Computational Science and Engineering
(International Master's Program)

Technische Universität München

Master's Thesis

Enabling the Adaptive Particle Representation on GPU: integrating a dynamic sparse data structure for the GPU in OpenFPM

Tommaso Bianucci

Computational Science and Engineering
(International Master's Program)

Technische Universität München

Master's Thesis

Enabling the Adaptive Particle Representation on GPU: integrating a dynamic sparse data structure for the GPU in OpenFPM

Author:	Tommaso Bianucci
1st examiner:	Univ.-Prof. Dr. Hans-Joachim Bungartz
2nd examiner:	Univ.-Prof. Dr. Ivo F. Sbalzarini
Assistant advisor:	M.Sc. Pietro Incardona
Submission Date:	September 6th, 2019

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

September 6th, 2019 Tommaso Bianucci

Acknowledgments

First of all I would like to thank Prof. Ivo Sbalzarini and Prof. Hans-Joachim Bungartz for giving me the opportunity to work on this project and for being my advisors and examiners for this thesis.

I would like to thank Pietro Incardona for his amazing guidance and patience throughout the last six months, allowing me to tap into his knowledge and teaching me so much, from C++ template-metaprogramming black-magic and CUDA tricks to the importance of setting up and working with proper processes when developing scientific software. I am also infinitely grateful to him for the countless code-review and bug-hunting sessions: it was great fun working together!

A big thank you also to the entire MOSAIC group: it is a great and fun working environment and, most importantly, they are all great and fun friends.

I would also like to thank Keefe, Laura and Miguel for all the friendship, support and food they generously offered during the last two years. Sharing with them the joys and difficulties of the student life was by far the most worthwile and enriching consequence of moving to Munich.

I would like to thank my family for their support and trust, especially my brother Elia.

And finally, I would need to thank Elisa for way too many reasons, but there is one that summarizes them all: *endless patience*.

"People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on each other, like a wall of mini stones."

-Donald E. Knuth

Abstract

GPUs conveniently bundle high performance parallel computing capabilities into a single device, allowing single workstations to deliver comparable performance to that of a small conventional cluster. However GPUs are designed to work with dense and regular data, while many applications – as the *Adaptive Particle Representation* (APR), which is the main driver of this work – produce data with irregular boundaries and some degree of sparsity. These applications could still benefit from the added performance coming from a GPU if they could use a dynamic sparse data structure directly from the device itself, however there is little research on and no ready-to-use implementation of such a data structure in CUDA a user can rely on.

Here we present the *SparseGridGpu*, a sorted-array-based dynamic dictionary for the GPU which allows handling generic data types, addressing its elements with arbitrary-dimensional indexing and which provides integrated facilities for the parallel execution of elementwise operations. Furthermore, we integrate this data structure into the *OpenFPM* framework for high-performance computing, making it directly available for simulations and data-processing applications.

When tested on locally-clustered sparse data or on dense data with an irregular domain, *SparseGridGpu* was able to insert and search billions of elements per second and to apply a test stencil operation with a two orders of magnitude speedup with respect to a single threaded host-side implementation.

This work provides GPU users with a pre-packaged dictionary which allows for highly parallel inserts and lookups, as well as a simple way to efficiently apply stencils and other elementwise operations on irregular and sparse grids. These features allow to use GPUs to enhance the performance of single workstations for this kind of simulations and data-processing tasks.

Furthermore, this work can be used as a building block for more complex multi-layer sparse data structures, as the APR, which can in turn be used for simple implementations of *Adaptive Mesh Refinement* (AMR) schemes.

Contents

Part I.

Introduction and State of the Art

1. Introduction

1.1. Motivation

Graphics Processing Units (GPUs) are specialized hardware devices which have been developed with the primary purpose of offloading rendering and image processing tasks from the *Central Processing Unit* (CPU). Over time they have evolved from being simple and fixed-function hardware pipelines to complex, massively parallel and highly programmable computing workhorses [1]. With the introduction of CUDA and OpenCL, the general purpose computing capabilities of GPUs have become easily available to programmers and it is now common for scientific computing applications to use GPUs for increased performance.

Today, some GPU models are specifically manufactured to be used in servers and computing nodes[1] and five out of the ten fastest supercomputers in the TOP500 list feature GPU acceleration using NVIDIA devices [2]. Furthermore, GPUs are a simple and cost-effective way to add considerable computational power to workstations, providing the computing capabilities of a small computer cluster at a fraction of the cost [3] and without the burden of managing a distributed environment.

This is particularly important in scientific computing and data processing settings, where scientists often need to execute medium-sized simulations or data analisys tasks on single workstations. Increasing the single node performance is also important in all the cases where a computer is used to control and collect data from scientific equipment: modern microscopes, for example, can generate data at a rate in the order of gigabytes per second which may need or benefit from being processed and analysed at real-time or near-real-time[2].

As detailed in section 1.4, the GPU hardware and programming model has specific characteristic that makes it especially suited to work on dense arrays and to apply operations uniformly on them: GPUs were indeed developed for processing graphics, which involves computing dense linear algebra operations. For this reason, the type of applications which are usually and most successfully accelerated with GPUs involve some vector-vector, matrix-vector, matrix-matrix or dense stencil operations. Looking at popular CUDA introductory

[1] As the NVIDIA Tesla V100.
[2] This topic is further discussed in section 1.5.

tutorials or books, such as *CUDA by Example* [4], we can find that all the main examples involve some simple vector sum or dot product or matrix-vector multiplication, because these problems map naturally to the CUDA programming model and the reader can easily discover the massive arithmetic performance a GPU can attain.

However, while dense problems are easy to solve on GPUs, not every problem in computing is naturally based on dense data, or it even becomes intractable if represented in a dense way. The main issue with dense data is that the memory footprint is fixed by the dimensions of the array and is independent from its actual content of information. In case of a naturally sparse problem, using a dense data representation leads to an inefficient use of storage and memory space, and it also hampers computational performance because of the need to access and process useless data.

Within the class of sparse problems, an important subclass comes from sparse linear algebra. How to best support the sparsity in these problems is a long-standing field of research and various sparse vector and matrix representation formats have been developed to allow it: for example the NVIDIA-developed library for sparse linear algebra *cuSPARSE* supports, among others, the *Coordinate* (COO), *Compressed Sparse Rows* (CSR) and *Blocked Compressed Sparse Rows* (BSR) formats [5]. These formats are however inherently *static*, as the size and sparsity pattern has to be known at construction time, and they are often used to store matrices that do not change over the time they are used on the GPU. The typical use case sees the matrix being assembled on the host CPU and main memory and then transferred to the GPU device, where it is used to perform a number of linear algebra operations before transferring the results back.

There exist, however, other problems that require a data structure to be both sparse and *dynamic*, meaning that it should allow its size, content and sparsity pattern to be changed directly by code running on the GPU itself. An example of such a problem comes from the implementation of the *Adaptive Particle Representation* (APR, see section 1.5), which is our reference application for this work.

These problems require a GPU implementation of an *associative array*, also commonly called *map* or *dictionary*, which allows for parallel *get* and *insert* operations, as well as the possibility to efficiently apply elementwise or *stencil-like* operations to all stored elements. While for traditional sequential CPU computing it is well established, this field of work is less explored on GPUs and there is little support for this kind of data structure in publicly available libraries [6].

1.2. State of the art

Associative arrays are not a novelty: they have been studied and implemented since the dawn of computing. There exist countless designs to suit different performance needs and,

for CPU, countless implementations: in most programming languages the most common types of associative arrays are available as part of the built-in or standard libraries. When it comes to GPU programming, however, associative arrays become more elusive: CUDA has no built-in associative array implementation, there is very limited support from third-party libraries and most of the theoretical works on this kind of data structures for GPU have only produced proof-of-concept code. Therefore, general purpose data structures on GPUs are usually built upon one of the following types: *arrays*, *sorted arrays* and *hash tables* [6]. There have been also attempts to use more complex data structures on the GPU, however these were commonly built on the CPU, then transferred onto the GPU and used statically as read-only objects [7–10].

One important work that focused on efficiently constructing a data structure on the GPU comes from *cuckoo hashing* [11]: this hash table features parallel bulk build and search operations, while not supporting table resizing at runtime nor element deletion. Another work focused on a method for efficiently constructing, on GPU, *bounding volume hierarchy* (BVH) trees and *octrees* [12].

In 2018 *Ashkiani et al.* proposed *GPU LSM* dictionary data structure for GPU, which is based on the structure of a *Log-Structured Merge* (LSM) tree, but uses sorted arrays at each of its levels, as in *Cache-oblivious Lookahead Arrays* (COLA) [6]. This data structure supports batch *insert*, *delete*, *lookup*, *count* and *range* operations, allows for dynamic resizing and shows especially good performance in insertion/deletion.

In 2019 *Awad, et al.* presented an high-performance *GPU B-Tree* implementation, comparing it with the GPU LSM: they show it can achieve the theoretically expected better performance in lookups but also a better insertion performance for small to medium batch sizes[3] [13].

While both the GPU LSM and GPU B-Tree data structure show very good performance[4], they were implemented only as a proof-of-concept stand-alone code and they make strong assumptions about the data they will contain[5] [6, 13]. This means that they are currently not available to users in the form of libraries.

A different approach has been taken by NVIDIA with their *GVDB Voxels* [14] library, first released in 2017. This library focuses on the management of sparse 2D and 3D voxel data for rendering, 3D-printing and computing, and stores the data in a *block-sparse* fashion, i.e. by storing sparse *bricks* of data into sorted arrays and operating with a block of threads on each brick of data [14]. As we will see in chapter 2, this is similar to the approach we have taken in our work, as we also chose to develop a block-based data structure. However GVDB Voxels has a design that does not fit with the requirements of our target application

[3] Up to approximately 100k.
[4] Performance in terms of rates of key insertions and deletions per unit of time.
[5] As e.g. accepting only 32-bit values.

(see 1.5 and 2): it is not a generic dictionary and it is intrinsically associated with the concept of space, it does not allow for dynamic insertions of new elements from within kernels and has a different way of managing ghost layers which implies a relevant memory overhead. Furtermore it does not meet the general design and flexibility offered by OpenFPM, the framework we are adding our dictionary feature to. For these reasons we set out to design and implement our own flavour of an associative array on GPUs.

In the following section (1.3) we briefly describe the OpenFPM framework and its main characteristics, then in section 1.4 we provide a primer on the CUDA architecture and programming model and how they affect algorithm design and implementation choices. Then in section 1.5 we describe the application driving the development of this work and in section 1.6 we finally state the aim of my Master's Thesis work and of the following chapters.

1.3. OpenFPM

OpenFPM[6] is 'an open-source C++ framework for parallel particles-only and hybrid particle-mesh simulations' [15]. The philosophy behind its design is to allow easier and faster development of high-performance codes for particle-based simulations or, more in general, particle-based algorithms. It therefore qualifies as a *domain-specific abstraction* framework.

OpenFPM provides facilities for domain decomposition, ghost-layer management in distributed settings, dynamic load balancing, GPU-accelerated computing and even for writing data structures to VTK files, freeing the application developer from the complexity of dealing with the lower-level aspects of parallelism and high-performance computing and allowing them to fully focus on the logic of the simulation or algorithm at hand.

OpenFPM allows a very general definition of particles and meshes as 'an arbitrary-dimensional space that carries an arbitrary number of arbitrary data structures' [15], therefore allowing the implementation of general algorithms that fall under the particle formalism. OpenFPM also allows greater flexibility with respect to the hardware an application needs to run on, since all OpenFPM distributed data structures are based on local[7] data structures whose parameters and memory layout can be chosen and tuned at compile time [15].

1.3.1. C++ Template Meta-Programming

One of the main technical innovations of OpenFPM lies in the pervasive use of the *generic programming* capabilities offered by C++ templates. On the one hand, leveraging templates

[6] OpenFPM and its documentation are available at `http://openfpm.mpi-cbg.de`

[7] Defined as residing on a shared memory environment.

allows OpenFPM to achieve greater generality, e.g. with respect to dimensionality and number and type of properties that can be stored in the internal data structures. On the other hand, templates also allow to shift parameter-based choices and optimizations to compile-time, reducing the need of indirections at run-time[8] and allowing a greater degree of optimization by the compiler.

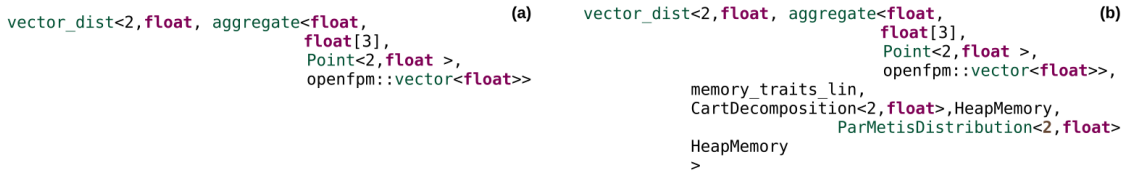


Figure 1.1.: 'Example of a parametric data structure in OpenFPM for a particle set stored as a distributed vector. This includes mandatory parameters (a), such as the space dimension and the data types of the particle properties, but may also include optional parameters (b), such as the internal memory layout or the type of domain decomposition used for this data structure. This renders all data structures generic and independent of their actual implementation, with optimized code automatically generated by the compiler'.
Figure and caption directly taken from [15].

In particular, data structures in OpenFPM are parametric (Figure 1.1), allowing them to target 'different space dimensionality, data types and hardware platforms' [15]. This allows for example for compile-time optimized choice of memory layout according to the hardware being targeted and again to avoid the cost, in terms of performance, of run-time polymorphism. These optimizations have default parameters that fit the most common use cases, however they can be fine-tuned by means of optional template parameters, as it is shown in Figure 1.1 (b).

[8] As e.g. for the resolution of virtual functions.

1.4. GPU computing and CUDA

Graphics Processing Units are devices that allow for fast and massively parallel computing and memory operations. They can yield amazing performance and be more cost- and energy-efficient than other setups of similar performance. However, achieving this performance requires an understanding of the specificity of the GPU hardware and programming model.

As with any other computing device, the problem and data at hand need to be translated into a form that suits the hardware's structure and way of working.

The purpose of this section is to provide the reader with the basics of

- how a GPU works,
- how it can be programmed and
- general strategies for obtaining good performance on it.

As the field of general-purpose GPU computing is dominated by NVIDIA, CUDA is its de-facto standard programming language. This work is therefore focused on CUDA and we have written our software in CUDA/C++.

1.4.1. Hardware characteristics

Multithreaded architecture A modern GPU is a SIMD[9] device. Since NVIDIA's architecture has slightly different assumptions and behaviour than the traditional SIMD model, NVIDIA refers to it as *Single Instruction - Multiple Thread* (SIMT). In this work we will use the terms SIMD and SIMT both to refer to the NVIDIA SIMT model.

At high level, NVIDIA devices are based on a certain number of *Streaming Multiprocessors* (SM), each containing several dedicated INT32, FP32 and FP64 cores. Most recent microarchitectures – *Volta* and *Turing* – also feature *Tensor Cores*, which are capable of delivering up to 64 FMA[10] operations per clock on FP16 data [16, 17].

In the Volta architecture, which we use as a reference from now on, each SM (see Figure 1.2) is partitioned into four processing blocks, each containing 16 FP32, 8 FP64, 16 INT32 and 2 Tensor cores. Each processing block forms one SIMD unit, since the cores in it share one warp scheduler, dispatch unit and register file [16].

Computation is organized in groups of 32 threads, called *warps*, which execute one common instruction at a time [18]. For analysis purposes, threads within a warp can be assumed to be executed in lockstep.

[9] Single Instruction - Multiple Data.
[10] Fused Multiply-Add.

Figure 1.2.: Volta Streaming Multiprocessor diagram. The SM is composed of four processing blocks, each containing 16 FP32, 8 FP64, 16 INT32 and 2 Tensor cores, as well as one warp scheduler, one dispatch unit and one register file. Each processing block forms one SIMD unit.
Figure directly taken from [16] - © NVIDIA Corporation.

When different threads within a warp need to execute different instructions due to a data-dependent conditional branch, then each branch is serially executed on the respective group of threads (which are called *active* threads) and threads which are not involved in the current branch are temporarily disabled (*inactive* threads). This phenomenon is called *branch divergence* (see Figure 1.3). When the instructions belonging to the conditional branches are executed, all threads in the warp are *reconverged* and normal parallel execution takes place again. Branch divergence can be a major cause for performance degradation: algorithms for the GPU should be designed in a way that minimizes it and ensures that the same instruction is uniformly executed in blocks of 32 threads.

The Volta architecture altered the branch divergence and reconvergence behaviour by introducing the *Independent Thread Scheduling*: to allow for more flexible data dependencies,

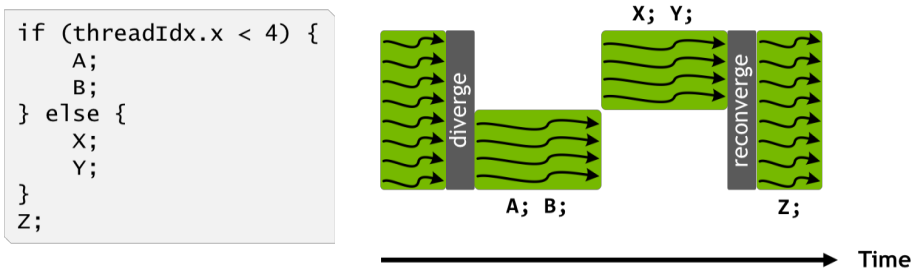


Figure 1.3.: 'Thread scheduling under the SIMT warp execution model of Pascal and earlier NVIDIA GPUs. Capital letters represent statements in the program pseudocode. Divergent branches within a warp are serialized so that all statements in one side of the branch are executed together to completion before any statements in the other side are executed. After the else statement, the threads of the warp will typically reconverge'.
Figure and caption directly taken from [16] - © NVIDIA Corporation.

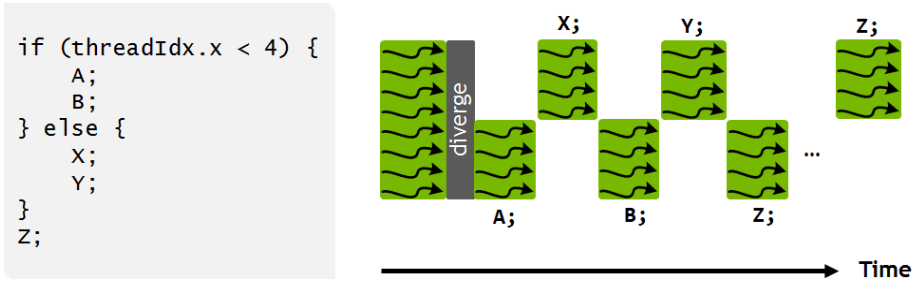


Figure 1.4.: 'Volta independent thread scheduling enables interleaved execution of statements from divergent branches. This enables execution of fine-grain parallel algorithms where threads within a warp may synchronize and communicate'.
Figure and caption directly taken from [16] - © NVIDIA Corporation.

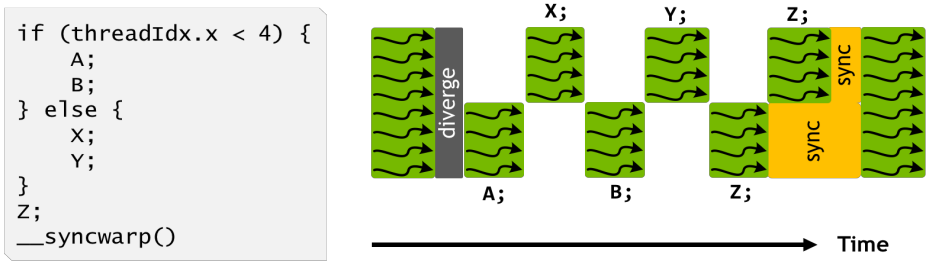


Figure 1.5.: 'Programs use Explicit Synchronization to Reconverge Threads in a Warp'.
Figure and caption directly taken from [16] - © NVIDIA Corporation.

reconvergence is not enforced anymore at the end of a conditional branch (see Figure 1.4) and the programmer must explicitly call the `__syncwarp()` function to request for synchronization and reconvergence of all the threads of a warp (see Figure 1.5) [16].

Memory In the Volta architecture, memory is structured in the following layers [16, 19]:

Global Memory serving all SMs:

- DRAM, physically provided by on-die 3D-stacked HBM2 memory.
- L2 Cache.

SM Memory located on each single SM:

- L1 Instruction Cache.
- A 128KB high-bandwidth memory bank serving both the L1 Data Cache and the Shared Memory.

Processing Block Memory located on each Processing Block:

- L0 Instruction Cache.
- Register File, physically provided by 16,384 32-bit registers.

1.4.2. Programming model

The CUDA programming model is based on the concept that the program is started and controlled by the *host* and some of its tasks can be offloaded to a 'physically separate *device* that operates as a coprocessor' [18]. As such, CUDA offers an interface to heterogeneous computing involving a *host* (CPU+MEM) and a *device* (GPU).

Kernels and thread hierarchy In CUDA, functions can be designated to be called from the host and run on the device. These functions, called *kernels* and tagged by the `__global__` declaration specifier, 'are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions' [18].

The threads executing a kernel are grouped into blocks of regular size, with the programmer specifying the number of blocks taking part in the computation, the *grid size*, and the number of threads composing each block, the *block size*. Both the block and grid sizes can be specified as 1D, 2D or 3D size vectors, allowing for thread blocks to spatially correspond to blocks of the data that has to be processed.

Each thread executing a kernel is given a unique *thread ID* within the block and each block gets a unique *block ID*, both accessible from within a kernel. In this way it is possible to have each thread execute different instructions or access different data depending on its ID.

Memory hierarchy Memory is conceptually organized into a hierarchy composed of three layers [18]:

Global memory This memory space is shared across the whole device, meaning that any thread can access any address in global memory. It is physically stored on the device's DRAM, aided by the L2 and L1 data caches and is accessed through 32-, 64- or 128-byte transactions [18].

Shared memory This memory space is shared among a block of threads and it is stored in on-chip memory banks, ensuring higher bandwidth and lower latency than global memory. This memory is not persistent across kernel calls, therefore data must be explicitly loaded from or stored to global memory for persistence. From a programmer's standpoint, shared memory is like a manually managed data cache.

Local memory This memory space is local to a single thread, i.e. it is private to each thread and it cannot be used to share values across threads. Variables in local memory are usually stored on the register file unless when they do not fit into the available registers[11] or in case of arrays which are not indexed by constant quantities [18]: in these cases it is physically stored in the same way as global memory, potentially causing a performance penalty.

1.4.3. Attaining performance: basic techniques

In light of the characteristics of GPUs mentioned above, we can list some basic and commonly employed patterns and techniques that allow increasing the performance of a GPU-accelerated application. They can be basically divided into two categories: *memory-access-related* and *parallelism-related*.

Coalesced memory access As mentioned earlier, access to memory is structured into *transactions* which read 32-, 64- or 128-bytes at once, corresponding to 8, 16 or 32 floats. This means that if each thread accesses one float without any specific access pattern, each thread will issue one memory transaction and waste at least $7/8$ of the memory bandwidth. The best access pattern is instead to have neighbouring threads[12] accessing neighbouring data from memory, without striding. This allows the GPU to pack – or *coalesce* – memory accesses from several neighbouring threads into the same memory transaction, ensuring better usage of the memory bandwidth and reducing the latency. In case float-typed data is accessed, coalesced memory access can allow all threads of a warp to load data from memory through one single transaction.

[11] Using more local memory than the available registers is called *register spilling*.
[12] In case of 2D or 3D indexing, threads are neighbours if they are adjacent in the x index.

Use of shared memory If the same data elements have to be accessed repeatedly by different threads within a block, then relying on L1 cache may not be the best choice, performance-wise, especially on architectures older than Volta[13]. In this scenario it is good practice to first load the data chunk for the entire thread block into shared memory – by taking advantage of coalescing – and then repeatedly access data, within each thread, from the shared memory.

The same pattern can be used for writing to global memory, e.g. in a scenario where multiple writes from different threads need to be reduced or where the original writing pattern would not be favorable for coalescing.

Minimize thread divergence We have seen that if different groups of threads within a warp take different conditional branches, then the execution of these branches takes place sequentially on the GPU. This can lead to the loss of parallelism and high performance costs. Care should therefore be taken to ensure that all threads within a warp can execute the same branches, avoiding divergence.

Minimize idling threads If some threads return from a kernel earlier than others belonging to the same warp, they remain idling and not available for other computations until the whole warp has returned from the kernel. Minimizing the possibility for threads to become idle allows avoiding this performance degradation.

1.5. Adaptive Particle Representation

The *Adaptive Particle Representation* (APR) is an image representation format developed by *Cheeseman et al.* [20] with the main purpose of storing and processing fluorescence microscopy images.

Modern microscopy imaging techniques, together with advancements in fluorescent labelling and genetics, allow recording biological structures and events with high resolution, both in space and time. In particular, advanced in-vivo live-imaging techniques allow the investigation of complex spatio-temporal biological processes, in 3D, over long periods of time [20].

However, while unlocking the potential for important discoveries, these techniques produce huge amounts of raw data, potentially gigabytes per second and terabytes per day [20]. Storing and processing this amount of data is slow and technically challenging on typical workstations, even for simple and common image manipulation tasks, often

[13]The Volta architecture has both L1 and Shared memory physically hosted on the same on-chip banks, ensuring the same bandwidth and latency for both of them and allowing programs that do not use shared memory to allocate the entire bank for L1 usage [16].

requiring the user to rely on projections as a means to reduce dimensionality and thus the memory footprint.

Furthermore, this huge quantity of raw data does not typically correspond to the quantity of information contained by the image: all pixels that fall in an area of the image with uniform intensity, as e.g. the background, will not carry their information in an efficient way and cause a waste of memory.

In light of these needs, the APR has been developed with the goal of providing not only a means to compress the data for storage, but also to allow data processing to take place directly in the APR format, without the need of going back to pixel [20].

1.5.1. The concept

The main idea behind the APR is to adaptively sample the image with a spatially varying resolution which depends on the image content. Intuitively we would like the resolution to be high in the areas where the content varies rapidly with respect to the spatial dimensions and to be low where the content is more uniform. This is exactly what the APR delivers and it does this while guaranteeing a desired error bound with respect to the original image [20].

The APR is designed to fulfill the following four *representation criteria* [20]:

1. It must guarantee a user-controllable representation accuracy for noise-free images and must not reduce the signal-to-noise ratio of noisy images.

2. Memory and computational cost of the representation must be proportional to the information content of an image and not to its number of pixels.

3. It must be possible to rapidly convert a given pixel image into that representation with a computational cost at most proportional to the number of input pixels.

4. The representation must reduce both computational cost and memory cost of image-processing tasks with a minimum of algorithmic changes and without requiring use of the full original pixel representation.

This is implemented through a quad- or oct-tree-based representation, where only the leaves are stored.

For what concerns our work, it is sufficient to keep in mind that the APR can be seen as a collection of layers, each representing the spatial domain with increasing resolution and each populated in a sparse fashion.

1.6. Aim of this work

The aim of this work is to extend the OpenFPM library with a data structure supporting the implementation of the APR and its algorithms on the GPU. This data structure needs to transparently support both scalar and vector values and to allow for massively parallel lookups and inserts. On top of this, we also want it to support the addressing of its elements in a general number of dimensions and to provide an easy-to-use interface for stencil application.

The main use cases we want to address with this data structure are dynamic storage of sparse data and the possibility to efficiently apply elementwise or stencil operations on this sparse data.

In this work we discuss the design choices we made to allow this data structure to suit both the needs of our problem and the characteristics of GPUs (see chapter 2), then we illustrate the implementation of it in two main classes (see chapters 3 and 4) and finally we measure its performance on a set of synthetic tasks (see chapter 5) reproducing parts of typical application scenarios.

Part II.

A general purpose block-sparse data structure for the GPU

2. Analysis and General Design

The core requirement of this work is to extend the OpenFPM library in a way that allows supporting the APR and its main operations on NVIDIA GPUs.

The natural first step of my thesis work was therefore to analyse the APR in terms of computational tools and *algorithmic building blocks* that are necessary for its succesful implementation on a GPU. The steps of this analysis are reported in the following sections.

2.1. From the APR to an actual data structure

A d-dimensional APR with n levels can be seen as a collection of n sparse d-dimensional grids, each carrying the information of a single level. This means that managing an APR on GPU requires the implementation of a data structure that is able to efficiently represent a sparse set of points in a d-dimensional space. Furthermore, the pulling scheme algorithm proceeds by inserting points at arbitrary levels, meaning that this data structure should also be dynamic, allowing at least insertions to take place, in parallel, on the GPU. We therefore need an *associative array*[1] implementation for the GPU.

The main challenges of implementing a dynamic and sparse data structure on the GPU comes from the specificity of GPU architecture and memory layout. As discussed in section 1.4, GPUs are SIMD devices where thread divergence is possible at the cost of incurring in a performance penalty. Also, memory accesses need to be organized in a way that is aware of both the physical memory layout of the device and the CUDA memory hierarchy.

Due to these specificities, algorithms for GPUs should ideally have the following properties:

1. access data in contiguous chunks, with neighbouring threads in a warp accessing neighbouring data in memory,
2. minimize global memory access by temporarily storing in shared memory the data that needs to be accessed several times during the lifetime of a kernel and
3. minimize thread divergence by arranging data as to allow groups of 32 threads (i.e. whole warps) to execute the same stream of instructions.

[1] Associative arrays are also commonly called *maps* or *dictionaries*.

Property (1) is generally referred to as *coalesced memory access* and allows minimizing the number of transactions with global memory per byte of data, since a single memory transaction will carry data for several threads[2]. Property (2) ensures better performance by exchanging redundant accesses to (slow) global memory with accesses to (fast) shared memory. Property (3) maximises the degree of parallelism of the algorithm but also requires data to be organized in a dense fashion.

It is clear from these properties how GPUs are specifically tailored to process dense arrays of data and this constitutes our main challenge: building a sparse data structure while retaining a good performance on a GPU.

2.1.1. The necessary primitive operations

Each level of the APR is a sparse spatial domain which needs to allow for:

- insertion of new elements,
- access to existing elements (r/w),
- application of stencil operations (as e.g. convolutions) and
- application of element-wise operations which may access/insert on neighbouring positions[3].

Due to the nature of the APR and its main use as an image representation format, we can expect:

- Accesses to existing elements to be more frequent than insertions of new elements.
- Stencil operations [4] to be the main way to perform changes to the values of the APR.
- Neighbour-inserting stencil operations to be mainly used when initially building the APR (pulling scheme).

It is therefore important for us to choose a data structure ensuring first of all fast access to existing elements over insertion performance. Then it should provide an efficient way to perform operations over all the existing elements and their spatial neighbours while minimizing the number of element searches.

[2] This is the same reasoning as behind the need for spatial data locality when dealing with cache-lines in CPUs.

[3] This is required by the pulling scheme.

[4] Here we use the term *stencil operation* in a broader sense, meaning any elementwise operation which may access neighbouring elements and is to be applied to all elements.

2.2. A map based on a sorted array

As mentioned previously, associative arrays on GPU are mainly based on arrays, sorted arrays and hash tables. We can quickly exclude unsorted arrays, as they provide $\mathcal{O}(1)$ time inserts but $\mathcal{O}(n)$ time searches. Hash tables can also be excluded, as the only fully developed implementation for GPU is based on *cuckoo hashing* [11], which does not allow for the table to be resized [6, 11]. This leaves either plain sorted arrays or the derived GPU LSM [6] as candidates for building our map upon[5].

Sorted arrays ensure $\mathcal{O}(\log n)$ search and require a slow $\mathcal{O}(n)$ operations for single element inserts (worst case). However, since we are working on an highly parallel setting, we can assume insert operations to be batched: the buffer of the new m elements can sorted and merged into the the existing sorted array at the cost of $\mathcal{O}(n + m \cdot (1 + \log m))$. GPU LSM provides a faster insertion behaviour than sorted arrays, however the search is more expensive ($\mathcal{O}(\log^2 n)$) and experimental results confirm that sorted arrays provide better search performance in practice on GPU devices [6].

Since we expect our client applications to rely on searches more often than on inserts, we decided to proceed with a sorted-array-based implementation. Our code is however designed in a way that will allow swapping the underlying map implementation and use, e.g., GPU LSM or GPU B-TREE in the future, if required.

2.2.1. A block-sparse map

One of our main design decisions has been to build a *block-sparse* data structure, meaning that the data is actually stored in small dense equilateral blocks which in turn are sparsely stored into an associative array. The main reason for this choice is that we can then choose a block size of an integral multiple of 32 elements, which in turn ensures that:

- neighbouring threads within a warp process data which can be accessed with a single memory transaction and
- all threads within a warp perform the same binary search on the underlying map, leading to no thread divergence during lookups and insertions.

This block-sparsity allows the data structure to be more suitable to the characteristics of GPUs and reduces the number of lookups required on the underlying map. On the other hand we pay these advantages with an higher memory usage and, in the worst case, having to store one entire block for inserting just one element. However we argue that, given the characteristics of the APR and the data it is meant to store, we can assume the levels to

[5] GPU B-TREE [13] would also be an appealing candidate, but we were not aware of its development when this design phase took place.

be generally *locally dense*, that is, we can assume the sparsity pattern of the APR levels to display clustering and therefore to lead to an average high occupancy of the blocks.

3. BlockMapGpu

In this chapter we describe how we built a univariate associative array that is able to transparently handle blocks of scalar-, vector- and matrix-valued elements. This structure, *BlockMapGpu*, is the first of the two units that consitute this work, with the second one being its specialization *SparseGridGpu*, which we cover in chapter 4.

We first describe the already existing data structure prototype we are using as foundation (3.1), listing the necessary extensions we implemented on it to support our requirements, and then we describe the actual *BlockMapGpu* we built on top of it (3.2).

3.1. Sparse Vector

BlockMapGpu is based on the `vector_sparse_gpu` class, a sorted-array-based map for the GPU developed by Pietro Incardona, main developer and maintainer of OpenFPM and my direct supervisor in this work. Sparse Vector allows for highly parallel retrieval, insertion and removal operations, therefore providing a general purpose associative data structure for the GPU which can be accessed and managed both from host and device code.

Conceptually, Sparse Vector is built upon two arrays, one containing the (sorted) indices and one containing the respective data, and provides two main algorithms: *get* and *insert*.

get The *get* operation allows the retrieval of an existing element of the data structure, for both read and write purpose. This operation is based on a binary search on the array of indices and then on returning the reference to the corresponding element. In case the element does not exist, a *background value*, i.e. a default value, is returned. An arbitrary number of threads can perform this operation in parallel and data consistency on write must be ensured by the client code.

insert The *insert* operation allows inserting elements at both new and existing indices. It is performed as a batch operation and consists of three phases: first an insertion buffer must be allocated on the device, then the device code performing the insertions is executed and finally a *flush* operation is performed, merging the insertion buffer with the actual data structure. It is allowed to insert the same element several times during the same insertion round. In order to manage data consistency in this case, as well as

when inserting an element which is already present, the *flush* operation can perform a reduction and store its result. A set of common reduction operators is already defined and available, but user-defined reductions can also be used.

Sparse vector relies on the *Modern GPU* library [21] for GPU-efficient parallel implementations of basic general purpose algorithms, such as *scan*, *sort* and *merge*.

In its original version, the sparse vector implementation accepted only scalars or small vectors as elements: this limitation comes from the way the underlying Modern GPU library relies on shared memory for moving the values around, since it is designed for holding scalar values. Using bigger data structures as values of for the sparse vector quickly led to shared memory exhaustion and failures at compile time.

To overcome this limitation, we[1] have changed substantial parts of the flush workflow so that Modern GPU's algorithms can be called independently of the values and that the corresponding operations can then be mirrored to the values buffer in a transparent way. Further details on the insertion workflow are in 3.1.2 and 3.1.3.

3.1.1. Storing several properties in a single map

An important feature of OpenFPM is to allow storing an arbitrary number of *properties* for each particle or spatial element of a grid. This allows the implementation of general particle-based applications. This also holds for the sparse vector: if the map is defined as to allow the storage of a certain number of properties, then each element is associated with the same number of values. Logically this is exposed to the user as each element having an *aggregate* of the properties as value. This is then translated under the hood into having several arrays of values on the GPU, one for each property, where at the same position one can find the various properties of a certain element.

This design has two important beneficial consequences: first, it allows to access any property of a given element without having to perform several binary searches. Second, it provides a convenient way to store meta-data for each element, which can be done simply by adding and accessing a special property for this purpose (see 3.2.1).

Since our sparse vector is implemented using sorted arrays, one for the indices and one for the corresponding data, we can leverage the support that OpenFPM vectors have for aggregate data types.

[1] This was a joint programming effort with equal contribution from Pietro Incardona and me.

3.1.2. Insert implementation

The insertion workflow is composed of three phases:

allocate allocation of an insertion buffer,

insert actual insertions taking place in parallel on the GPU and

flush the insertion buffer is processed and merged into the data structure.

The insertion buffer is composed of m slots of allocated space for a certain amount, say n, of elements where each slot is assigned to one thread block that will perform the inserts. The inserts are then performed on the GPU, by calling the *insert* method as part of a user-defined CUDA kernel. Here each thread block adds the inserted data to its respective segment of the insertion buffer. Finally, the user calls the *flush* method, which takes care of:

1. sorting the insertion buffer by index,
2. reducing the values inserted for the same index according to the user-specified reduction operator and
3. merging the result together with the existing data, again taking care of reducing duplicate entries.

Each of the above steps uses several parallel algorithms – like e.g. *scan*, *merge*, *mergesort*, *segreduce* – as building blocks to achieve the desired result, with good performance on the GPU. The *segreduce* algorithm is especially important for us as we needed to re-implement it in order to suit our needs, as detailed in 3.1.3.

3.1.3. Segmented reduction

The segmented reduction, or *segreduce*, operation allows reducing subsets of a given data array according to predefined *segments*. That is, for each segment – i.e. a range of adjacent elements – of the data array, all elements belonging to the segment are collapsed and reduced together with the given reduction operator. Then, the reduced results for all segments are compacted into a shorter array which is eventually returned.

Segreduce is a commonly used parallel computing primitive on the GPU and an optimized implementation of it is available in the Modern GPU library [21]. OpenFPM uses the segreduce function from Modern GPU in its sparse vector implementation, which is designed to handle scalar elements, however this breaks as soon as blocks of elements or vector-valued elements are introduced. The reason lies in the fact that Modern GPU, assuming a typical use case where elements are float- or int-typed, heavily relies on the use of shared memory: as soon as elements start having a bigger memory footprint, the

available shared memory cannot contain the required data anymore, leading to compilation errors. Furthermore Modern GPU is not aware of our block-based design, meaning that it would let a single thread process one entire block of data: this would prevent any coalesced memory access to take place, heavily impairing performance.

This forced us to implement our own *block-friendly* version of segreduce. Along with it, we have also implemented a new algorithm for determining, in parallel and on the GPU, the segments of elements with the same keys in a sorted array of keys.

3.1.4. Generic handling of scalar, vector and matrix element values

One of the most challenging requirement of our design is the need to store blocks of either scalars, vectors or even small matrices transparently, parallelizing the operations accordingly on several threads without the need for any special handling from the user side.

Following the general design of the OpenFPM library, we can achieve this level of generality by leveraging C++ templates. The type and dimensionality of the values, as well as the size of the data blocks, are known at compile time and can therefore be used as template parameter. Furthermore, templates allow us to define compile-time switches for choosing different implementations of our algorithms that best suit the parameters at hand.

A simple example of how templates allow us to achieve generality and proper parallelism is our implementation of a general assignment operator which can be used inside our basic algorithms to transparently handle 0-, 1- and 2-dimensional values. In Source Code 3.1 we can see how we implemented the `assignWithOffset` generic assignment operator inside the the `generalDimensionFunctor`, a functor that groups several generic operators. This assignment operator handles blocks of n-dimensional values and recursively ensures that each thread goes through the entire n-dimensional tensor corresponding to a given offset within the data block.

We used the same approach also for applying binary operators to single elements of a data block, which is crucial for the reduction phase of the insertion workflow.

3.2. BlockMapGpu

Now that we have extended the Sparse Vector to have the capability to store blocks of data, we can proceed building the *BlockMapGpu* data structure and interface on top of it.

The block map is an associative array allowing 1D or *linear* indexing of its values. It allows addressing single elements via their index, while under the hood it stores these values in a block-sparse fashion by grouping them into *data blocks* or *chunks* and storing these chunks into a sparse vector. The user provides a linear ID for the elements they want

```
1  template <typename T1>
2  struct generalDimensionFunctor
3  {
4      ...
5      template <typename T1b, typename T2, typename T3>
6      __device__ __host__ inline static void assignWithOffset(
7              T1b &dst, const T2 &src, T3 offset)
8      {
9          dst[offset] = src[offset];
10     }
11     ...
12 };
13
14 template <typename T1, unsigned int N1>
15 struct generalDimensionFunctor<T1[N1]>
16 {
17     ...
18     template <typename T1b, typename T2, typename T3>
19     __device__ __host__ inline static void assignWithOffset(
20             T1b dst, const T2 &src, T3 offset)
21     {
22         for (int i = 0; i < N1; ++i)
23         {
24             generalDimensionFunctor<T1>::assignWithOffset(
25                     dst[i], src[i], offset);
26         }
27     }
28     ...
29 };
```

Source Code 3.1.: The implementation of the `assignWithOffset` operator inside the `generalDimensionFunctor`. Lines 1-12 show the implementation for scalar elements, while lines 14-29 show the template specialization for a `N1`-long array type, which recursively relies on the same templated operator, allowing to transparently handle arbitrary-dimensional elements. The `offset` argument points to a specific element within the block and is computed from the thread-id in order to ensure coalesced memory access. This use of templates allows for both the loop and the recursion to be resolved and inlined at compile time, therefore minimizing any performance overhead related to loop execution and function call.

to address and simple integer division and modulo operations allow addressing the right data block and element within the block.

The block map also keeps track of the actually existing elements of each block by storing a 1 Byte bit-mask along with each element as an additional property.

The interface provides elementwise *get* and *insert* method, as well as blockwise versions which can be used to load an existing or new data block into shared memory and allow all threads of a block to coalesce their changes through one single search or insert operation[2].

The interface allows also to access both the index and data vectors directly, which is useful for operations that have to be applied to all existing elements, removing the need to perform searches.

3.2.1. Data-type transformations

DataBlock The first and most important template parameter of BlockMapGpu is the aggregate of data types to be used for the various properties the map should store. As the map contains blocks of data, we want the aggregate to contain types that represent blocks of the basic data types we use for each element. To achieve this, we define the *DataBlock* struct: it provides a simple way to wrap a fixed number of elements of a specific basic type and allows indexing the element with the `[]` operator.

Wrapping an array of elements with the *DataBlock* struct has several benefit over using a plain array. First of all it allows adding meta-data, as the size of the block or the base type of the elements, which can then be easily used in metaprogramming. Then it allows the programmer to easily to easily distinguish the data block from other arrays. Finally it hides the array-nature of the block from the compiler, preventing blocks to be matched by array-typed template parameters, which could lead OpenFPM's metaprogramming design to perform unwanted transformations.

A simplified version of the DataBlock type wrapper can be see in the Source Code 3.2: it allows the definition of a fixed-size array of data, exposing the basic type and size through the `scalarType` and `size` members respectively and providing indexed access via the overloading of `operator[]`.

Source Code 3.3 then shows a sample instantiation of `BlockMapGpu` and how `DataBlock` can be used.

Metadata property As already mentioned, the BlockMapGpu needs to also store, for each data block, which elements of the block actually exist and what do not. This is achieved by adding an additional property to the data structure that stores 1 Byte per element: the

[2]It is possible to call *insert* from each thread of a block, however this causes each thread to use an entire data chunk from the insertion buffer to only write one element.

```
template<typename ScalarT, unsigned int DataBlockSize=64>
struct DataBlock
{
    typedef ScalarT scalarType;
    static const unsigned int size = DataBlockSize;
    ScalarT block[size];

    __device__ __host__ inline ScalarT &operator[](unsigned int i)
    {
        return block[i];
    }
};
```

Source Code 3.2.: Simplified version of the `DataBlock` type wrapper. It allows storing the data of the block as a plain array in memory, while preventing it to be caught by all the templated transformations which are designed to match array types. Overloading `operator[]` allows the transparent addressing of the elements within the block.

```
BlockMapGpu<
        aggregate<
                DataBlock<float, 64>,
                DataBlock<float[3], 64>
                >
        > blockMap;
```

Source Code 3.3.: Sample instantiation of `BlockMapGpu`. This instance stores two properties, being a block of `float` and a block of `float[3]` respectively, both blocks containing 64 elements of the specified type. These could, e.g., represent pressure values and 3D velocities for each point of a given simulation domain.

```
template<typename newT, typename T>
struct AggregateAppend {};

template<typename newT, typename ... list>
struct AggregateAppend<newT, aggregate<list ...>>
{
    typedef aggregate<list..., newT> type;
};
```

Source Code 3.4.: The `AggregateAppend` meta-function. Here the variadic template parameter `list` is used to match and extract the set of types composing the user-supplied aggregate. The given new type `newT` is then appended to the extracted list in order to form the new, extended aggregate type.

```
template<typename AggregateBlockT, ...>
class BlockMapGpu
{
    ...
    typedef typename AggregateAppend<
            DataBlock<unsigned char, BlockT0::size>,
            AggregateBlockT
            >::type AggregateInternalT;
    ...
}
```

Source Code 3.5.: Using the `AggregateAppend` meta-function to append the meta-data property `DataBlock<unsigned char, BlockT0::size>` at the end of the user-supplied aggregate `AggregateBlockT`, defining an `AggregateInternalT` data type.

0-th bit is used to mark an element as existing and the remaining bits remain available for storing additional per-element flags in derived classes. This property is injected at compile time using C++ Template Meta-Programming (TMP) features and placed as last property of the user-defined aggregate.

Source Code 3.4 shows how we define such an injection meta-function, and Source Code 3.5 shows how to use it to define a new aggregate type to be used internally in the BlockMapGpu.

Extracting block and scalar types from aggregates Using C++ TMP capabilities, we can also define meta-functions to extract the type information from aggregates of blocks.

In Source Code 3.6 we can see the two meta-functions – `BlockTypeOf` and `ScalaTypeOf` –

```
template<typename AggregateT, unsigned int p>
using BlockTypeOf =
        typename std::remove_reference<
                typename boost::fusion::result_of::at_c<
                        typename AggregateT::type, p
                        >::type
                >::type;

template<typename AggregateT, unsigned int p>
using ScalarTypeOf = typename BlockTypeOf<AggregateT>::scalarType;
```

Source Code 3.6.: The `BlockTypeOf` and `ScalaTypeOf` meta-functions allow retrieving the block and scalar types, respectively, of the property at position `p` in the given `AggregateT`.

```
template<unsigned int p>
auto insert(unsigned int linId)
        -> ScalarTypeOf<AggregateT, p> &;
```

Source Code 3.7.: Example of a method – `insert` – with generic return type, depending on the `AggregateT` type and the index of the property `p`.

that are used throughout our code to extract type information from the user-provided aggregate. OpenFPM aggregates internally store the list of the given types as *Boost*[3] sequence, making it easy to access them by position using the `result_of::at_c` metafunction. In this way `BlockTypeOf` can retrieve the `p`-th block type of the aggregate and `ScalaTypeOf` can further extract the type of scalar elements contained in the block.

The types extracted in this way are then used to generically define member variables or methods' return types – as shown in 3.7 – that match those provided by the user in the aggregate when *BlockMapGpu* is instantiated.

[3] See `https://www.boost.org`

4. SparseGridGpu

We present *SparseGridGpu*, a specialization of the *BlockMapGpu* class and second unit of this work. *SparseGridGpu* adds the notion of dimensionality and geometry on top of the sparse data structure and it also provides facilities for easily applying stencil-like operations on the data.

4.1. Geometry

The block map provides a working implementation of a block-sparse map, but its elements are indexed in a linear, 1D fashion. Since we need to be able to represent spatial domains in arbitrary dimensions, we need to add the concept of dimensions and coordinates on top of the block map.

The *SparseGridGpu* takes care of this first of all by accepting a template parameter specifying the dimensionality, d, of the sparse grid we want to instantiate. Then it exposes the same main methods as *BlockMapGpu*, *get* and *insert*, in its interface, but accepting d-dimensional coordinates as addressing parameter. The sparse grid takes care of the conversion from coordinates to linear indices both at the block level and within a single block.

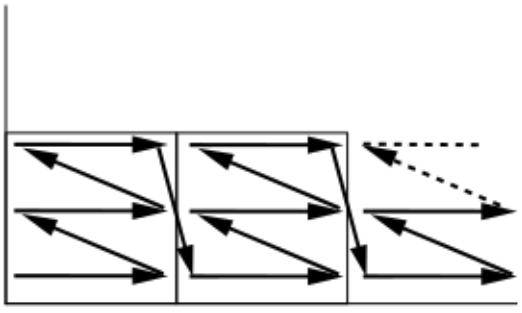


Figure 4.1.: Sketch showing how the linearization takes place in 2D. The linear index is first incremented by going over the same block in a row-wise fashion and then by moving to the next block and repeating.

Since the sparse grid is designed to store spatial data, we want each data block of the underlying map to represent spatially contiguous data: in 2D a data block should store

```
template<unsigned int dim, unsigned int blockEdgeSize>
class BlockGeometry
{
    ...
    template<typename indexT>
    __host__ __device__ mem_id
            LinId(const grid_key_dx<dim, indexT> coord) const;

    __host__ __device__ grid_key_dx<dim, int>
            InvLinId(const mem_id linId) const;

    template<typename indexT>
    __host__ __device__ mem_id
            BlockLinId(const grid_key_dx<dim, indexT> blockCoord) const;

    __host__ __device__ grid_key_dx<dim, int>
            BlockInvLinId(mem_id blockLinId) const;
    ...
}
```

Source Code 4.1.: The interface of the `BlockGeometry` class, which handles coordinates-to-linear and linear-to-coordinates transformations, both for elements and for blocks.

the data of a small square "patch" of the domain and in 3D it should store a small "cube". The linearization takes therefore place first at the block level, where blocks are stored with lexicographic ordering with respect to their coordinates. This step involves dividing all the coordinates by the block edge length and linearizing this value in order to find the data block where the element lies. Then the remainders for each dimension are linearized again in order to find the offset of the element within the data block, as shown in Figure 4.1.

This way of performing the linearization allows for a better use of spatial data locality[1] in all the scenarios where it is necessary for a thread to access a neighbouring element along any dimension, e.g. when performing stencil-like operations on the data.

The linearization and its inverse transformation are taken care of by a special `BlockGeometry` class we implemented (see Source Code 4.1), which provides the `LinId`, `InvLinId`, `BlockLinId` and `InvBlockLinId` methods. This allows for easily plugging in a different linearization scheme, in case there is the need for a different one[2].

[1] Compared to the naive *row-wise* linearization scheme.

[2] Pietro Incardona also implemented a linearization scheme based on the Z-order curve. The results for this alternative linearization are included in chapter 5 for comparison purpose.

4.2. Stencil application

The application of stencils is one of the most common way to operate on data, both in simulations and in image processing. One important requirement for the development of our sparse grid was therefore providing the user with a simple and convenient interface for applying stencil-like operations, while hiding the internal layout of the data structure as much as possible. Also, we wanted to provide the user with good out-of-the-box performance without the need for them to write complex optimizations in their code.

Our interface for stencil application is designed around the following principles:

1. The user should code the stencil operator in the form of a functor object exposing a `__device__`-tagged `stencil` method. This method should be coded in a way to be called on each existing element of the domain.

2. The user should define shared memory regions – which size can be automatically computed at compile time, based on the block size and stencil support size – according to the needs of the specific stencil. Shared memory is used for loading the whole chunk of data in a coalesced way and then allowing each thread to access its neighbouring data in a faster way.

3. The *SparseGridGpu* provides the infrastructure for launching the user-provided stencil on all the existing elements, passing the reference to the block as well as the coordinates of the element being considered down to the stencil function.

4. The *SparseGridGpu* also provides functions for loading the block's data as well as loading also a ghost layer from neighbouring blocks, into a shared memory region. It also provides a function to store the block updated content from the shared memory again into the data structure in global memory.

This design provides the user with total flexibility as to what properties to load for each element, how to operate these values together and in which properties to store the results. It frees the user from the burden of managing ghost layers, which is especially relevant at higher dimensions, and from the need to code coalesced load and store access to the block in all the stencils.

4.2.1. Stencil launching infrastructure

Since stencil operations need to be applied on all the existing elements of the data structure there is no need to perform expensive searches: we can just access the underlying index and data buffers – which by design are plain arrays – and distribute all elements evenly between thread blocks. Since our data structure is block-based, we assign each data chunk

to a thread block and then each thread verifies if the corresponding element is actually existing – i.e. if it has the exist-bit set to 1 – before actually applying the stencil function.

Our stencil launching infrastructure is composed of:

device-side code which computes the coordinates of the corresponding element, checks if the element is existing and, if it is the case, calls the user-defined stencil,

host-side code which gets the indices and data buffers and launches the specific stencil-executing CUDA kernel mentioned above on each of the blocks contained in the buffer.

Modes The stencils can be applied in two modes, either performing an *insert-on-write*, i.e. writing on a new block which will then be inserted into the data structure, or writing *in-place*, i.e. directly altering the existing data. The *insert* mode allows several threads to add new elements for the same spatial position and then operating a reduction on all the values. This however has the additional cost of having to flush the inserts and merge them into the data structure, as explained in 3.1.2, to ensure consistency of the data. The *in-place* mode instead allows for faster execution, under the assumption that the threads are mapped one-to-one to already existing elements which they can modify directly, without the need to perform an insertion.

Interface The stencil operator is given by the user in the form of a functor structured as in Source Code 4.2: it has to specify a support radius[3], a `stencil` function containing the actual operations and a `flush` function where the user can specify the reduction operator to be used in case the stencil is applied in insert mode. The user then passes the stencil for application as a template parameter to the `applyStencils` method, as shown in Source Code 4.3.

4.2.2. Load & Store facilities

SparseGridGpu provides functions to allow for easy loading of a block's content or ghost layer from global into shared memory and to store a block's content from shared into global memory.

The idea behind these functions is that the user allocates a shared memory area with enough space to store the content of a block – plus its ghost layer – for a given property `p`, then they load the block inner content and/or the ghost layer, in a coalesced way, for the same property. After the load phase the user can then apply the stencil operator directly on the shared memory region and finally store the result back – again via coalesced

[3] I.e. how many neighbouring elements in each direction are used by the stencil kernel.

```
template<unsigned int dim, unsigned int p>
struct GenericStencil
{
    static constexpr unsigned int supportRadius = 1;

    template<typename SparseGridT, typename DataBlockWrapperT>
    static inline __device__ void stencil(
            SparseGridT & sparseGrid,
            const unsigned int dataBlockId,
            const openfpm::sparse_index<unsigned int> dataBlockIdPos,
            unsigned int offset,
            grid_key_dx<dim, int> & pointCoord,
            const DataBlockWrapperT & dataBlockLoad,
            DataBlockWrapperT & dataBlockStore,
            bool applyStencilHere,
            args...)
    {
            ...
    }

    template <typename SparseGridT, typename CtxT>
    static inline void __host__ flush(SparseGridT & sparseGrid, CtxT & ctx)
    {
            ...
    }
};
```

Source Code 4.2.: The interface a stencil must implement. It has to specify a **constexpr** `supportRadius` which can be used to compute the thickness of the ghost layer at compile time, together with a `__device__`-tagged `stencil` method to be applied elementwise and a `__host__`-tagged `flush` method which is used to commit the changes to the data structure when the stencil is used in *insert mode*.

```
...
sparseGrid.applyStencils<
                GenericStencil<3, 0>
        >(STENCIL_MODE_INPLACE, args...);
...
```

Source Code 4.3.: A sample application of a stencil. The `GenericStencil` functor is passed as template parameter.

memory access – to global memory. If the stencil requires reading from and writing to several properties, the user can simply allocate and load/write them into shared memory independently.

4.2.3. User-defined stencils

A stencil to be applied on a *SparseGridGpu* using our infrastructure must implement the interface as per 4.2.1 and can use the load/store functions (4.2.2) to reduce the impact of memory accesses in a simple way. A typical stencil structure looks like the one in Source Code 4.4 and it can be broken down into the following parts:

template meta-programming The first part (lines 17-22) takes care of inferring the actual data types and computing, at compile time, the size of the shared memory region which needs to contain both the block and its ghost layer.

load In the second part (lines 23-26) the shared memory region is declared and both the block and ghost data are loaded from global memory into it with the `loadGhostBlock` method. All threads of the block cooperate in this load operation. Finally all threads in the block are synchronized, making sure the data is fully loaded before any computation can take place.

computing In the third part (lines 27-38) the actual stencil computation can take place by only accessing the shared memory region and avoiding any global memory access.

store In the fourth part (lines 39-40) another blockwise synchronization takes place, ensuring all computation is complete before proceeding to storing the data to global memory with the `storeBlock` method.

Following this scheme, the user can easily code stencil operations to work on the *SparseGridGpu*.

```
1  template<unsigned int dim, unsigned int pLoad, unsigned int pStore>
2  struct SampleStencil
3  {
4      ...
5      template<typename SparseGridT, typename DataBlockWrapperT>
6      static inline __device__ void stencil(
7              SparseGridT & sparseGrid,
8              const unsigned int dataBlockId,
9              const openfpm::sparse_index<unsigned int> dataBlockIdPos,
10             unsigned int offset,
11             grid_key_dx<dim, int> & pointCoord,
12             const DataBlockWrapperT & dataBlockLoad,
13             DataBlockWrapperT & dataBlockStore,
14             bool applyStencilHere,
15             args...)
16     {
17         typedef typename SparseGridT::AggregateBlockType AggregateT;
18         typedef ScalarTypeOf<AggregateT, pLoad> ScalarT;
19         typedef decltype(sparseGrid.getLinIdInEnlargedBlock(0)) IndexT;
20         constexpr unsigned int enlargedBlockSize = IntPow<
21                 SparseGridT::getBlockEdgeSize() + 2 * supportRadius, dim
22                 >::value;
23         __shared__ ScalarT enlargedBlock[enlargedBlockSize];
24         sparseGrid.loadGhostBlock<pLoad>(
25                 dataBlockLoad, dataBlockIdPos, enlargedBlock);
26         __syncthreads();
27         ScalarT result; IndexT linId;
28         if (isActive)
29         {
30             linId = sparseGrid.getLinIdInEnlargedBlock(offset);
31             result = myComputations(...);
32         }
33         __syncthreads();
34         if (isActive)
35         {
36                 enlargedBlock[linId] = result;
37         }
38         __syncthreads();
39         sparseGrid.storeBlock<pStore>(dataBlockStore, enlargedBlock);
40     }
41     ...
42 };
```

Source Code 4.4.: A typical stencil structure. Block and ghost layer data are first loaded to the `enlargedBlock` shared memory region (24-25), then the stencil actual computations are applied only by active threads (27-38) and finally the result is stored back to global memory (40).

Part III.

Results and Discussion

5. Performance Results

In chapter 3 we have shown how we designed and implemented a general purpose associative array allowing for massively parallel lookup and insert operations on the GPU. In chapter 4 we have then shown how we built a block-sparse grid to be used to store sparse spatial data which show some degree of clustering. This *SparseGridGpu* allows – under the assumption that data consists of locally-dense *patches* within a sparse domain – the application of stencil-like operators.

We have therefore measured the performance of the *SparseGridGpu* against three metrics: its ability to sustain a high parallel insertion load – i.e. measuring the insertion rate it can achieve –, its ability to sustain a parallel lookup load – i.e. measuring the lookup rate it can achieve – and its ability to apply stencils on its elements – i.e. the rate of elements it can process per second.

Test setup All the results reported in this work come from tests run on an NVIDIA GeForce GTX 1070 (Pascal). It features 1920 CUDA cores, 8GB GDDR5 SDRAM and 256 GB/s memory bandwidth. The GPU is mounted on an Asus ROG GL502VS laptop with 16GB DDR4 SDRAM and an Intel Core i7 6700HQ with four cores.

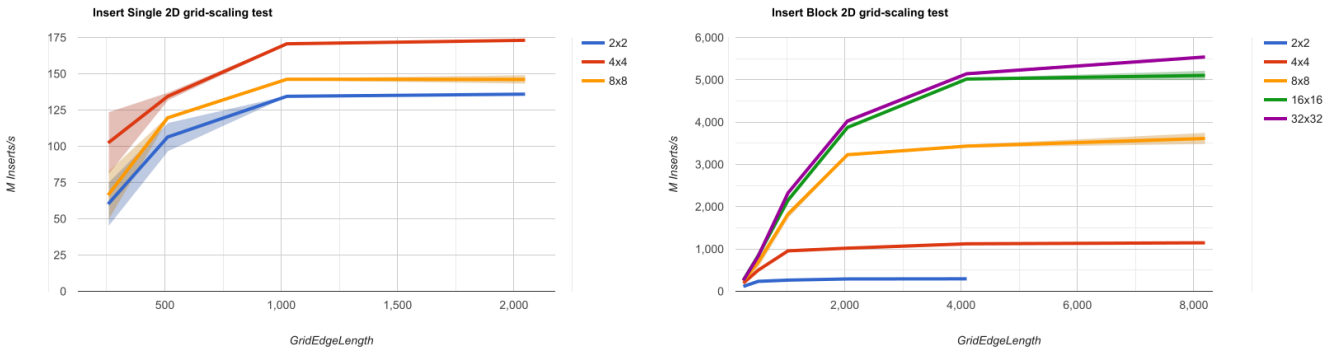
As official figures are not available, we measured the single-precision peak performance of our GPU using, as a proxy, the `cublasSgemm`[1] matrix-matrix multiplication from the *cuBLAS* library. Our result is 5.8 TFlops/s.

All performance plots are automatically produced with the *Google Charts*[2] API as part of a test suite we developed for monitoring performance during continuous integration testing in OpenFPM development. All plots include the mean values over 100 measurements, in the form of line-connected points, as well as the area contained within the standard deviation, in the form of shaded area around the line.

5.1. Insertion performance

Since the *SparseGridGpu* is a dynamic data structure that allows for massively parallel insertions on the GPU, the first way to measure the performance our design and implementation

[1] See https://docs.nvidia.com/cuda/cublas/index.html#cublas-lt-t-gt-gemm
[2] See https://developers.google.com/chart/



(a) Single insertion. Tested on 2x2, 4x4 and 8x8 blocks.

(b) Block insertion. Tested on 2x2 to 32x32 blocks.

Figure 5.1.: Scaling behaviour of 2D insertion with respect to the edge length of the grid.

can achieve is by testing the insertion rate it can sustain.

There are two ways to perform insertions, at the single element level or blockwise:

single In this case insertion is performed on device code by calling the `insert` method: this retrieves a new empty block from the insertion buffer, marks the corresponding element as existing and returns a reference to the value. For each thread calling the insert method – and for each call by the same thread – a new entire block is used from the insertion buffer. This allows for the flexibility to have threads inserting randomly placed elements, without any relation to the thread layout and grouping into blocks, however this flexibility is paid in terms of insertion inefficiency, since an entire block is used – and will be reduced and merged – just for inserting one single element.

block In this case insertion is performed by calling the `insertBlock` method from just one thread of the thread-block: this retrieves a new empty block from the insertion buffer and returns its reference[3]. The pointer to the data block can then be shared – via shared memory – to all the threads of the same CUDA thread-block, which in turn will be able to independently write their values and set the elements as existing. This allows to optimize the usage of the insertion buffer and to minimize the cost associated with the subsequent reduction of the inserted elements, under the assumption that threads within the same thread-block are mapped one-to-one to the elements of a data block.

We have tested insertion performance in both ways in the scenario of a dense-block insertion, i.e. inserting full blocks.

[3]This reference is actually encapsulated by an OpenFPM `encap` object.

In Figure 5.1a we measure the rate of insertions obtained on 2D grids with edge lengths ranging from 256 to 2048 and in the case of 2x2, 4x4 and 8x8 blocks. The best scenario for single-element insert turns out to be using 4x4 blocks, where the sparse grid is able to sustain a rate of around 170 millions insertions per second.

These insertions are performed on a pre-filled data structure, therefore the figures shown in the above plot also take into account the time spent in the reduction operation with the pre-existing value for the same element.

When we move to the blockwise insertion scenario, we can see an increase in the number of elements which can be inserted per unit of time, as expected. In Figure 5.1b we can see the rate of insertions obtained, by blockwise insertion, on 2D grids with edge lengths ranging from 256 to 8192 and in the case of 2x2 to 32x32 blocks.

Here we can notice how the bigger block sizes are able to yield better performance, achieving rates up to around 5.5 billions insertions per second, corresponding to an effective insertion throughput of 22 GB/s[4][5]. We can also notice that we can insert many more elements, at the same time, in the blockwise mode than in the single mode. This is due to the smaller and more efficient memory footprint that the former has, while the latter has to use an entire block of memory for each element that is to be inserted. Therefore, the blockwise insertion can achieve, for the same number of elements inserted (2048), up to 24x speedup and, in case the number of elements to insert is higher, the insertion rate can speedup even 32x when compared to the best one which can be achieved with single-element insertions.

5.2. Lookup performance

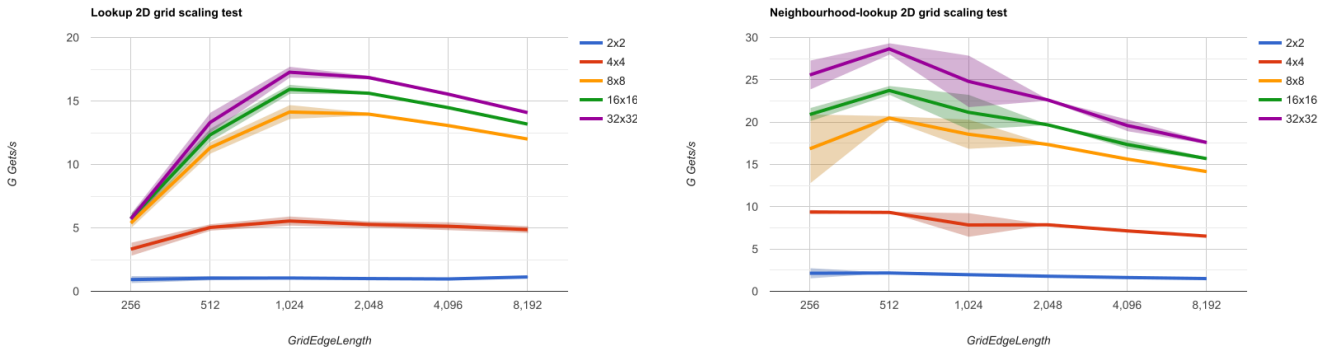
Our second performance metric is the rate of parallel lookups the *SparseGridGpu* can sustain. For this purpose we perform two different tests which reproduce two typical access patterns:

single The test kernel performs one single `get` call per thread, for the corresponding element. Therefore the lookups within each block do not produce any divergent execution, as they access elements belonging to the same data block.

neighbourhood The test kernel performs nine `get` calls per thread, retrieving the corresponding element and its full 2D neighbourhood. The neighbouring blocks are also accessed, by the threads at the block boundary. Here there is no specific optimization and the same element is looked up multiple times by different threads: the performances of this test highlights the capability of the GPU data caches.

[4]The elements are *float*-typed, therefore each element accounts for 4 Bytes.

[5]Since the insertions take place on a pre-filled data structure, each insert also implies a read from global memory, therefore the memory bandwidth which is occupied by data transfers is 44 GB/s.



(a) Single-element lookup. Tested on 2x2 to 32x32 blocks.

(b) Neighbourhood lookup. Tested on 2x2 to 32x32 blocks.

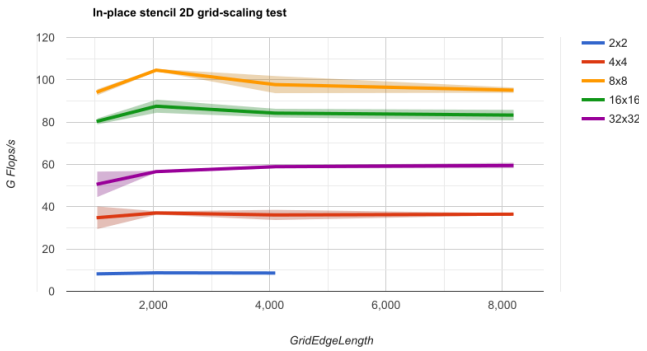
Figure 5.2.: Scaling behaviour of 2D lookup with respect to the edge length of the grid. The scale of the horizontal axis is logarithmic for better readability.

Figure 5.2 shows the rate of single element calls to the `get` method our GPU can sustain, for both scenarios and with respect to the size of the domain edge. In both cases we can see an initial increase of performance with increasing size, as the device is not fully utilized, while then the increasing cost of the binary searches takes over, bringing the performance down again. We can also see the effect of the data caches by comparing the two cases, as the peak performance for the 8x8 block in the single case is around 14 G Gets/s, while the neighbourhood lookup, for the same size, reaches 18.5 G Gets/s. We can also clearly see how a bigger block size is advantageous, since for storing the same amounts of elements having a bigger block size causes a reduction in the underlying data structure size, allowing for faster searches.

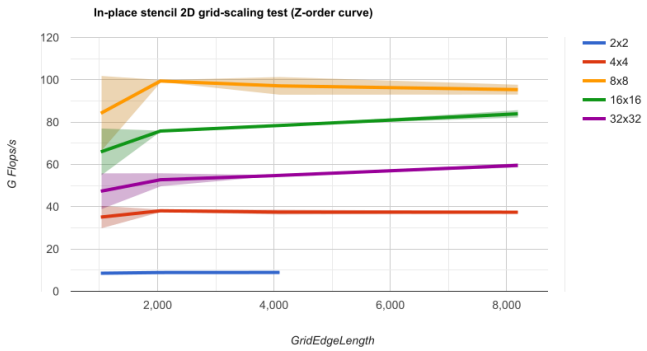
5.3. Stencil performance

The third metric we need to measure the performance of *SparseGridGpu* against is the application of elementwise operations, as stencils. We measure the performance of stencil application as the rate of elements that can be processed per unit of time (Elem/s). The rate of *floating-point operations per second* (Flops/s) performed on each element depend on the actual number of operations performed by the specific stencil at hand. In our tests we use a simple explicit Euler time-stepping iteration to solve the Poisson equation for heat transfer modelling – which has 7 Flops in 2D and 9 Flops in 3D for each element – and we will also show figures in GFlops/s for this stencil. The stencil uses our `loadGhostBlock` and `storeBlock` methods for accessing global memory, then it performs the operations on shared memory.

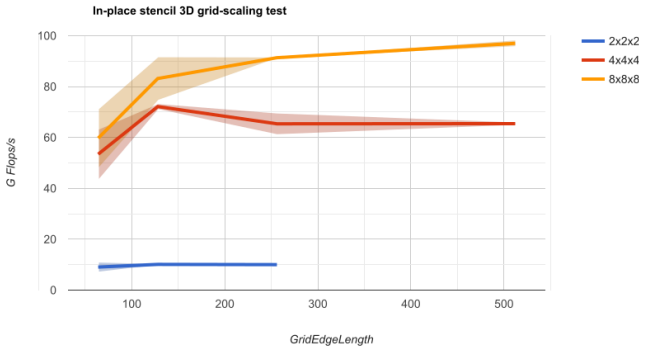
As detailed in 4.2.1, stencils can be either applied *in-place* or by an *insert* operation.



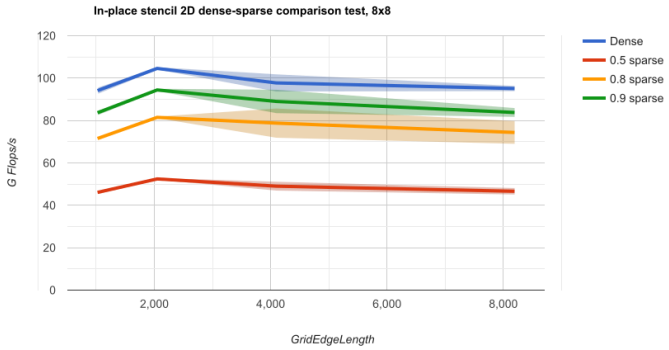
(a) Standard linearization, 2D, 2x2 to 32x32 blocks.



(b) Z-order curve linearization, 2D, 2x2 to 32x32 blocks.



(c) Standard linearization, 3D, 2x2x2 to 8x8x8 blocks.



(d) Comparison with sparse grids: 0.5, 0.8 and 0.9 occupancy.

Figure 5.3.: Scaling behaviour of stencil application on a dense grid, with respect to the edge length of the grid. Different colours indicate different block sizes in (a), (b), (c), while in (d) they indicate different degree of sparsity.

5.3.1. In-place stencil application

We apply our *HeatStencil* in the in-place mode to a variety of scenarios, in order to gauge how dimensionality and sparsity affect performance.

Dense 2D In Figure 5.3 we can see the performance of the stencil computation applied to a dense 2D domain. The 8x8 block results as the optimal size for 2D and can sustain a performance around the 100 GFlops/s – which is 1/58 of the measured peak performance for the device – with a peak at 104 GFlops/s: this means that the stencil is applied at a rate that ranges between 14 and 15 GElem/s on all but the smallest grids.

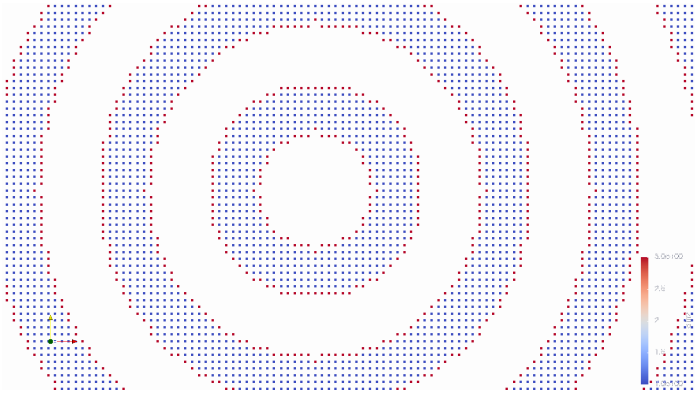


Figure 5.4.: A view of the 2D sparsity pattern used in our tests. This image shows the elements of the *SparseGridGpu*: the inner elements are marked in blue while the boundary elements are marked in red. *SparseGridGpu* can write all its properties, including the metadata, to VTK format through OpenFPM's VTK writing facility.

Comparable results can be obtained if swapping our linearization scheme with one based on the Z-order curve (or Morton curve) which has been developed by Pietro Incardona. This can be seen in Figure 5.3b.

Dense 3D The next scenario we tested was to apply the stencil on a dense 3D domain, to assess the impact that dimensionality increase has on performance, especially because of the increase in the size of the ghost layer that has to be loaded.

In Figure 5.3c we can see how, in terms of Flops/s, the 3D performance is almost on par with the 2D one. In terms of rate of elements processed per unit of time, 3D stencil application reaches around 11 GElem/s. In this scenario, the optimal block size appears to be 8x8x8, which means a total of 512 inner and 488 ghost elements in the block.

Sparse 2D After 3D we explored how sparsity within the block affects performance. Here we need to stress the fact that only sparsity within a single block matters for performance, as empty blocks are simply nonexistent in the data structure and therefore no operation is applied on them.

We have applied the *HeatStencil* on a 2D sparse domain composed of several concentric circular crowns (see Figure 5.4), with their thickness and spacing being the parameters used to tune the average block *occupancy* – i.e. the ratio of existing elements over the total block size.

As we can see in Figure 5.3d, sparsity causes a reduction in performance, with respect to the dense case, that is approximately proportional to the average occupancy of the data blocks. This is consistent with our expectations and with the observation that, in case of

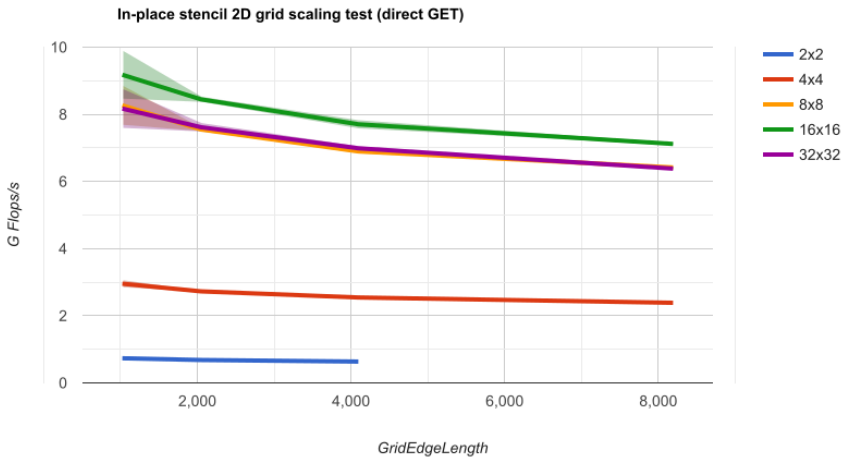


Figure 5.5.: Performance of 2D stencil application using direct element lookups instead of coalesced load into shared memory. Different colours indicate different block sizes, from 2x2 to 32x32.

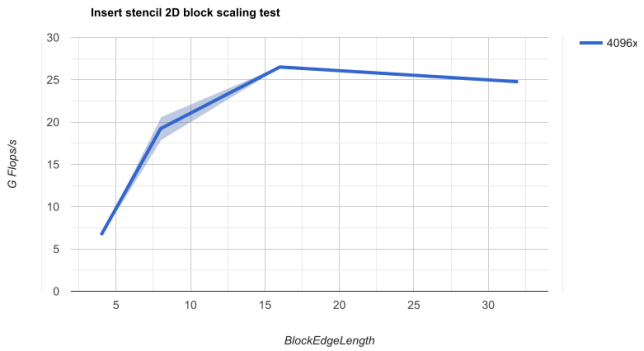
nonexisting elements, the corresponding threads of the block still contribute to the load and store operations while not contributing to the actual arithmetic operations. The above results for the 0.8 and 0.9 occupancies correspond to a rate of actual processed elements of around 12 GElem/s.

5.3.2. Shared memory vs. data cache

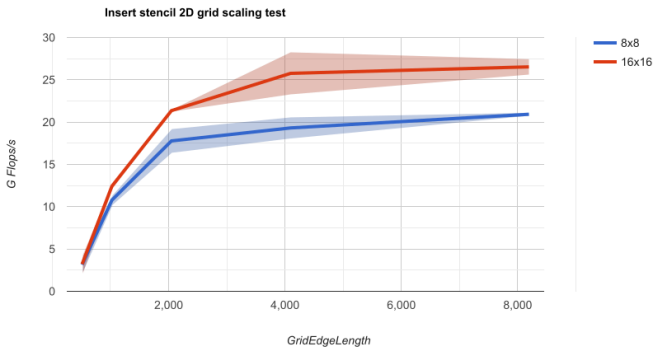
We have made the design decision of providing an interface for retrieving blocks and their ghost layer and copying them into shared memory in a coalesced way. It is now important to test wether this use of shared memory is actually advantageous when compared to the approach of looking up all the required neighbouring elements with a `get` method call.

In Figure 5.5 we can see the performance that can be achieved by applying a variant of the *HeatStencil* which uses single element lookups instead of the coalesced *loadGhostBlock* method. Specifically, we can see how the performance degrades with increasing domain size, as the lookups become more and more expensive[6] and how in this case a 16x16 block size would be optimal. However the most important observation is that the overall performance, even in the best scenario, basically never exceeds 9 GFlops/s, which is less than 1/11 of the peak performance that can be achieved by using the load/store methods provided by our interface.

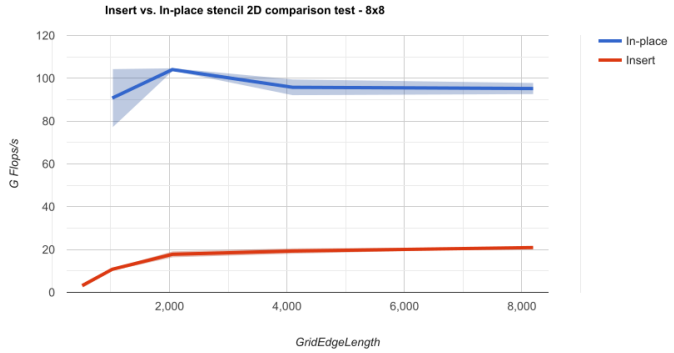
[6]The cost of the lookup is logarithmic with respect to the size of the dictionary, being it implemented as a binary search.



(a) Block-scaling behaviour.



(b) Grid-scaling behaviour.



(c) Comparison of insert and in-place modes on a 8x8 block, grid-scaling. Here we can see how by using insertions the performance drops to about one fifth of that achieved with the in-place mode.

Figure 5.6.: Performance of 2D insert stencil application on a dense grid.

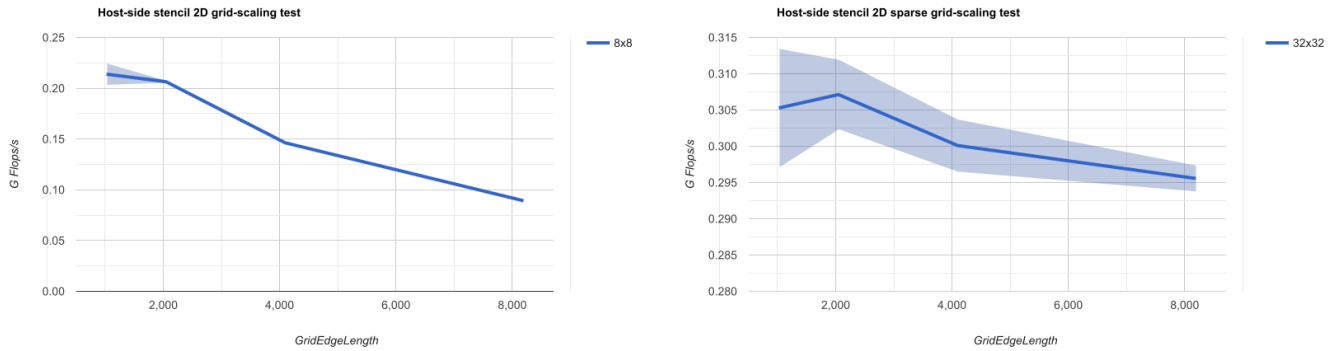
5.3.3. Insert stencil application

We have tested the insert mode for stencil application in the 2D dense case, to gauge how the insertion workflow impacts on the stencil application performance.

As shown in Figure 5.6a, we have first assessed the performance at various block sizes, for a fixed grid edge length of 4096: here we can see how performance increases until a block size of 16x16, then starting to drop again. We have therefore analysed the scaling of the 8x8 and 16x16 with respect to the grid size.

In Figure 5.6b we can see how in both cases the performance increases with the grid size, topping at 21 GFlops/s and 26.5 GFlop respectively, which correspond to 3 GElem/s and 3.7 GElem/s respectively.

Finally, with Figure 5.6c we show the impact the insert workflow has on stencil application performance, which in this case is reduced roughly fivefold.



(a) 2D dense, with 8x8 blocks.

(b) 2D sparse, with 0.5 block occupancy and 32x32 blocks.

Figure 5.7.: Scaling behaviour of 2D stencil application on a dense grid, performed on the host, with respect to the edge length of the grid. (a) shows the case of dense blocks, with 8x8 blocks, that is the best case scenario for GPU execution. (b) shows the case of sparse blocks, with 0.5 occupancy and 32x32 blocks, which is the best case for host-size execution.

5.3.4. Comparison with single-threaded stencil application on host

As a final comparison, we made a simple version of our stencil to be run, single-threaded, on the host. This is to have a baseline reference of what level of performance we would be able to achieve by iterating on our data structure in a sequential fashion.

In Figure 5.7a we measure the performance behaviour on dense 8x8 blocks. This is the most favorable scenario for GPU execution, according to our results in 5.3.1. We can see how host-side single-threaded execution can only peak at around 0.2 GFlops/s, while for the same scenario our GPU code can reach 100 GFlops/s, meaning approximately a 500x performance gain.

In Figure 5.7b we instead measure the performance on half-filled 32x32 blocks, which is the scenario that allows for best host performance and at the same penalizes GPU performance. Here we can see how host-side execution peaks at around 0.3 GFlops/s, while in the same scenario device code reaches 25 GFlops/s, which is still approximately 83x faster.

Of course this comparison is not really fair, as the host implementation is quite naïve and it could be further optimized and made parallel. However, it is anyway important as it highlights the magnitude of the difference in performance a user can obtain, out-of-the-box, by using *SparseGridGPU* on OpenFPM instead of simple loop iterations. This makes high performance available to users which happen to have a GPU device but lack the necessary HPC training and skills to leverage it.

6. Discussion

We have designed and implemented a dynamic associative array for the GPU, which allows for general-dimensional spatial indexing, parametric and arbitrarily typed data and provides for a simple to use yet powerful infrastructure for applying stencil-like operations on its elements. The results we reported in chapter 5 show that the *SparseGridGPU* is a versatile block-sparse data structure which:

- supports fast massively parallel insertions and lookups,
- can achieve good stencil application performance on dense blocks and retains it proportionally on partially populated blocks,
- allows the flexibility of applying elementwise operations where one element can be inserted by several threads and then reduced, while retaining acceptable performance.

Insertions & Lookups Specifically, insertions can sustain rates of 5.5 GElem/s, corresponding – in our test case – to an effective insertion throughput of 22 GB/s. This is around 1/10 of the total memory bandwidth of the device used in our tests, however this performance cost includes the reduction operations and the coherence guarantees that a parallel data structure must ensure.

Lookups are performed at a rate of 14 GElem/s for single-element access and can reach 18.5 GElem/s in case of full neighbourhood access (in the 8x8 block case[1], over 1M total elements in the data structure). This implies an effective data throughput of 54 GB/s and 74 GB/s respectively, meaning around 1/3 and 1/5 of the total memory bandwidth.

Stencils When applying stencils in-place, *SparseGridGpu* was able to sustain a rate of 14.9 GElem/s in a dense 2D scenario with 8x8 blocks, which means approximately 233 MBlock/s. This translates to the ability to actually process 60 GB/s of data and to an effective data transfer rate (load + store) of 153 GB/s. As reference, a naive stencil implementation on a dense matrix, with 6 global memory accesses per element, would have required a memory bandwidth of 358 GB/s to sustain the same processing rate, exceeding the available 256 GB/s of our device.

[1] Bigger block sizes can yield better performance, but we consider 8x8 as "standard" 2D block size.

In terms of Flops/s, in the best scenario[2] *SparseGridGpu* reached 104 GFlops/s on a device that can peak at 5.8 TFlops/s, meaning that it can use 1.8% of the peak performance. To put this result into perspective, we can perform a quick analysis using the *roofline model* [22], taking into account the global memory bandwidth, assuming instantaneous access to the shared memory and assuming all accesses to global memory to be perfectly coalesced. A naïve access scheme, where each thread triggers 6 memory accesses, has an *operational intensity*[3] of 0.29. This value determines – at the available memory bandwidth of 256 GB/s – a maximum attainable performance of 74.6 GFlops/s. If we instead look at the access scheme of the 2D case with 8x8 blocks that *SparseGridGpu* performs, we can count – for each block – 100 loads (block + ghost) and 64 stores, accounting for 656 Bytes, and 448 Flops. This implies an operational intensity of 0.68, determining a maximum attainable performance of 174.8 GFlops/s in this case. This means that *SparseGridGpu* is actually reaching 60% of the theoretical performance in this scenario and that missing 40% is the potential improvement we can aim at with future optimizations of the *SparseGridGpu*.

Sparsity The results on the stencil application on sparse blocks also confirm that the performance penalty is proportional to the average block occupancy, meaning that users will be able to predict the performance they can attain, based on the average data block occupancy implied by their problem.

As already discussed in section 1.5, we can expect APR data to be sparse but locally clustered, which ensures data blocks to have a high average occupancy. Our results on stencil application on sparse blocks indicate that we can expect good computational performance from processing APR data, once it will be implemented on top of the *SparseGridGpu*.

Dense applications The results we presented suggest that the *SparseGridGpu* could be a suitable general purpose solution to the problem of performing simulations on dense domains with irregular – i.e. non-rectangular – boundaries. In such scenarios the average data block occupancy would be close to 1, but the data would fit poorly in a standard rectangular domain. Here the sparse-blocked nature of our data structure would automatically take care of properly mapping the data to memory, while the user should only insert the actually existing elements of the grid and then code the stencils to apply the required elementwise operations for the simulation at hand.

[2] That is the 2D dense case with 8x8 blocks.

[3] *Operational intensity* is defined as the ratio of operations performed per Byte of memory traffic [22].

6.1. Conclusions

As we have seen in Chapter 1, support for general purpose associative arrays on GPU is, at best, elusive. On the one hand there are interesting proof-of-concept implementations that are not distributed in a ready-to-use form and are limited by very strict assumptions. On the other hand there are few *pre-packaged* implementations that however do not allow the data structure to be built or modified from device-side code.

SparseGridGpu fills this gap by providing an easy-to-use solution, fully integrated into the open-source OpenFPM framework and available as part of it. It is a dynamic data structure, allowing modifications to be performed, in parallel, from the device[4] and it provides an infrastructure for efficiently applying elementwise and stencil-like operations. It accepts arbitrary data types and therefore it supports not only scalar, but also vector- and tensor-valued quantities, which are a typical requirement of simulations and scientific computing.

Importantly, it also delivers good performance out-of-the-box, fulfilling OpenFPM's design goal of allowing easier and faster development of HPC applications.

Finally, our results prove that the *SparseGridGpu* could be successfully used to allow leveraging GPUs to accelerate sparse simulations and data processing tasks, especially in the scenarios that primarily drove our work: single nodes performing medium-sized simulations or workstations processing data coming from scientific equipment in near-realtime.

6.2. Outlook and future work

The presented work constitutes the first iteration of the development and integration of a sparse dynamic data structure for the GPU into OpenFPM. While this specific project can be considered complete and was able to reach tangible results, as shown in chapter 5, the code should still be properly refined and there are further performance optimizations that should be implemented.

Specifically, the natural first future step for OpenFPM maintainers should be to complete the implementation of a *deletion* operation on *BlockMapGpu*: although it is not specifically required in case the client application is the APR, it would complete the interface and allow for more flexible use. Together with this, the *SparseGridGpu* interface should be polished further, to allow for a more coherent integration into OpenFPM.

Furthermore, there are still *performance hotspots* in our code which we believe could greatly benefit from further optimization and allow for better overall performance: in particular the *flush* workflow for insertions and the way ghost layers are loaded for each block are

[4] With the exception of the *deletion* operation, which is still under development and not yet available.

the most relevant performance-limiting factors and therefore the first parts of the code that should undergo further optimization.

Another possible development that we think would be worth exploring is swapping the *sorted-array*-based data structure at the heart of the *SparseGridGpu* with an implementation of *GPU B-Tree* [13] and compare their performance.

Beside the aforementioned improvements to *SparseGridGpu* itself, we should also consider possible future developments *on top* of it. The first and most natural one would be to actually use *SparseGridGpu* as the data structure to provide the layers for the APR, together with the *pulling scheme* algorithm for its construction and the interpolation operators to propagate information across different layers.

Another interesting possibility is to use this *SparseGridGpu*-based APR data structure to implement *Adaptive Mesh Refinement* (AMR) simulations on the GPU. Our work would allow all required dynamic operations to be natively performed on the GPU.

Finally, since OpenFPM already has facilities for handling the data transfers and synchronizations required in distributed settings, it should easy to employ *SparseGridGpu* itself or the *SparseGridGpu*-based APR in distributed and multi-device machines.

This work lays the foundations for all the above developments, which we hope will soon be integrated into OpenFPM, providing users with more easily accessible computational performance for their simulations and data processing tasks.

Bibliography

[1] Chris McClanahan. 'History and evolution of GPU architecture - A paper survey'. In: (2010).

[2] TOP500.org. *TOP500 list, June 2019*. 2019. URL: https://web.archive.org/web/20190731133659/https://www.top500.org/lists/2019/06/ (visited on 07/31/2019).

[3] Lorenzo Dematté and Davide Prandi. 'GPU computing for systems biology'. In: *Briefings in bioinformatics* 11.3 (2010), pp. 323–333.

[4] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[5] NVIDIA Corporation. *cuSPARSE API Reference Guide*. 2019. URL: https://web.archive.org/web/20190801120940/https://docs.nvidia.com/cuda/cusparse/index.html (visited on 08/01/2019).

[6] Saman Ashkiani et al. 'GPU LSM: A dynamic dictionary data structure for the GPU'. In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2018, pp. 430–440.

[7] Ke Yang et al. 'In-memory grid files on graphics processors'. In: *Proceedings of the 3rd international workshop on Data management on new hardware*. ACM. 2007, p. 5.

[8] Changkyu Kim et al. 'FAST: fast architecture sensitive tree search on modern CPUs and GPUs'. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM. 2010, pp. 339–350.

[9] Jordan Fix, Andrew Wilkes, and Kevin Skadron. 'Accelerating braided B+ tree searches on a GPU with CUDA'. In: *2nd Workshop on Applications for Multi and Many Core Processors: Analysis, Implementation, and Performance (A4MMC), in conjunction with ISCA*. 2011.

[10] Lijuan Luo, Martin DF Wong, and Lance Leong. 'Parallel implementation of R-trees on the GPU'. In: *17th Asia and South Pacific Design Automation Conference*. IEEE. 2012, pp. 353–358.

[11] Dan A Alcantara et al. 'Real-time parallel hashing on the GPU'. In: *ACM Transactions on Graphics (TOG)* 28.5 (2009), p. 154.

[12] Tero Karras. 'Maximizing parallelism in the construction of BVHs, octrees, and k-d trees'. In: *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*. Eurographics Association. 2012, pp. 33–37.

[13] Muhammad A Awad et al. 'Engineering a high-performance GPU B-Tree'. In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. ACM. 2019, pp. 145–157.

[14] NVIDIA Corporation. *GVDB Voxels - Programming Guide*. 2018. URL: https://developer.nvidia.com/gvdb.

[15] Pietro Incardona et al. 'OpenFPM: A scalable open framework for particle and particle-mesh codes on parallel computers'. In: *Computer Physics Communications* 241 (2019), pp. 155–177.

[16] NVIDIA Corporation. *NVIDIA TESLA V100 GPU ARCHITECTURE*. 2017. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[17] NVIDIA Corporation. *NVIDIA Turing GPU Architecture*. 2018. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[18] NVIDIA Corporation. *CUDA C Programming Guide*. 2019. URL: https://web.archive.org/web/20190725160034/https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 07/25/2019).

[19] NVIDIA Corporation. *NVIDIA TESLA P100*. 2016. URL: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[20] Bevan L Cheeseman et al. 'Adaptive particle representation of fluorescence microscopy images'. In: *Nature communications* 9.1 (2018), p. 5160.

[21] Sean Baxter. *Modern GPU*. 2013. URL: https://nvlabs.github.io/moderngpu.

[22] Samuel Williams, Andrew Waterman, and David Patterson. *Roofline: An insightful visual performance model for floating-point programs and multicore architectures*. Tech. rep. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2009.