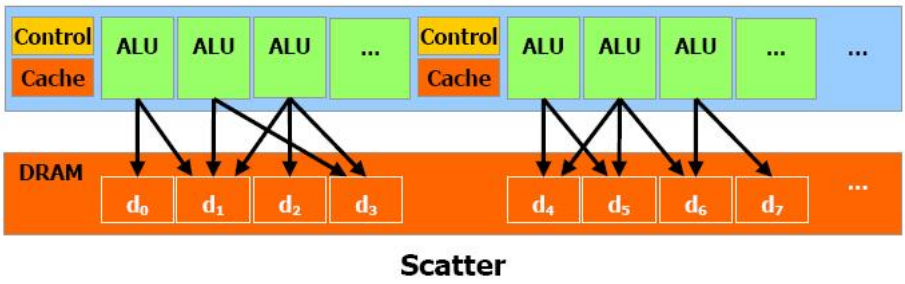

Figure 1.2: Writting in DRAM

The gather is a direct reading, i.e. a reading of a value at a given index in the stream. The gather corresponds to operation a=b[i].

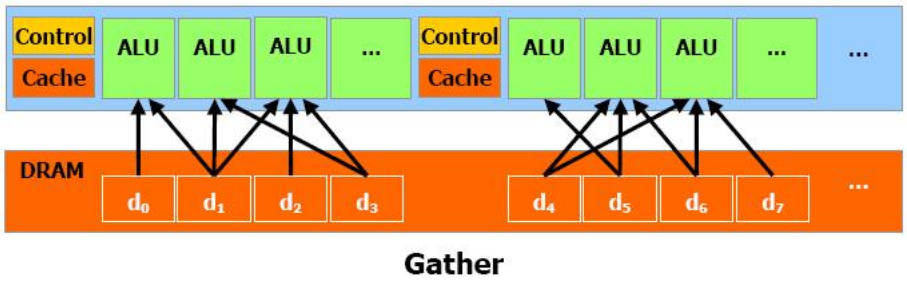

Figure 1.3: Reading in DRAM

Chapter 2

CUDA 6.2

2.1 CUDA Presentation

In November 2006, NVIDIA introduced CUDA (Computed Unified Device Architecture), an architecture for parallel computing that leverages the computing capabilities of GPUs. The CUDA development team has therefore developed a set of software layers to communicate with the GPU, and is used in a software environment that allows developers to use the C language. Thus, the graphics card is seen as a device capable of executing a large number of threads in parallel. it will be able to run a program several times at the same time on different data.

2.1.1 CUDA Architecture

CUDA exploits the computing capabilities of GPUs by having them process kernels (programs) on a number of threads. It is represented by a driver, a runtime, mathematical libraries (CUFFT and CUBLAS), an API based on an extension of the C language and the associated compiler (which redirects the non-executed part on the GPU to the classical default compiler of the system). CUDA offers two types of APIs:

- a high level API: the CUDA runtime API.
- a low level API: the CUDA driver API

The high level CUDA API globally ignores the hardware although taking into account its specificities is required to obtain an interesting yield. It is implemented "above" the low level API, each call to a runtime function is broken down into more basic instructions managed by the driver API.

The API driver is more complex to manage, it requires more work to run processing on the GPU, but in return it is more flexible, offering additional control to the programmer who wants it. It therefore acts as an intermediary between the compiled code and the GPU.The CUDA runtime is an intermediary between the developer and the driver, which facilitates development by hiding certain details.

CUDA proposes either to go through the runtime API, or to directly access the driver API, itsthat these APIs are mutually exclusive. The runtime API can thus be seen as the high level language, and the API driver as an intermediary between the high and low level, which allows you to manually optimise the code in more depth.

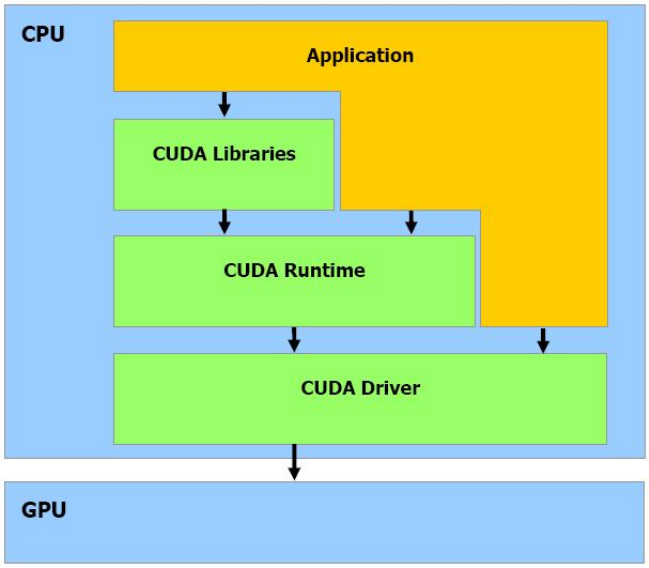

Figure 2.1: CUDA Structure

2.1.2 Some definitions

First, let's define a kernel, it's a function called by the CPU and executed by the GPU. A thread is an instance of a kernel. CUDA threads are expected to be much more numerous than in other environments. Indeed, a GPU thread performs more precise calculations than a CPU, and always in parallel (a thread is intended to be executed a large number of times in parallel). The design of threads derives from that of data parallelisation. Indeed, a thread is a portion of code that handles part of the data, all the threads handle all the data in parallel.

A warp is a group of 32 threads. But this granularity is still not enough to be easily usable by a programmer, so in CUDA one does not directly manipulate warps, one works with blocks that can contain from 64 to 512 threads.

These blocks are gathered in grids. The advantage of this grouping is that the number of blocks processed simultaneously by the GPU is closely linked to the hardware resources. The number of blocks in a grid allows to completely abstract this constraint and to apply a kernel to a large quantity of threads in a single call, without worrying about fixed resources. The other terms frequently encountered in the CUDA API are used to designate the CPU, called host, and the GPU designated as device.

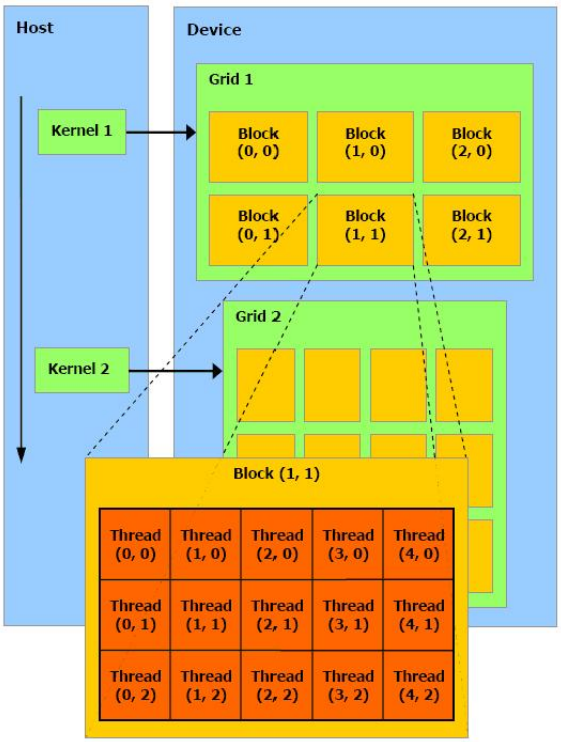

Figure 2.2: Organisation of grids, blocks, threads, and kernels.

Thread Block : A thread block is a set of threads that cooperate by efficiently sharing data through shared memory and synchronise their executions to harmonise memory access. More precisely, synchronisation points can be specified in the kernel, where the threads of a block are interrupted when they reach the synchronisation point. In each block, a thread is identified by its threadID, which is the thread number in the block. Thread identification can be done on one, two or three dimensions. By for example, a thread will be identified by :

- index (x) for one dimension.
- the index (x, y) for two dimensions, so in a size block (Dx, Dy) its threadID is (x + y.Dx).
- the index (x, y, z) for three dimensions, so in one size block (Dx, Dy, Dz) its threadID is (x + y.Dx + z.Dx.Dy).

Block Grid : A block can only contain a limited number of threads. However, blocks of the same size that execute the same kernel can be grouped into a grid of blocks, so the total number of threads that can be thrown to summon a kernel is higher. However, this implies a reduced thread cooperation, since threads in different blocks of the same grid cannot communicate or synchronise.

As for threads in a block, blocks in a grid have their identifier : the blockID. This model allows kernels to run efficiently without recompiling on various peripheral devices with different parallelisation capabilities. A device can run all blocks of a grid sequentially if it has few parallelisation capabilities, or in parallel if it has high parallelisation capabilities, or it can combine both. It's the CUDA runtime that breaks it down for us.

2.1.3 Memory Management

A thread running on the device has access to the following memory spaces:

- registers by thread, reading and writing.
- local memory by thread, read and write.
- shared memory by block, read and write.
- global memory by grid, reading and writing.
- constant memory per grid, read only.
- texture memory by grid, read only.

Global, constant and texture memory spaces can be read or written by the host and persist at kernel launches by the same application.

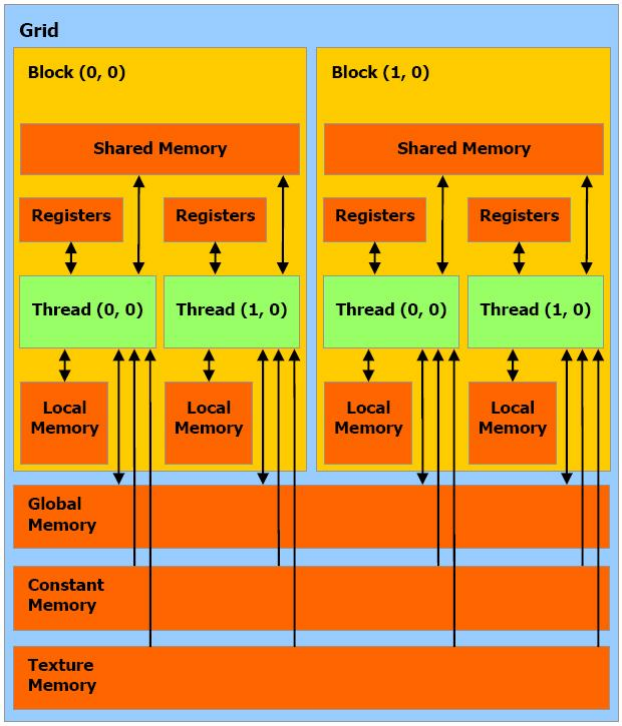

Figure 2.3: Memory Model

Local memory : This memory is, like the global memory, not hidden, and with a very high latency. This memory is only used for certain variables, which are automatically placed there when there are too many register variables, or when a structure would take up too much space in the registers, or if the compiler cannot determine if an array is indexed with constant values. This memory is automatically managed during compilation.

Global memory : CUDA is able to read and write to the memory embedded in the graphics card. These operations bear, respectively, the names of gathering and scattering. Global memory (DRAM) is the usable memory of any CUDA location, with the same performance at the key: this memory is not hidden, and you have to wait 400 to 600 cycles before accessing it. This leaves a multiprocessor inactive during this time.

Constant memory : The constant memory is hidden: reading from this memory costs only one cycle. For all half-warp threads, read from constant memory is as fast as from a register, as long as all threads read the same memory location. The cost of reading increases linearly with the number of different addresses requested by the threads. It is recommended that all threads in a warp use the same address, not just half warps, as future devices will require it for optimal operation. It can only be allocated from the host and only the host

can write data to it. The GPU has permission to read only from the non-volatile memory, so being in a kernel, it is impossible to modify data from that memory.

Texture memory : This memory space is hidden, so the cost of reading is very low. This memory is optimised for a two-dimensional space, so threads of the same warp that read at close addresses will have optimal performance. Also, it is intended for flow requests with constant latency. Reading the device's memories through the texture mechanism can be an advantageous alternative to reading from the global or constant memories.

Shared memory : Each multiprocessor has a small memory area called Shared memory. Its management is entirely the responsibility of the programmer. It allows to break in part the limitations imposed by a parallel processing of threads by allowing them to communicate and thus interact between them quickly, without going through the memory of the graphics card. All threads in the same block are guaranteed to be executed by the same multiprocessor. Conversely, the allocation of blocks to different multiprocessors is completely indefinite, so two separate block threads cannot communicate during their execution. Using this memory well is therefore complicated but can be profitable because, except in the case where several threads try to access the same memory bank which causes a conflict, the rest of the time access to the shared memory is as efficient as access to the registers. Thus, more threads per block means less memory per thread and fewer threads per block means fewer threads will be able to communicate. In addition, it is generally advisable to allow each multiprocessor to work on several blocks so that a second block can be processed when the first is paused to optimise resources. This reduces the size of shared memory, in the recommended case where 2 blocks are in each multiprocessor.

It is faster to access than global and local memory, and is used to exchange and share data between threads in the same block. Its use also makes it possible to limit access to DRAM, and thus to increase the performance of certain treatments. The following figure shows the difference between direct access to DRAM and access to data from shared memory.

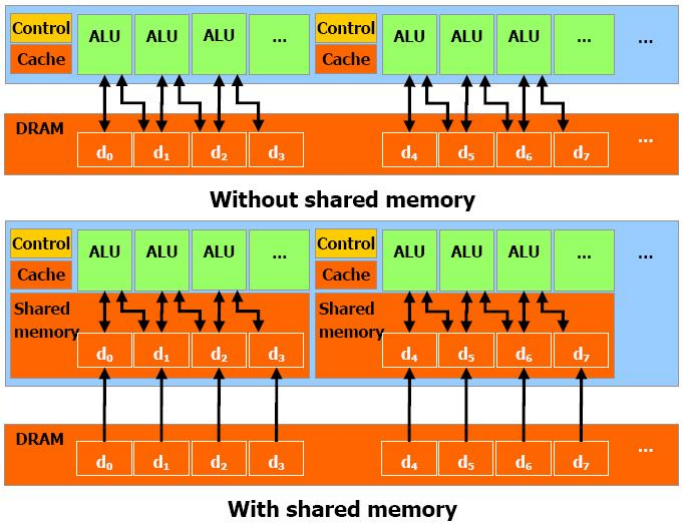

Figure 2.4: Memory access with and without shared memory

Registers : Generally, access to a registry does not take a single additional cycle per instruction, but delays may occur, due to read after write dependencies, and conflicts may occur. Delays entered by dependencies can be ignored, as soon as there are at least 192 active threads per multiprocessor, which allow to hide them. The compiler and thread scheduler organise the instructions for optimal performance, which requires avoiding conflicts with banks. The best way to get good performance is to use a multiple of 64 as the number of threads per block. An application has strictly no way of controlling these conflicts.

Shared memory is therefore not the only memory to which multiprocessors have access, they can of course use video memory but it offers lower bandwidth and higher latency. As a result, in order to limit too frequent access to this memory, NVIDIA has equipped its multiprocessors with caches (approximately 8 KB per multiprocessor) for access to constants or textures. The GPU has a cache memory at the texturing units, they can be used to efficiently read data.

Multiprocessors have 8192 registers to share between all threads of all active blocks on this multiprocessor. The number of active blocks per multiprocessor cannot exceed 8, the number of active warps is limited to 24 (768 threads). Thus, the more threads there are per block the better the latency of some operations is hidden but the less registers they have. Optimising a CUDA program consists essentially in balancing the number of blocks and their size. Moreover, a block of 512 threads would be particularly inefficient because only one block could be active on a multiprocessor, potentially wasting 256 threads. NVIDIA recommends using blocks of 128 to 256 threads that offer the best compromise between latency masking and enough registers for most kernels.

CUDA threads can access data in various memory locations. Thus, each thread has a local private memory. Each thread block has a shared memory visible to all threads in the block and with the same block life. All threads have access to the same global memory. Two read-only memory spaces are also available for all threads: constant and texture memory spaces. The global,

constant or texture memory spaces are optimised for different memory uses. The texture memory offers various addressing modes for specific data formats. Global, constant or texture memory spaces persist at kernel launches by the same application.

The CUDA programming model assumes that CUDA threads run on physically separate devices that operate as a co-processor on the host running the C program. For example, the kernel runs on the GPU and the rest of the C program runs on the CPU. In addition, the host and the device have their own DRAM, which is called host memory or device memory respectively. So a program manages the global, constant and texture memory spaces so that they are visible by the kernel through the CUDA runtime calls. This includes memory allocations and de-allocations, as well as data transfers between host and device memory.

2.2 Hardware

The device is implemented as a set of multiprocessors. Each multiprocessor has a Single Instruction Multiple Data (SIMD) architecture: at a given cycle, each multiprocessor processor executes the same instruction, but operates on different data. Each multiprocessor has on its memory :

- A set of 32-bit registers per processor.
- A parallel data cache or shared memory, which is shared by all processors and implements shared memory space.
- A read-only constant cache, shared by all processors and which accelerates the reading of the constant memory space, which is implemented as a read-only region of the device memory.
- A read-only texture cache, shared by all processors and that accelerates the reading of the texture memory space, which is implemented as a read-only region of the device memory.

Local and global memory spaces are implemented as read-write regions of the device memory and are not hidden. Each multiprocessor has access to the texture cache via a texture unit that implements different addressing modes and data filtering.

The number of blocks that a multiprocessor can process simultaneously (the number of active blocks per multiprocessor) depends on the number of registers per thread and the amount of shared memory required per block for a given kernel, as the multiprocessor registers and shared memory are shared between all threads of the active blocks.

If there are not enough registers or shared memory available per multiprocessor to process at least one block, the kernel will fail to launch.

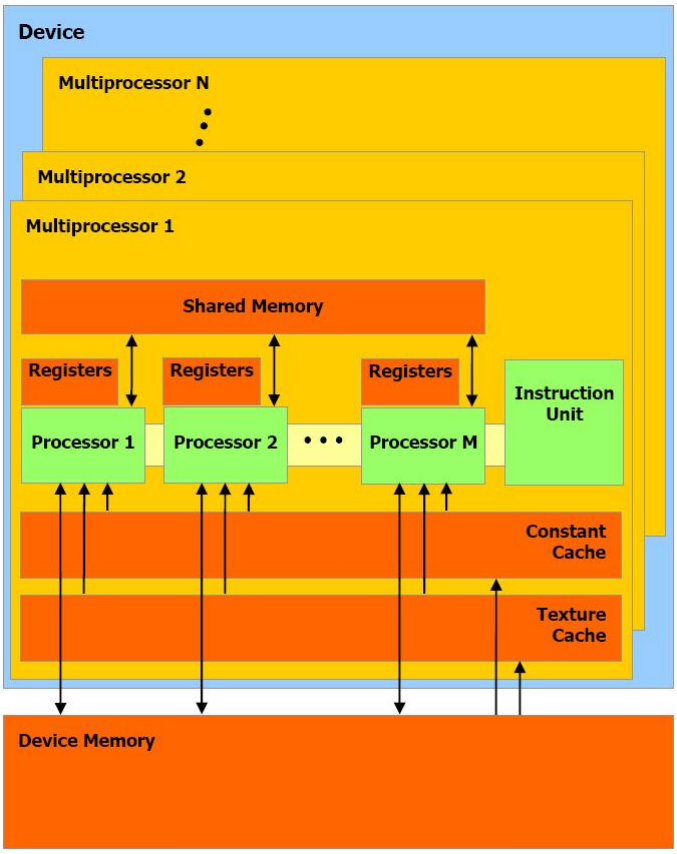

Figure 2.5: Hardware Model

2.3 Language and Compilation

2.3.1 C language extensions

The CUDA programming interface is intended to provide an easy way to write programs to be executed on the device. It consists of :

- A set of extensions to the C language, which allow the programmer to target portions of code that will run on the device.
- A runtime library that is divided into :
  - A host component that runs on the host and provides functions to control and access one or more devices from the host.
  - A device component, which runs on the device and provides GPU-specific functions.
  - A common component, which provides vector types and a subset of the C library supported by the host and device.

The only functions of the standard C library that run on the device are the functions of the common runtime component. The C language extensions are as follows:

- Function IDs that specify whether a function is running on the host or device and whether it is called by the host or device.
- Variable identifiers to specify the memory location of a variable on the device.
- A new directive that specifies how a kernel is executed on the device from the host.
- Four construction variables that define the grid and block dimensions, and block and thread clues.

Each source file that contains these extensions must be compiled with the compiler of CUDA nvcc.

Function identifiers : A kernel, a function called by the CPU and executed by the GPU, will be referenced by **__global__** . Such a function is executed on the device and can only be called by the host.

A function referenced by **__device__** is a function used in a kernel, so it will be executed by the GPU and can only be called from the GPU.

A classic function (function called by the CPU and executed on the CPU) will be referenced by **__host__** , a keyword that can be omitted since it represents the default behavior. However, this qualifier can be used with the qualifier __device__if a function is compiled by the host and the device.

The __device__and __global__functions do not support recursion.
The __device__and __global__functions cannot declare static variables.
The __device__and __global__functions cannot have a variable number of parameters.
The __global__and __host__identifiers cannot be used together.
The memory address of the functions __device__cannot be requested.
The __global__functions have a void return type.
At execution, the configuration for the __global__functions must be specified.
A call to a __global__function is asynchronous, the kernel returns before the device has completed its execution.
The parameters of the __global__functions are stored in the shared memory and limited to 256 bytes.

Variable identifiers : Variables also have new qualifiers to control the memory area in which they will be stored.

The **__device__** identifier declares a variable that is and remains on the device. It resides in the global memory space, will live no longer than the application, is accessible to all threads in the grid, and to the host through the runtime library. This type can be combined with the following two types.

The qualifier **__constant__** declares a variable that is and remains in constant memory space. It will live no longer than the application, is accessible to all threads in the grid, and to the host through the runtime library. These variables can only be declared from the host but not from the device.

A variable preceded by the keyword **__shared__** indicates that it will be stored in the shared memory of a thread block, so will have the lifetime of the block, and will only be accessible to threads in the block. The shared memory being much faster than the global memory, one must try to replace all the accesses to the global memory by the accesses to the shared memory.

Runtime Configuration : The execution configuration must be specified for the __global__ functions. It defines the size of the grid and blocks used to execute the function on the device, and the associated stream. The definition of the kernel is done as follows:
*__global__ void func(float * parameter)* ;
And its use via :
func<< Dg, Db, Ns, S >>(parameter) ;

Dg is of type Dim3, and specifies the size and dimension of the grid, such that **Dg.x * Dg.y** equals the number of blocks thrown, **Dg.z** is not used.

Db is of type Dim3, and specifies the size and dimension of each block, such that **Db.x *Db.y * Db.z** equals the number of threads per block.

Ns is of type size_t, and specifies the number of bytes in the shared memory that are dynamically allocated per block in addition to the statically allocated memory. This is an optional parameter that is 0 by default.

S is of type cudaStream_t, and specifies the associated stream. This is an optional parameter that is 0 by default.

Grid construction variables : gridDim is of type dim3 and contains the dimensions.
BlockIdx is uint3 type and contains the block index in the grid.
BlockDim is dim3 type and contains the block dimensions.
ThreadIdx is of type uint3 and contains the thread index in the block.

We can't take the addresses of these construction variables, nor assign values to these variables.

Compilation : nvcc is a compiler for CUDA, it provides command line options, and executes them by invoking tools that implement the various compilation steps. It consists of separating the peripheral code from the host code and compiling the peripheral code into a binary form. The generated host code is either output as C code that will be compiled using another tool or as object code by calling the host compiler directly during the last build step.

Applications can either ignore the generated host code, load and execute the Cubin object on the device using the CUDA driver API, or can link to the generated host code, which includes the Cubin object as an initialised global data array and contains a translation of the runtime configuration syntax into the CUDA runtime start code to load and run each compiled kernel.

Kernels must be written using the CUDA instruction architecture, called PTX. However, it is more efficient to use a high-level programming language, such as C. In both cases, kernels must be compiled in binary code via nvcc. It is a compilation driver that simplifies the C compilation process for CUDA code. It provides command line options and executes them by invoking a series of tools that implement the various compilation steps.

Chapter 3

Dynamic Parallelism

The first CUDA programs had to conform to a parallel, flat, mass programming model. Programs had to run a sequence of kernel launches, and to get the best performance, each kernel had to exhibit enough parallelism to use the GPU effectively. For "parallel for" loop applications, the parallel mass model is not too limiting, but some parallel models - such as nested parallelism - cannot be expressed as easily. Nested parallelism occurs naturally in many applications, such as those that use adaptive grids, which are often used in real-world applications to reduce computing complexity while capturing the relevant level of detail. Flat and solid parallel applications must either use a fine grid and perform unwanted calculations, or use a coarse grid and lose finer details.

CUDA 5.0 introduced Dynamic Parallelism, which allows kernels to launch from threads running on the device; threads can launch other threads. An application can launch a coarse grain core which, in turn, launches finer grain cores to do the job where it is needed. This avoids unwanted calculations while capturing all the interesting details.

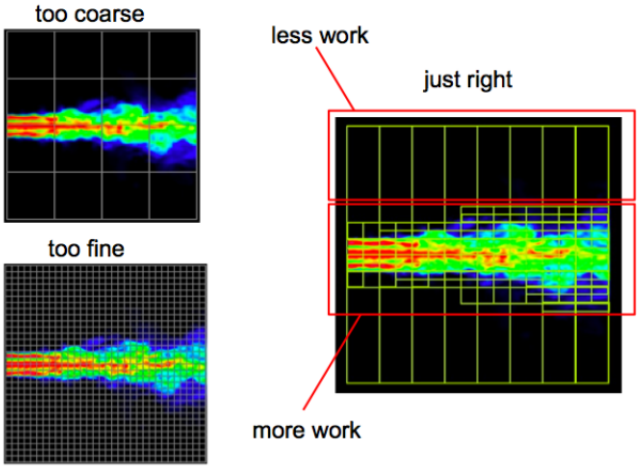

Figure 3.1: A fluid simulation that uses adaptive mesh refinement performs work only where needed

A device thread that configures and launches a new grid belongs to the parent grid, and the grid created by the invocation is a child grid. the invocation and completion of child grids is correctly nested, which means that the parent grid is not considered complete until all child grids created by its threads have been completed. Even if invoking threads are not explicitly synchronised on launched child grids, the runtime guarantees implicit synchronisation between parent and child.

On the host and device, the CUDA runtime provides an API for launching kernels, waiting for the work to be completed and tracking dependencies between launches through streams and events. On the host system, the launch status and CUDA primitives referencing streams and events are shared by all threads within a process however, processes run independently and cannot share CUDA objects. A similar hierarchy exists on the device: launched kernels and CUDA objects are visible for all threads in a thread block, but are independent between thread blocks. This means for example that a feed can be created by one thread and used by any other thread in the same thread block, but cannot be shared with threads in any other thread block.

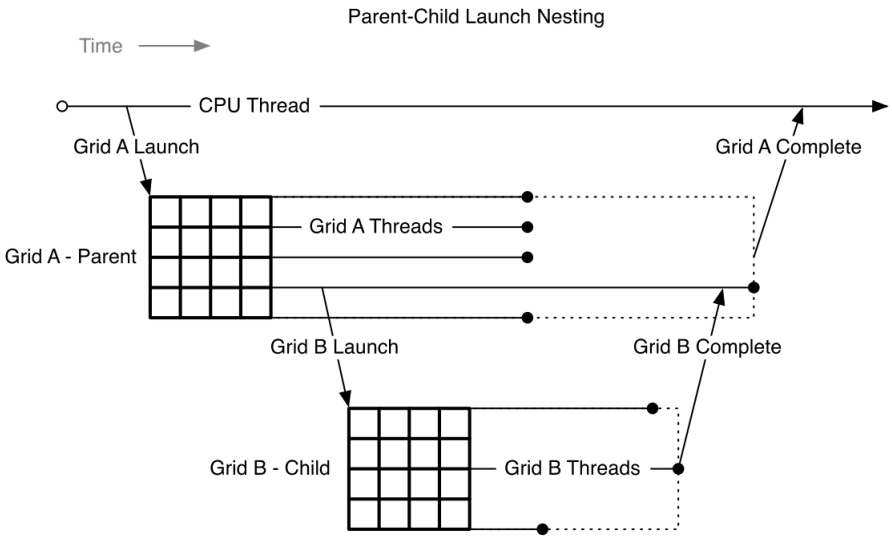

Figure 3.2: Parent Child launch nestling

Chapter 4

Further Documentation Resources

This section aims to help further work on the subject by gathering a handful of useful information about CUDA programming, how to use the Jetson TX2 and works on Interval programming.

4.1 Jetson TX2

4.1.1 Hardware

The Jetson TX2 module integrates:

- **256 core NVIDIA Pascal GPU**. Fully supports all modern graphics APIs, unified shaders and is GPU compute capable. The GPU supports all the same features as discrete NVIDIA GPUs, including extensive compute APIs and libraries including CUDA. Highly power optimised for best performance in embedded use cases.
- **ARMv8 (64-bit) Multi-Processor CPU Complex**. Two CPU clusters connected by a high-performance coherent interconnect fabric designed by NVIDIA; enables simultaneous operation of both CPU clusters for a true heterogeneous multi-processing (HMP) environment. The Denver 2 (Dual-Core) CPU clusters is optimised for higher single-thread performance; the ARM Cortex-A57 MPCore (Quad-Core) CPU clusters is better suited for multi-threaded applications and lighter loads.
- **Advanced HD Video Encoder**. Recording of 4K ultra-high-definition video at 60fps. Supports H.265 and H.264 BP/MP/HP/MVC, VP9 and VP8 encoding.
- **Advanced HD Video Decoder**. Playback of 4K ultra-high-definition video at 60fps with up to 12-bit pixels. Supports H.265, H.264, VP9, VP8 VC-1, MPEG-2, and MPEG-4 video standards.
- **Display Controller Subsystem**. Two multi-mode (eDP/DP/HDMI) outputs and up to 8-lanes of MIPI-DSI output. Multiple line pixel storage

allows more memory-efficient scaling operations and pixel fetching. Hardware display surface rotation is also provided for bandwidth reduction in mobile applications.

- **128-bit Memory Controller**.128-bit DRAM interface providing high bandwidth LPDDR4 support.
- **8GB LPDDR4 and 32 GB eMMC memory** integrated on the module
- **1.4Gpix/s Advanced image signal processing**: Hardware accelerated still-image and video capture path, with advanced ISP.
- **Audio Processing Engine**. Audio subsystem enables full hardware support for multi-channel audio over multiple interfaces.

Module Datasheet link.
Jetson TX2 tutorials,guide and datasheet link.
Jetson TX2 developper kit link.

4.1.2 Instal

NVIDIA JetPack SDK is the most comprehensive solution for building AI applications. Use the JetPack installer to flash your Jetson Developer Kit with the latest OS image, to install developer tools for both host PC and Developer Kit, and to install the libraries and APIs, samples, and documentation needed to jumpstart your development environment.

Jetpack for Jetson link.
Jetson Download Center link.

NVIDIA makes available a large range of development tooll such as NSIGHT, it enables to cross compile from the host device with Eclypse to the Jetson.
NSIGHT tutorial link.

4.2 CUDA Toolkit

NVIDIA provides lots of ressources for programming with CUDA (CUDA Toolkit). For the embedded systems such as Jetson, a kit with every tools that could be use is also provided (Jetpack) which includes CUDA Toolkit.

CUDA Toolkit link.
CUDA Toolkit documentation link 1. link 2.

NVIDIA also provides plenty of documentation for CUDA. Here is a list with the most detailed infomration about the commonly used features.
Programming guide for CUDA link.
Advice for CUDA programming link.
Linking and COmpiling link.
Compute capability and their meaning link.
CUDA Dynamic parallelism guide link 1. link 2.

4.3 Interval papers

I have found two Master thesis dealing with SIVIA on parameter estimation with intervals while working with CUDA and GPUs.

The first one is by Siddharth Sharma on Parameter Estimation for System Biology Models on GPU Clusters. It deals on SIVIA and shows all the issues about wordload that comes with this dynamic code. link.

The second one is by Maxime LASTERA, it is in French but it can be still be use by one of the student of ENSTA. The thesis is on the use of GPUs for scientific computation. link.