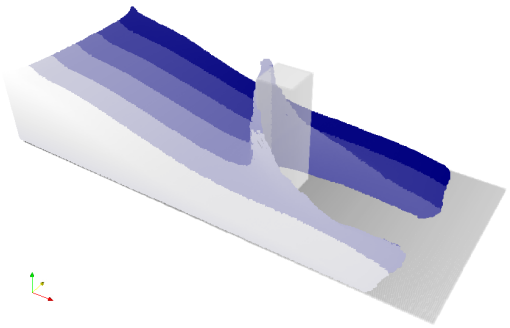
UNIVERSITÀ DEGLI STUDI DI CATANIA
Facoltà di Scienze Matematiche, Fisiche e Naturali
DOTTORATO DI RICERCA IN INFORMATICA

FLUID DYNAMICS SIMULATIONS ON MULTI-GPU SYSTEMS

EUGENIO RUSTICO



A dissertation submitted in partial fulfillment of the requirements for the degree of "Research Doctorate in Computer Science"

COORDINATOR
Chiar.mo Prof. Domenico Cantone

XXIV cycle

Acknowledgements

This thesis would not have been possible without the help, encouragement, patience and love of my family.
Thank you. You are the start and the goal of my path.

I would like to thank all my friends and colleagues of IPLab and DMI and in particular prof. Giovanni Gallo, for his precious tips and advices.

A special thank goes to the friends and colleagues of the INGV Catania. I would like to thank especially Dr. Ciro Del Negro for his help, esteem, assistance and trust.

Many people helped me growing professionally and culturally along the path of the Ph.D. In particular, I owe my deepest gratitude to prof. Robert A. Dalrymple, prof. Alexis Hérault and Dr. Giuseppe Bilotta. Your help was more than priceless.

Developers and dreamers all around the world made possible GNU/Linux, LaTeX, Inkscape and a huge number of high quality tools I could use for free. I feel Free Software was a fundamental basis for my work. I wish I could thank you all.

Contents

Whether you think you can, or you think you can't - you're right.

Henry Ford

Chapter 1

Introduction

The realistic simulation of fluid flows is fundamental in a number of fields, from entertainment to engineering, from astrophysics to city planning. In particular, there are a number of applications related to civil protection: simulating a tsunami, an ash cloud or a lava flow, for example, has an important role both in disaster prevention and damage mitigation.

The Sezione di Catania of the Istituto Nazionale di Geofisica e Vulcanologia (INGV-CT) is a leading research center in the fields of natural hazard assessment and lava flow simulation. In the past 10 years, in collaboration with the University of Catania, two lava simulators have been developed: one uses a Cellular Automaton (CA) approach, while the other is based on the Smoothed Particle Hydrodynamics (SPH) method.

The CA simulator, called MAGFLOW, has been widely used for hazard assessment and forecasting, as it is capable of simulating lava flow with Bingham rheology on an arbitrary DEM (Digital Elevation Model, a 2D representation of a 3D topography). The SPH-based one, called GPUSPH, has been recently developed for its capability to model a fluid in a truly three-dimensional environment while simulating complex phenomena such as solid-fluid interaction, crust formation and tunneling.

Such complex and flexible simulators come at the price of a high level of complexity and a significant computational cost. The simulation of an eruption, to be useful in any application (ranging from scenario forecasting to statistical analysis, from the validation of the model itself to its sensitivity analysis) should take significantly less time than the simulated event; an efficient implementation is thus needed for any practical application.

Among the possible high-performance computing solutions available, GPU computing is nowadays one of the most cost-effective in terms of Watt per FLOPS and nevertheless one of the most powerful ones. For this reason, the MAGFLOW simulator has been ported for the execution on GPU while GPUSPH, as the name suggests, has been natively written with GPU computing in mind. This has greatly improved the performance of MAGFLOW with respect to the correspondent CPU implementation and allowed GPUSPH to complete a simulation in a reasonable time (faster than 1:1). A multi-GPU version of both simulators has been recently developed to increase the performance and, for GPUSPH, to allow simulations that would not fit the memory of a single GPU.

1.1 Contribution

The subject of this thesis is the original multi-GPU implementation of both MAGFLOW and GPUSPH simulators. As later explained, it is not trivial to exploit more than one device simultaneously to solve complex problems like SPH or cellular-automaton based fluid simulations. We had to overcome several technical and model-related challenges mainly due to the locality required by the problem computational units (cells, particles) and to the latencies introduced by using off-CPU devices. The resulting multi-GPU simulators allow faster simulations and enable GPUSPH to simulate sets of particles bigger than a single GPU could fit. To the best of our knowledge, these are the first multi-GPU Cellular-Automaton

based lava simulator and the first multi-GPU Navier-Stokes SPH fluid simulator.

In the first parts we introduce the concepts of GPU computing, GPGPU, multi-GPU and CUDA. The second part is dedicated to the MAGFLOW simulator; the single-GPU version is briefly described and then the multi-GPU implementation is introduced. A similar structure is used in the third part, where the single- and multi-GPU implementations of GPUSPH are described. Finally, in the fourth part, the results are briefly discussed.

The results presented in sections 6.8 and 8.11 show a performance gain almost linear with the number of GPUs used. The simulators have been tested for using up to 6 GPUs simultaneously and the execution times differ from the ideal ones only by a small cost function which is itself linear with the number of GPUs.

Two load balancing approaches have been designed, implemented and tested: an *a priori* balancing for MAGFLOW and an *a posteriori* one for GPUSPH. The former dynamically changes according to the bounding box variations of the simulated fluid. The latter one makes GPUSPH perform steadily even in very asymmetric simulations, where a static split would perform about 50% worse.

The performance results are formally analyzed and discussed. While the results are close to the ideal achievable speedups, some future improvements are hypothesized and open problems are mentioned.

Part I

GPU computing

Chapter 2

The Graphics Processing Unit

A GPU (Graphics Processing Unit) is a microprocessor specialized in graphics-related operations. GPUs have been traditionally used to offload and accelerate graphic rendering from the CPU by providing direct hardware support for common 2D and 3D graphics primitives.

The market of videogames has probably been the most important pushing factor towards faster and faster GPUs, which are nowadays incredibly faster than CPUs in parallel, arithmetic-intensive tasks like matrix manipulations and coordinate projections.

Figure 2.1: Screenshots of *Castle Wolfenstein* (1981), *Wolfenstein3D* (1992), *Return to Castle Wolfenstein* (2009) and *Wolfenstein* (2011).

Fig. 2.2 shows the trend of peak computational power of CPUs vs. GPUs, from 2003 to 2010, with an always increasing gap. However, as the GPUs are typically many-cores with a lower clock rate, it is more difficult to reach the peak value of a GPU, as not all the computational problems can be split in independent, parallel tasks. The red vertical line marks the introduction of the CUDA GPGPU

computing platform.

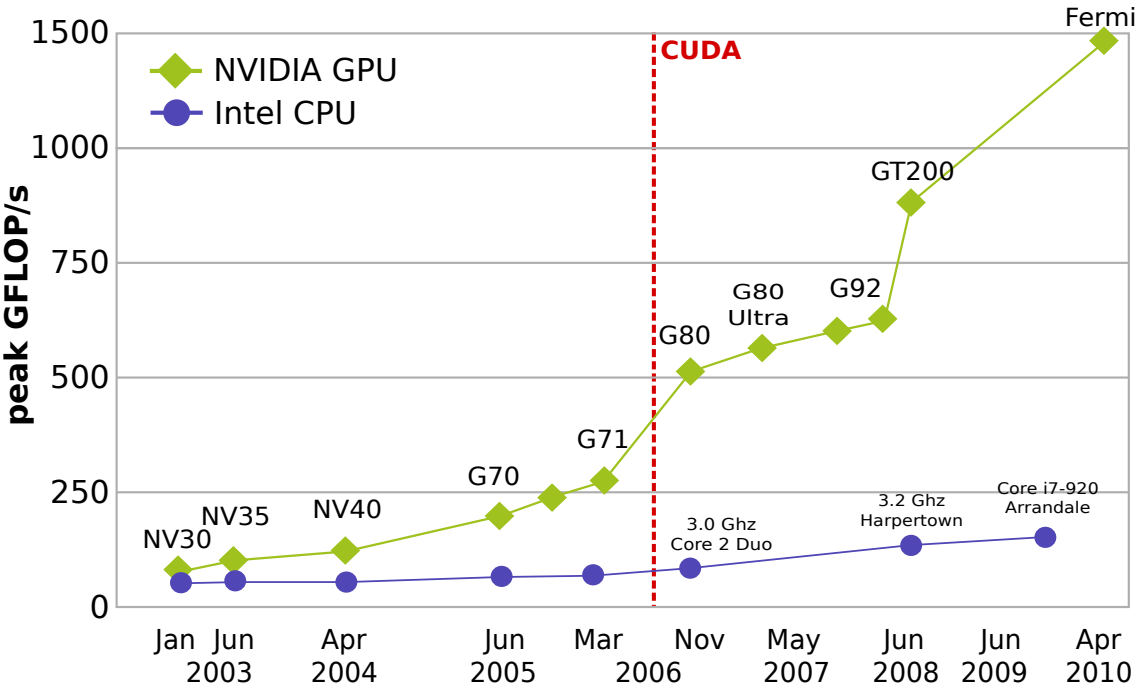


Figure 2.2: Trend of theoretical computational power of CPUs and GPUs, 2003-2010

2.1 Birth of GPGPU

In 2001 NVIDIA released the first chip capable of *programmable shading*, i.e. a GPU with which it was possible to customize the arithmetic pipeline to obtain very sophisticated surface renderers. Programmable shading later became a standard GPU feature also for other brands.

While programmable shaders were meant to be used for graphical purposes, there was a growing interest in using graphic cards also as general-purpose stream processing engines [78, 33]. Engineers and scientists started using them for non-graphical calculations by mapping a non-graphic problem to a graphic one and letting the GPU compute it. To cite a few examples, in [12] a programmable shader is used to simulate the behavior of a flock; in [34] pixel shaders are used for both visualization and evolution of Cellular-Automaton based simulations; in [50] a framework for linear algebra is implemented on programmable GPUs.

The possibility to use a specialized microprocessor for generic purpose problems marked the beginning of the GPGPU paradigm, where the acronym GPGPU stands for General-Purpose Graphics Processing units. The *hack* of mapping a non-graphic problem to a graphic one, however, was complex and cumbersome, and required the programmer to know many hardware details as well as to code using the hardware assembly language.

2.2 From hacking to CUDA

In 2007 NVIDIA released CUDA (acronym for Compute Unified Device Architecture), a hardware and software architecture with explicit support for general purpose programmability. The ground-breaking feature of CUDA was the possibility to code for the GPU using an extension of the C programming language[1].

```
1 for (i=0; i<4096; i++)
2     cpu_array[i] = i;
```

Listing 2.1: Array initialization in C++ for serial execution on the CPU.

```
1 #define BLKSIZE 64
2 ...
3 __global__ void init_kernel(int* gpuPointer) {
4     index = blockDim.x*blockIdx.x + threadIdx.x;
5     gpuPointer[index] = index;
6 }
7 ...
8 init_kernel<<< 4096/BLKSIZE, BLKSIZE >>>(gpu_array);
```

Listing 2.2: Array initialization in CUDA C for parallel execution on the GPU.

[1] *CUDA C* is C extended with a few GPU-specific keywords and implicit library calls, but with some limitations like no possibility to call external libraries, no direct CPU memory access, no recursion (this limit has been overcome in recent versions), etc.

Listing 2.1 is an example of serial array initialization on the CPU, in C; listing 2.2 is an equivalent CUDA C array initialization for parallel execution on the GPU.

Also ATI, NVIDIA's main competitor, released soon after a GPGPU architecture called Stream; instead of a high-level language, Stream requires to program the GPU with a low-level assembly language.

A modern GPU typically hosts more cores then a CPU, up to a one thousand, and runs with a slightly lower clock rate. This setting is ideal to implement a SIMD (Single Instruction Multiple Data) paradigm, where a series (stream) of instructions is executed in parallel on multiple data. Fig. 2.4 is a graphical representation of the SISD (Single Instruction Single Data), SIMD and stream processing approaches.

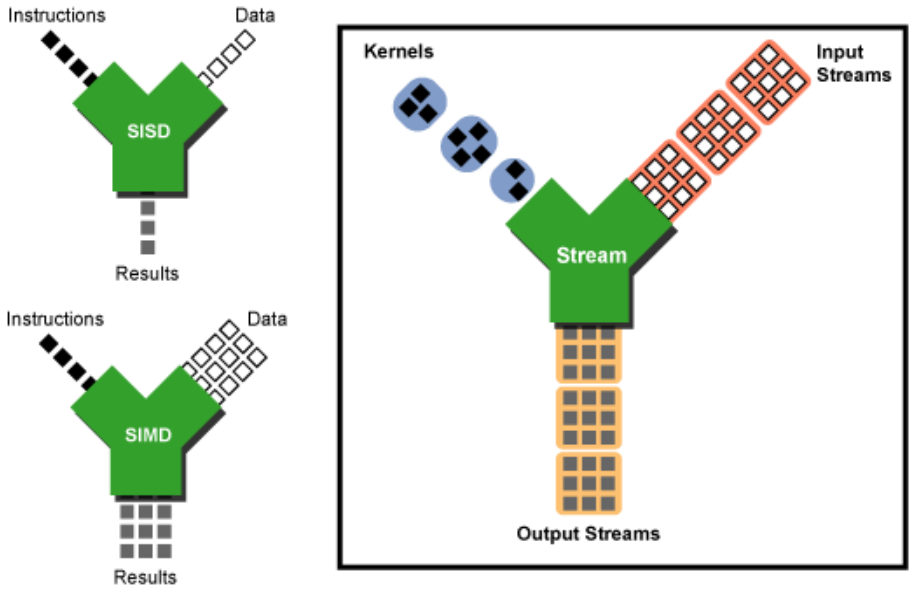


Figure 2.3: SISD (Single Instruction Single Data), SIMD (Single Instruction Multiple Data) and Stream Processing.

From a programmer's perspective, a GPU may be seen as an external computer with its own RAM memory and processors; the typical workflow consists in transferring the input data from the CPU memory to the GPU memory, requesting a computation and transferring back the resulting output. Following a common convention, we will refer to the GPU and its RAM as *device* and to the CPU and its memory as *host*.

Some classes of problems can be solved much faster on GPUs and achieve speedups of two orders of magnitude with respect to the correspondent CPU implementations, as long as they expose a high level of parallelism on a fine data granularity. The price of such a speed benefit is a slight change in the programming logic or, sometimes, a complete rewrite of the computational core. The high cost-effectiveness of GPU-based solutions is, however, the main breakthrough of this phenomenon. A single workstation with modern consumer GPUs easily reaches the theoretical speed of one TFLOPS (10^{12} Floating Point Operations Per Second), while costing less than €1,000 and consuming less than 800W.

Figure 2.4: A consumer-level multi-GPU workstation reaching the theoretical speed of one TFLOPS

Even devices dedicated to scientific computing like the NVIDIA Tesla boards, which share the chipset with the consumer-level GPUs, are still a very cost-effective solution and have been recently used also in supercomputing environments. In may, 2010 a hybrid CUDA-enabled Chinese supercomputer called Nebulae reached the theoretical peak performance of 2.98 PFlops, ranking first in the list of the most powerful commercial systems in the world, and second by means of actual measured performance [53]. Later in 2010 another GPU-based Chinese supercomputer, Tianhe-1A, ranked first in the TOP500 list with an actual performance of 2.57 PFlops. In June 2011 another GPU-based supercomputer called TSUBAME 2.0

entered the top ten; in November 2011 Nebulae, Tianhe-1A and TSUBAME 2.0 still rank among the top 5 most powerful supercomputers in the world.

The cost-effectiveness and the ease of utilization of modern GPUs, along with the wide spread of GPU computing even outside the commercial and academic world, led some to claim that the *GPU Computing Era* has begun [62]. It is worth mentioning, however, that some others criticized the excess of enthusiasm for GPGPU and proved that a well-tuned CPU implementation, optimized for execution on recent multi-core processors, often reduces the flaunted 100× speedup reported by many works [51].

2.3 Numerical precision

Another little price of porting an application to GPGPU, apart from the effort of a little re-engineering, regards numerical precision.

It is useful to recall that floating point arithmetic introduces a systematic error that is important to take into account and mitigate through the use of specific numerical techniques [25]. For example, a common and underestimated source of error is the non-commutativeness of floating point operations: this is particularly visible when the numbers involved differ by orders of magnitude such as, for example, when adding alternatively very large numbers and numbers very close to zero. Different implementations of the same algorithm may result in a little change in the order of additions, and if the method is not numerically stable these errors will propagate and amplify, becoming "mysterious" big changes in the output. This is a common issue in GPU computing, as porting a software for the execution on a many-core GPU often requires slight changes in the order of computations and the output may not match with the one produced with the reference serial code. Techniques such as the Kahan summation [46] can reduce the numerical differences, but a specific numerical analysis is often necessary to guarantee specific

levels of accuracy.

Another potential cause of differences between a CPU and a GPU implementation relies in the hardware precision. Among the formats defined in IEEE 754 standard for floating point arithmetic [44], *single precision* and *double precisions* are the most common ones. Their length is respectively 32 and 64 bits and their correspondent C/C++ types are `float` and `double`. While most desktop and workstation CPUs have had hardware support for both types since the '90s, GPUs have, until recently, only featured hardware support for single precision, sometimes offering the possibility to emulate double precision (in NVIDIA cards, an emulated double precision operation is $8\times$ slower than a hardware single precision one). As the computations performed on the GPU were mainly devoted to compute pixel coordinates for graphics, single precision was totally acceptable. Because of the increased request for numerical precision, especially in GPGPU scientific computing, ATI and NVIDIA afterwards recently introduced on-chip support for double precision also in GPU cards. Most non high-end cards, however, still use single precision as default. This is an issue to take into account for applications requiring double precision.

2.4 Recent advances

Both NVIDIA's CUDA and ATI's Stream run on specific GPU hardware, with practically no interoperability between the two. Because of the growing interest for parallel computing and the wider range of hardware available, an architecture-independent parallel computing platform, OpenCL, was proposed in 2008 by Apple and released in 2009 as the result of a collaboration with AMD, IBM, Intel, and NVIDIA [28].

OpenCL was designed to offer an abstraction layer between the applications and the underlying hardware, allowing an application to be transparently executed

on heterogeneous hardware (GPUs as well as multicore CPUs and other hardware). The language is derived from C99 and the API is similar to the low-level CUDA C API. Coding in OpenCL is in general more complex than CUDA C, due to the wider variety of hardware on which it runs, but does not force the programmer to bind to a specific hardware class or brand.

We started approaching the GPGPU paradigm in 2007, when CUDA was the only platform allowing for generic purpose programmability in a C-like language. For this reason, our GPU-enabled simulators are at moment CUDA-based, but we are considering the possibility to port our softwares to the OpenCL platform in a near future.

Chapter 3

CUDA

As mentioned before, the CUDA platform consists of a hardware and a software part. The hardware required to run a CUDA routine can be found quite easily as any graphic card produced by NVIDIA since the release of the GeForce 8 series (2006) is CUDA-capable[1]; many old PC's are therefore already CUDA-enabled, while it is possible to buy a low-end CUDA-enabled card with less than €50.

On the software side, a CUDA application relies on two layers: a CUDA-enabled video driver, which is capable of communicating with the graphic card at the operative system level, and the CUDA Toolkit, which is a set of shared libraries, tools and bindings needed to compile the code for the GPU and integrate it into a "normal" application. There are two different programming interfaces to the CUDA runtime libraries: the low-level API, that exposes to the programmer a higher number of structures to access and use a GPU, and the high-level API, that automatizes the most common operations. The latter one allows a programmer to drive a GPU with just a few C functions that closely follow the naming convention and parameter order of similar functions from the C standard library.

[1]The NVIDIA website states that CUDA-enabled GeForce cards are "GeForce 8, 9, 100, 200, 400-series, 500-series GPUs with a minimum of 256MB of local graphics memory".

3.1 Compute Capabilities

CUDA-enable applications are in general backwards-compatible with all CUDA-enabled hardware, but new generation cards may offer additional features not supported in previous devices. The set of capabilities of a given generation has version number referred as *compute capabilities* and it is 2.1 in the latest generation cards (*Fermi* architecture) as of this writing. As an example, one needs a card with at least compute capability 1.1 to use atomic operations and 1.3 for hardware support for double precision. One could define the compute capabilities as the hardware version of a card, the major number referring to the generation; the Toolkit has an independent version number, which recently has arrived to 4.0.

The toolchain used to implement a function for the GPU consists of the following steps:

1. Write the function (*kernel*) to be executed on the GPU in CUDA C with any IDE/text editor, even mixing CPU and GPU functions in the same source files;

2. Compile the files containing GPU code with the Toolkit compiler (`nvcc`);

3. `nvcc` transparently compiles the GPU code into the card's assembly[2], lets a predifined C/C++ compiler (`gcc` in Unix-based systems) compile the CPU code and links together the GPU and CPU binary objects.

It is possible to run the resulting application on any hardware with compute capability equal or greater than the target one, as long as a possibly up-to-date video driver is installed. From the programmer's perspective, the only change from the usual toolchain is using `nvcc` in place of `gcc`.

[2] `nvcc` compiles the GPU code into both PTX (a proprietary assembly which abstracts from the hardware, always forward compatible) and CUBIN (the actual binary code for the GPU, forward-compatible only with CUDA architectures of the same compute capability major version); when a CUDA application is run, it loads the CUBIN to the GPU, if compatible, or does a just-in-time compilation to produce a proper CUBIN from the PTX.

3.2 Multiprocessors

Although programming with CUDA platform does not require the knowledge of the details fo the underlying hardware, the way GPU threads are launched and the characteristics of features like shared memory strongly reflect the way GPU cores and memory banks are organized in the card. It is therefore advisable to learn at least the basic principles in order to be able to fully exploit the hardware capabilities.

Similarly to other NVIDIA and non-NVIDIA GPUs, CUDA-enabled cards host one or more SIMD multiprocessors each embodying several computational cores, for a total of tens or even hundreds of cores. Recent cards such as the GTX 590 host up to 1024 CUDA cores, for a total 3 billion transistors. More specifically, each multiprocessor hosts 8 to 32 ALUs and one instruction unit that decodes one instruction at a time to be executed in *lockstep*[3] by multiple threads. Distinct multiprocessors execute thread batches (called *warps* in CUDA terminology) in parallel but independently from each other.

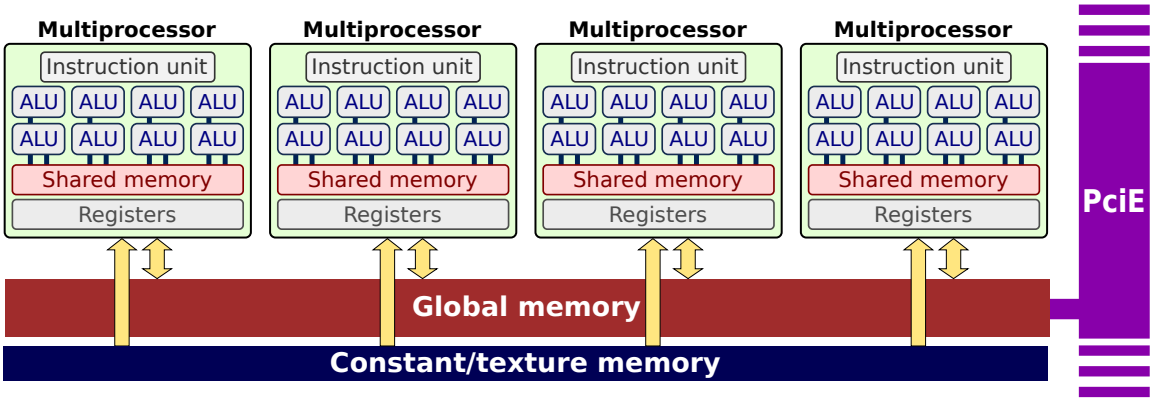


Figure 3.1: Schematic representation of how GPU cores are grouped into multiprocessors and linked to shared and global memory banks.

Recent consumer-level cards host up to 1.5Gb GDDR5 memory and support hardware double precision; high-end cards like Tesla host larger RAM memory

[3]Here intended in the parallel computing meaning, as the same instruction is performed synchronously by different cores, not in the meaning of redundant computing.

with optional ECC support. There are other memory resources, such as a global constant memory (cached, read-only) and textures (spatially cached, read only until rebound, although more recent hardware allows writing to texture too). Each multiprocessor also hosts a bank of *shared memory*, directly accessible by all ALUs and much faster than global memory (RAM). Shared memory, however, has a limited size (16-48Kb) and its temporal scope is limited to the execution of one block. Because of its high speed and low latency, shared memory is often used to exchange data between threads of the same block, handle race conditions or simply avoid multiple reads in global memory by storing them in a shared buffer. It is up to the programmer to carefully evaluate which kind(s) of memory it is convenient to exploit for a specific problem and how to access them.

3.3 Kernels, warps, blocks

We already referred to the series of instructions to be executed on a GPU as *kernel*; we can see a kernel as a function compiled for the GPU and meant to be executed in parallel on multiple data streams (SIMD).

The instance of a kernel is called *GPU thread*, or simply *thread* when the context does not allow for any ambiguity. Kernels are instantiated in parallel threads and threads are grouped into *blocks* with one, two or three dimensions; blocks are in turn arranged in a one-, two- or (only with the most recent compute capabilities) three-dimensional grid. The block and grid sizes are up to the programmer, and can be decided at runtime. The position of a kernel in a block and the one of the block in the grid, as well as the block and grid sizes, are available inside the kernel in the form of implicitly defined variables, so that it is possible for each thread to compute the address of the data to be accessed.

Blocks have a size limit that may depend on the underlying architecture and this limit is in general lower than the number of threads a typical application

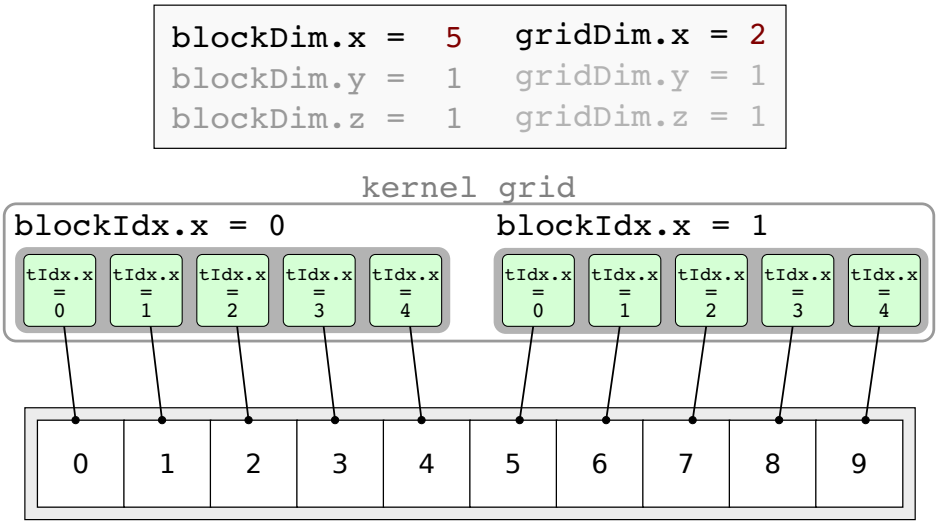


Figure 3.2: Simple example of two one-dimensional blocks each assigned to a portion of a global array.

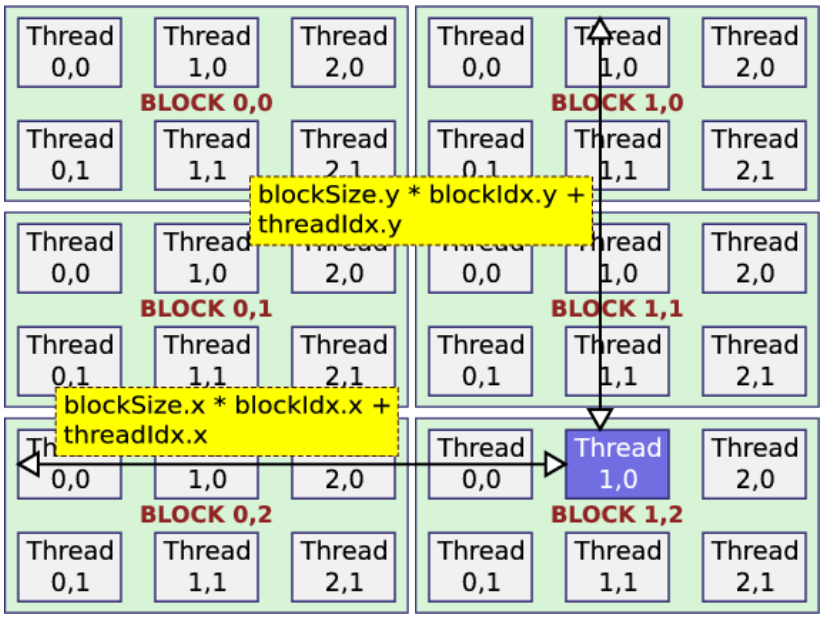


Figure 3.3: Example thread indexing on a 2D grid of 2D blocks.

would instantiate. This means that grouping the threads into blocks is not just possible, but often necessary, and it is related to the hardware structure.

Each block is assigned to a multiprocessor which, as already mentioned, executes multiple threads simultaneously (a warp). When the number of threads is greater than the number of cores, a minimal context switch is executed to let all threads complete. When a thread issues a memory request that may take several cycles (e.g. accessing a word in the RAM memory of the card takes about 400 clock

cycles in 2.0 hardware, 800 clock cycles in previous cards) the context switch lets other threads execute while one is waiting for the resulting data. Thus, instantiating more threads than cores (and, in general, issuing enough computations to *saturate* the hardware) leads to a higher throughput as time-consuming memory requests are covered by computations.

Another time-consuming factor is *divergency*. When two or more threads of the same warp need to execute different instructions because of a branch divergence, those operations are serialized. The programmer should take care to avoid as much as possible that threads of the same warp diverge.

3.4 Memory types

There are several kinds of memory available on the device, each with specific features and limits:

- **Registers**: R/W, very fast, not cached, local to each thread; it is not possible to access them directly. The total number of registers on a card is fixed and it is divided among the threads; this limits the complexity of kernels and the number of concurrent threads.

- **Shared**: R/W, fast, not cached, local to a multiprocessor and shared among all the threads of a block; usually only a few tens/hundreds of Kb are available per multiprocessor.

- **Global**: R/W, slow, usually not cached[4], accessible to all threads; it is the RAM memory of the card, usually sized in terms of gigabytes (minus a small buffer used to paint the active screen) in modern GPUs.

- **Constant**: R/O and cached for the GPU, R/W for the CPU-side; useful for small global constants. It is as fast as registers if all threads in a warp access

[4]The Fermi (2.x) CUDA architecture introduces L1 and L2 caches for global memory.

the same datum.

- **Texture**: R/O and cached for the GPU, can be initialized only by the CPU. Supports built-in interpolation, clipping, and special address modes. The caching is spatial (probably using a Z-curve [27]).

GPU threads can not access directly any host memory area[5]; to exchange data between the host and the device, specific CUDA functions must be called from the host side. We will conform to a CPU-centered nomenclature and call *download* any copy operation from device to the host and *upload* any transfer in the opposite direction.

3.5 Coalescence

It is up to the programmer to choose the most appropriate location for the data, but it is even more important the way those data are accessed, especially when accessing the global memory: the controller is able to pack multiple requests to adjacent memory addresses into one long read, causing multiple potential memory access to collapse to a single one. Correctly aligned memory requests are referred to as *coalesced*. These pattern slightly change across different compute capabilities, but as a rule of thumb consecutive threads should access consecutive words in memory and memory requests should begin with addresses that are multiples of a word. Figures 3.4, 3.5 and 3.6 summarize, as showed in the CUDA Programming Guide, how the hardware controllers of different compute capabilities are able to optimize non-consecutive or misaligned accesses, with important consequences on the overall performance.

One of the practices suggested by NVIDIA's to improve coalesced accesses, for example, is to arrange the data in a *structure of arrays* fashion rather than an *array*

[5]It is possible with some specific hardware configurations, e.g. integrated GPUs sharing RAM with the CPU.

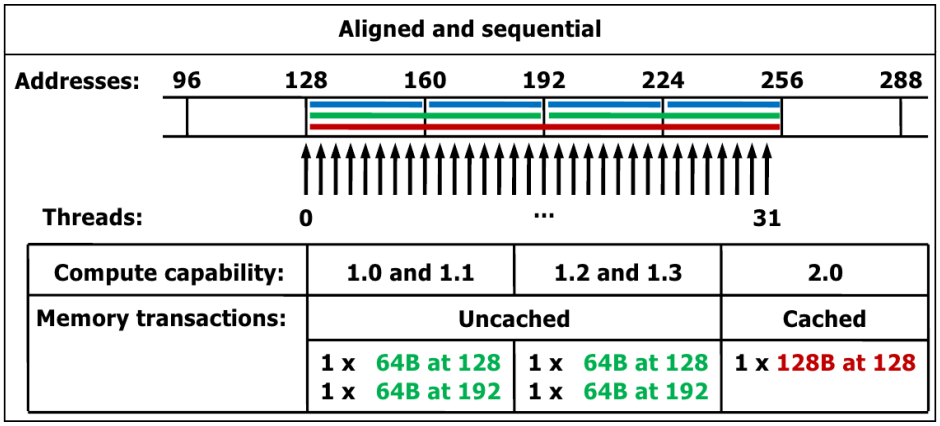


Figure 3.4: Coalescence - case 1: accesses are aligned and sequential (best case)

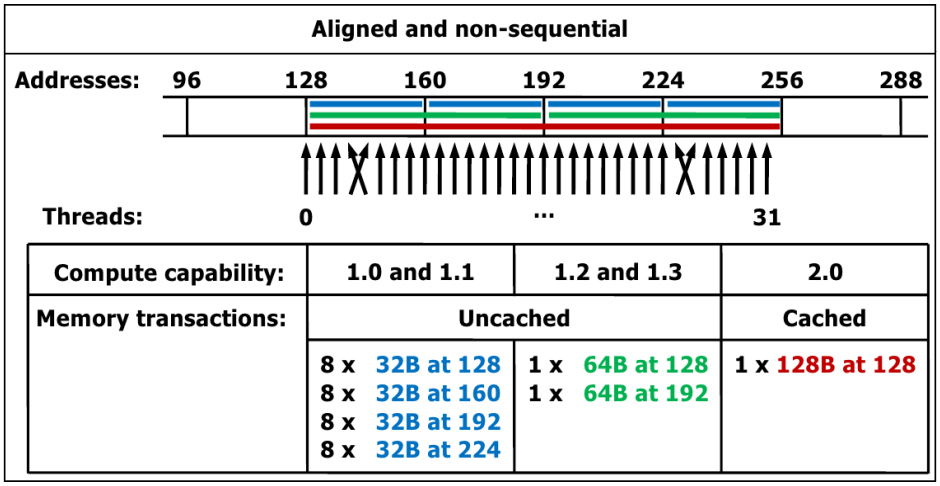


Figure 3.5: Coalescence - case 2: accesses are aligned but not sequential

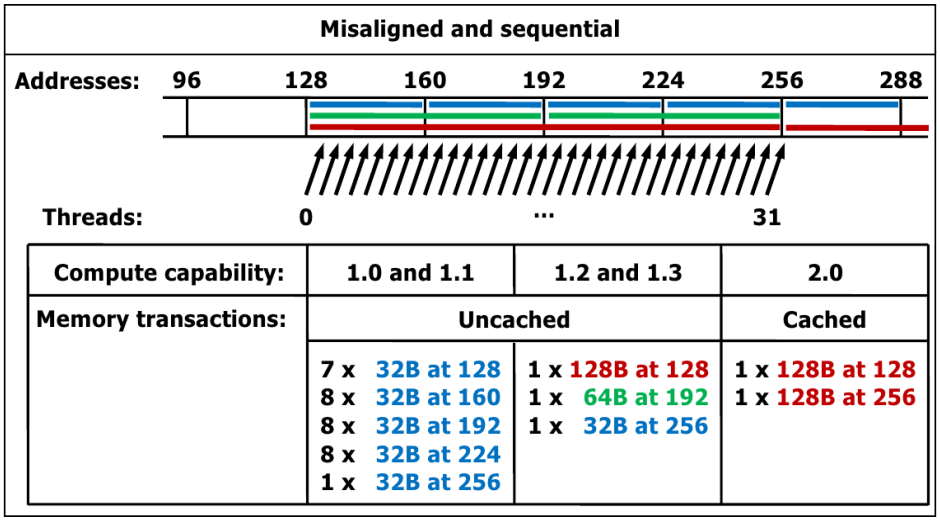


Figure 3.6: Coalescence - case 3: accesses are not aligned but sequential

of structures [63]. Both MAGFLOW and GPUSPH are implemented following this recommendation.

In some classes of problems, however, it is not possible to predict or correct the sparsity of some accesses. In the SPH model, for example, accessing the neighbors of consecutive particles often leads to misaligned memory accesses. We could think of putting in the list of neighbors a *copy* of the data instead of the addresses of the neighbor particles; however, this could require up to one hundred times more GPU memory and is thus unfeasible.

3.6 Contexts

A *CUDA context* is the set of variables and structures used internally by the CUDA runtime libraries to communicate with a specific device. The high-level API automatically initializes a CUDA context when a CPU thread first tries to access a GPU, in a one-context-per-thread fashion; the low-level API instead allows to create multiple CUDA contexts and to switch from one to another within the same CPU thread. Using multiple GPUs with the high-level API requires therefore one CPU thread per device.

Since the recent release of CUDA 4.0, this is not necessary anymore, as it is possible to switch from one device to another within the same CPU thread with a specific high-level call [64]. As we implemented the simulator with CUDA 3.2 using the high-level API, we chose to launch one CPU thread per device. This also have the advantage of allowing a cleaner design and a more robust code, while transparently exploiting multiple CPU cores, if available, to perform the host tasks.

3.7 Streams

A CUDA Stream is an abstraction for a logical sequence of operations that it is possible to run on the GPU. In a CUDA Stream it is possible to enqueue a kernel launch, a data transfer or a CUDA Event, a dummy structure needed for GPU timing and synchronization of streams. If multiple streams are run in parallel, and the pending operations of a stream are compatible with those of another stream (e.g. kernel and data transfer in older cards, two different kernels in devices with compute capability 2.0), the compatible operations are run simultaneously.

3.8 Asynchronous API

All kernel calls are non-blocking, i.e. the CPU code keeps executing after launching a kernel and CPU and GPU can work independently. To make the CPU code wait for a kernel to complete, for example when we need the result of the kernel computation, we can call the `cudaThreadSynchronize()` method. Memory transfers, instead, are by default blocking: a `cudaMemcpy()` method only returns when the transfer is complete. The CUDA platform offers the capability to run memory transfers concurrently with CPU and GPU computations. We access this functionality through the *asynchronous API*, which is a set of library calls mostly terminating with the `-async` suffix.

Listing 3.1 shows an example of concurrent kernels and memory transfers, where the computations on the input data start as soon as the first part has been uploaded. Note that three different `for` cycles are used to iterate on a set of streams, enqueueing the operations in a breadth-first fashion. Although in some applications a depth-first fashion could be preferable (e.g. processing video streams), using distinct `for` cycles is necessary for actual asynchronous execution because of the way the CUDA library was designed. The CUDA runtime enqueues the operations in two internal queues, one for kernels and one for memory trans-

fers. If all the operations of the same stream are enqueued in a row, the runtime scheduler will not find any compatible operation to the one being issued as they belong to the same stream. Listing 3.2 shows an "incorrect" depth-first enqueueing leading to a synchronous execution.

```
1  // enqueue the upload of parts of the data
2  for (int i = 0; i < NSTREAMS; ++i)
3      cudaMemcpyAsync(inputDevPtr + i * partial_size,
4          hostPtr + i * partial_size, size,
5          cudaMemcpyHostToDevice, stream[i]);
6  // enqueue the partial computations
7  for (int i = 0; i < NSTREAMS; ++i)
8      MyKernel<<<100, 512, 0, stream[i]>>>
9          (outputDevPtr + i * partial_size,
10         inputDevPtr + i * partial_size, size);
11 // enqueue the download of the results
12 for (int i = 0; i < NSTREAMS; ++i)
13     cudaMemcpyAsync(hostPtr + i * partial_size,
14         outputDevPtr + i * partial_size, size,
15         cudaMemcpyDeviceToHost, stream[i]);
```

Listing 3.1: Example code for issuing concurrent kernels and memory transfers in a breadth-first fashion.

There are two requirements to use asynchronous transfers. The first is the usage of streams. A kernel and a memory transfer must be issued on different, non-zero streams. Issuing two operations on the same stream (or not specifying a stream, that results in using stream 0 for both) tells the GPU scheduler that the second depends on the first, and thus it has to wait for it to finish.

The second is that all CPU buffers involved in the transfers must be allocated as *page-locked*. When a CPU process is switched or paused by the scheduler, its memory area may be swapped in virtual memory by the operative system, and it is

restored as soon as the process is active again. This may cause concurrency and/or inconsistency problems for asynchronous transfers, that remain alive despite the state of the issuing process. For this reason, we must ask the operative system to allocate a special kind of memory that is not paged (page-locked) even if the owner process is paused. CUDA offers some simplified methods to issue this special kind of allocation (`cudaHostAlloc()` or `cudaMallocHost()`). Page-locked memory is also faster than non page-locked, as it prevents page faults and enables direct GPU-RAM DMA transfers. However, it is advisable to minimize its use, as page-locking a large amount of RAM memory may result in increased swapping and reduced overall system performances.

```
1  // enqueue the upload of parts of the data
2  for (int i = 0; i < NSTREAMS; ++i) {
3      // upload input data
4      cudaMemcpyAsync(inputDevPtr + i * partial_size,
5          hostPtr + i * partial_size, size,
6          cudaMemcpyHostToDevice, stream[i]);
7      // kernel launch
8      MyKernel<<<100, 512, 0, stream[i]>>>
9          (outputDevPtr + i * partial_size,
10         inputDevPtr + i * partial_size, size);
11     // download output data
12     cudaMemcpyAsync(hostPtr + i * partial_size,
13         outputDevPtr + i * partial_size, size,
14         cudaMemcpyDeviceToHost, stream[i]);
15 }
```

Listing 3.2: Example code for issuing concurrent kernels and memory transfers in a depth-first fashion. Transfers are likely to be performed in a synchronous way.

3.9 Libraries

Numerical problems often require high-level operations like sorting big arrays or finding the minimum/maximum element. As it is not trivial to implement such operations in an efficient way on parallel hardware, some pre-optimized libraries have been released to this aim.

For example, CUDPP (CUDA Data Parallel Primitives) is "*a library of data-parallel algorithm primitives such as parallel prefix-sum (scan), parallel sort and parallel reduction*" [65]. It runs on CUDA-enabled hardware and it is released under the New BSD license [6] [77]. In particular, we use the CUDPP Radixsort [71] in GPUSPH and the CUDPP minimum scan [72] in both MAGFLOW and GPUSPH.

Other libraries that are worth citing include CUFFT for parallel Fast Fourier Transform, CUBLAS for linear algebra solvers, CUSPARSE for efficient handling of sparse matrices, CURAND for quasi-random number generation and Thrust, a template library offering similar functions to CUDPP with a higher level of abstraction.

[6] The BSD license is a variant of GNU GPL allowing the code to be used in commercial, closed-source applications.

Chapter 4

Multi-GPU computing

We already mentioned that only problems exposing some degree of parallelism are suitable for a parallel implementation on the GPU. Numerical problems exposing a high level of intrinsic parallelism, however, can be mapped to more than one level of parallelism. In general, if a problem is modeled as a set of independent subproblems, then it is possible to split the set of required computations in two or more partitions that can be executed in parallel on separate devices.

4.1 Motivation

The main motivation to split a problem into multiple devices is the possibility to achieve a gain in performance. The highest theoretical speedup obtainable with D devices is $D\times$, but in rare cases it can be even higher [1].

For some problems there is also a memory reason. Some numerical problems can benefit from the increased total memory of a multi-device systems by working on a wider range of data or on inputs with higher density. It is the case of very wide or very dense SPH simulations, whose number of particles would not fit in the memory of one GPU.

[1] For example, the split may relieve a congested PCI bus, increasing the data trasfer speed between the CPU and the devices and therefore causing a *superlinear* speedup, as in [11].

In some cases, a substantial speedup or memory gain can even have consequences on the model level. With a problem that is solved one order of magnitude faster than the prospected time, for example, it is possible to think of applying a stochastic super-model to refine or bound the numerical results. A multi-device implementation of a numerical problem is more likely to reach such a result if it is robust and truly scalable, as it allows to increase performance proportionally to the available computing power.

However, while exploiting the computational power of a GPU is today far easier than at the beginning of the GPGPU era, porting a numerical solver to GPU in an efficient way is still far from trivial and exploiting more than one GPU at a time is even more difficult due to technical and model constraints. There exist multi-GPU softwares mainly for data compression and visualization [3, 45, 76], molecular dynamics [82, 52, 67] and a few grid-based models [5, 68, 83, 81, 66]. To the best of our knowledge, however, no Navier-Stokes SPH simulator has been implemented for multi-GPU yet, although some works claim that a port to multi-GPU is in progress.

4.2 Naïve approach

A trivial approach to run a problem in parallel on multiple devices could be to split the problem domain, whatever it is, in a number of partitions equal to the number of devices to be exploited. Recalling that the graphic processor has its own, coupled memory, one could copy the necessary, partial data to the devices, perform the computations and finally transfer back the results; this should be repeated for every set of computations to be performed (e.g. every step of a simulation).

Given the bus data rate and the mathematical throughput of the computing units, it is easy to estimate the ratio between the two and the potential convenience of the approach. Unfortunately, except for very particular cases, the time required

to transfer the problem data largely overcomes the time required to perform the computations, making the whole process disadvantageous. Moreover, as direct inter-device communication is in general expensive or even not possible, any inter-dependence among parts of the problems adds a technical constraint that must be carefully taken into consideration.

4.3 Subproblems inter-dependence

Some numerical problems, such as astrophysical n-body simulations, present completely inter-dependent subproblems and therefore can not be easily split, as each element requires accessing sparse data of the whole domain. In these cases, one should rethink the model, if possible, or use an approximate solution. Some classes of problems expose instead a complete subproblem independence; in this cases, it is enough to partition the problem with a greedy strategy and run the subparts in parallel on different devices. We refer to these fully parallel problems as *embarrassingly parallel*.

Most problems are somewhere in the middle. The particle methods we will describe later, for example, are by definition made up of quasi-independent subproblems: the state of a particle depends on the previous state of the same particle and on the state of neighbor particles only. It is possible to split such a problem as long as a proper overlap is left between the subproblems; we need to properly allocate, handle and keep updated this overlap.

4.4 Overhead

Running a problem on multiple devices introduces an unavoidable overhead due to the split operations and load balancing. Most problems, especially the ones involving some subproblem overlap, also require a variable amount of data to be

continuously transferred during the computations, to keep the overlaps updated and deal with dynamic changes (e.g. a particle "moving" from one device to another). Data transfers between a device and the host, as well as device-to-device transfers, are the predominant overhead factor in most multi-GPU implementations. Moreover, as in the general case it is not possible to transfer data from one device to another in a direct manner, data must be first copied to the host and then copied back to the recipient device, thus doubling the already significant transfer times. Making all these overheads negligible is the main challenge of any multi-GPU system where a continuous inter-device communication is needed.

4.5 Speedup metrics

Comparing the performance of different problems is usually tricky and sometimes even unfeasible, due to the possible differences in the type of data or the measured parameters (e.g. speed vs. accuracy or input size). It is however possible to estimate the efficiency of a parallelization by means of measured speedup with respect to a single-core execution. We refer to cores for simplicity, but the following considerations are still valid for either multi-core, multi-GPU or generic many-core parallel computing.

A common starting point to analyze a parallel implementation is to model the problem as made by a parallelizable part and a serial, non parallelizable one. The latter includes operations which are performed anyway in a serial manner (e.g. file I/O) and operations that are introduced with the parallel implementation (*overhead*, e.g. load balancing). Let α be the parallelizable fraction of the problem and β the non parallelizable one; it is $\alpha + \beta = 1$, with α, β non-negative real numbers. For embarrassingly parallel problems α is equal or very close to 1; problems intrinsically serial, where each step depends on the outcome of the previous one, have $\beta \approx 1$. It is advisable, when the model allows for it, to formulate a problem

in a way to minimize β to the strictly necessary.

4.5.1 Amdahl's law

Amdahl's law, introduced in 1976 by the computer architect Gene Amdahl [2], measures the speedup in terms of execution time and gives an upper-bound inversely proportional to β. Although it is mainly used in parallel programming, where number of available computing units is often directly related with the obtainable speedup, it can be used in the analysis of a serial problem when only a portion of it is being improved.

With a slight change in the notation, Amdahl's law can be written as

$$S_A(N) = \frac{1}{\beta + \frac{\alpha}{N}} \tag{4.1}$$

where N is the number of used cores and S_A the theoretical maximum speed-up. Fig. 4.1 plots the outcome of the law with different ratios α/β, the asymptotic limit of each being $1/\beta$.

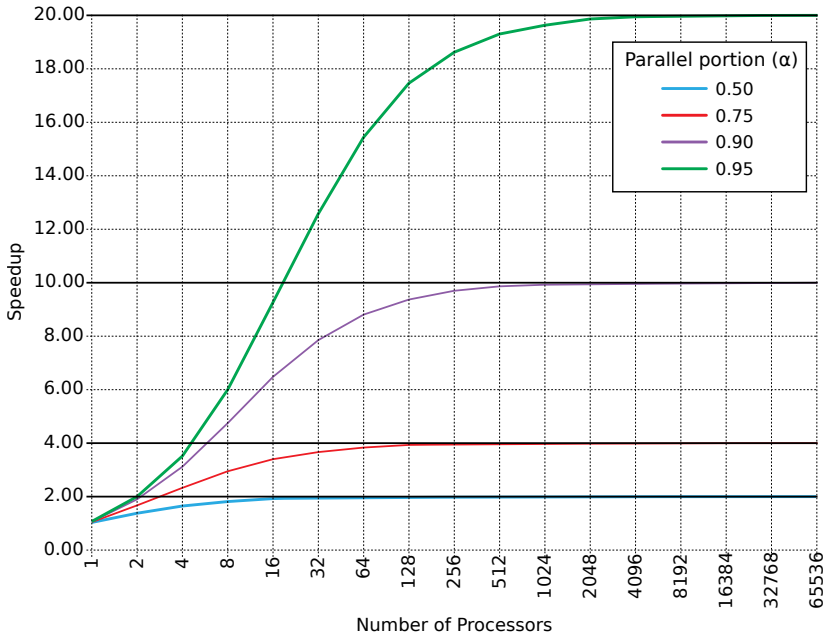


Figure 4.1: Plot of Amdahl's law with different values of α

Amdahl's law assumes that the problem size is fixed and the number of cores is the only variable. The intuitive meaning is rather simple: the time T_p required to solve a problem with N cores is $T_p = \beta T_1 + \alpha T_1 / N$, and the ratio T_1 / T_p (equivalently, $1 / T_p$ with $T_1 = 1$) is the expected speedup.

In case the parallel fraction is not known *a priori*, it is possible to rewrite the formula for α:

$$\alpha_{\text{estimated}} = \frac{\frac{1}{S_e} - 1}{\frac{1}{N} - 1} \tag{4.2}$$

where S_e is the empirical speedup obtained with N cores.

4.5.2 Gustafson's law

In 1988 J. Gustafson et. al. proposed a different speedup model [29], claiming that Amdahl's law was way too restrictive. In particular, they challenged the assumption that the problem size is always fixed, remarking that the size of a problem often scales with the available raw computing power. In other words, as they clearly explain in the paper, "*rather than ask how fast a given serial program would run on a parallel processor, we ask how long a given parallel program would have taken to run on a serial processor*".

Gustafson's law can be formulated as:

$$S_G(N) = N - \beta \cdot (N - 1) \tag{4.3}$$

The obtained speedup is equal to the number of used cores minus the non parallelizable fraction, that makes $N - 1$ cores unexploited for β fraction of the problem. The plot of Gustafson's law is just a line less steep than $x = y$. With this formulation, there is no theoretical upper-bound to the speedup, as long as it is always possible to indefinitely increase the workload. When compared to Amdahl's law, this is often referred to as *scaled speedup*.

4.5.3 Karp–Flatt metric

The Karp–Flatt metric was proposed in 1990 as a measure of the efficiency of the parallel implementation of a problem [47]. The new efficiency metric is the experimental measure of the serial fraction β of a problem. As it can be imagined, this can be proven to be consistent with Amdahl's approach, and can be derived from it.

With another slight notation change, it is:

$$\beta_{KF} = \frac{\frac{1}{S_e} - \frac{1}{N}}{1 - \frac{1}{N}} \tag{4.4}$$

where S_e is the empirically measured speedup and N is the number of cores, as usual. This can also be obtained from Amdahl's law by rewriting for β. Since this *a posteriori* metric measures both serial time and introduced overhead, it is especially useful to apply to at least two executions with different N: a constant β_{KF} means the the parallelization was efficient, while a growing β_{KF} means that some overhead is being introduced by the growing number of cores.

4.6 Linear overhead

We already mentioned that the highest theoretical speedup obtainable with D devices is $D\times$ and that, except for the so called embarrassingly parallel problems, we usually introduce some overhead due to the synchronization and data transfer operations. We now informally analyze the consequences of a linear cost (i.e. β_{KF} linear with the number of devices), which occurs when a non perfect parallelization is implemented or no dynamic load balancing is used.

We may wonder, for a specific problem, at what point the introduced overhead exceeds the advantage of increasing the number of devices by one, thus determining an upper-bound to the number of devices that is convenient to use. Please

note that in this context we use the word *convenient* referring to the performance metrics (using D devices is convenient if the problem is solved faster than using $D-1$ devices), and not to the memory gain.

Let us compute first the performance gain of using D devices instead on $D-1$, in an ideal case. Say T the time required to solve a problem with one device. It will take T/D time to solve the problem with D devices, and $T/(D-1)$ with $D-1$ devices. Using D devices takes

$$\frac{T}{D-1} - \frac{T}{D} \quad = \quad T(\frac{1}{D-1} - \frac{1}{D}) \quad = \quad \frac{T}{D(D-1)}$$

time less than using $D-1$ devices. Dividing by T we obtain the gain in percent:

$$\frac{1}{D(D-1)} \tag{4.5}$$

This number decreases quite fast. Table 4.1 shows the progressive gain, in percent, from 2 to 8 devices. For example, using 8 devices we expect a problem to be solved in $1/(8(8-1)) \approx 1,8\%$ less time than using 7 devices.

Let Q be the overhead introduced each time we increase by one the number of used devices; it is possible to estimate the mentioned upper-bound D_{max} by choosing the maximum D for which the inequality

$$\begin{aligned} Q/T < \frac{1}{D(D-1)} \quad &\Rightarrow \\ Q < \frac{T}{D(D-1)} \end{aligned} \tag{4.6}$$

holds. It is implicit that for the inequality to be defined it must be $D > 1$ (indeed, we can safely assume that it is convenient to use one device instead of zero) and $T > 0$ (no need to split a simulation that takes no time).

Devices	2	3	4	5	6	7	8
Progressive gain	50,00%	16,7%	8,3%	5%	3,3%	2,4%	1,8%

Table 4.1: Ideal progressive gain, in percent, when using 2 to 8 devices

4.7 Timeline profiling

While debugging and testing our first multi-GPU prototypes, we soon felt the need to profile accurately the execution of the simulations across multiple devices to find out the effectiveness of asynchronous operations and to detect potential bugs and unwanted race conditions.

In year 2008 NVIDIA released the Visual Profiler, a multi-platform profiling utility now provided as part of the CUDA SDK tools [16]. The Visual Profiler runs a CUDA program in a special environment that makes the CUDA runtime log information about the ongoing operations. Example of information include the number of cache hits and misses, timestamps and duration of kernel executions and memory transfers, number of coalesced and uncoalesced memory access, *occupancy* of a kernel, and other. Is is also possible to add custom fields by using special registers within the kernel code. However, the Visual Profiler did not provide until recently a graphical plot and comparison of the execution timelines, especially for multi-GPU environments. Moreover, all asynchronous operations were made synchronous for logging purposes, voiding any possibility to check the actual overlap between transfers and computations.

The CUDA runtime offers the possibility to log, optionally in CSV format, information such as the beginning and ending timestamps of any kernel or transfer performed on the GPU, as long as some special environment variables are set. It is not easy to interpret the logged information, as this feature is very poorly documented and the meaning of some fields is still unknown, but this allows not to lose the overlap among asynchronous operations. We then implemented a little tool that takes in input the log of an execution and produces a custom multi-device

timeline in SVG format.

The output is a fully configurable timeline with indications about the absolute and relative execution times. It shows the time at which an operation has been issued on the CPU and the time it was actually performed on the device. Thanks to the powerful ECMA script feature of the SVG standard, it is possible to see detailed informations about an event just by hovering the mouse on an event, as shown in fig. 4.2. It is possible to configure via command-line a number of graphical parameters (e.g. horizontal drawing scale) and filtering options (e.g. hide CUDPP calls or replace decorated kernel names); the visualization can be organized by method name (fig. 4.2) or by stream (fig. 4.3) and it is possible to show the timelines of different GPUs in the same view. The only important limit currently known is the lack of an accurate synchronization between timelines of different devices. Unfortunately, each device uses its own clock-driven timeline and there is no documented way to fix an absolute relation between any two. We thus had to synchronize by means of different estimates, e.g. knowing the process start time or timing a dummy kernel launch as a reference.

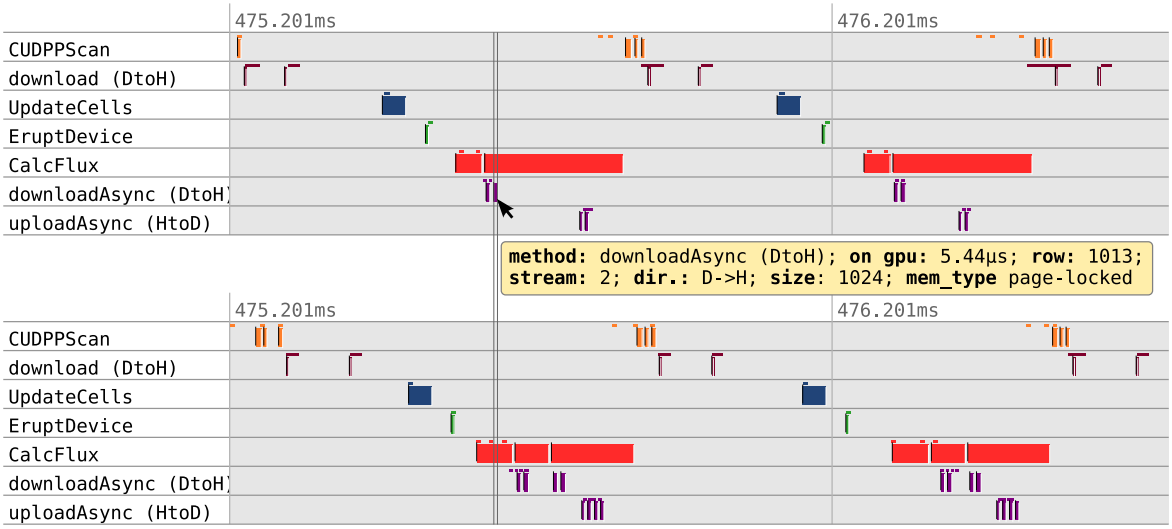


Figure 4.2: Timeline produced for a 3-GPUs MAGFLOW simulation (only 2 are shown). The overlap between the bordering slice transfer (in purple) and the `CalcFlux` kernel (in read) is immediately recognizable. Together with the yellow popup, two event delimiters appear on hovering, to compare the event time with other events even in different GPUs.

This little tool boosted the testing and debugging process of both simulators, allowing to rapidly find minor bugs that prevented some operations or even different devices to overlap their execution. In general, a detailed graphical representation of what is really happening in the device helps to evaluate quickly the efficiency of new design hypotheses and fine tune execution parameters.

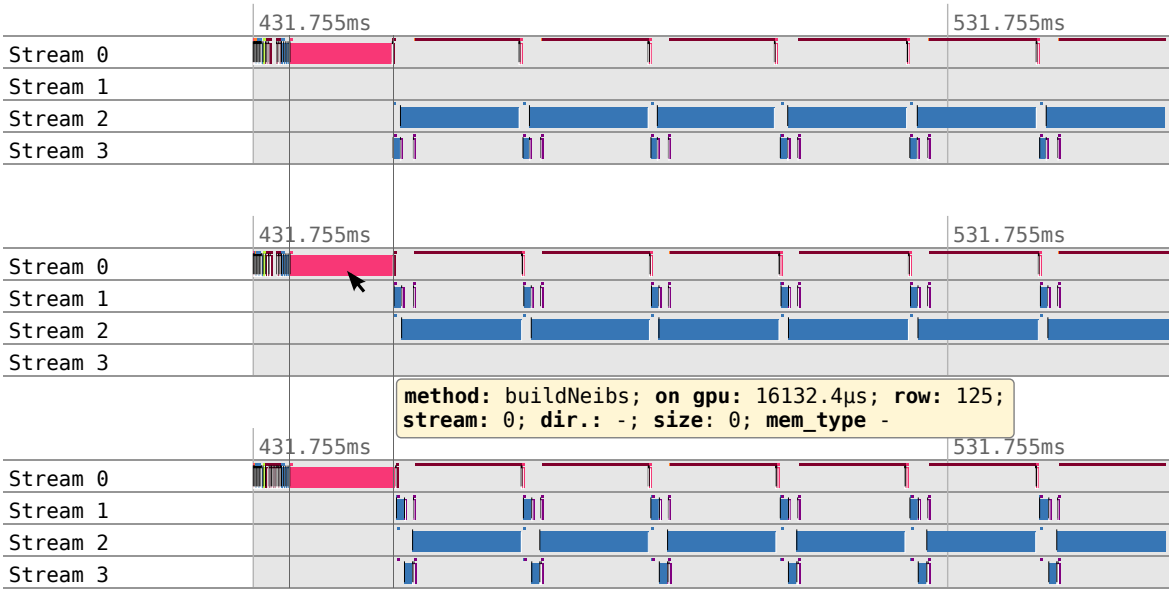


Figure 4.3: Timeline produced for a 3-GPUs GPUSPH simulation, aligned by stream. Note the dependency of memory transfers on previous kernels of the same stream, esp. in GPU n.3 (the central one of the simulation, with 2 bordering slices).

The timeline feature, which at the time we started working on multi-GPU was only available as a plugin for the Microsoft Visual Studio commercial IDE for Windows [15], seems to have been finally introduced in recent CUDA 4.1RC release [14]. Although one of the counters is supposed to measure the overlapping ratio of calls to the asynchronous API, it seems many calls are still made synchronous, and this is confirmed by the bad performance of programs being profiled. It is therefore still not possible to profile a multi-GPU program and track the actual amount of asynchronous operations unless using a non-official profiler like the one we developed.

In the future, the timeline tool might be improved for general use with a user friendly interface and, most important, an automatic system to synchronize the

timelines of different devices (e.g. with a timed dummy kernel launch).

4.8 Testbed

4.8.1 MAGFLOW

Our test-bed for the MAGFLOW simulator was the simulation of the lava eruption on Etna volcano in the year 2001 on DEMs with 2m and 5m resolution. In this scenario there is a single vent emitting up to $30m^3/s$ fluid lava with a pseudo-gaussian trend. The lava was modeled with a Bingham rheology [9] and a Giordano-Dingwell viscosity [24] with a solidification temperature of 800° and a 0.2% water percentage. Some sets of simulations were resumed from one already 19% complete, where the bounding box of the active flow had already reached its maximum extent.

4.8.2 GPUSPH

In GPUSPH the scenario to be simulated (topology plus various parameters) is encapsulated into the C++ virtual class `Problem`. A class that inherits from `Problem` must define the environment (e.g. walls) and the amount and initial shape of fluid. Optionally, it is possible to override the default physical and simulation parameters. We used two classes in particular, `DamBreak3D` and `BoreInABox`, both defining a $0.43m^3$ parallelepiped of water in a box. The former containes a column-shaped obstacle almost in the center; the latter has a wall on one side partially containing the fluid and an optional second wall making a corridor. In the `DamBreak3D` the fluid flows almost evenly in the domain, while in `BoreInABox` the flow soon becomes strongly asymmetric and flattens afterward in the non-corridor variant. The SPH is set to use a Wendland kernel with a density in the range $4 \cdot 10^{-3}...4 \cdot 10^{-3}$; the fluid was modeled with an artificial viscosity law

($\alpha = 4$) and a speed of sound $c = \sqrt{300 \cdot 0.4 \cdot g}$ m/s.

4.8.3 Hardware

Our testing platform is a TYAN-FT72 rack mounting 6×GTX480 cards on as many 2nd generation PCI-Express slots. The system is based on a dual-Xeon processor with 16 total cores (E5520 at 2.27GHz, 8Mb cache) and 16Gb RAM in dual channel. Each GTX480 has 480 CUDA cores grouped in 15 multiprocessors, 48Kb shared memory per MP, 1.5Gb global memory with a measured datarate of about 3.5Gb/s host-to-device and 2.5 Gb/s device-to-host (with 5.7 Gb/s HtD and 3.1 Gb/s DtH peak speeds for pinned buffers).

The operative system is Ubuntu 10.04.2 LTS Lucid Lynx 2.6.32-33-server SMP x86_64, gcc 4.4.3, CUDA runtime 3.2 and NVIDIA video driver 285.05.09.

Part II

The MAGFLOW simulator

Chapter 5

The MAGFLOW simulator

The MAGFLOW model is an advanced state-of-the-art Cellular Automaton (CA) for lava flow simulation. Its physics-based cell evolution equation guarantees accuracy and adherence to reality, at the cost of a significant computational time for long-running simulations. We here provide an overview of the related work, then describe the model behind the simulator and finally the single- and multi-GPU implementations.

5.1 Related work

The CA-based approach is probably one of the most successful for the simulation of a lava flows [17, 43, 54, 4]. The computational domain is represented by a regular grid of 2D or 3D cells, each characterized by properties such as lava height and temperature; an evolution function for the properties of the cells models the behavior of the fluid.

The MAGFLOW model [80] is a cellular automaton developed at INGV-Catania and capable of modeling many physical phenomena of lava flows like thermal effects and temperature-dependent viscosity change.

It has been successfully used both to reproduce well known past events (vali-

dation) and to predict the behavior of the lava flow in real-time during the Etna eruptions of 2004 [20], 2006 [37, 79] and 2008 [10]. It has been possible to use it for forecasting purposes thanks to its fast GPU implementation, which can simulate several days of eruption in a few hours.

5.2 Model

We here describe briefly the MAGFLOW model as necessary basis to understand the single- and multi-GPU implementation; for a detailed description please refer to [80].

The simulation domain is modeled as a regular 2D grid of cells whose side matches the resolution of the DEM (Digital Elevation Model) available for the area. Each cell carries five scalar properties: ground elevation, fluid lava thickness, solidified lava thickness, heat and temperature. Cells marked as "vent" also have a lava emission rate, which is usually time-dependent.

The state of a cell at step s depends on the state of the eight neighboring cells (Moore neighborhood) and on the state of the same cell at step $s-1$. The lava flux between each pair of neighboring cells is determined according to the height difference using a steady-state solution for the one-dimensional Navier-Stokes equations for a fluid with Bingham rheology.

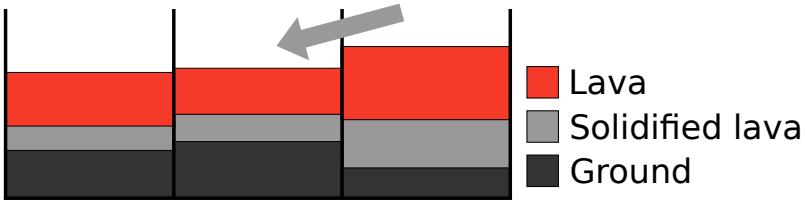


Figure 5.1: Section of three adjacent cells with ground elevation, solidified lava and fluid lava.

The total flux of each cell depends on the lava emission (for vent cells) plus any lava flux with neighboring cells. The actual lava variation is given by the total flux

Q multiplied by the timestep δt for that iteration. A numerical constraint upper-bounds δt to prevent nonphysical simulations. Each cell computes its maximum and the minimum of the maxima is chosen as global timestep.

Each iteration of simulation consists of the following steps:

1. Compute the lava emitted by active vents;
2. Compute the cross-cell lava flux between each pair of neighbors;
3. Find the minimum timestep among the computed maxima;
4. Integrate the flux rate over the chosen timestep: compute new lava thickness, solidified lava and heat loss.

The MAGFLOW simulator was initially developed only for serial CPU execution. As the model suits well a parallel implementation, it was ported to be executed on a many-core GPU.

5.3 Single-GPU MAGFLOW

Porting MAGFLOW to CUDA was relatively easy due to the implicit parallel nature of the model. Each cell only depends on the previous state of the same cell and of a small number of neighbors (eight); the only global operation is finding the minimum δt. We can therefore run one thread per cell and let them run in parallel, each computing its next state.

The CPU implementation of MAGFLOW defined a `Cell` structure holding the data for each cell; the automaton was essentially a list of Cells. The GPU version of MAGFLOW defines instead an array for each property of the cells. This follows the NVIDIA recommendation to use *structures of arrays* rather than *arrays of structures* [63], and it is aimed to improve coalescence in memory accesses.

The four steps of the CPU implementation were straightforwardly implemented as GPU kernels:

1. **`Erupt`**: for each cell, compute lava emission if it is a vent (parallel);
2. **`CalcFlux`**: for each cell, compute the final flux by iterating on neighbors, sum the partial fluxes and compute the maximum δt (parallel);
3. **`MinimumScan`**: find the minimum δt (global scan);
4. **`Update`**: compute new lava thickness, solidified lava and heat loss (parallel).

As showed in the list, the only operation that is not truly parallel is the minimum scan, for which the CUDPP library is used.

The main difference between the serial CPU-based implementation and the parallel GPU-based one is the number of times interactions are computed. The CPU code computes the interaction between two cells once and adds the contribution to both cell (with proper change in sign). The GPU benefits instead from totally independent and parallel computations, and it is more efficient to let each cell compute the interactions with all its neighbors, even if this means computing each interaction twice. This approach simplifies the implementation substantially, since it avoids the concurrency problems that have to be dealt with when partial results are stored on cell by multiple neighbors.

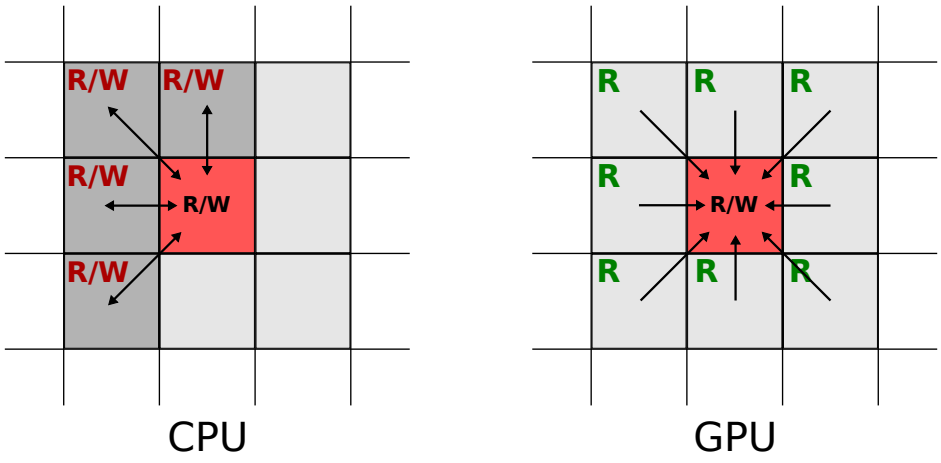


Figure 5.2: The CPU code computes the interaction of a pair of neighbors once and adds the partial on both; the GPU code computes all the interactions for each cell and only writes self total.

As mentioned in section 2.3, numerical results may vary a little with respect to the CPU implementation as partial fluxes are added together in a different order. We use the Kahan summation algorithm to mitigate this difference. Moreover, the CPU implementation used double precision while most GPU simulations are run on cards featuring hardware single precision. The numerical differences, however, do not alter the simulation significantly.

The performance results of the GPU implementation are quite impressive. The speedup factor depends on the hardware used, the topology of the flow and the DEM resolution. For example, the higher the DEM resolution, the more saturated is the GPU, thus better covering all transfers and other GPU-related overheads. In the general case, it is safe to say that on recent hardware (i.e. GTX480) the speedup over the reference serial implementation for single-core CPUs varies from 50× to 100×, taking only tens of seconds to simulate a one week eruption.

More details on the implementation, validation and performance gain of the GPU-based MAGFLOW can be found in [8].

Chapter 6

Multi-GPU MAGFLOW

We here describe how the parallel, GPU-based MAGFLOW was adapted to run on multiple GPUs simultaneously, thus exploiting a second level of parallelism.

6.1 Splitting the problem

There are in general at least two ways to exploit a second level of parallelism for an already parallel problem: splitting the set of data to be elaborated (space domain) or split the set of computations to be performed (computational domain).

The simplest and most intuitive is to split the automaton domain into separate areas and assign each area to a different device. Because the flux computation for every cell requires to access its immediate neighbors, the split areas need to have some degree of overlap.

We could instead divide the problem in the domain of computations, assigning each phase of the computation to a different device (in *pipeline*). This approach presents unfortunately three major drawbacks: it is not possible to scale to an arbitrary number of devices; it is not trivial to balance computational load among the devices; it requires the complete domain to be continuously transferred through all active devices during the whole simulation. For these reasons, we chose the

former (spatial) division.

The two-dimensional arrays of cells holding all the properties needed for a simulation are stored in row-major order, following the C convention. This must be taken into account for the optimization of memory transfers, since accessing a burst of consecutive cells is faster than accessing the same amount of data sparsely distributed. We therefore split the domain only horizontally, in stripes, so that we will need to transfer only whole rows. In special cases of very narrow, rectangular domains, we can work on transposed data depending on which choice will lead to shorter rows.

Steps 1 (`Erupt`) and 4 (`Update`) do not require any special consideration when splitting the simulation domain; each cell runs the `Erupt` and `Update` kernels in a completely independent fashion. No significant changes are thus required with respect to the single-GPU implementation.

Step 3 (`Minimum scan`) easily scales to multiple GPUs as each GPU provides its local minimum and the global one is chosen with a quick comparison. Step 2 (`CalcFlux`) is the only one having consequences on the split as each cell requires to access its immediate neighbors. Because it is not possible to access a cell from one device to another in an efficient and completely transparent way, each device must hold a copy of an entire row of cells (the neighbors of the edging subdomain cells) along each edge shared with another device subdomain. This copied row is read-only, and needs to be updated at every iteration. Because also neighbor devices need a row to read from, the total overlap is 2 cells thick. Fig 6.1 represents such a subdomain division.

Let D be the number of available GPU devices; a workstation usually has 1 to 3 devices, while a dedicated rack may host up to 8 devices per PCI domain. Let also R be the number of rows and C the number of columns of the simulation domain. We split our R $\times$ C grid in horizontal subdomains and we assign bordering subdomains to consecutive GPU indices. If we split the domain in equal parts, assuming

for simplicity that R is a multiple of D, the device of index d will be assigned to zero-based rows $d(\mathrm{R}/D)$ to $(d+1)(\mathrm{R}/D)-1$.

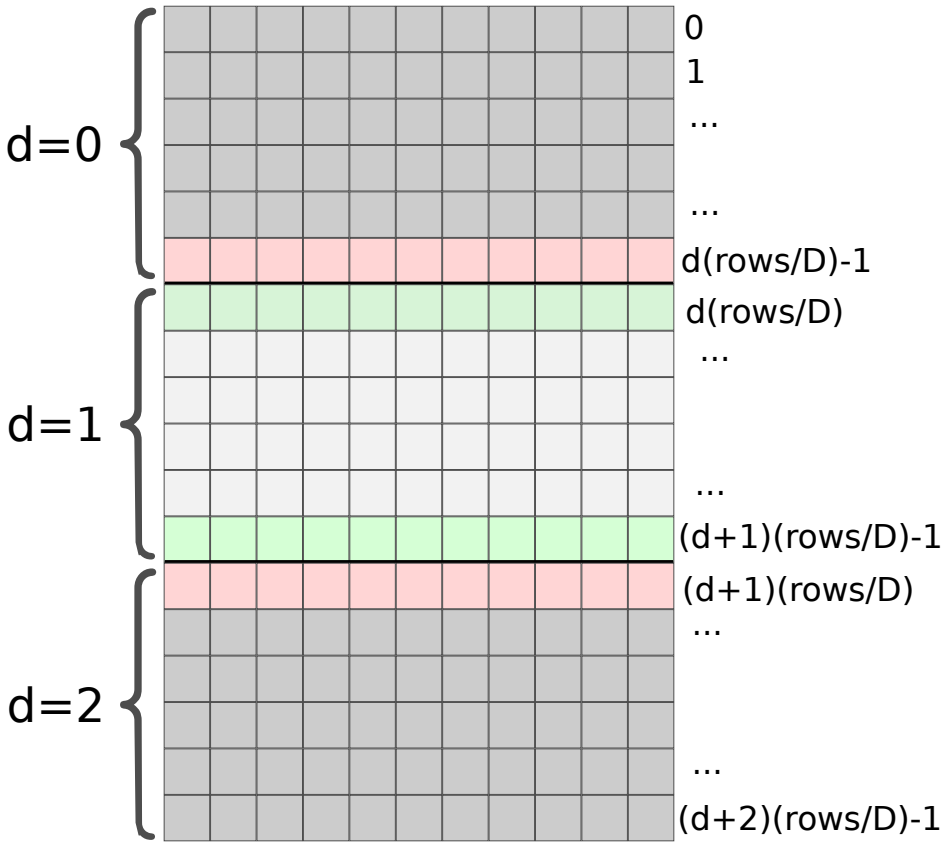


Figure 6.1: Representation of subdomains and overlapping rows. The internal rows of stripe assigned to device $d = 1$ are green, the external ones (i.e. one internal of device $d = 0$ and one internal for $d = 2$) are light red.

However, device d needs to read cells in row $d(\mathrm{R}/D)-1$ to compute cross-cell lava fluxes of row $d(\mathrm{R}/D)$, i.e. it needs to read the last row of device $(d-1)$; the same reasoning applies to row $(\mathrm{R}/D)(d+1)-1$, that requires reading access to row $(d+1)(\mathrm{R}/D)$. Devices $(d-1)$ and $(d+1)$ need in turn to read respectively the first and the last row assigned to device d; the needed overlapping between contiguous subdomains is therefore two rows high. We call rows $[d(\mathrm{R}/D)\ldots(d+1)(\mathrm{R}/D)-1]$ *internal* rows for device d, while rows $[d(\mathrm{R}/D)-1]$ and $[(d+1)(\mathrm{R}/D)]$ are *external* rows for the same device. Every device computes and writes the status of all its internal rows, while the external ones are just copies accessed for reading.

6.2 Hiding row transfers

Transferring one row of cells takes a variable time depending on the underlying hardware, the PCI bus utilization, the DEM resolution, the direction of the transfer, and many others. We observed an average time of 10% the total GPU time required by one iteration on a 2-GPU simulation. The more the number of devices, the more edges we need to transfer, the more PCI bus will congest and slow down. Moreover, PCI bus is full-duplex but multiple transfers in the same direction can not interlace and are thus serialized.

We need a technique to cover these latencies. As already mentioned, CUDA offers a simple though efficient feature: *asynchronous data transfers*, where "asynchronous" refers to the possibility to perform them concurrently with kernel executions. We can organize the simulator so that the transfer of borders is "hidden" by a kernel execution.

The key idea is to compute the borders first, and begin to download them as soon as they are ready, while still working on the rest of the subdomain. As soon as all borders have been downloaded, it is possible to upload them to neighbor devices, so that every device will have an updated copy of external cells at the beginning of the next iteration. To exchange the final, integrated amount of fluid lava we should run the transfers in parallel with the `euler` kernel. However, this takes too little time to complete and does not saturate the device enough to be split in portions.

It would be more efficient to run the transfers in parallel with the computation of fluxes, that takes most of the simulation time and would be surely long enough to hide the transfers. Lava amounts are not ready in this phase, as fluxes have not been integrated yet, but we can transfer fluxes instead; we will then need to run the integration also on external borders, as we have most recent fluxes but not quantities of lava.

The key idea is to compute the flux of the internal bordering rows first, and begin to download them as soon as they are ready; while they are being transferred, the GPU can work on the remaining subdomain. We need to split the `CalcFlux` kernel in two or three executions, the first two of which running on the internal borders only and the third one computing the remaining cells. Fig. 6.2 represents the way concurrent transfers and computations are issued and performed.

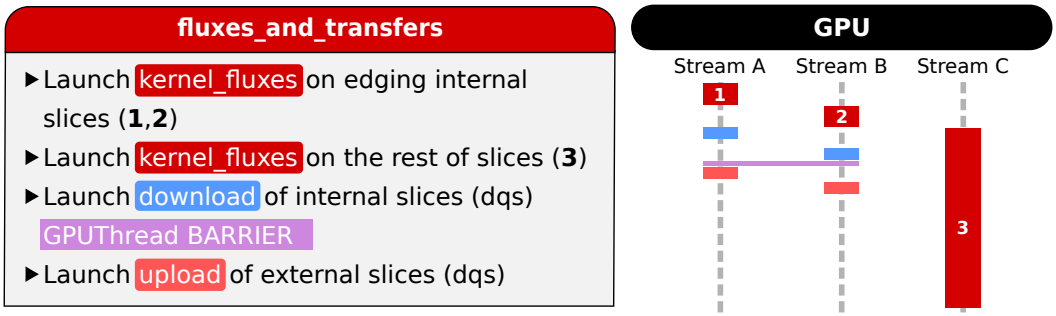


Figure 6.2: Final design of `fluxes_and_transfers` method. Note that the fluxes are exchanged instead of the integrated lava amounts, which are not reay yet. As a consequence, `euler` kernel must be run also on external cells.

Since transferring the borders (downloading and uploading) takes much less time than computing the fluxes on the rest of cells, data transfers are expected to be completely covered by the second `CalcFlux` call. The amount of data computed by the first call is actually more than one row as it is not convenient to run a kernel which does not saturate the hardware; moreover, it is more robust to run `CalcFlux` one time for each border than once for both. The most efficient and elegant solution is therefore to split the subdomain in three equal stripes and launch the kernel one time for each stripe.

6.3 Barriers

A barrier is needed in between the download and the upload of borders to keep data integrity and avoid r/w conflicts. The barrier has been implemented as a stream flushing call (`cudaStreamSynchronize`), only on the streams assigned

to borders, and a CPU thread barrier (`pthread_barrier_wait`), to ensure that all the devices reached the same point. In fig. 6.2 these are called *GPUThread barrier*. Note that these barriers do not stop nor wait for the `CalcFlux` kernel, that in the mean time computes the fluxes non non-bordering cells. The host copies of the borders are double buffered to prevent the unfortunate case of a CPU thread so fast that the internal rows are already being written by the next iteration before they have been upload by neighbor GPUs (e.g. if the scheduler of the operative system pauses one of the GPUThreads for too long).

Another barrier is required when selecting the minimum δt: as soon as they selected their local minimum, all GPUs must communicate this value and wait for the global one to be selected by the CPU. This time we call a general device flushing call (`cudaThreadSynchronize`), which waits for all kernels up to the `MinimumScan` to be complete, and a CPU thread barrier (`pthread_barrier_wait`), to ensure that all local `dt`s have been written. In fig. 6.3 these are called *BARRIER*.

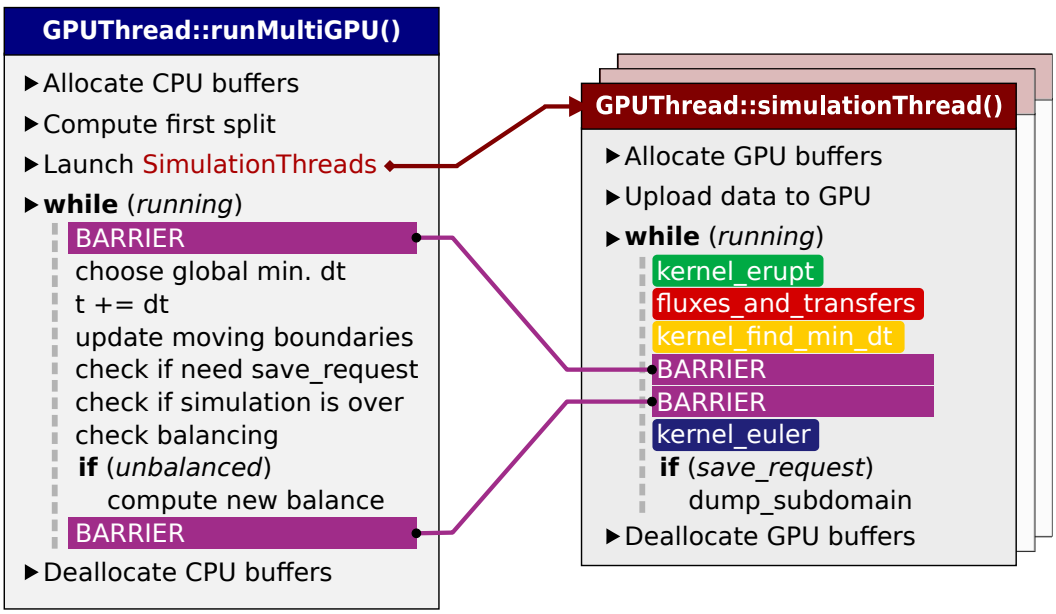


Figure 6.3: Simplified scheme of the simulator structure. NPTL barriers are purple; the red fluxes_and_transfers method encapsulates also asynchronous slice exchange.

6.4 From serial to parallel code

The CUDA parts of the single-GPU MAGFLOW implementation underwent almost no changes while porting to multi-GPU. The CPU code, as well as the general structure, was instead completely redesigned.

In the multi-GPU implementation we introduce the class `GPUThread` (not to be confused with GPU threads, that are GPU instances of a kernel), that encapsulates all pointers and CUDA calls needed to handle a GPU and work on the assigned subdomain. The main thread allocates one `GPUThread` per device and each `GPUThread` starts a dedicated `pthread` in constant communication with the associated device. Threads are mainly synchronized with the `pthread_barrier_wait()` primitive of the NTPL pthread implementation [22]. We also tried other sets of primitives, like a far more CPU-consuming busy-wait mechanism, but `pthread_barrier_wait()` performed quite well with an overhead measured in some seconds every hour of simulation.

The main thread tracks the simulation time and periodically requests a subdomain dump from the GPUs according to a user-defined save frequency. An optional auxiliary thread checks periodically the status of the system for debugging purposes.

It is possible to specify at command line several options such as the simulation percentage to run, an interrupted simulation to restore and the list of devices to be used. Devices may be heterogeneous and even belong to different generations (e.g. Tesla and Fermi). It is possible to specify a device more than once; this causes two or more `GPUThread`s to use the same physical device, thus emulating a multi-GPU behavior on a single-GPU machine. Obviously, this is useful only when developing/debugging and gives no performance gain. The data is not multi-GPU aware: it is possible to save the state of a single-GPU simulation and to load it in a multi-GPU environment, and vice-versa.

6.5 Preliminary performance analysis

We could think that splitting a simulation on D devices would complete $D\times$ faster than on a single GPU. Unfortunately, things are not that simple. The first aspect we need to consider is the different device saturation each kernel reaches. The `CalcFlux` kernel is the only one capable of saturating the hardware. The `Erupt` kernel does almost no mathematical operations but accesses the global memory at least twice; this means most of the time is spent waiting for the data transfer to complete, and working on a subdomain smaller than the global one does not yield a proportional performance gain. The same consideration holds for the `MinimumScan`, but for a different reason: a parallel reduction usually takes $\log(N)$ operations and even halving the size of the input we still need $\log(\frac{N}{2}) = \log(N) - 1$ operations. Also the `Update` kernel barely saturates the hardware. The plot in fig. 6.4 shows how much time the kernels take with respect to the number of input cells, from 100 to $6 \cdot 10^4$ (corresponding to the number instantiated GPU threads).

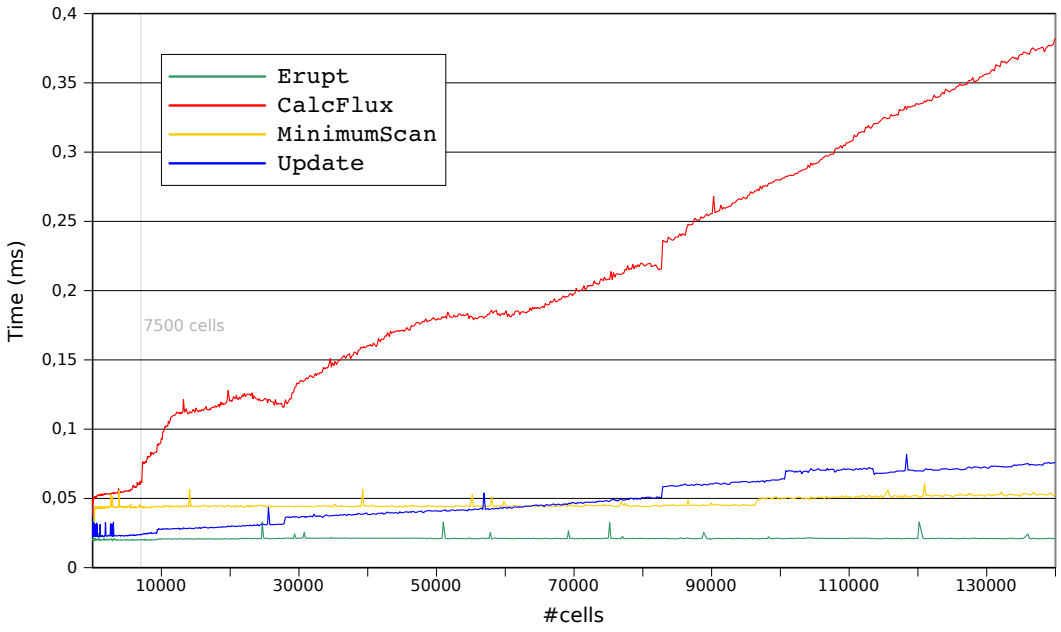


Figure 6.4: Execution times of each step with respect to the number of active cells in the domain.

We notice a sudden change in the time of `CalcFlux` kernel at about 7500 cells and a smaller one around 10^4. The first "step" is the saturation point of the GPU; before the hardware is saturated, we get no benefit in decreasing the size of the assigned subdomain. It is now clear that it is not convenient to split a simulation on multiple devices if the resolution of the underlying DEM is very small.

We also notice that although the growth of `CalcFlux` and `Update` kernels is almost linear, their fit corresponds to a line of the kind $ax + by + c = 0$ with $c > 0$. This means that even for kernels actually saturating the hardware, say N is the size of the input and T the time required to compute it, it is

$$T(N/D) > T(N)/D \tag{6.1}$$

Let us call T_E, T_C, T_M and T_U the time required by kernels `Erupt`, `CalcFlux`, `Minimum Scan` and `Update`, respectively. The time T required to complete an iteration of the automaton on an input big N is

$$T(N) = T_E(N) + T_C(N) + T_M(N) + T_U(N) \tag{6.2}$$

If we compute the average time taken by each kernel over the whole simulation we find that the the `Erupt` kernel takes about 2% of the total GPU time; `CalcFlux` 73.0%; `MinimumScan` 9.0% and `Update` 16%. From the observed trend (linear or constant) we also know that the approximate time taken by each kernel on a fraction $1/D$ of the input is

$$\begin{aligned} T_E(N/D) &\approx T_E(N) \\ T_C(N/D) &> T_C(N)/D \\ T_M(N/D) &\approx T_M(N) \\ T_U(N/D) &> T_U(N)/D \end{aligned}$$

We can thus estimate the total time T_D required by an iteration using D GPUs as

$$\begin{aligned} T_D(N/D) &> 0.02 + 0.73/D + 0.09 + 0.16/D \quad \Rightarrow \\ T_D(N/D) &> (0.11 + 0.89/D)T \end{aligned} \tag{6.3}$$

We have a substantial fraction of the problem that just does not scale, as if it was non-parallelizable. Even assuming zero-overhead, the maximum speedup we can achieve is $S_{max}(D) = 1/T_D(N/D)$ with $T = 1$, i.e.:

$$S_{max}(D) = \frac{1}{0.11 + \dfrac{0.89}{D}} \tag{6.4}$$

More formally, we estimated $\beta = 0.11$ and $\alpha = 0.89$. This limits the maximum achievable speedup, in Amdahl's speak, to $\sim 3.87\times$ with 6 devices, with an asymptotic upper bound of $1/0.11 \approx 9.1\times$ (with a number of cores that goes to infinite). We should remark, however, that among the limitations of the Amdahl's law there is the simplicity of assumptions such as that the parallelizable part scales linearly. In this case, for example, this is not true, as each kernel scales with a different rate.

We never experienced the need for a simulation with a higher resolution than 2m. As the memory required by a simulation up to this resolution always fits the memory of one devices, the multi-GPU MAGFLOW allocates in each GPU enough memory to contain the whole simulation. Though not memory-efficient, this is very robust and straight to implement. For this reason, it is not proper for MAGFLOW to estimate the *scaled speedup* (in Gustafson's speak) of the multi-GPU implementation, as the size of the problem can not increase indefinitely.

6.6 Load balancing

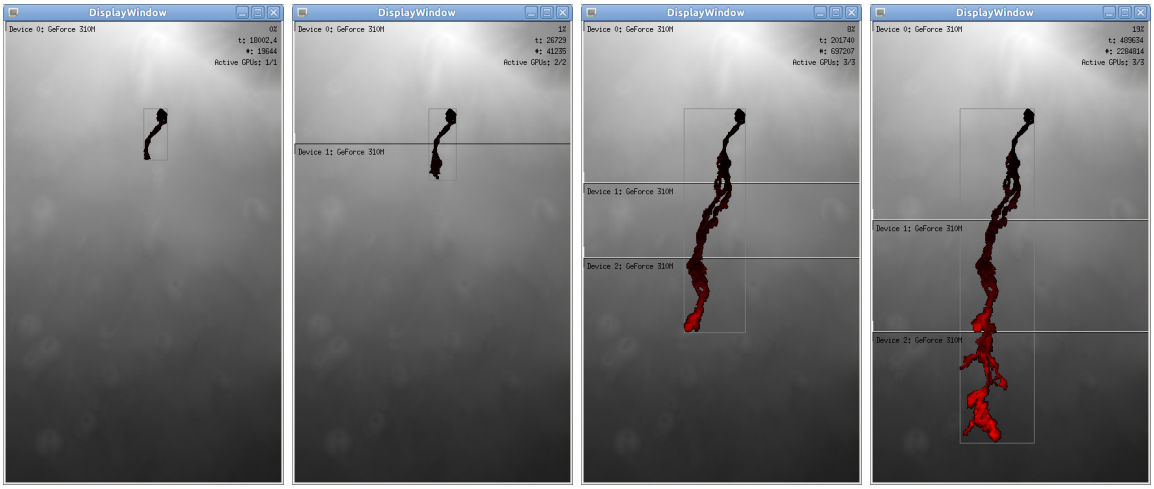
It is trivial to prove that an even division performs better than one in which the time taken by one or more devices largely differs from the average. As the topology of the flow may vary greatly, a static subdomain assignment made at the beginning of a simulation would end up in a completely unbalanced division, undermining the overall performance. A dynamic load balancing system is necessary.

As the simulation begins, the main thread makes a rough estimation of the computational power of each active GPU by retrieving the number of their multiprocessors, then assigns parts of the domain proportionally with the computational power of the devices. As the bounding box of the lava flow grows, the proportions are checked again and if the distribution of rows does not reflect anymore the computational power ratio, the splits dynamically move by one row at a time. While dynamic, this only an *a priori* load balancing, i.e. it tries to estimate what is best to do but does not check if the consequences are actually positive.

Unfortunately, there are some unpredictable runtime factors that influence the performance of the devices (CPU thread scheduling, GPU block scheduling, PCI bus conflicts, particular lava flow topologies, etc.). The only way to take into account all these factors is to change the split proportions according to the execution times of previous iterations (*a posteriori* load balancing). However, because the current approach is quite robust and performs reasonably well, we did not implement yet such a complex balancer for the multi-GPU MAGFLOW.

Lava flows often start with a very small or even single-cell extent; the lava erupted by vent cells, then, makes them quickly expand until they eventually stabilize to a certain extent. Splitting the automaton immediately after a simulation is started would give us no speedup, or even slow down the performance. For this reason, the simulator always starts a simulation on a single GPU and then enables one GPU at a time, as long as all of the already active devices are saturated. As

an *a priori* saturation threshold, we roughly estimated that a device should not be considered saturated unless the number of issued kernel blocks is about one order of magnitude bigger than the number of multiprocessors. Every time a new device is activated, the domain is split again among the active GPUs. Fig. 6.5 shows how the number of devices and the size of the subdomains dynamically changes as the bounding box of the simulated flow grows.



$1.8 \cdot 10^4$s (1 GPU) $2.6 \cdot 10^4$s (2 GPUs) $2 \cdot 10^5$s (3 GPUs) $4.9 \cdot 10^5$s (3 GPUs)

Figure 6.5: Snapshots of different phases of a simulation with MAGFLOW. The available devices are enabled one at a time while the previous ones are saturated. As the bounding box of the lava flow grows, the subdomains are dynamically reassigned.

6.7 Interface

Though not fundamental for the purposes of the simulations, we took the opportunity of redesigning the simulator to add a real-time display of the ongoing simulations. Its main purpose is to visually monitor the topology and size of a simulation while it runs, without the need to export the temporary dumps and import them into an external visualization tool. It was also extremely useful for debugging purposes and as an insight to the simulator internals.

Fig. 6.6 is a screenshot of the mentioned interface. It is just a single window where it is possible to draw the running lava flow as well as the DEM underlying

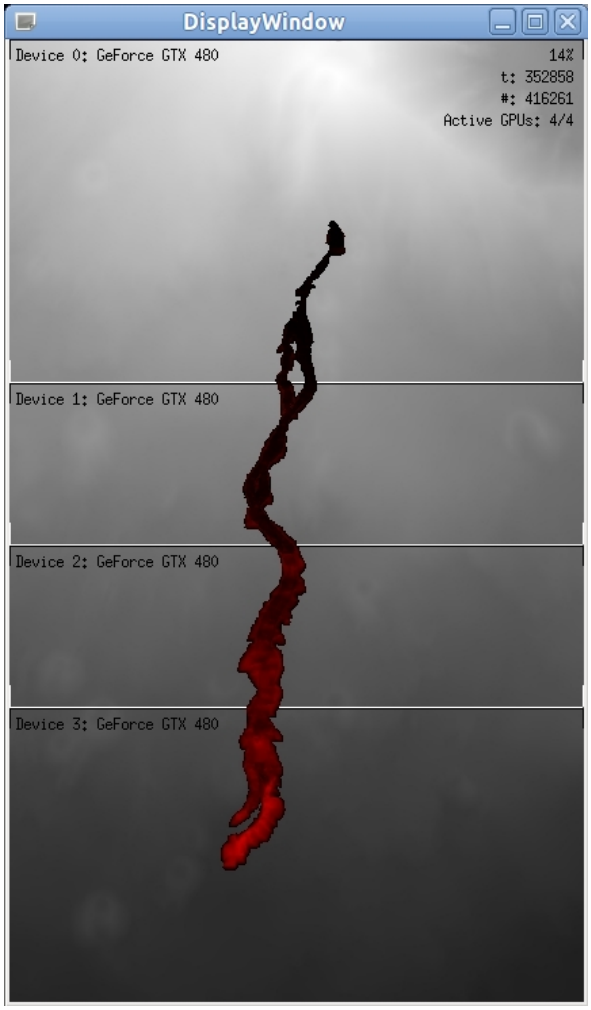


Figure 6.6: Screenshot of the simulator interface while running a 4-GPUs simulation.

the simulation and the solidified lava. It outlines the bounding box of the active eruption and, in case of a multi-GPU simulation, the subdomain division. The palette reflects the amount of lava contained in a cell and some text information about the state of the simulation are drawn in the upper right corner.

6.8 Results

The multi-GPU MAGFLOW has been tested by simulating the lava eruption on Etna volcano in year 2001 on the rack described in section 4.8. We ran the simulations on DEMs with 5m and 2m resolutions, loading from a simulation that was already 19% complete, so that the active box already reached its maximum extent.

DamBreak3D	1 GPU	2 GPUs	3 GPUs	4 GPUs	5 GPUs	6 GPUs
Real time (s)	14,377	7,737	5,813	4,993	4,562	4,500
Ideal time (s)	14,377	7,189	4,792	3,594	2,875	2,396
Real cost (s)	0	549	1,021	1,399	1,687	2,104
Karp-Flatt (β_{KF})	0	0.076	0.106	0.130	0.147	0.176

Table 6.1: Multi-GPU MAGFLOW execution times, overhead in seconds and Karp-Flatt metric, while simulating on a 2m DEM

As a single simulation may require days to complete, for performance comparison most tests performed up to 20% or 22% of simulation time. All tests have been numerically validated and the comparison refers to the same simulation with the same options (e.g. state dumping frequency) as computed by the single-GPU code. Table 6.1 shows the execution times in seconds.

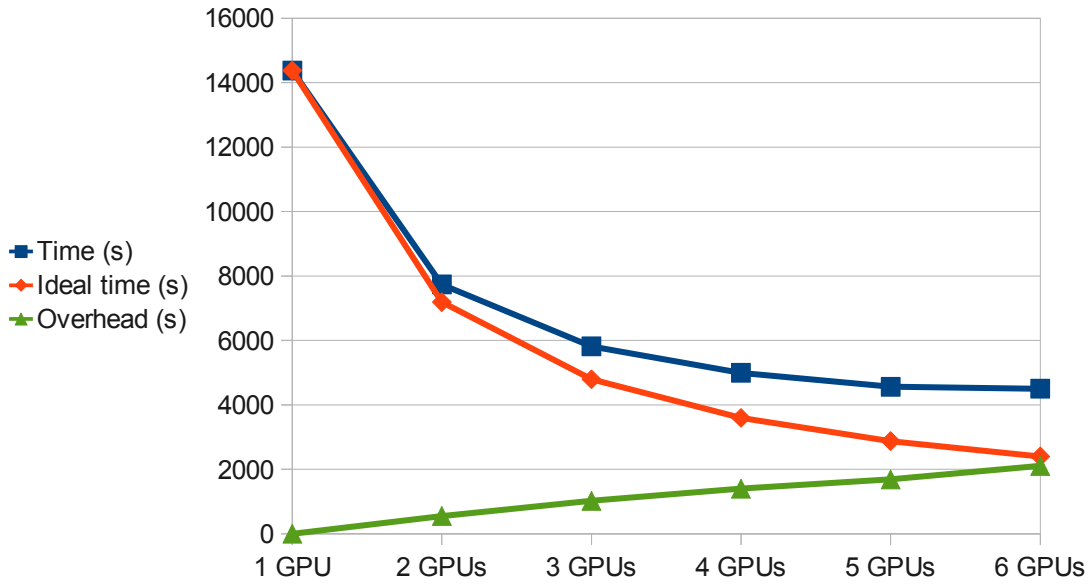


Figure 6.7: Simulation time: real, ideal and difference between the two, 1 to 6 devices.

Table 6.1 lists the actual and ideal execution times, overheads in seconds and serial fraction estimated with the Karp-Flatt metric. As β_{KF} keeps growing, we are introducing some liner overhead, the source of which we identified with the lack of an *a posteriori* load balancing.

Fig. 6.7 plots the same data as table 6.1. Being T the total simulation time and D the number of GPUs used, T differs from the ideal time T/D only for a constant

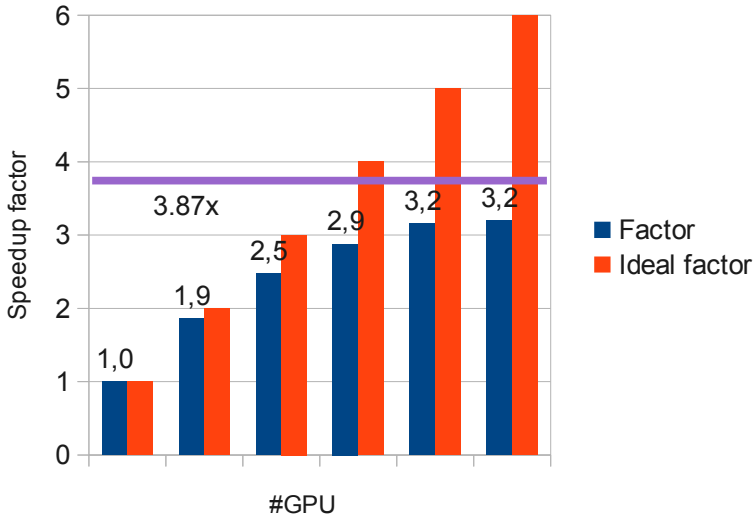


Figure 6.8: Speedup: observed and ideal, 1 to 6 devices. The purple line marks the theoretical maxim estimated with equation 6.4.

overhead Q approximately equal to $4 \cdot 10^{-2} \cdot T \cdot D$.

Because the overhead is approximately linear, we can apply the estimation technique mentioned in section 4.6. The overhead introduced for each new device is $\approx 4 \cdot 10^{-2} \cdot T$; the maximum number of devices satisfying the inequality 4.6 is $D_{max} = 5$.

Indeed, table 6.1 confirms our estimate: if we split the simulation across more than 5 GPUs, the cost becomes comparable with the simulation time and it is not convenient to exploit the sixth, available GPU.

The overhead size with respect to the total simulation time depends on the simulation settings, and sometimes even on the input data (e.g. topology of the flow). For example, the same eruption on a 5m DEM can benefit at most from being distribute across 3 GPUs, as at that resolution even a single-device simulation barely saturates the GPU. More in general, the maximum number of GPUs across which it is convenient to distribute the computation depends on the total number of active cells (i.e. the extent of the simulated flow) and the DEM resolution.

Part III

The GPUSPH simulator

Chapter 7

The GPUSPH simulator

GPUSPH is the CUDA implementation of fluid-dynamics models based on the Smoothed Particle Hydrodynamics (SPH) numerical method. It allows for simulations with a superior adherence to fluid physics and enables simulating phenomena like fluid-solid interaction and crust formation that would not be possible to simulate with a model based on the cellular automaton approach. This flexibility comes at the price of increased computational and memory requirements with respect to the MAGFLOW simulator. Luckily, SPH too exposes an intrinsic parallelism that makes it convenient for implementation on a many-core GPU.

Similarly to the arrangement of part II, here follows an overview of the related work, the model and the single- and multi-GPU implementations.

7.1 SPH

SPH is a mesh-free, Lagrangian numerical method for fluid modeling, originally developed by Gingold, Monaghan and Lucy [23, 56] for astrophysical simulations.

The mathematics behind the SPH method rely on the properties of convolutions. For any scalar field u in a domain $\Omega \subseteq \mathcal{R}^d$ we can write, by definition of the

Dirac's delta distribution δ:

$$u(\mathbf{x}) = \int_\Omega u(\mathbf{y})\delta(\mathbf{x}-\mathbf{y})d\mathbf{y}.$$

Let $W(\cdot, h)$ be a family of *smoothing kernel*s, parametrized by their *smoothing length* h, satisfying

$$\int_{\mathcal{R}^d} W(\mathbf{x}, h)d\mathbf{x} = 1$$
$$\lim_{h\to 0} W(\cdot, h) = \delta$$

We can approximate δ with a W as follows:

$$u(\mathbf{x}) \simeq \int_\Omega u(\mathbf{y})W(\mathbf{x}-\mathbf{y}, h)d\mathbf{y}. \tag{7.1}$$

Let Ω represent the body of the fluid with density $\rho\colon \Omega \to \mathcal{R}$. We can discretize Ω with a set of points $\mathbf{x}_i$, each with an associated mass m_i. The volume associated with the point $\mathbf{x}_i$ is then $V_i = m_i/\rho(\mathbf{x}_i)$ and the integral 7.1 can be approximated with

$$\begin{aligned}\int_\Omega u(\mathbf{y})W(\mathbf{x}-\mathbf{y}, h)d\mathbf{y} \;&\simeq \\ \sum_i u(\mathbf{x}_i)W(\mathbf{x}-\mathbf{x}_i)V_i \;&= \\ \sum_i u(\mathbf{x}_i)\frac{m_i}{\rho_i}W(\mathbf{x}-\mathbf{x}_i, h)\end{aligned}$$

we arrive thus at the standard SPH discretization for field values:

$$< u(\mathbf{x}) >= \sum_i u(\mathbf{x}_i)\frac{m_i}{\rho_i}W(\mathbf{x}-\mathbf{x}_i, h) \tag{7.2}$$

which is the standard SPH interpolation formula for scalar fields.

The smoothing kernels are usually chosen with compact support so that the

summation is extended only to particles in a spatial *neighborhood* of $\mathbf{x}$; they are chosen positive, so that the interpolated density

$$< \rho(\mathbf{x}) >= \sum_i m_i W(\mathbf{x} - \mathbf{x}_i, h)$$

is always strictly positive; and they are chosen center-symmetric to ensure first-order accuracy of interpolation.

7.1.1 SPH for derivatives

It is possible to transfer the ∇ operator to the kernel by exploiting the divergence theorem and the properties of convolutions. We then obtain the first form of the SPH interpolation for the gradient:

$$< \nabla u(x) >= \sum_i u_i \frac{m_i}{\rho_i} \nabla W(r_i, h), \tag{7.3}$$

that is first order accurate for particles for which the kernel support is entirely contained in Ω.

The SPH gradient is generally written using slightly different formulations than (7.3). In particular:

$$< \nabla u(x_j) >= \sum_i (u_i \pm u_j) V_i \nabla W_{ij}, \tag{7.4}$$

$$< \nabla u(x_j) >= \frac{1}{\rho_j} \sum_i (u_i - u_j) m_i \nabla W_{ij} \tag{7.5}$$

$$< \nabla u(x_j) >= \rho_j \sum_i \left(\frac{u_i}{\rho_i^2} + \frac{u_j}{\rho_j^2} \right) m_i \nabla W_{ij}. \tag{7.6}$$

The preference of a formulation over another has to be evaluated case by case as they present slightly different properties. For example, in (7.4) (with the plus sign) and (7.6) it is guaranteed that the influence of particle j over particle $\bar{\imath}$ is opposite the influence of particle $\bar{\imath}$ over particle j. On the other hand, (7.4) (with a minus

sign) and (7.5) guarantee that the gradient of a constant function evaluates to zero. The formulation (7.5) is computationally more efficient but (7.4) is more stable in the case of big differences in the density values between particles (as in the case of multiple fluids). Further details about these formulations and their applications can be found in [57].

For second order derivatives, the standard SPH interpolation introduces very high errors near the boundaries of Ω. In the case of the Laplacian operator, a possible approach to solve this issue derives from the Taylor expansion of the SPH gradient. It was introduced in [13] to model thermal conduction and adapted for the dynamic case in [60]. It has the form

$$< \nabla^2 \mathbf{v}_j >= \sum_i m_i \frac{(\rho_i + \rho_j)\mathbf{x}_{ij} \cdot \nabla W_{ij}}{\rho_i \rho_j (\mathbf{x}_{ij}^2 + \varepsilon h^2)} \mathbf{v}_{ij} \tag{7.7}$$

where $\varepsilon \simeq 0.01$, h is the smoothing length and the εh^2 term prevents singularities in the case of overlapping particles.

7.1.2 SPH for Navier Stokes

The simulation of a quasi-compressible and isotropic fluid is driven by the Navier-Stokes equations paired with an equation of state:

$$\frac{D\mathbf{v}}{Dt} = -\frac{\nabla P}{\rho} + \nu \nabla^2 \mathbf{v} + \mathbf{g} \tag{7.8}$$

$$\frac{D\rho}{Dt} = -\rho \nabla \cdot \mathbf{v}, \tag{7.9}$$

$$P = \frac{\rho_0 c_s^2}{\gamma} \left(\left(\frac{\rho}{\rho_0} \right)^{\gamma} - 1 \right),$$

Here, $\mathbf{g} = (0, 0, -g)$ is gravity, ρ_0 is the density at rest, γ is a constant generally chosen in the range $[1, 7]$ and the speed of sound c_s is chosen to be an order of magnitude higher than the maximum speed expected during the flow.

We can discretize these equations by applying (7.7) and (7.4) (with plus sign)

to (7.8) and (7.5) to (7.9), resulting in:

$$\frac{D\rho_j}{Dt} = \sum_i \frac{\rho_j}{\rho_i}\mathbf{v}_{ij}\cdot\mathbf{x}_{ij}F(r_{ij},h)m_i,$$
$$\frac{D\mathbf{v}}{Dt} = -\sum_i\left(\frac{P_i+P_j}{\rho_i\rho_j}+\Pi_{ij}\right)\mathbf{x}_{ij}F(r_{ij},h)m_i + \nu\sum_i\frac{m_i(\rho_i+\rho_j)\mathbf{v}_{ij}}{\rho_i\rho_j}F_{ij}$$

Here, Π_{ij} is an artificial viscosity that takes the form

$$\Pi_{ij} = \begin{cases} -\alpha h\dfrac{\bar{c}_s}{\bar{\rho}}\dfrac{\mathbf{x}_{ij}\cdot\mathbf{v}_{ij}}{r_{ij}+\varepsilon h^2} & \mathbf{x}_{ij}\cdot\mathbf{v}_{ij} < 0, \\ 0 & \text{otherwise} \end{cases}$$

where $\bar{c}_s$ is the average speed of sound computed at the particles i and j, and $\bar{\rho}$ their average density. More details on the artificial viscosity in SPH can be found in [55].

7.1.3 Integration

The integration relies on an explicit predictor/corrector scheme with a dynamic timestep limited by standard CFL conditions. Being f_M the maximum particle force per unit mass, the timestep is limited by:

$$\Delta t \leq k\min\left\{\frac{h}{c_s}, \sqrt{\frac{h}{f_M}}, \frac{h^2}{\nu}\right\}$$

with k a safety factor in range $]0,1[$.

From the particle positions X_n and velocities V_n at time t_n we compute the particle accelerations $F^*_{n+1}(X_n,V_n)$ and the maximum timestep Δt^*. The predicted

new positions and velocities are then:

$$V^*_{n+1} = V_n + \Delta t_n F^*_{n+1},$$
$$X^*_{n+1} = X_n + \Delta t_n V^*_{n+1}.$$

with the timestep Δt_n selected from the previous iteration. The correction step therefore computes $F^{**}_{n+1}(X^*_{n+1}, V^*_{n+1})$ and a new maximum timestep Δt^{**}. The corrected velocity V_c to be used is computed as

$$V_c = V_n + \Delta t_n F^{**}_{n+1}/2.$$

The final velocities and positions are computed as

$$V_{n+1} = V_n + \Delta t_n F^{**}_{n+1}$$
$$X_{n+1} = X_n + \Delta t_n V_c$$

and the timestep for next iteration is chosen as $\Delta t_{n+1} = \min(\Delta t^*, \Delta t^{**})$.

This kind of two-step integration scheme adds some complexity when splitting an SPH simulator on multiple GPUs, since additional memory transfers might be necessary to keep the overlapping subdomain areas updated in-between the steps.

7.2 Related work

The SPH method has been used in many fields of research including astrophysics, volcanology and oceanography. It has recently seen increasing interest with application to several aspects of fluid dynamics, from free surface [58] to thermal problems [13], from gaseous media [75] to hair [31] and deformable objects [21, 74, 6]. It has been applied to simulate phase transitions [48, 74], viscous fluids [60] and interaction of miscible and immiscible fluids [61, 73]. Other partially explored fields

include fluid-solid interaction and solid mechanics, where it is often referred as SPAM (Smooth-Particle Applied Mechanics) [41]. Recent reviews on SPH can be found in [59] and [26].

The great flexibility of the SPH method comes at the price of a significant computational cost. Fortunately, SPH is an excellent candidate for implementation on high performance parallel computing platforms, and in particular on GPUs. One of the first SPH solvers partially exploiting a GPU was developed by Amada [1]: in this case, the force computation was offloaded to the GPU while the CPU took care of other tasks such as neighbor search. A multicore-CPU version has also been recently developed [42].

The first implementation of the SPH method completely on the GPU is due to Kolb and Cuntz [49] and Harada et al. [32]. A strong limitation in these implementation, as with Amada's work, was the use of OpenGL and Cg (C for graphics) to program the GPU. There was no official platform support for GPGPU programming and this lead to some technical limitations. As an example, it was not possible to write to three-dimensional textures, and this imposed to use a flattened two-dimensional structure to the textures holding position and velocity of each particle.

The introduction of the CUDA architecture for NVIDIA cards in 2007 allowed to fully exploit the computational power of modern GPUs without the limitations imposed by having to go through the graphical engine.

The first CUDA implementation of the SPH model [35] was developed at the Sezione di Catania of the Istituto Nazionale di Geofisica e Vulcanologia (INGV-CT) in cooperation with the Department of Civil Engineering of the Johns Hopkins University. It was inspired by the Fortran SPHysics code [30] and has been recently published as the open source project GPU-SPH [36].

The first GPU-based implementation achieved two order of magnitude in speedup over the reference single-core CPU implementation at that time. A comparison

with an implementation optimized for modern multi-core CPUs would probably lead to a smaller speedup; however, no such performance comparison has been conducted as it would require to port the entire source code to OpenCL or to a CPU-specific platform.

GPU-SPH applications range from coastal engineering [38, 18, 19] to lava flow simulation, for which a specialized version with support for non-Newtonian viscous fluids and temperature-dependent parameters has been developed [7, 39, 40]

7.3 Single-GPU GPUSPH

Here we give an overview of the simulator design as a necessary basis to understand the challenges posed by the multi-GPU implementation and how they have been overcome. For a detailed description of the porting process and the performance results please refer to [35].

7.3.1 Kernels

The organization of the SPH method in CUDA kernels directly reflects the computing phases of the SPH model:

1. **`BuildNeibs`** - For each particle, build the list of neighbors;
2. **`Forces`** - For each particle, compute the interaction with neighbors;
3. **`MinimumScan`** - Select the minimum `dt` among the maxima provided by each particle;
4. **`Euler`** - For each particle, update the scalar properties integrating over selected `dt`.

The kernel for force computation(`Forces`) and the kernel for integration (`euler`) are actually run twice because of the two-step integrator scheme. All kernels

have been written from scratch, except for the particle sorting during the neighbor search phase, that uses CUDPP Radixsort, and for the minimum scan that is also done using CUDPP.

7.3.2 Fast neighbor search

We could totally skip step one and look for neighbors only when we need to access them (e.g. force computation). However, because we need to access them more than once within each iteration, it is faster to build the list once and iterate on it when needed.

The simplest way to find the neighbors of a particle is to compute the euclidean distance to every other particle of the simulation and compare this to the *influence radius*. This naïf approach, however, has a $O(n^2)$ complexity that makes it too expensive even for small simulations. To speedup this process, the simulated volume is partitioned in virtual cells with side equal to the influence radius and all the particles are sorted and indexed with respect to such grid. When looking for neighbors of a given particle it is enough to iterate only on neighbor *cells* instead of the whole simulation domain. Fig. 7.1 is a schematic representation of the virtual cells in 2D. It is worth to highlight that this grid serves only to speed up the neighbor search, and has nothing to do with the SPH model.

Building the list of neighbors for every particle of the simulation still requires about 50% the total simulation time, even using the above mentioned grid for fast neighbor search. Another optimization is to update the neighbor list every $k-th$ iteration; this causes a negligible loss of precision while decreasing the time required for the neighbor list construction from 50% to about 20% of the total simulation time.

An alternate model has been also tests, based on a completely different approach [70]. For each cell of the virtual grid, some slots are reserved to store potential particles. In other words, the list of particles is not compressed; empty

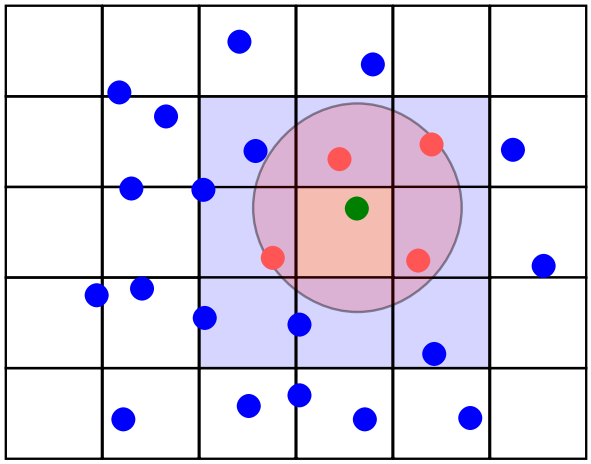


Figure 7.1: If cells have side equal to the influence radius, neighbors of the green particle must reside inside in the immediate neighbor cells (light blue).

cells reserve anyway free slots for all potential particles. A particular indexing method allowed to keep track of the particles outgoing from the cell or ingoing in it without having to iterate on them again. No neighbor list was constructed, as neighbors were listed only when needed.

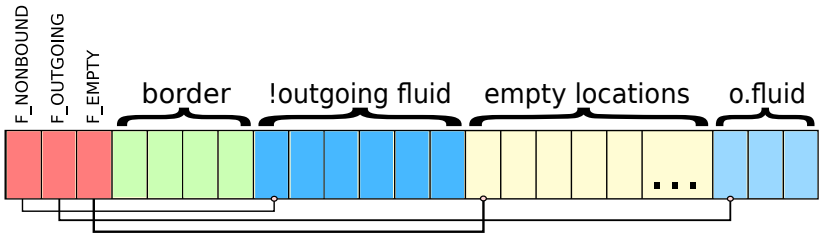


Figure 7.2: Organization and indexing of particles in a cell in the alternate model.

The approach resulted to have a memory consumption and a computational cost similar to the traditional and nevertheless simpler method, hence researches in this directions have been stopped by now.

7.3.3 Memory requirements

For each particle in the simulation we need to allocate in the GPU memory three vectors, three scalar values, a `particleInfo` and a set of slots to store the neighbor list. More specifically, we need:

- A `float4` array (`pos`) holding three-dimensional position and scalar mass (double buffered because of the two-step integration scheme)
- A `float4` array (`vel`) holding three-dimensional velocity and scalar density (ditto)
- A `float4` array (`forces`) holding three-dimensional force and scalar pressure derivative
- A `short4` array (`particleInfo`) holding the particle type, the object ID and a global particle ID (useful for tracking a particle through a simulation when post-processing the output)
- An array of `MAXNEIBSNUM` `uint` holding the list of neighbors, where `MAXNEIBSNUM` is the maximum number of neighbors per particle, selected at compile time
- Two `uint` arrays holding the particle indices and hashes for sorting
- Other auxiliary arrays such as cell indices for fast neighbor search and buffers for `MinimumScan`, plus some arrays that depend on specific simulation settings (correction factors, vorticity vector, etc.). These other arrays may not be necessarily proportional to the number of particles.

The overall memory requirement is about 1Kb per particle, more than half of which being occupied by the neighbor list. The total number of particles fitting the global memory of a GPU ranges from $\sim 5 \cdot 10^5$ for a card with 512Mb of RAM to $\sim 1.8 \cdot 10^6$ for a card with 1.5Gb of RAM (e.g. a GTX480).

If P is the number of particles to be simulated in D devices, we need to allocate more than P/D particles in each card because of the overlapping slices. If P_o is the average overlap size, we need to allocate at least $P/D + 2 \cdot P_o$ particles. However, because the split is at the slice level, even if load balancing is enabled it is still

possible that the fluid topology causes a device to hold many more particles than at the beginning of a simulation. Let us define a margin factor M_f; we can allocate $M_f \cdot P/D$ particles, with M_f empirically estimated. In our tests it was enough to set $M_f = 1.4$; only the corridor version of `BoreInABox` problem required to slightly increase M_f. If the number of particles fits in one device, it is still possible to start a multi-GPU simulation with the option `--alloc-max`, that causes allocating in each device enough memory for the whole simulation.

On the host side, we only need a subset of these arrays as temporary storage to periodically dump the state of the simulation. Moreover, the underlying operative system offers a virtual memory mechanism swapping with the hard disk. The only memory limits are therefore on the GPU side.

7.3.4 Speedup

By fine-tuning the thread block size and making most memory access coalesced, while iterating on neighbors still requires sparse accesses that probably can not be further optimized, it was possible to reach a speedup of two orders of magnitude (i.e. $100\times$) with respect to a single-core CPU implementation. It is worth keeping in mind, however, that when this comparison was conducted the CPU implementation was used a single-core CPU in a serial way. A comparison with respect to a modern, multi-core CPU, running a fully multi-threaded version of the simulator, would definitely bring a quite smaller speedup factor. A detailed comparison would require to port the whole simulator to OpenCL or to a CPU-specific platform; we would not expect the CPU to be faster than a hundred-cores GPU anyway.

Chapter 8

Multi-GPU GPUSPH

Exploiting a second level of parallelism required some structural changes to the single-GPU GPUSPH code. It became a multi-threaded application and many modules needed to be re-engineered and encapsulated in a different class structure. Some minor features were temporarily disabled for testing purposes, like periodic boundaries, and will be enabled again soon.

Here follows an overview of the challenges we had to overcome on the model side and on the technical side.

8.1 Splitting the problem

The key idea for exploiting multiple GPUs for the same simulation is that the total computational burden must be fairly split among the devices. This is far from trivial if the computational elements of the problems are not completely independent of each other. The way the problem is split and the path the data has to follow depends on the nature of the problem. There could even be no unique optimal solution for a certain problem, as different splits may perform quite differently according to the characteristics of a specific problem *instance*.

SPH is a purely parallel method except for the fact that every particle needs to

interact with other particles closer than the influence radius (neighbors). This locality constraint, together with the need to search for a globally minimum timestep, required a special consideration when designing the parallel version of the simulator. Planning to exploit multiple GPUs simultaneously brings the challenge to a higher level, as the level of parallelism increases and the inter-particle communication becomes less simple. With this in mind, we analyze now different possibilities for splitting GPUSPH.

We first wonder if it is convenient to use the naïve approach described in section 4.2. The `Forces` kernel, that is the longest step, takes about 84μs for each 1K particles. If we consider about 128 bytes per particle, downloading 1K particles takes about 24μs, while uploading them takes 40μs. We are still under 84μs but we did not take into account yet the neighbor list. Its size is at least $128 \cdot 4$ bytes per particle and it makes the transfer time overcome the kernel time. As in most cases, this approach is not convenient here; we need a more sophisticated model, specialized for the SPH problem.

There are at least three different ways to split a SPH simulation across multiple GPUs. As each iteration of the simulation is made of at least 4 different steps, we could split the problem in the domain of the computations: we could assign each phase of the computation to a different device, thus arranging a *pipeline*. This method, however, still needs the entire set of particles to be transferred at every iteration across all the devices, and does not scale easily as the number of devices increases.

The second possibility comes naturally from the organization of particles in memory. SPH is a lagrangian, meshless model, where particles are arranged in a *list*. We can see the list as an enumeration of all the particles with no connection at all with their spatial position: list locality does not correspond to spatial proximity. We could just split the list in subsets and assign each subset to a different device. Unfortunately, this is not feasible because we have no guarantee that the neighbors

of a particle reside on its same device, and accessing single neighbors in different devices would bring way too much overhead.

The last possibility is to split the problem in the spatial domain. Though SPH is a meshless method, we have seen in section 7.3.2 that particles are sorted and indexed by means of a grid of virtual cells of the same size of the influence radius. The SPH model is absolutely unaware of this, but keeping the particles sorted thought the simulation greatly speeds up the neighbor search. We can exploit this ordering to split the fluid on a spatial basis and handle a minimum overlapping of subdomains needed by the locality requirement. This is actually an extension of the previously described list-split that takes into account a pre-existent order constraint. We choose the spatial split because of its simplicity, scalability and robustness.

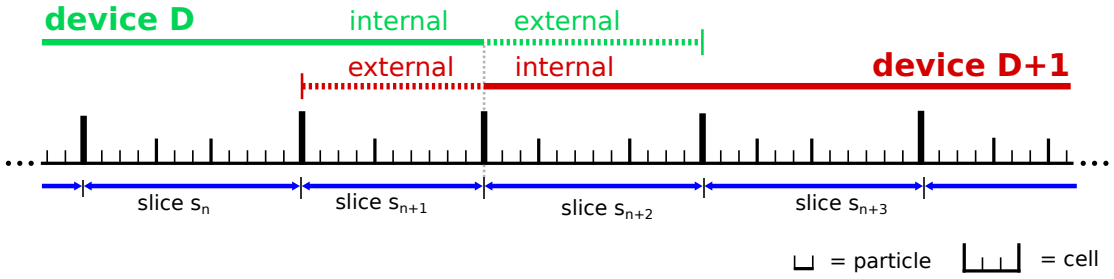


Figure 8.1: Representation of the the list split technique. Once particles are sorted by cells and 3D cells are linearized, it is possible to split the 3D domain by splitting the list of particles in specific addresses when 3D slices begin.

8.2 Split planes

While a spatial split in theory could operate on any three-dimensional plane, we focus on the three cartesian ones (the planes orthogonal with the cartesian axes), as the split is based on the (virtual) cubic cells used for fast neighbor search.

Although splitting along different, orthogonal planes simultaneously could be technically possible, transferring the edge of a subdomain would be very expen-

sive. Fig. 8.2 illustrates the problem in a simpler two-dimensional case. Assuming that the domain is linearized in a row-first fashion, it is possible to transfer every green edge by requesting a single memory operation, while the red edges require many small transactions. As a general rule, it is recommended that all the split lines (or planes in 3D) are aligned to the way data are linearized in memory. It is thus not allowed to operate progressive splits along different planes within the same simulation; we can imagine the 3D domain being split in slices all parallel with each other.

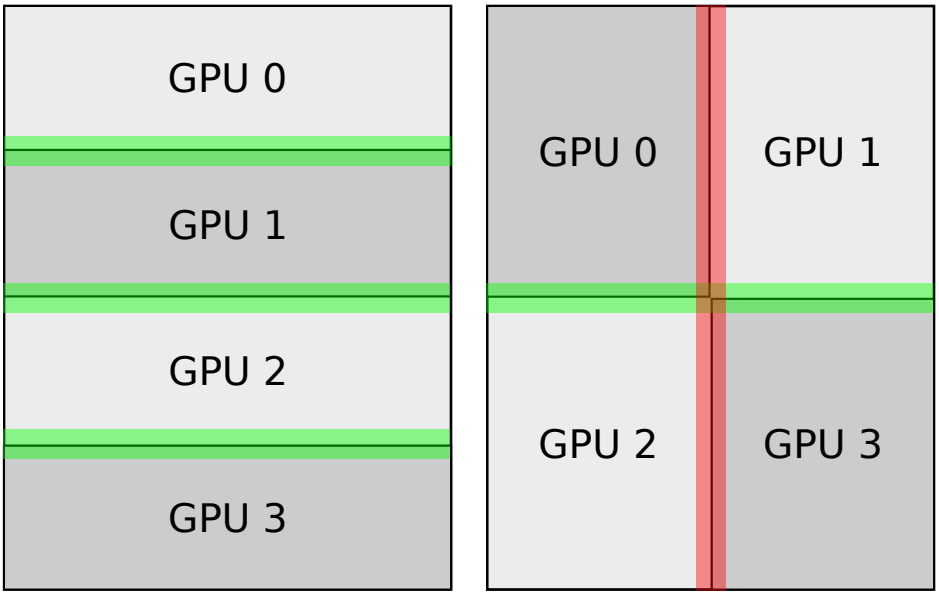


Figure 8.2: Splitting a 2D domain along the same axis or along orthogonal axes. Assuming the data are linearized per row, it is possible to copy each of the green edges in one memory transfer, while the red edges require many small transactions.

It is still possible to choose a different split plane for each simulation at compile time. This is implemented by defining three different compiler macros and letting the programmer enable any of the three; the way cells are enumerated and 3D grid is linearized changes accordingly. The compiler macros for the device code are reported in listing 8.1. The only simplification with respect to the actual code is the use of the normal multiplication instead of `__mul24()` method, which is a faster replacement performing a 24bit multiplication.

```
#if defined(PROBLEM_SIMMETRY_PLANE_XY)
    #define PSA z
    #define CALC_SIMMETRY_HASH return \
        gridPos.z * (gridSize.y * gridSize.x) + \
        gridPos.y * (gridSize.x) + \
        gridPos.x;
#elif defined(PROBLEM_SIMMETRY_PLANE_XZ)
    #define PSA y
    #define CALC_SIMMETRY_HASH return \
        gridPos.y * (gridSize.z * gridSize.x) + \
        gridPos.z * (gridSize.x) + \
        gridPos.x;
#elif defined(PROBLEM_SIMMETRY_PLANE_YZ)
    #define PSA x
    #define CALC_SIMMETRY_HASH return \
        gridPos.x * (gridSize.y * gridSize.z) + \
        gridPos.y * (gridSize.z) + \
        gridPos.z;
#else
    #error No split plane defined in the selected problem.
// the following avoids further compile errors
    #define PSA x
#endif
```

Listing 8.1: Compiler defines for different linearization of $3D$ cells according to the split plane; PSA stands for Principal Split Axis.

It is up to the programmer to decide which split plane is the most convenient. As a rule of thumb, one should choose the split that minimizes the number of particles per slices (i.e. the sections of fluid being cut). For most problems the choice has no big consequences as the time required for transferring an overlapping slice, whatever the chosen plane, is usually completely covered by the force computation. Very asymmetric topologies, however, may benefit from a proper cut when load balancing, as the balancing granularity is at the slice level (see section 8.10).

8.3 Subdomain overlap

To ensure that all the neighbors of a particle p reside on the same device as p we need a small section of the subdomain of each GPU to overlap with neighbor devices. An overlapping the size of the influence radius for each GPU is necessary and sufficient to have this condition guaranteed. As either one of two neighboring subdomains must overlap with the other, the total overlap is twice as wide as the influence radius. Recalling that the split is done by means of the grid of virtual cells, and that cells have size equal to the influence radius, the total overlap is exactly two cells wide.

In three-dimensional space, we can think of *slices* of volume to be exchanged at every iteration (see fig. 8.3). Each device needs a copy of the first slice of neighboring devices as read-only information for the interaction with neighbors, and sends its edging slices as read-only copy to neighboring devices. We refer to the assigned read/write particles as *internal* ones and to the read-only particles as *external*. When referring to slices, *internal* and *external* also means edging (i.e. first or last slice of the device subdomain).

Figure 8.3 represents one possible split of a simulation domain. The subdomains assigned to two devices and their overlaps are highlighted. The position of the particles is taken from an actual simulation and particles are colored by velocity. The blue spot in device n.2 is due to the impact with an obstacle not drawn for visualization purposes.

8.4 Kernels

We now discuss about the kernels and what changes were necessary to pass from one GPU to multiple GPUs.

Working on a subset of the global domain is trivial for step 1 (neighbor search) and 4 (integration) of the model. We have to be careful only in running them on the

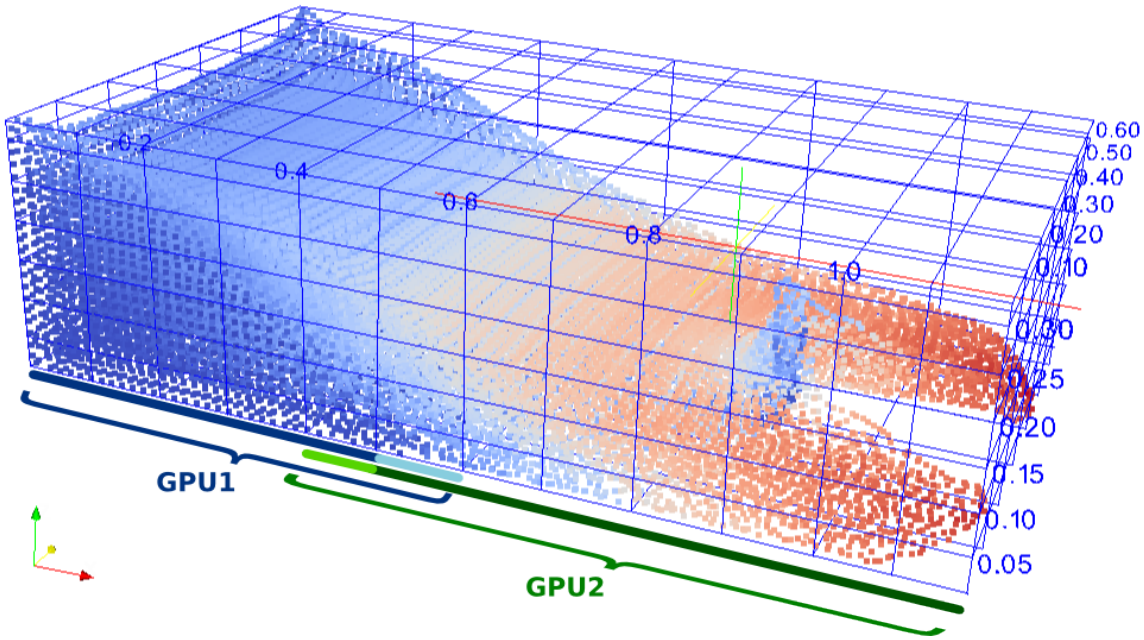


Figure 8.3: Fluid volume with virtual cells, split on YZ plane. The blue line shows the internal cells of GPU n.1; light blue the external read-only slice updated by GPU n.2; internal cells of GPU n.2 are green, while the external slice is light green. Particles are colored by velocity and cells are not in scale for visualization purposes.

appropriate particles subset. In particular, the neighbor search reads all internal and external particles but produces the neighbor list of internal particles only. The integration, instead, is run on all particles, as the external ones are exchanged as forces, as later explained, and new positions have to be computed. Running the integration also on the external particles means that in the whole multi-GPU simulation the overlapping slices are integrated twice. Fortunately, this overhead is really narrow, as the integration step barely saturates the GPUs.

Step 3 (minimum scan) is straightforwardly extended to n GPUs: each device finds its local minimum and sends it to the CPU, that quickly compares the few local minima and finds the global one. We need a barrier to wait for all the local minima to be ready. Step 2 requires a further remark. The neighbors of the external particles may not be available, as the next 3D slice reside on a different device. Therefore, any force computed on an external particle would be *partial* and should not contribute to the computation of the final dt. We therefore run the force computation kernel only on the internal particles; external ones are accessed only

as neighbors of internal ones.

8.5 Hiding slice transfers

Each device needs an updated copy of the neighboring slices at each iteration. More specifically, we need updated positions and velocities of particles in neighboring slices each time forces are computed and integrated; due to the two-step integration, this update has to be done twice for every iteration. This introduces a conspicuous overhead. The contribute to the whole simulation time greatly varies according to the density and topology of the simulated fluid. In some cases it can even make a multi-GPU simulation perform worse than a single-GPU one.

To overcome this problem we exploit the hardware capability of performing concurrent computations and data transfers, similarly to what we did for the multi-GPU MAGFLOW. We use the *asynchronous API* described in section 3.8 to begin the transfers as soon as the edging slices are ready, while the other one are still being computed.

We initialize three *CUDA streams* for each device; we will use two of them to enqueue operations about the edging borders and one for the remaining slices. Except for the first and the last device, which only have one neighboring device and thus one edging slice, all devices have two edging slices. We will describe the behavior of a device with two neighbors, as the devices with only one neighbor are just a simpler case. A first, simple design would be to issue first a `Forces` kernel on edging slices; download the edges as soon as they are ready, while computing the forces on the remaining slices; run the `MinimumScan`; run `Euler` kernel on the non-edging slices while uploading the updated external edges; finally, integrate also the edging slices as soon as the uploads are complete. Fig. 8.4 contains a graphical representation of this scheme and they way the GPU performs the issued operations.

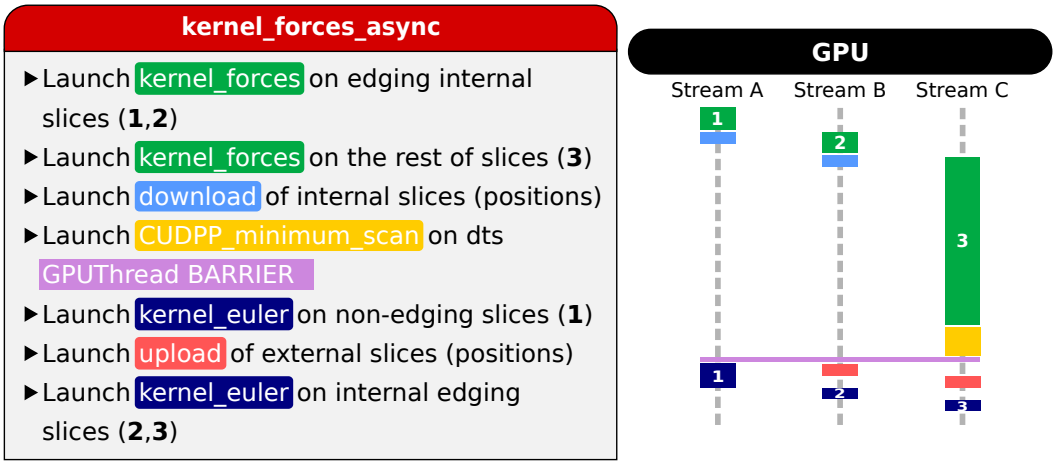


Figure 8.4: Early design of `kernel_forces_async` method. Note that the positions (and velocities) are exchanged, and the method includes the `MinimumScan` and the integration.

Another possibility could be to exchange the forces instead of positions and velocities. This would have two major advantages:

1. We can start uploading the external slices while the forces on the non-edging slices are still being computed; because the `Forces` kernel takes longer than euler, transfers are more likely to be completely hidden.

2. We need to transfer less data (forces instead of positions and velocities). This is true unless we need additional structures, such as τ coefficients for SPS correction [69].

As already mentioned, this comes at the small price of running the integration kernel also on external particles. We chose the latter approach, that is represented in fig. 8.5. The minimum scan and the integration are not included in the method anymore, as all asynchronous transfers finish before the `Forces` kernel does.

In both figures 8.4 and 8.5, CUDA events are omitted for simplicity. They are enqueued in every stream after each operation to allow fine control of completed operations. As an example, streams A and B are synchronized (i.e. waited for) just before the purple colored barrier, that is a "soft" barrier that only `simulationThreads` reach. Once the barrier has been reached by all threads we are thus sure that the exchange buffers are updated and ready to be uploaded to the device.

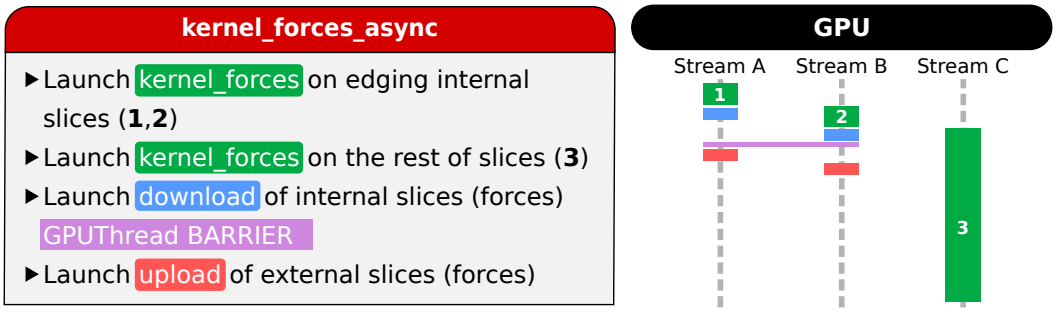


Figure 8.5: Final design of `kernel_forces_async` method. Note that the forces are exchanged instead of the positions; as a consequence, `euler` kernel must be run also on external particles.

Launching the `Forces` kernel on a single slice is in general not convenient, as it very likely does not saturate the device; in other words, running the kernel on more than one row would require the same time though reducing the workload of the "central" launch. It is thus convenient to launch the kernels on the edging *stripes*, i.e. a small number of slices with enough particles to saturate the GPU. In our tests 1/16 of the total subdomain was an arbitrary size performing excellent in any setting. The transfer of edges, instead, will only affect single edging rows.

8.6 Simulator design

Similarly to the implementation of the MAGFLOW simulator, we encapsulate all pointers and CUDA calls needed to handle a GPU into the class `GPUThread` (not to be confused with the homonymous GPU instances of a kernel). The main thread allocates one `GPUThread` per device and each `GPUThread` starts a dedicated `pthread` in constant communication with the associated device. The main thread tracks the simulation time and periodically requests a subdomain dump from the GPUs according to a user-defined save frequency. Treasuring the experience we had with MAGFLOW, we synchronized the threads through a GPU flush (`cudaThreadSynchronize`) and the NTPL implementation of `pthread_barrier_wait()` as CPU barrier.

As for the MAGFLOW simulator, it is possible to specify at command line several simulation options and it is possible to run a simulation on a heterogeneous set of devices, even belonging to different hardware generations. The data is not multi-GPU aware: it is possible to save the state of a single-GPU simulation and to restore it in a multi-GPU environment, and vice-versa.

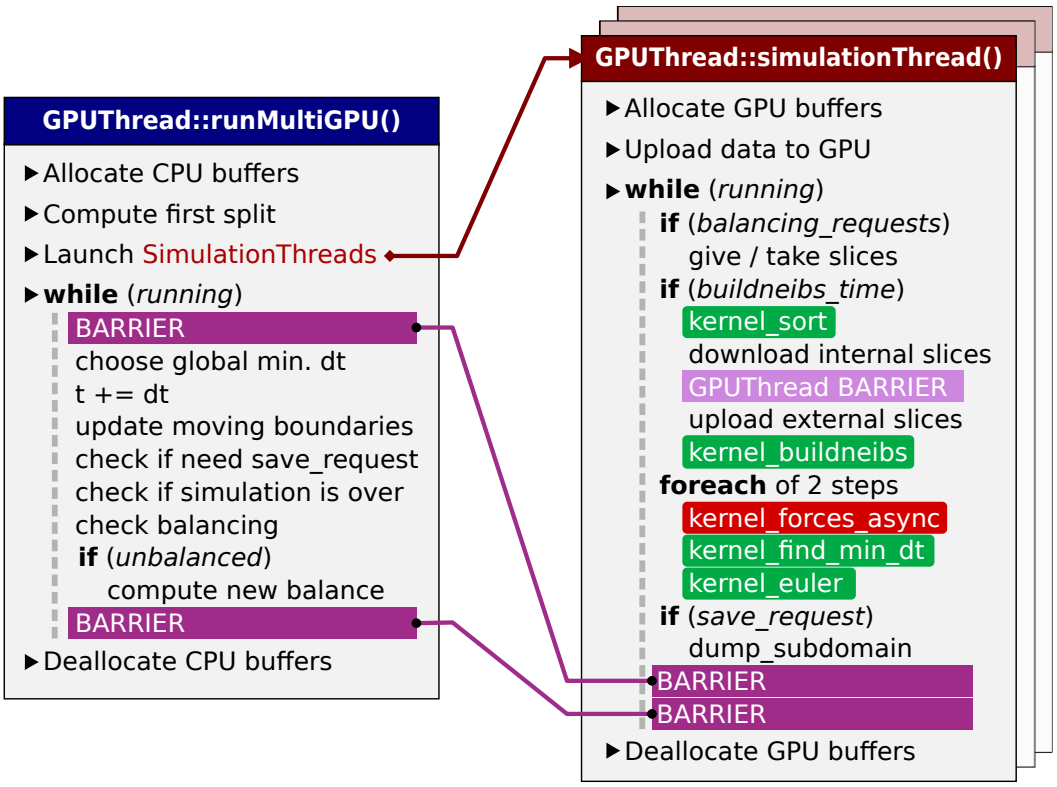


Figure 8.6: Simplified scheme of the simulator structure. GPU kernels are green; NPTL barriers are purple; the red kernel_forces_async method encapsulates also asynchronous slice exchange. Light purple barriers are only reached by `simulationThread`s for inter-GPU synchronization while darker ones serve for communication between the main thread and the `simulationThread`s.

8.7 Performance metrics

As we often simulate problems exhibiting different densities, topologies and number of particles, the absolute execution time required by a simulation to complete is too simple a performance metric to our purposes. It is not even useful to measure the straight speedup in the Amdahl's or Gustafson's meaning, unless comparing

simulations with identical settings and simulated time. We needed a more versatile metric abstracting from the specific simulation settings.

We propose to measure the amount of work done within a simulation as the number of fulfilled iterations times the number of particles. This can be considered as the number of single iterations completed (imagining, for example, a hypothetical single-core GPU). To abstract from the simulation length we can simply divide by the real time. We chose seconds as time units and, as the number of particles often exceeds the million, we found useful to consider thousands of them. We are thus measuring *thousands of iterations on particles per second*; in short form, we call this unit *kip/s* or simply *kips*. Another advantage of this metric compared to the mere speedup is that it is possible to compute the instantaneous speed at runtime, with no need to wait for a simulation to complete. This metric, however, is specific for SPH simulations and may not be suitable for other models.

In the following section we also give the performance measure in kips, although for the final performance evaluation we still use the speedup in the Amdahl's meaning, to ease a possible comparison with any future or independent work. It is worth recalling that for any comparison to be accurate the same integration scheme and physical settings must be used. It is also advisable to simulate similar fluid topologies, as different topologies can still affect memory coalescence and thread/block scheduling.

8.8 Preliminary results

The testbed was the simulation of 0.43m^3 of water divided into 1 million particles for 1.5s of simulated time, on the rack described in section 4.8. Up to six devices were used. No load balancing techniques were used in this phase of the test; the list of particles is fairly divided among the devices and the subdomain size remains constant through the whole simulation. Videos of the reference simulation run

DamBreak3D	1 GPU	2 GPUs	3 GPUs	4 GPUs	5 GPUs	6 GPUs
Kip/s	10,304	19,71	27,981	35,376	40,861	45,675
Real time (s)	3,641	1,885	1,331	1,072	914	823
Ideal time (s)	3,641	1,821	1,214	910	728	607
Real cost (s)	0	65	117	162	186	216

Table 8.1: Multi-GPU GPUSPH *kips* and execution times and cost in seconds, 1 to 6 devices, 1 million particles

on different number of devices are available at http://www.dmi.unict.it/~rustico/sphvideos.

Table 8.1 shows the execution time in seconds of the same simulation running on a variable number of GPUs. The execution time of two simulations with the same settings (including the number of GPUs used) did not vary more than a couple of seconds over the total time (∼0.1%), so there was no need to average several instances of the same simulation. Fig. 8.7 is a graphical representation of values in table 8.1.

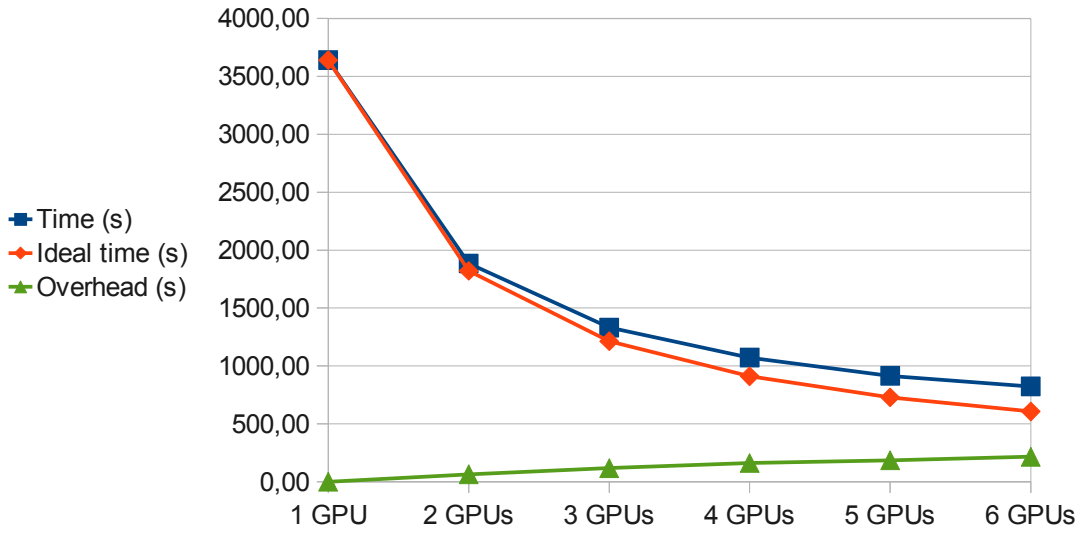


Figure 8.7: Simulation time: real, ideal and difference between the two, 1 to 6 devices

The simulation scales almost linearly with the number of GPUs. The overhead due to the transfer of overlapping slices at each iteration is completely covered (as shown in fig. 8.8); however, the lack of load balancing causes a tangible difference

from the ideal times. The cost is linear with the number of GPUs and, with the above settings, it is roughly equal to one minute for every additional device used. In terms of simulation time, each device adds an overhead approximately equal to $T \cdot 1.6 \cdot 10^{-2}$, with T being the reference single-GPU simulation time. According to the estimation (4.6), it would be convenient to use up to 8 GPUs.

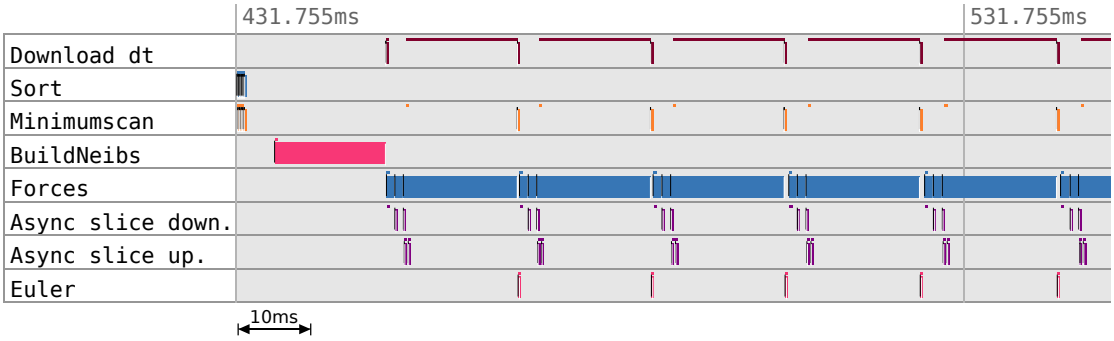


Figure 8.8: Actual timeline of a multi-GPU GPUSPH simulation, with kernel lengths in scale (lengths smaller than 1 pixel have been rounded up). Only one GPU with 2 neighboring devices is shown. The little dots mark the moment operations were issued from the CPU. Downloading the `dt` is blocking for the CPU (from which, the long line) until the minimum scan is complete.

Fig. 8.8 shows the actual sequence of events as profiled during the simulation of a `DamBreak3D` with 1 million particles on 3 GPUs. The exchange of slices (purple) is actually performed concurrently with computation of forces, and effectively start as soon as the computation of forces on edges is completed. While the rectangles plot the start time and duration of the events on the device, the little dots above them mark the timestamps of the same operations as issued on the host. All operations are asynchronous to the host except for the download of the `dt`, which is blocking for the time represented by the long brown line.

We also reached the second aim of the multi-GPU GPUSPH: the possibility to run simulations that would not fit in the memory of a single device. Taking into account the additional buffers needed for a multi-GPU simulation, such as the overlapping stripes and the exchange buffers, we have been able to run a simulation with about 10 million particles on 6 GPUs.

8.9 Numerical precision

It is worth mentioning that the numerical results may vary a little bit when changing the number of GPUs used. We already mentioned in section 2.3 that addition is not commutative in floating-point arithmetic. Particles are sorted by means of the containing cell index; during the exchange of borders, many slices are moved along the particle array and we have no guarantee that the radixsort applied afterwards preserves the inner-cell particle order.

The numerical difference observed in summing floats in a different order depends on the values being sorted; in our case, the error should be no higher than 10^{-7}. Through thousands of iterations, this error propagates in the fluid and amplifies up to $\sim 10^{-5}$ in the computed timesteps. It is therefore acceptable to observe a slightly different number of iterations and some minor differences in the final particle positions, as it is possible to observe in the videos.

8.10 Load balancing

In the ideal case of all the GPUs taking the same amount of time to complete each step, the simulation time is expected to speedup in a quasi-linear way (with the only exception of constant factors such as kernel launch latency). In general, this is difficult to guarantee, and even small differences from the average time may lead to a considerable performance loss. A device taking n% more time than the average to perform all the operations between two synchronization barriers will worsen the whole simulation time by exactly n%.

Dividing the fluid in parts of the same size (i.e. same number of particles) does not always lead to the optimal workload balance. Indeed, many unpredictable elements may influence the total computation time, such as the sparsity of neighbor particles since last sort, the fluid topology, branch divergences inside a kernel and even hardware factors such as PCI interrupts and bus congestion. No balancing

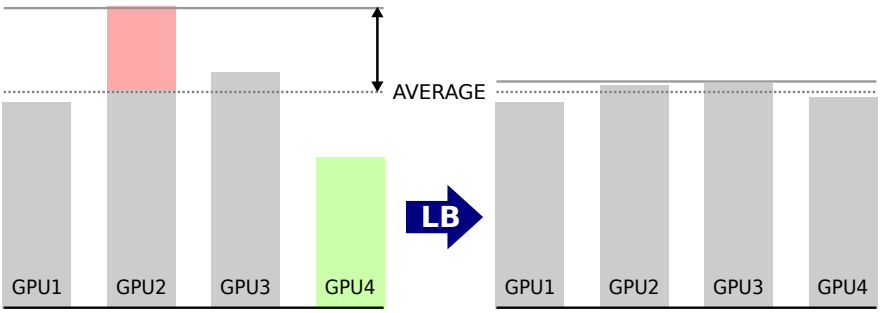


Figure 8.9: Average execution times before and after load balancing.

model can take all these factors into account without relying on the execution time of the previous steps. An *a posteriori* load balancing technique is more appropriate here.

The key idea is very simple: we keep track of the time required by each GPUs for the `Forces` kernel and we ask the GPUs taking longer than the average to *give* a slice of their subdomain to GPUs taking less.

More in detail, we consider the average time A_g taken by a single GPU g to complete the forces kernel on the central set of particles over the last k iterations. A smart choice of k could be a multiple of the number of iterations between two reconstructions of the neighbor list, to minimize the number of sorts; in our case, $k = 10$. We then compute the cross-GPU average

$$A_G = \sum_{d=1}^{D} A_d / D$$

and $A_{\delta g} = A_g - A_G$.

If $|A_{\delta g}| \geq T_{LB}$, with T_{LB} as balancing threshold, we mark the GPU g as *giving* (if $A_{\delta g} > 0$) or *receiving* (if $A_{\delta g} < 0$). A giving GPU "sends" one slice to the receiving one; if they are not neighboring, every intermediate GPU gives one slices and takes another at the appropriate edge. A_g is reset to wait for next k iterations. Listing 8.2 represents this algorithm in pseudo-code.

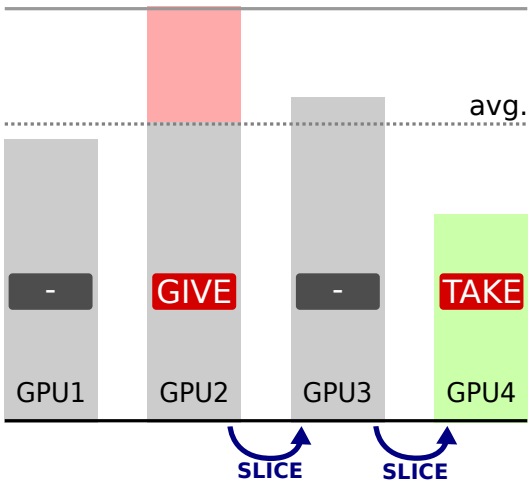


Figure 8.10: Example of balancing: GPU n.2 is marked as GIVING, GPU n.4 as TAKING and GPU n.3 is requested to take a slice from GPU n.2 e give one to GPU n,4.

```
1  AG ← global avg. duration of kernel Forces
2  Ts ← avg. duration of one slice = AG / num. slices
3  foreach GPU
4      Ag ← self avg. duration of kernel Forces
5      delta ← Ag - AG
6      if (abs(delta) > Ts * HLB_THRESHOLD)
7          if (delta > 0)
8              mark as GIVING_CANDIDATE
9          else
10             mark as TAKING_CANDIDATE
11 foreach GPU
12     if a couple GIVING/TAKING is found
13         mark GIVING_CANDIDATE as GIVING
14         mark TAKING_CANDIDATE as TAKING
15         foreach intermediate GPU
16             mark as GIVING and TAKING
17         break
```

Listing 8.2: Load balacing pseudocode

The threshold T_{LB} must be big enough to avoid sending slices back and forth and small enough to trigger the balancing when needed. Let T_{slice} the average time required to compute the `Forces` kernel on a single slice; because the granularity

is at the slice level, it convenient to set T_{LB} proportionally to T_{slice}:

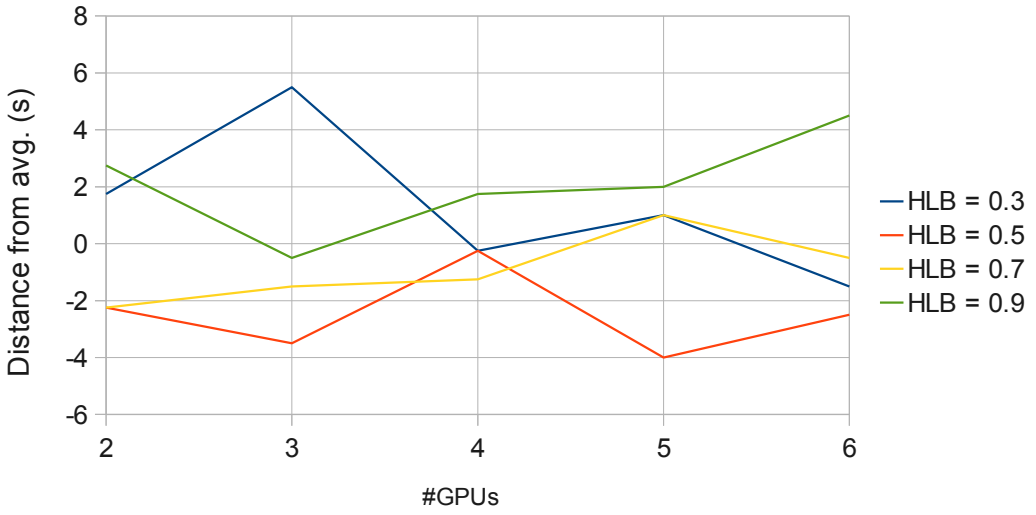
$$T_{LB} = H_{LB} \cdot T_{slice}$$

with H_{LB} as *balancing threshold*. In our tests $H_{LB} = 0.5$ performed quite well in the general case, but might need a manual fine-tuning in specific simulations.

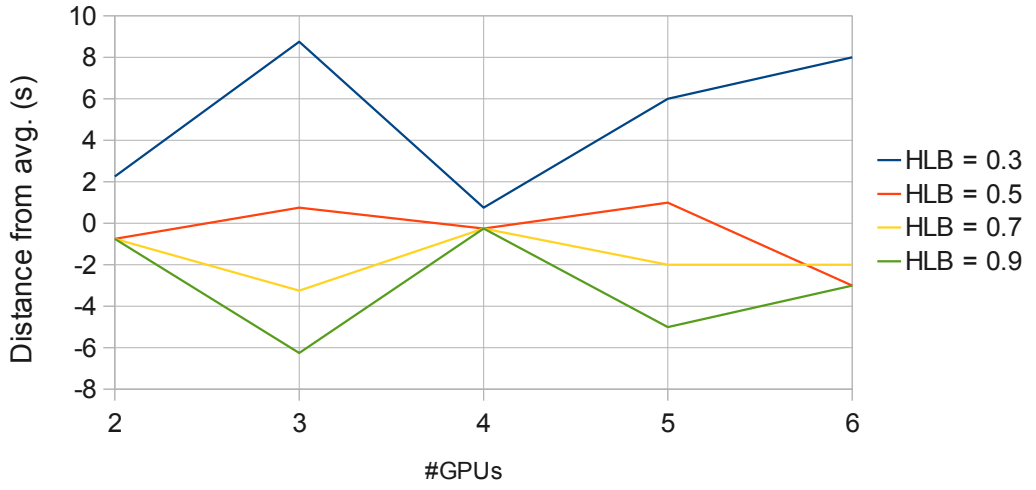
Fig. 8.11a shows the variation of performance (distance in seconds from the average time with the same number of GPUs) with different values of H_{LB} when simulating `BoreInABox` problem. Choosing $H_{LB} = 0.5$ is a good choice in the general case, performing better than higher as well as lower thresholds. Fig. 8.11b shows the variation of performance with problem `DamBreak3D`. We first notice an oscillating behavior due to the presence of a narrow obstacle that changes the general subdivision of the fluid: when simulating with an odd number of GPUs, one of them entirely contains the obstacle, while with an even number of devices the obstacle is "shared" between two of them. The `DamBreak3D` is roughly symmetric along the split axis and it is quite balanced even with a static split. It is therefore not surprising that in this case a higher threshold (i.e. less balancing overhead) slightly increases the performance.

It is worth mentioning, however, that these considerations are only interesting for the sake of theoretical research. The performance variations refer to simulations lasting several hundreds of seconds (from about 1,200s with 2 devices to about 400s with 6 devices) and are thus almost negligible in practice.

Figures 8.13 and 8.12 show different snapshots of two `BoreInABox` simulations with about 1.1 million particles, each Each particle is colored according to the device it belongs to, so that the dynamics of balancing are highlighted. In the corridor variant the workload keeps becoming more and more asymmetric, while in the standard `BoreInABox` the originary symmetry is partially restored when the fluid redistributes evenly on the floor.



(a) `BoreInABox`



(b) `DamBreak3D`

Figure 8.11: Performance of load balancing (distance in seconds from the average time) with respect to threshold H_{LB}.

The load balancing policy and algorithm described so far is not perfect and, while it works reasonably well in practice, may not converge to the optimal balance in some classes of situations, remaining stuck in a local minima or in a "ping-pong" slice exchange. This has to be considered as a first attempt to overcome the technical difficulties arising from on-the-fly subdomain resizing and needs further improvements especially in the balancing policy.

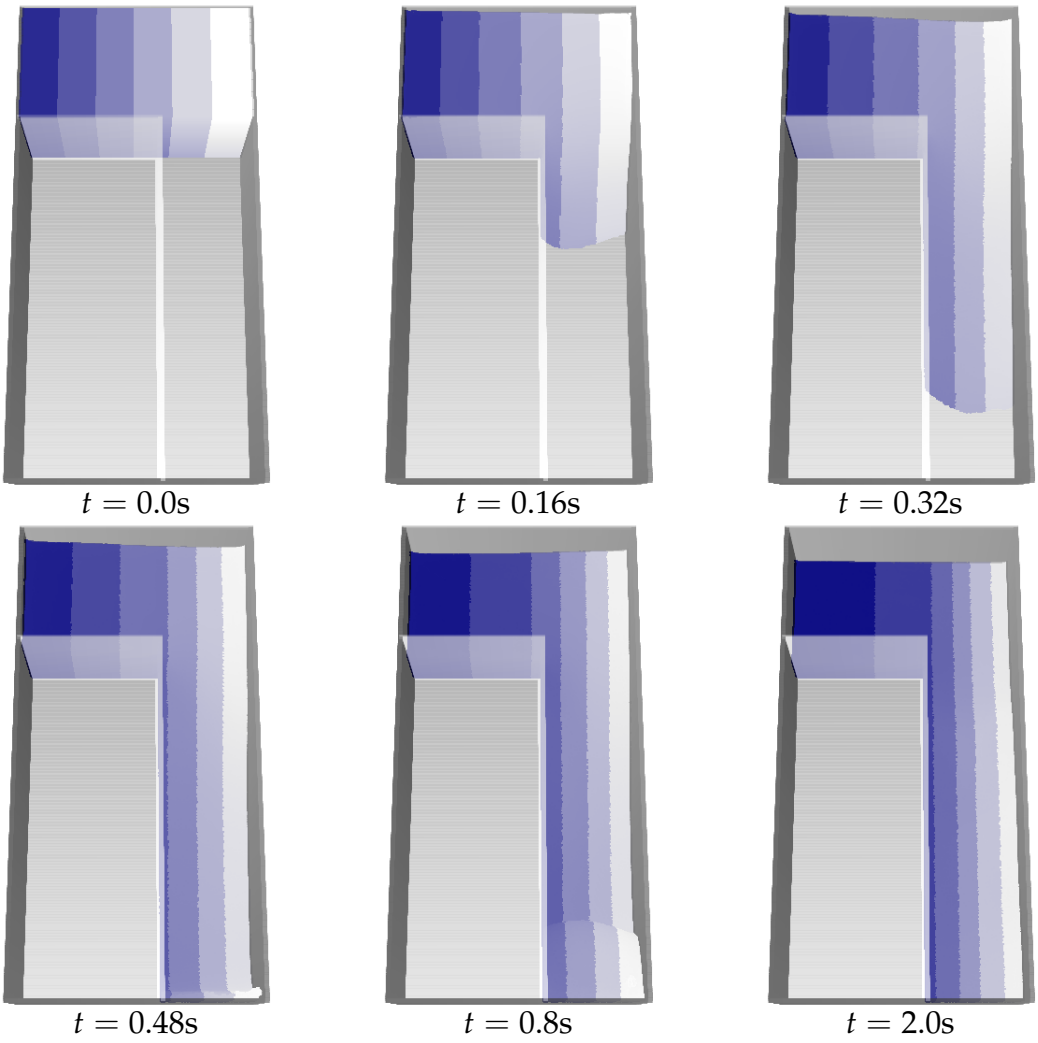


Figure 8.12: Snapshots of different phases of the `BoreInABox` problem simulation, corridor variant, on 6 GPUs. Each particle is colored according to the device number it belongs to.

8.11 Final results

In GPU-SPH terminology, a *problem* is the definition of a physical domain, fluid volumes and geometrical shapes (a scene) to simulate. The effectiveness of the implemented load balancing policy has been tested by measuring the achieved performance during the simulation of two problems, `DamBreak3D` and the `BoreInABox`, with about 1.6 million particles each, from 1 to 6 GPUs. The resulting times are showed in tables 8.2 and 8.3; tables 8.4 and 8.5 show the measured kips/s; charts 8.14 and 8.15 plot the execution times for visual comparison.

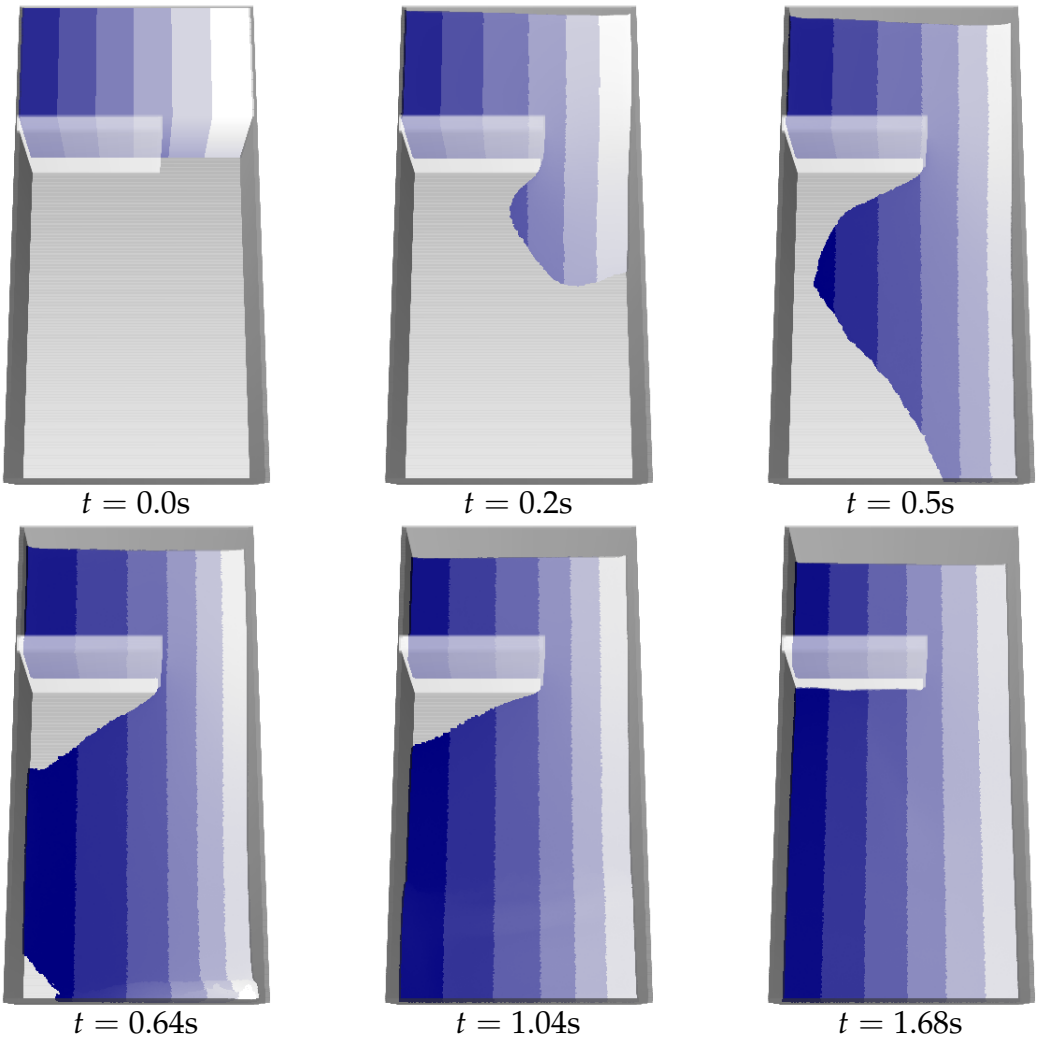


Figure 8.13: Snapshots of different phases of the `BoreInABox` problem simulation, with no lateral corridor, on 6 GPUs. Each particle is colored according to the device number it belongs to.

The `DamBreak3D` problems presents a high level of symmetry, especially when splitting the problem along the Y axis. The number of particles assigned to each GPU is roughly constant during the whole simulated time (1.5s) and the load balancing is not expected to make a big change. Simulating over 2 and 4 GPUs lead to an advantage of roughly 1% total simulation time; on 5 and 6 GPUs simulations run between 6% and 13% faster than without any balancing. Nevertheless, simulating on 3 GPUs is surprisingly about 1% *longer* with load balancing activated. This is probably a consequence of the domain shape and distribution of particles. With 3 GPUs, two of them are assigned to lateral stripes (thus, have a higher

percent of static particle-border interactions) while the central one deals with the obstacle. The naturally balanced particle distribution highlights the little overhead of balancing attempts (112 slices moved over about 35k iterations), without any performance gain.

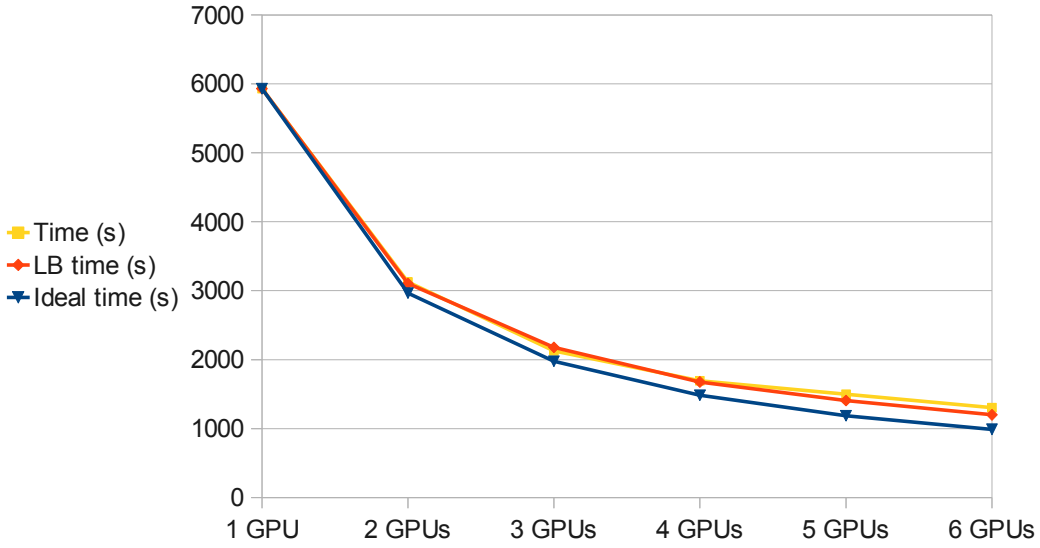


Figure 8.14: Multi-GPU GPU-SPH execution times for a simulation with 1.6 million particles, `DamBreak3D` problem, 1-6 GPUs.

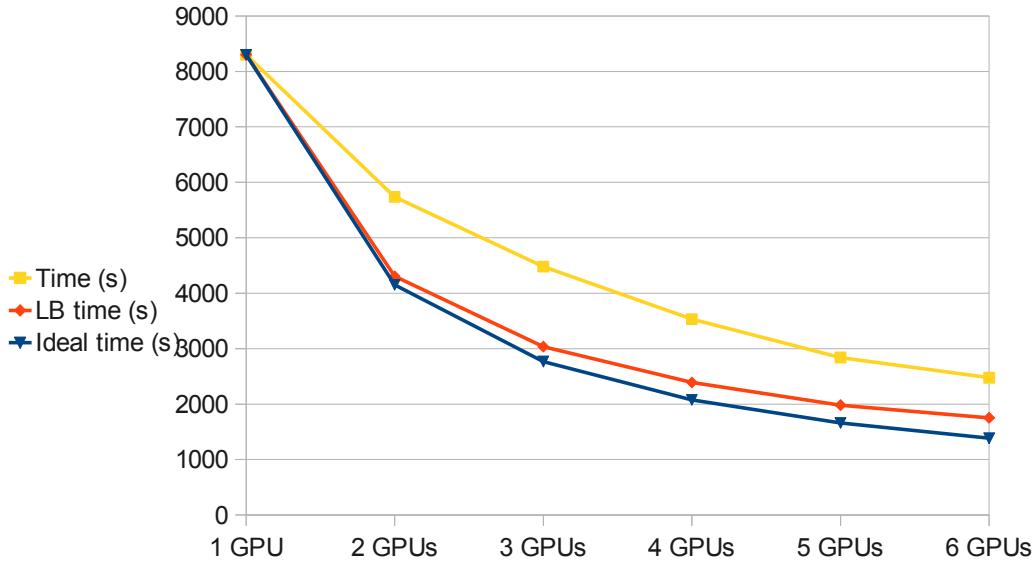


Figure 8.15: Multi-GPU GPU-SPH execution times for a simulation with 1.6 million particles, `BoreInABox` problem, 1-6 GPUs.

When simulating an asymmetrical problem like `BoreInABox`, however, load

balancing makes a big performance difference. During the 2.5s simulated time of `BoreInABox`, the particles flow in the lateral corridor and one third of the simulation domain, considering Y as split axis, receives two thirds of the total fluid. about In fig 8.15 it is possible to see how load balancing keeps the performance close to the ideal ones, while without balancing the performance drops (or, execution time jumps up). In such cases, load balancing also allows for bigger simulations, as it is possible to reduce the allocation margin factor M_f and save space for further particles.

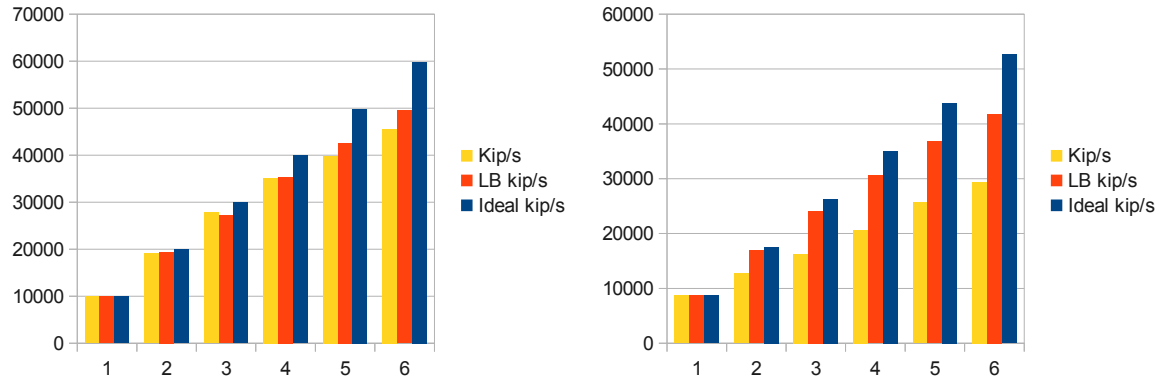


Figure 8.16: Multi-GPU GPU-SPH performance in kips, 1.6 million simulation, `DamBreak3D` and `BoreInABox` problems, 1-6 GPUs

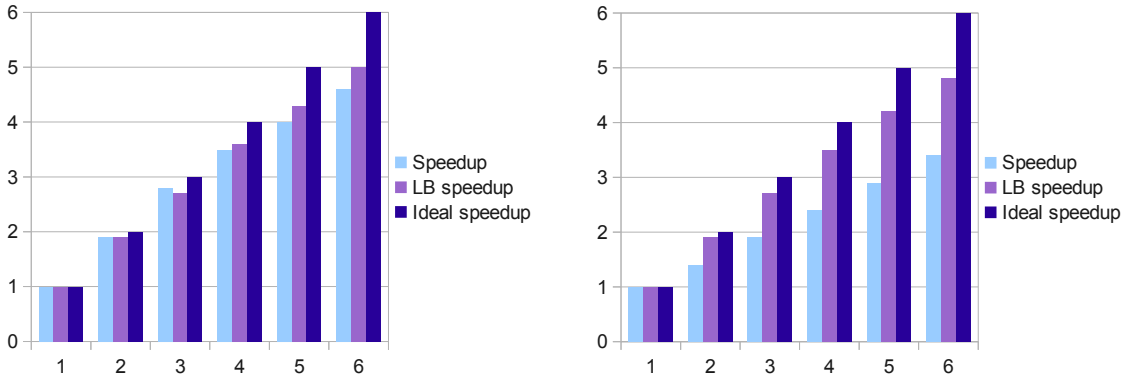


Figure 8.17: Multi-GPU GPU-SPH speedups, 1.6 million simulation, `DamBreak3D` and `BoreInABox` problems, 1-6 GPUs

DamBreak3D	1 GPU	2 GPUs	3 GPUs	4 GPUs	5 GPUs	6 GPUs
LB off (s)	5,932	3,127	2,128	1,693	1,500	1,304
LB on (s)	-	3,103	2,179	1,676	1,406	1,200
Ideal time (s)	-	2,966	1,977	1,483	1,186	989

Table 8.2: Multi-GPU GPU-SPH execution times, without and with load balancing (LB), `DamBreak3D` with 1.6 million particles.

BoreInABox	1 GPU	2 GPUs	3 GPUs	4 GPUs	5 GPUs	6 GPUs
LB off (s)	8,301	5,737	4,479	3,532	2,841	2,477
LB on (s)	-	4,304	3,037	2,391	1,982	1,752
Ideal time (s)	-	4,150	2,767	2,075	1,660	1,383

Table 8.3: Multi-GPU GPU-SPH execution times, without and with load balancing (LB), `BoreInABox` with 1.6 million particles.

DamBreak3D	1 GPU	2 GPUs	3 GPUs	4 GPUs	5 GPUs	6 GPUs
Kip/s	9,977	19,149	27,802	35,213	39,784	45,599
LB kip/s	-	19,380	27,191	35,336	42,578	49,491
Ideal kip/s	-	19,955	29,932	39,910	49,887	59,865

Table 8.4: Multi-GPU GPU-SPH kip/s, without and with load balancing (LB), `DamBreak3D` with 1.6 million particles.

BoreInABox	1 GPU	2 GPUs	3 GPUs	4 GPUs	5 GPUs	6 GPUs
Kip/s	8,770	12,713	16,275	20,649	25,657	29,418
LB kip/s	-	16,940	24,115	30,548	36,800	41,745
Ideal kip/s	-	17,541	26,311	35,082	43,852	52,623

Table 8.5: Multi-GPU GPU-SPH kip/s, without and with load balancing (LB), `BoreInABox` with 1.6 million particles.

DamBreak3D	2 GPUs	3 GPUs	4 GPUs	5 GPUs	6 GPUs
β_{KF}, no LB	0.053	0.036	0.048	0.063	0.061
β_{KF}, LB	0.053	0.056	0.037	0.041	0.040

Table 8.6: Multi-GPU GPU-SPH measured β_{KF} with Karp-Flatt formula, without and with load balancing (LB), **DamBreak3D** problem.

BoreInABox	2 GPUs	3 GPUs	4 GPUs	5 GPUs	6 GPUs
β_{KF}, no LB	0,429	0,289	0,222	0,181	0,153
β_{KF}, LB	0,053	0,056	0,048	0,048	0,050

Table 8.7: Multi-GPU GPU-SPH measured β_{KF} with Karp-Flatt formula, without and with load balancing (LB), **BoreInABox** problem.

The empirically measured β_{KF} confirms the efficiency of the implementation (table 8.7). The simulator performs quite well for `DamBreak3D` with load balancing either enabled or disabled. With no balancing β_{KF} reaches approximately 6% with 5 and 6 GPUs, while load balancing bounds it to about 4%. In general, β_{KF} does not show any significant growth and this confirms that no tangible overhead is being introduced for the execution on multiple GPUs. Problem `BoreInABox` shows a more peculiar trend, with β_{KF} inversely proportional to D. β_{KF} is pretty stable around 5% when load balancing is enabled, while a static split exposes an initially high but *decreasing* β_{KF}. In fact, it is $\beta_{KF} \approx 43\%$ with 2 GPUs and it decreases to 15% with 6 (yet, three times bigger than with load balancing). This is due to the simplistic assumption that the parallelizable part α scales in a homogeneous way, which is no more than a rough approximations for meshless fluid simulations. The strong asymmetry in the distribution of the fluid in the `BoreInABox` problem affects the performance in a non-linear way, as if we had portions of α each with its own scaling factor. Increasing the number of devices corresponds to a performance improvement, as we are slightly refining the subdivision granularity. In general, using a slightly inaccurate notation, the condition $\beta_{KF}(d_1) > \beta_{KF}(d_2)$ with $d_1 < d_2$ suggests that part of β is probably due to asymmetry in the workload rather than in serial, non parallelized portions of the problem. In such cases, load balancing is necessary for an efficient parallelization.

The current balancing implementation still exposes some uncovered overheads. In particular, the slice transfers right after the construction of the neighbor list (i.e. when a particle "moves" from one device to another) are not covered. Moreover, as the load balancing operations must be complete before kernel `BuildNeibs` starts, the load balancing itself is an operation with a little uncovered cost. In the same simulations which were used to estimate β_{KF}, the number of slices moved across different devices for balancing purposes ranges from about 700 in the 2- and 3-GPUs simulations to 1,700 in the 6-GPUs ones. Each slice transfer takes from tens

to hundreds of microseconds and needs to be performed twice; the total overhead easily reaches a couple of minutes per simulation. With a more complex simulator design it could be possible to cover all these transfers or even merge them into the operations for the updated of neighboring slices.

Part IV

Conclusions

Chapter 9

Conclusions

We have implemented and tested a multi-GPU version of two fluid simulators based on two rather different numerical models.

The experience maturated with MAGFLOW highlights how important is the ratio between the mathematical throughput of a numerical model and the amount of memory accesses. Although most of the accesses to global memory are coalesced, devices are not fully saturated, and this becomes evident when increasing the number of GPUs used. The small, linear cost needed to increase by one the number of GPUs used can become quickly comparable with the expected gain and voids the advantage of using more than 4 GPUs. On the other hand, the *a priori* balancing technique adopted to distribute the simulation domain according to the computational power is not flexible as a *a posteriori* balancing could be. We did not have the need to further improve the simulator performance, as the highest DEM resolutions we used barely saturated the hardware; however, a more dynamic balancing policy should be the first step for any future improvement in case the computational burden is increased.

The GPUSPH simulator has a far higher mathematical throughput, due to the heavy simulation model on which it relies. Although the absolute number of global memory accesses is higher than in MAGFLOW, the large amount of mathematical

operations performed, together with the concurrent transfer/execution hardware capability, makes it very convenient to execute a simulation even on 6 GPUs simultaneously. Getting close to the ideal speedup was not trivial and required using many optimizations and some advanced programming techniques. In particular, the implemented *a posteriori* load balancing policy allowed to maintain a high performance level even in adverse, very dynamic simulations, by fairly distributing the computational load as the fluid dynamically changes.

In both cases, the results are quite promising as the execution time of simulations exploiting multiple GPUs is close to the ideal one and the unavoidable overheads, negligible for GPUSPH and tangible for MAGFLOW, were easily foreseen and bounded. Many of the design, programming and GPU-specific techniques we developed can be easily exploited in different applications, similarly to what we did by applying to GPUSPH the know-how aquired during the MAGFLOW development.

The overall experience confirms how difficult is to exploit the available raw computational power in a scalable and efficient way. Especially for non-trivial problems with small granularity and locality requirements, more computational power does not directly lead to a linear performance gain, unless a robust design is planned and the appropriate hardware features are exploited. This goes through a deep knowledge of the underlying platform and a proper usage of parallel programming techniques.

9.1 Publications

The research conducted so far has been published in international journals and conferences.

ISI journals:

- *Scalable multi-GPU implementation of Cellular Automata based lava simulations*,

E. Rustico, G. Bilotta, A. Hérault, C. Del Negro and G. Gallo, Annals of Geophysics, Vol. 54, No. 5, 2011

- *Porting MAGFLOW to CUDA*, G. Bilotta, E. Rustico, A. Hérault, A. Vicari, C. Del Negro and G. Gallo, Annals of Geophysics, Vol. 54, No. 5, 2011
- *LAV@HAZARD: A web-gis interface for volcanic hazard assessment*, A. Vicari G. Bilotta, S. Bonfiglio, A. Cappello, G. Ganci, A. Hérault, E. Rustico, G. Gallo and C. Del Negro, Annals of Geophysics, Vol. 54, No. 5, 2011
- *Numerical Simulation of lava flow using a GPU SPH model*, A. Hérault, G. Bilotta, E. Rustico, A. Vicari and C. Del Negro, Annals of Geophysics, Vol. 54, No. 5, 2011
- A journal article with title *Advances in multi-GPU Smoothed Particle Hydrodynamics simulations* is is being finalized for submission to IEEE Transactions on Parallel Computing

Proceedings:

- *Smoothed Particle Hydrodynamics simulations on multi-GPU systems*, proceedings, E. Rustico, G. Bilotta, A. Hérault, C. Del Negro and G. Gallo, 20th Euromicro International Conference on Parallel, Distributed and Network-Based Computing, Special Session on GPU Computing and Hybrid Computing, 2012, Munich (accepted, forthcoming event)
- *Wave Forces and Fragility for Elevated Residential Structures*, proceedings, A. Kennedy, T. Tomiczek, T. Kijewski-Correa, R. Dalrymple, A. Hérault, G. Bilotta e E. Rustico. In collaboration with the Johns Hopkins University (USA) and the University of Notre Dame (USA). It is being finalized for submission to the International Conference on Coastal Engineering 2012 (ICCE)

Other:

- *Realistic rendering of Smoothed Particle Hydrodynamics lava flows*, poster, E. Rustico, G. Bilotta, A. Hérault and C. Del Negro, Pericolo Vulcani 2010, Roma
- *Scalable multi-GPU implementation of Cellular Automata based lava simulations*, poster, E. Rustico, G. Bilotta, A. Hérault and C. Del Negro, Pericolo Vulcani 2010, Roma
- *Advances in SPH-based lava flow simulation on GPU*, extended abstract, A. Hérault, G. Bilotta, E. Rustico, A. Vicari, C. Del Negro, GNTS 2010
- *A Web Interface for Volcanic Hazard on Mt Etna*, poster, A. Cappello, S. Bonfiglio, A. Vicari, G. Ganci, G. Bilotta, A. Hérault, E. Rustico, G. Gallo and C. Del Negro, EuroVis 2010
- *Alternate cell-based neighbour search method*, technical report. Eugenio Rustico, in collaboration with UNICT-DMI, INGV-CT and Univerisité de Marn-La-Vallée

9.2 Further improvements

We have seen that simulating a lava flow on a DEM with 2m resolution with the MAGFLOW simulator barely saturates a couple of modern GPUs. There is currently no need for any performance improvements on the multi-GPU aspect, unless new tests are required by a slight change in the model (e.g. approximating the dt and avoiding the synchronization barrier needed for the minimum reduction). It is seldom needed to simulate on a 1m resolution DEM; in that special case, a big lava flow with a very wide extent could require more memory than a GPU can hold, as the entire domain is currently uploaded in each device (unlike GPUSPH, where the amount of required memory has to be split). Although we have never encountered such a need, a possible extension could be to upload to each GPU only the active subdomain.

The most important extension we are willing to implement to the GPUSPH simulator is instead the possibility to run a simulation on a cluster of multi-GPU nodes, reaching a third level of parallelism. This would allow for simulations with several millions of particles, enabling very wide simulations (e.g. tsunamis) and/or the simulation of very dense fluids for a superior physical accuracy. The main differences of such a scenario with current multi-GPU single-node implementation would reside in the spatial subdivision policy and the inter-node synchronization mechanism, while the communication latencies could even be smaller than the intra-device ones [1].

Another interesting change would be in the problem granularity. We are currently evaluating the possibility to handle the splits at a finer resolution than the 3D slice. Working at cell granularity, although requiring a more complicated slice addressing system, would allow a finer load balancing and pave the way for accurate second-level splits (e.g. for simulations on GPU clusters).

The load balancing algorithm implemented for GPUSPH is still to be considered a preliminary though successful attempt toward a more sophisticated balancing system. Many possible improvements would need a detailed theoretical analysis and a deep testing: adaptive threshold, history analysis and detection of local minima, just to mention a few.

Finally, we are evaluating the possibility to employ the techniques and know-how developed so far to design a general-purpose task-driven scheduler allowing other computationally expensive numerical problems to exploit a multi-GPU system.

[1] An Infiniband link data rate can exceed 1TB/s with a latency of $\sim 1\mu s$. For comparison, the PCI-Express 2.0 bus can achieve 16GB/s and a CUDA kernel launch requires $\sim 6\mu s$.

Bibliography

[1] T. Amada, M. Imura, Y. Yasumuro, Y. Manabe, and K. Chihara. Particle-Based Fluid Simulation on GPU. *ACM Workshop on General-Purpose Computing on Graphics Processors and SIGGRAPH 2004 Poster Session*, 2004.

[2] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[3] EA Attardo, A. Borsic, and RJ Halter. A multi-gpu acceleration for 3d imaging of the prostate. In *Electromagnetics in Advanced Applications (ICEAA), 2011 International Conference on*, page 1096–1099. IEEE, 2011.

[4] Maria Vittoria Avolio, Gino Mirocle Crisci, Salvatore Di Gregorio, Rocco Rongo, William Spataro, and Giuseppe A. Trunfio. SCIARA $\gamma 2$: An improved cellular automata model for lava flows and applications to the 2002 Etnean crisis. *Computers & Geosciences*, 32(7):876 – 889, 2006. Computer Simulation of natural phenomena for Hazard Assessment.

[5] Ronald Babich, Michael A. Clark, and Bálint Joó. Parallelizing the quda library for multi-gpu calculations in lattice quantum chromodynamics. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[6] Markus Becker, Markus Ihmsen, and Matthias Teschner. Corotated SPH for deformable solids. In Eric Galin and Jens Schneider, editors, *NPH*, pages 27–34. Eurographics Association, 2009.

[7] G. Bilotta, A. Hérault, C. Del Negro, G. Russo, and A. Vicari. Complex fluid flow modeling with SPH on GPU. *EGU General Assembly 2010, held 2-7 May, 2010 in Vienna, Austria, p.12233*, 12:12233–+, May 2010.

[8] G. Bilotta, E. Rustico, A. Hérault, A. Vicari, C. Del Negro, and G. Gallo. Porting and optimizing MAGFLOW on CUDA. *Annals of Geophysics*, 54(5), 2010.

[9] E.C. Bingham. *An investigation of the laws of plastic flow*. Government Printing Office, 1916.

[10] A. Bonaccorso, A. Bonforte, S. Calvari, C. Del Negro, G. Di Grazia, G. Ganci, M. Neri, A. Vicari, and E. Boschi. The initial phases of the 2008–2009 Mount Etna eruption: A multidisciplinary approach for hazard assessment. *J. Geophys. Res.*, 116:1–19, March 2011.

[11] Ali Cevahir, Akira Nukada, and Satoshi Matsuoka. Fast conjugate gradients with multiple GPUs. In *Proceedings of the 9th International Conference on Computational Science: Part I*, ICCS '09, pages 893–903, Berlin, Heidelberg, 2009. Springer-Verlag.

[12] Rosario De Chiara, Ugo Erra, Vittorio Scarano, and Maurizio Tatafiore. Massive simulation using GPU of a distributed behavioral model of a flock with obstacle avoidance. In Bernd Girod, Marcus A. Magnor, and Hans-Peter Seidel, editors, *VMV*, pages 233–240. Aka GmbH, 2004.

[13] Paul W. Cleary and Joseph J. Monaghan. Conduction Modelling Using Smoothed Particle Hydrodynamics. *Journal of Computational Physics*, 148(1):227 – 264, 1999.

[14] NVIDIA Corporation. CUDA 4.1 release candidate announcement. http://developer.nvidia.com/content/cuda-41-release-candidate-now-available, November 2011.

[15] NVIDIA Corporation. NVIDIA paralle nsight. http://developer.nvidia.com/nvidia-parallel-nsight, November 2011.

[16] NVIDIA Corporation. NVIDIA visual profiler. http://developer.nvidia.com/nvidia-visual-profiler, November 2011.

[17] G. M. Crisci, S. Di Gregorio, O. Pindaro, and G. A. Ranieri. Lava flow simulation by a discrete cellular model: first implementation. *International Journal of Modelling and Simulation*, 6(4):137–140, 1986.

[18] R.A. Dalrymple and A. Hérault. Levee breaching with GPU-SPHysics code. In *Proc. Fourth Workshop, SPHERIC, ERCOFTAC, Nantes*, 2009.

[19] R.A. Dalrymple, A. Hérault, G. Bilotta, and R. Jalali Farahani. GPU-accelerated SPH model for water waves and other free surface flows. In *Proc. 31st International Conf. Coastal Engineering, Shanghai*, 2010.

[20] C. Del Negro, L. Fortuna, A. Herault, and A. Vicari. Simulations of the 2004 lava flow at Etna volcano using the magflow cellular automata model. *Bulletin of Volcanology*, 70(7):805–812, 2008.

[21] Mathieu Desbrun and Marie paule Gascuel. Smoothed particles: A new paradigm for animating highly deformable bodies. In *In Computer Animation and Simulation '96 (Proceedings of EG Workshop on Animation and Simulation*, pages 61–76. Springer-Verlag, 1996.

[22] Ulrich Drepper and Ingo Molnar. The Native POSIX Thread Library for Linux. February 2005.

[23] R. A. Gingold and J. J. Monaghan. Smoothed particle hydrodynamics - Theory and application to non-spherical stars. *Mon. Not. Roy. Astron. Soc.*, 181:375–389, November 1977.

[24] D Giordano and D Dingwell. Viscosity of hydrous etna basalt: implications for plinian-style basaltic eruptions. *Bulletin of Volcanology*, 65(1):8–14, 2003.

[25] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23:5–48, March 1991.

[26] Moncho Gomez-Gesteira, Benedict D. Rogers, Robert A. Dalrymple, and Alex J.C. Crespo. State-of-the-art of classical sph for free-surface flows. *Journal of Hydraulic Research*, 48(sup1):6–27, 2010.

[27] Simon Green. Post on NVIDIA Forums. http://forums.nvidia.com/index.php?s=e46ee6f488f2b54b6db79f3a17679409&showtopic=67452&view=findpost&p=381297, May 2008. Post by Simon Green, NVIDIA Corporation employee (accessed 1 November 2011).

[28] The Khronos Group. OpenCL - khronos launches heterogeneous computing initiative. http://www.khronos.org/news/press/khronos_launches_heterogeneous_computing_initiative, June 2008.

[29] John L. Gustafson, Gary R. Montry, Robert E. Benner, C. W. Gear, John L. Gustafson, Gary R. Montry, Robert, and E. Benner. Development of parallel methods for a 1024-processor hypercube. *SIAM Journal on Scientific and Statistical Computing*, 9:609–638, 1988.

[30] M. Gómez-Gesteira, B.D. Rogers, R.A. Dalrymple, A.J.C. Crespo, and M. Narayanaswamy. User guide for the SPHysics code v1.2, 2007.

[31] S. Hadap and Magnenat N. Thalmann. Modeling Dynamic Hair as a Continuum. *Computer Graphics Forum*, 20(3):329–338, 2001.

[32] T. Harada, S. Koshizuka, and Y. Kawaguchi. Smoothed particle hydrodynamics on GPUs. In *Computer Graphics International*, pages 63–70, 2007.

[33] Mark Harris. Mapping computational concepts to GPUs. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[34] Mark J. Harris, Greg Coombe, Thorsten Scheuermann, and Anselmo Lastra. Physically-based visual simulation on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '02, pages 109–118, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[35] A. Herault, G. Bilotta, and R. A. Dalrymple. SPH on GPU with CUDA. *Journal of Hydraulic Research*, 48(Extra Issue):74–79, 2010.

[36] A. Hérault, G. Bilotta, R.A. Dalrymple, E. Rustico, and C. Del Negro. GPUSPH. http://www.ce.jhu.edu/dalrymple/GPU/GPUSPH/Home.html.

[37] A. Herault, A. Vicari, A. Ciraudo, and C. Del Negro. Forecasting lava flow hazards during the 2006 etna eruption: using the MAGFLOW cellular automata model. *Computers & Geosciences*, 35(5):1050–1060, 2009.

[38] A. Hérault, A. Vicari, C. Del Negro, and R.A. Dalrymple. Modeling water waves in the surf zone with GPU-SPHysics. In *Proc. Fourth Workshop, SPHERIC, ERCOFTAC, Nantes*, 2009.

[39] Alexis Hérault, Giuseppe Bilotta, Ciro Del Negro, Giovanni Russo, and Annamaria Vicari. *SPH modeling of lava flows with GPU implementation*, volume 15 of *World Scientific Series on Nonlinear Science, Series B*, pages 183–188. World Scientific Publishing Company, 2010.

[40] Alexis Hérault, Giuseppe Bilotta, Annamaria Vicari, Eugenio Rustico, and Ciro Del Negro. Numerical simulation of lava flow using a GPU SPH model. *Annals of Geophysics*, 54(5), 2011. accepted.

[41] William G. Hoover. *Smooth Particle Applied Mechanics: The State of the Art*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2006.

[42] Markus Ihmsen, Nadir Akinci, Markus Becker, and Matthias Teschner. A parallel SPH implementation on multi-core CPUs. *Comput. Graph. Forum*, 30(1):99–112, 2011.

[43] K. Ishihara, M. Iguchi, and K. Kamo. Numerical simulation of lava flows on some volcanoes in Japan. In Jonathan H. Fink, editor, *Lava Flows and Domes: Emplacement Mechanisms and Hazard Implications (IAVCEI Proceedings in Volcanology)*, pages 174–207. Springer-Verlag, 1990.

[44] ISO. *ISO/IEC/IEEE 60559:2011 Information technology — Microprocessor Systems — Floating-Point arithmetic*. International Organization for Standardization, Geneva, Switzerland, 2011.

[45] Byunghyun Jang, David Kaeli, Synho Do, and Homer Pien. Multi gpu implementation of iterative tomographic reconstruction algorithms. In *Proceedings of the Sixth IEEE international conference on Symposium on Biomedical Imaging: From Nano to Macro*, ISBI'09, pages 185–188, Piscataway, NJ, USA, 2009. IEEE Press.

[46] W. Kahan. Pracniques: further remarks on reducing truncation errors. *Commun. ACM*, 8(1):40+, January 1965.

[47] Alan H. Karp and Horace P. Flatt. Measuring parallel processor performance. *Commun. ACM*, 33:539–543, May 1990.

[48] R. Keiser, B. Adams, D. Gasser, P. Bazzi, P. Dutre, and M. Gross. A unified lagrangian approach to solid-fluid animation. *Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics*, 0:125–148, 2005.

[49] Andreas Kolb and Nicolas Cuntz. Dynamic particle coupling for GPU-based fluid simulation. In *In Proc. of the 18th Symposium on Simulation Technique*, pages 722–727, 2005.

[50] Jens Krüger and Rüdiger Westermann. Linear algebra operators for GPU implementation of numerical algorithms. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 908–916, New York, NY, USA, 2003. ACM.

[51] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *SIGARCH Comput. Archit. News*, 38:451–460, June 2010.

[52] Benjamin G. Levine, John E. Stone, and Axel Kohlmeyer. Fast analysis of molecular dynamics trajectories with graphics processing units-radial distribution function histogramming. *J. Comput. Phys.*, 230:3556–3569, May 2011.

[53] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. The TOP500 project. http://www.top500.org/lists, November 2011.

[54] Hideaki Miyamoto and Sho Sasaki. Simulating lava flows by an improved cellular automata method. *Computers & Geosciences*, 23(3):283–292, 1997.

[55] J Monaghan and R Gingold. Shock simulation by the particle method SPH. *Journal of Computational Physics*, 52(2):374–389, 1983.

[56] J. J. Monaghan. Smoothed particle hydrodynamics. In *Annual Review of Astronomy and Astrophysics 30*, pages 543–574, 1977.

[57] J. J. Monaghan. Smoothed Particle Hydrodynamics. *Annual Review of Astronomy and Astrophysics*, 30(1):543–574, 1992.

[58] J. J. Monaghan. Simulating free surface flows with SPH. *Journal of Computational Physics*, 110:399–406, 1994.

[59] J. J. Monaghan. Smoothed particle hydrodynamics. *Reports on Progress in Physics*, 68(8):1703, 2005.

[60] Joseph P. Morris, Patrick J. Fox, and Yi Zhu. Modeling Low Reynolds Number Incompressible Flows Using SPH. *Journal of Computational Physics*, 136(1):214 – 226, 1997.

[61] Matthias Müller, Barbara Solenthaler, Richard Keiser, and Markus Gross. Particle-based fluid-fluid interaction. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '05, pages 237–244, New York, NY, USA, 2005. ACM.

[62] John Nickolls and William J. Dally. The GPU computing era. *IEEE Micro*, 30:56–69, 2010.

[63] NVIDIA Corporation. *CUDA C Best Practices Guide 3.2*, September 2010.

[64] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide 4.0*, May 2011.

[65] NVIDIA Corporation and University of California. CUDPP - CUDA data parallel primitives library. http://code.google.com/p/cudpp/, August 2011. (accessed 1 November 2011).

[66] C Obrecht, F Kuznik, B Tourancheau, and Jean-Jacques Roux. Multi-gpu implementation of the lattice boltzmann method. *Computers and Mathematics with Applications*, 2011.

[67] James C. Phillips, John E. Stone, and Klaus Schulten. Adapting a message-driven parallel application to gpu-accelerated clusters. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 8:1–8:9, Piscataway, NJ, USA, 2008. IEEE Press.

[68] Paul Richmond, Dawn Walker, Simon Coakley, and Daniela Romano. High performance cellular level agent-based simulation with flame for the gpu. *Briefings in Bioinformatics*, 11(3):334–347, 2010.

[69] B.D. Rogers and R.A. Dalrymple. Three-dimensional SPH-SPS modeling of wave breaking. In *Symposium on Ocean Wave Measurements and Analysis (ASCE)*, Madrid, 2005.

[70] Eugenio Rustico. Implementazione e valutazione metodo di memorizzazione particelle per cella. Technical report, Dipartimento di Matematica e Informatica, Università di Catania, Viale Andrea Doria n.6, 95125 Catania, October 2009.

[71] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore GPUs. *Parallel and Distributed Processing Symposium, International*, 0:1–10, 2009.

[72] Shubhabrata Sengupta, Mark Harris, and Michael Garland. Efficient parallel scan algorthms for GPUs, December 2008.

[73] B. Solenthaler and R. Pajarola. Density contrast SPH interfaces. In *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '08, pages 211–218, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[74] Barbara Solenthaler, Jürg Schläfli, and Renato Pajarola. A unified particle model for fluid–solid interactions: Research articles. *Comput. Animat. Virtual Worlds*, 18:69–82, February 2007.

[75] Jos Stam and Fiu Eugene. Depicting fire and other gaseous phenomena using diffusion processes. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '95, pages 129–136, New York, NY, USA, 1995. ACM.

[76] Magnus Strengert, Marcelo Magallón, Daniel Weiskopf, Stefan Guthe, and Thomas Ertl. Hierarchical visualization and compression of large volume datasets using gpu clusters. In *In Eurographics Symposium on Parallel Graphics and Visualization (EGPGV04) 2004*, pages 41–48, 2004.

[77] Berkeley University of California. The BSD 3-Clause License. http://www.opensource.org/licenses/BSD-3-Clause, July 1999. (accessed 1 November 2011).

[78] Suresh Venkatasubramanian. The graphics card as a streaming computer. *CoRR*, cs.GR/0310002, 2003.

[79] A. Vicari, A. Ciraudo, C. Del Negro, A. Herault, and L. Fortuna. Lava flow simulations using discharge rates from thermal infrared satellite imagery during the 2006 Etna eruption. *Natural Hazards*, 50(3):539–550, 2009.

[80] A. Vicari, A. Hérault, C. Del Negro, M. Coltelli, M. Marsella, and C. Proietti. Modeling of the 2001 lava flow at Etna volcano by a Celluar Automata approach. *Environmental Modelling & Software*, 22(10):1465–1471, 2007.

[81] Pablo Vidal and Enrique Alba. A multi-gpu implementation of a cellular genetic algorithm. *Computational Intelligence*, pages 18–23, 2010.

[82] Oreste Villa, Long Chen, and Sriram Krishnamoorthy. High performance molecular dynamic simulation on single and multi-gpu systems. In *International Symposium on Circuits and Systems (ISCAS 2010), May 30 - June 2, 2010, Paris, France*, pages 3805–3808. IEEE, 2010.

[83] Yichen Zhou, Shenyi Song, Tingxing Dong, and David A. Yuen. Seismic wave propagation simulation using accelerated support operator rupture dynamics on multi-gpu. *Computational Science and Engineering, IEEE International Conference on*, 0:567–572, 2011.

List of Figures

List of Tables

Listings