Accelerating Scientific Applications on GPUs

Pau Farré Gonzalez

A thesis submitted in partial fulfilment for the
degree of
Master in Innovation and Research in Informatics
High Performance Computing

in the

Facultat dínformàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Director: Daniel Jiménez Gonzalez
Departament d'Arquitectura de Computadors (DAC)

July 4, 2016

Contents

List of Figures

List of Tables

Chapter 1

Introduction

All around the world, new Supercomputers are introducing accelerators as a way of improving performance and energy consumption, among which we can find GPUs.

One of the reasons that GPUs are being introduced in the HPC world is that their raw performance (FLOPS) and performance/watt (FLOPS/W) is higher than in CPUs, since GPUs were originally designed to do graphics, which is an inherently parallel workload.

GPUs have evolved into throughput-oriented devices with a high number of functional units, small caches, and high memory bandwidth to achieve high parallel performance. While CPUs have evolved into devices with high frequencies and a deep memory hierarchy to achieve high single-thread performance.

Unfortunately, adapting the software of a supercomputer is not an easy task. It takes months of work to develop new HPC applications. Therefore, instead of writing brand new applications for GPUs, the current trend is to adapt existing applications. Nevertheless this process is slow and painful due to current applications being already optimized for SMPs.

We think that GPU computing is beneficial to HPC applications because GPUs offer higher peak performance (FLOPS) and efficiency (FLOPS/W) than current CPUs. The reason for this is that GPUs having longer SIMD vector length than CPUs and having a memory hierarchy devoted to obtain the maximum memory throughput, GPUs are able to exploit more DLP than CPUs.

Not all programs can be ported to GPUs, but most HPC applications, are data parallel applications that could benefit from GPUs.

In-situ visualization in a GPU cluster is another reason for using GPUs. Actual GPU compute APIs are able to interact with GPU graphic APIs such as OpenGL exploiting memory locality in the GPU.

In this thesis we have analyzed and accelerated two large scientific applications used at the Barcelona Supercomputer Center (BSC). With this, we want to show how two complex applications can be efficiently ported to GPUs, enumerating the most important lessons learned. In addition, we have developed a mechanism to manage the coherency of CPU & GPU memories. Memory coherency is necessary because CPU and GPU have separated memories and coherency is not maintained by any hardware, therefore delegating this work to the programmer. Usually GPUs are not integrated in the same chip as CPUs, but sold as an external device plugged-in in a low-bandwidth interconnection network (typically PCI-express). Therefore a good memory management is key to achieve high performance in GPU applications.

The first application is a Computational Fluid Dynamics (CFD) application called WARIS-Transport developed in the Computer Applications (CASE) department and is a drop-in replacement for a well-

known application called FALL3D used for analyzing the transport and deposition of volcanic ashes.

In order to avoid the performance & programmability of maintaining CPU/GPU memory coherency, we opted to port all computations present in the iterative algorithm to the GPU.

CPU/GPU synchronization points act as a blocking operation in the CPU freezing offloading of work into the GPU and causing it to become idle when the synchronization point is reached. Therefore in order to feed the GPU with enough work and prevent stalls, in WARIS-Transport we minimized the amount of CPU/GPU synchronization points.

The second application is called PELE, a Molecular Conformation application with the Monte Carlo method (MC), originally developed at the Life Sciences (LIFE) department of BSC.

This application is significantly bigger than WARIS-Transport and the computation pattern cannot be completely offloaded to the GPU.

For this case we decided to use an automatic memory manager to ease the programmability of CPU/GPU memory coherency in a partially accelerated application. We present a Semi-Automatic Memory Manager (SAMM) to manage copies between CPU and GPU, which outperforms the automatic memory manager provided by NVIDIA, called Unified Memory (UM).

1.1 Document Organization

In chapter 2 we discuss NVIDIA GPU architecture features and the current evolution of performance between different NVIDIA HPC-class GPUs. Also we explain the CUDA programming model.

In chapter 3 we introduce the experimental set-up and methodology followed in this thesis.

In chapter 4 we present WARIS-Transport. We show implementation details, GPU development strategy and we give a comparison between the CPU and GPU implementation.

In chapter 5 we present PELE. In this chapter, a description of the application structure is provided. We also introduce each accelerated part with GPUs. In this application we propose using Unified Memory (UM) to manage CPU and GPU coherence. Also, and to increase performance, we propose a Semi-Automatic Memory Manager (SAMM). A comparison between the original implementation, and the GPU implementation with NVIDIA UM and SAMM is provided.

Chapter 6.1 concludes this thesis with the conclusions and lessons learned we extract from our work.

Chapter 2

GPU architecture and CUDA Programming Model

2.1 GPU architecture

In the past, Graphics Processing Units (GPUs) were focused exclusively on graphics processing, but since the unification of Vertex shader and Pixel shader units in a single, programmable computing unit, GPUs are able to efficiently execute data-parallel general purpose computations.

For simplicity, we will use the CUDA/NVIDIA nomenclature since it's the system used in this work.

A GPU typically contains a number of highly multi-threaded vector computing cores named streaming multiprocessors (SMs), a front-end that distributes work between all the SMs, and DRAM. For instance, in Figure 2.1 we can see a schematic GPU layout of the newest NVIDIA GPU architecutre, *Pascal*.

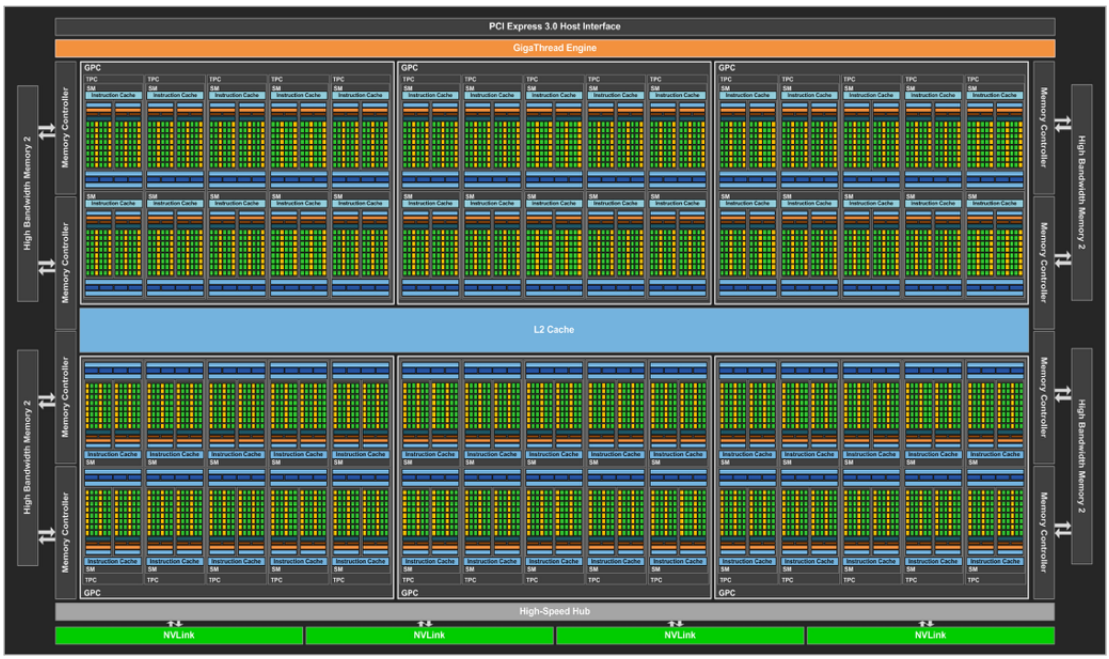

Figure 2.1: Pascal architecture [1].

Each of these SMs (shown in Figure 2.2) contains several vector pipelines that issue instructions that execute on vector functional units (CUDA cores). Each SM also contains a large register file, a small L1

data cache, a shared memory, and a specialized cache for textures that can also be used to store read-only data.

Figure 2.2: SM in Pascal architecture [2]

GPUs execute using a Single Instruction Multiple Thread (SIMT) model where threads are grouped in fixed-length sets of threads (i.e., *warps*). All threads within a warp (32 in NVIDIA GPUs) execute in lock-step the same instruction on different data in parallel. Each vector pipeline is able to handle several in-flight warps and has a scheduler that selects the warp to be executed in each cycle. This helps GPUs hide long latency operations by switching to another warp on a cache miss.

Architecture	**Tesla**	**Fermi**	**Kepler**	**Maxwell**	**Pascal**
Year	2008	2009	2012	2014	2016
Register File Size / SM	32 KB	128 KB	256 KB	256 KB	256 KB
L1 size & Shared / SM	16 KB	48 KB	48 KB	96 KB	64 KB
FP32 CUDA cores / SM	8	32	192	128	64
FP64 CUDA cores / SM	1	8	64	4	32
FP64:FP32 Ratio	1:8	1:4	1:3	1:32	1:2
GPU Card	**Tesla C1060**	**Tesla M2090**	**Tesla K40**	**Tesla M40**	**Tesla P100**
#SMs	30	16	15	24	56
#FP32 Cuda cores	240	512	2880	3072	3584
#FP64 Cuda cores	30	128	960	96	1792
Peak FP32 GFLOPs	622	2660	5040	6844	10608
Peak FP64 GFLOPs	77	665	1680	213	5304
TDP	187	225	235	250	300
FP32 GFLOPs/W	3.32	11.82	21.44	27.36	35.36
FP64 GFLOPs/W	0.41	2.95	7.14	0.85	17.68

Table 2.1: GPU Architectures over time. Data taken from NVIDIA.

In Table 2.1, in the upper part, we can observe a summary of the available resources per SM for the different CUDA architectures. And in the bottom part we see the most powerful single-chip HPC-class GPU from the each architecture[1] shown in the upper part.

NVIDIA has tree ways of improving performance (FLOPs) and efficiency (FLOPs/W) generation after generation. The first one is to scale the number of functional units (CUDA cores) per SM, increasing vector-level parallelism (e.g, from Tesla to Fermi and from Fermi to Kepler architectures, see FP32 CUDA cores/SM in Table 2.1). The second way is to scale the number of SMs, increasing TLP (e,g, from Kepler to Maxwell and from Maxwell to Pascal, see #SMs in Table 2.1). The third one is to scale the number of critical resources inside an SM, like the amount of shared memory or the register file size. Having more resources allows having more work in-flight, achieving more MLP and therefore, throughput.

For some HPC applications, support for double precision is mandatory to achieve good accuracy. Therefore, GPUs for HPC have to have a good FP64:FP32 ratio. As we can see in Table 2.1, the FP64:F32 ratio has improved from 1:8 in the first generation[2] to 1:2 in latest architecture coming this year. The exception is the Maxwell architecture. Maxwell was supposed to be released with 16nm integration scale (instead of 28nm like Kepler), but manufacturing problems forced NVIDIA to redesign the architecture for 28nm, prioritizing good FP32 performance over FP62 since their graphics market mainly demands high FP32 performance.

2.1.1 Memory hierarchy

The Memory hierarchy of a GPU is designed to provide throughput at the expense of memory latency. Therefore, GPU memory hierarchies tend to have small general-purpose caches and high memory bandwidth to feed all SMs. Since GPUs have been dedicated for years to do graphics, today we can still find specialized caches like the texture cache or the constant cache. In some situations, specialized caches like the texture cache are useful for exploiting spatial locality.

At the bottom of Figure 2.3, we can find the off-chip DRAM, often implemented with faster GDDR5 memory. Each DRAM chip is interfaced through a private on-chip memory controller (MC). Then, each MC talks to its L2 cache slice, which is connected to an interconnection network (usually represented as a crossbar). In the other side of the crossbar we can find all shared multiprocessors (SM) (only one represented for simplicity).

Each SM contains a instruction cache, a L1 data cache, a Texture-cache (TC) for read-only data, a shared memory and a big register file shared by all CUDA cores. Usually the register file is bigger than the other caches combined (see Table 2.1).

2.2 Why CUDA

We selected CUDA over other GPU programming models because it has a faster development and interesting features that we will explain in this section.

[1] NVIDIA classifies their GPUs in relation to the segment of market are made for, for instance, *GeForce* are for gaming, *Quadro* for professional rendering, *Tegra* for low-power chips like mobile phones and tablets and *Tesla* GPUs are for HPC and workstations where double precision is required. But *Tesla* is also the name of the first GPU generation that supported CUDA. Therefore don't confuse the Tesla architecture from the upper part of Figure 2.1 with the Tesla HPC GPUs in the lower part of Figure 2.1.

[2] Originally, Tesla architecture (G80) didn't have support for double precision (Compute Capability 1.1 and 1.2), but in the second revision with the GT200 gpu-core they added double precision support (Compute Capability 1.3).

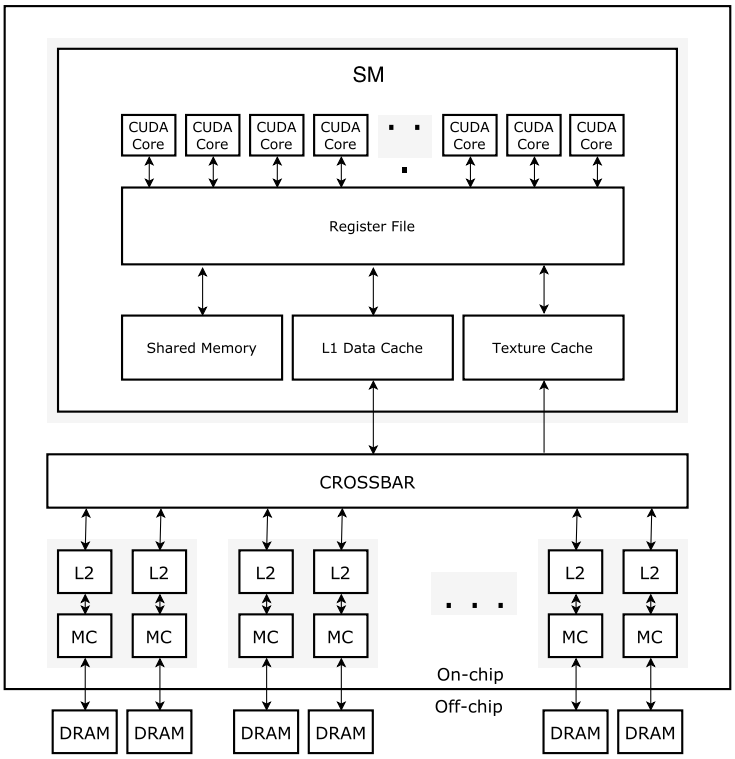

Figure 2.3: Schematic GPU Memory hierarchy.

GPUs have evolved over time and new capabilities that are good for HPC are added. For instance, page migration in newer Pascal GPUs. In a standardized language like OpenCL, whenever there is a big feature that does not conform to the standard, a new revision of the standard has to be made. This may suppose months of discussions, after which hardware manufacturers may decide to not to support the new functionality. In contrast, NVIDIA (having full control of the language) frequently adds new functionalities to CUDA when new GPU architectures are released. CUDA offers high performance and better programmability, while OpenCL offers portability between GPU manufacturers.

From the programmer point of view, CUDA is easier to write because CUDA programmers usually write code with the CUDA runtime API, which provides higher level of abstraction than the CUDA driver API, which is equivalent to OpenCL. Also CUDA is easier than other languages like SIMD intrinsics. Since CUDA is a C++ extension, it supports templates (but not the STL). Also, unlike OpenCL, CUDA kernels are translated into device instructions at compile time, making it easier to catch errors at compile time instead of at run-time.

CUDA offers a more complete software stack including things like GPU-side debuggers, GPU profilers (command line and visual), disassemblers, also use GPU accelerated libraries like the CUDA math library, cuBLAS, cuFFT, cuDNN, cuSPARSE, cuRAND, cuSOLVER, Thrust, nvGRAPH or CUB.

2.3 The CUDA Programming Model

NVIDIA's GPU architecture is coupled with a programming model that extends C/C++ to write high-performance SIMT parallel programs. A CUDA program contains functions that are executed on the CPU, and functions that are executed in the GPU (i.e, *kernels*). CPU code needs to make sure that data is available in the GPU memory before executing a kernel that accesses it. This is performed via

explicit memory transfers that move data through the PCI Express interconnect. In CUDA 6.0, NVIDIA introduced an automatic memory manager called Unified Memory (UM), where the CUDA runtime is in charge of copying data from CPU to GPU automatically (and from GPU to CPU at synchronization points).

A kernel is defined as a multi-dimensional (up to 3D) *grid* of parallel tasks called *thread blocks* that execute the same function (i.e., SPMD model). Each thread block can be scheduled on any available Shared Multiprocessor (SM), in any order, concurrently or sequentially. In fig 2.4 we show an example of the automatic scalability of the CUDA programming language. Suppose a kernel with 8 thread-blocks, if we execute this kernel in a GPU with 4 SMs each SM will execute 2 thread-blocks, and in a GPU with 2 SMs each SM will execute 4 thread-blocks. If we assume fix time to compute each thread block in a SM across the two GPUs, then scaling up the number of SMs will have a positive effect on performance without the programmer having to rewrite anything.

Therefore a CUDA program can be transparently executed in any number of SMs. This property makes the CUDA programming model truly scalable, and allows CUDA programs to be launched on both low-power (mobile) GPUs like the Tegra K1 (1 SM) and GPUs with dozens of SMs like the new Tesla P100 with 56 SMs.

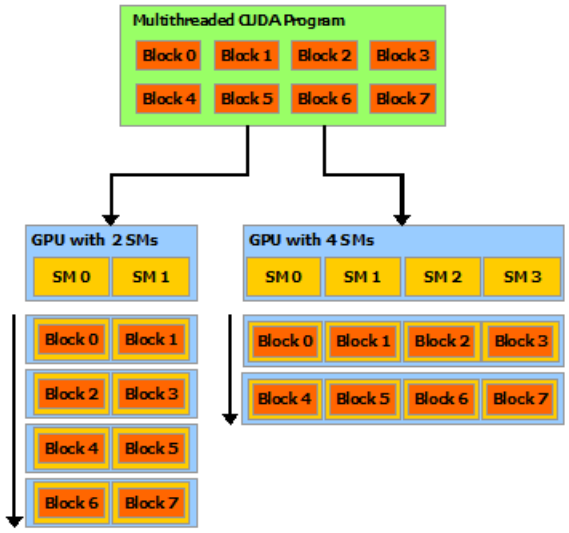

Figure 2.4: Automatic scalability [3]

This intrinsic scalability property is due to the inherent independence between computations in different thread blocks.

To exploit memory locality inside the thread block, CUDA provides access to shared memory, which acts as a cache managed by the programmer. There are also warp-level intrinsics that allow fast communication among threads in a warp without having to use shared memory (e.g., *shuffle intrinsics*).

Kernel execution and DMA transfer operations are asynchronously pushed by CPUs to the GPU using queues of commands called CUDA *streams*. Operations enqueued onto a stream are executed in order, but operations from different streams may be executed in parallel. To help programmers write CUDA programs while not caring about streams, the programming language defines a *default stream* (or stream zero) that is used by default when no other stream is specified (e.g. when using synchronous *cudaMemcpy* or when no stream is provided when launching a kernel).

This default stream has the property of serializing all other operations in other streams. Since asynchronous operation is key in order to overlap CPU and GPU computations or even GPU memory transfers and GPU kernel execution, using multiple streams is necessary to achieve maximum performance in a CUDA program.

2.3.1 Kernels

Kernels are C++ functions defined with the *__global__* specifier. A CUDA kernel can call other functions that have the *__device__* specifier, but a kernel cannot call functions executed in the CPU (functions with *__host__* specifier).

If *dynamic parallelism* is supported by the GPU, then a GPU kernel is able to launch other kernels without going to the CPU. This allows the GPU to generate more work without having CPU intervention. This functionality is available after CUDA 5.0 and for GPUs with compute capability[3] 3.5 or greater.

2.3.2 Thread hierarchy

Threads in a CUDA kernel are organized in a 3D hierarchy to form a one-dimensional, two-dimensional or three-dimensional grid of threads called thread block. Also, as a second layer, threads blocks can be organized in a one-dimensional, two-dimensional or three-dimensional grid of thread blocks. In Figure 2.5, we can see an example 2D grid of 2D thread blocks.

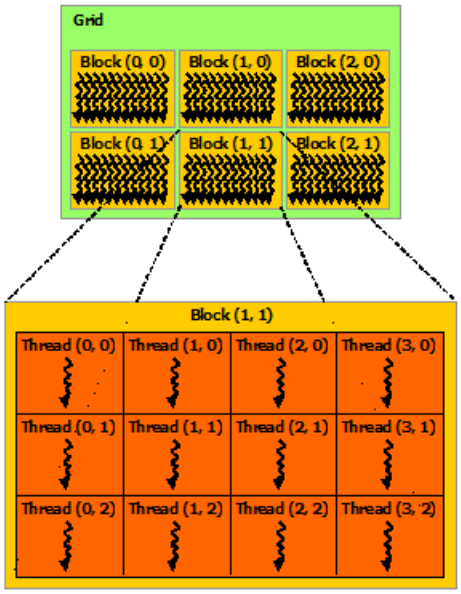

Figure 2.5: Thread hierarchy [3].

This enables CUDA programs to easily map threads into 1D/2D/3D problems. For higher conventionalities, programmers have to rely in sequential *for* constructs.

2.3.3 CUDA Memory Types

CUDA threads have access to several memory spaces:

[3]Compute Capability (CC) is the term used by NVIDIA to classify GPU architectures and their capabilities

- Global memory: Memory resident in GPU Main memory (GDDR) and cached in L2 (L1 caching enabled with compile flags).
- Local memory: Per-thread memory used to store register spills and stack memory. Cached in L1/L2 and physically resident in GPU Main memory (GDDR).
- Shared memory: Per-thread-block memory with high bandwidth used to share data between threads in a thread block (akin to local store in other architectures).
- Texture memory: Read-only memory, allocated in the off-chip memory used by global memory but accessed by dedicated hardware (Texture Functional Units) and cached in the texture cache. Also provides data filtering and interpolation by hardware.
- Constant memory: Read-only memory cached in the constant cache that presents benefits when all threads in a warp load the same data. Usually is implemented using Texture reads, therefore it uses the texture pipeline and is allocated in global memory.

Chapter 3

Experimental Setup and Methodology

This work has been performed mainly in a GPU server called *Hulk*, see Table 3.1. Also to compare the performance, other machines like *Minotauro* or *MareNostrum* have been used.

Hulk initially had four Tesla K40, GPUs that have been used in the first part of this thesis (WARIS-Transport), then the GPUs were upgraded to GeForce Titan X, which implement the later GPU architecture available (Maxwell). In both cases all GPUs are connected to the same PCI-e root complex which enables peer-to-peer communications without CPU intervention.

On the other hand Minotauro (see Figure 3.1) has two NVIDIA Tesla K80 per node. A K80 is a dual-pci-slot GPU that contains two GK210, that has double register file size than a K40 (512 KB/SM vs 256 KB/SM) but in exchange has less SMs per GPU (13 SMs instead of 15) and is programmed as two different GPUs from CUDA. Each GK210 has 12 GB of GDDR5 for a total of 24 GB per K80 card. Since CUDA exposes each GK210 as one "GPU K80", in this document we will show Minotauro as having 4 K80 GPUs per node.

	MareNostrum III	**Minotauro**	**Hulk (WARIS)**	**Hulk (PELE)**
CPU	Intel Xeon E5-2670	Intel Xeon E5-2630 v3	Intel Core i7-4820K	
Architecture	Sandy Bridge	Haswell	Ivy Bridge-E	
Freq	2.60GHz	2.40GHz	3.7GHz	
Cache size	20480 KB	20480 KB	10240 KB	
#cores/socket	8	8	8	
#sockets/node	2	2	1	
GPU card	-	Tesla K80	Tesla K40	GeForce Titan X
GPU Arch.	-	Kepler	Kepler	Maxwell
#GPUs/card	-	2	1	1
#cards/node	-	2	4	4
#GDDR/GPU	-	12 + 12 GB	12 GB	12 GB
RAMM	32 GB DDR3	128 GB DDR4	64 GB DDR3	
DISC	GPFS + SSD	GPFS + SSD	1TB HDD	

Table 3.1: Test machines with HW/node information.

WARIS-Transport (see Table 3.2), requires MPI implementations to support MPI_THREAD_MULTIPLE in order to allow multiple threads calling MPI at the same time. CUDA 7.5 was used, and the compilation flags were set to *-O3 -march=native*. In PELE (see Table 3.3) was compiled using a combination of

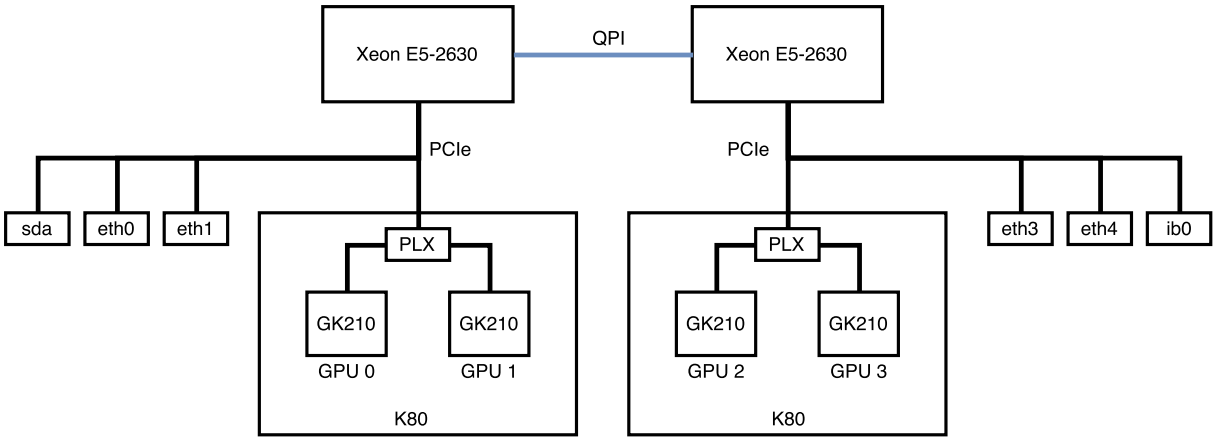

Figure 3.1: Minotauro K80 node configuration.

g++-4.9.3 and nvcc. In both cases, *gprof* was used as the CPU profiler and nvprof as the GPU profiling tool.

	Marenostrum	**Minotauro**	**Hulk**
MPI wrapper	impi-5.1.2.150	openmpi-1.6.5-mt	mvapich-2.0-mt
CPU compiler	icc-13.0.1	gcc-4.9.1	gcc-4.9.3
GPU compiler		nvcc-7.5.17	nvcc-7.5.17
CFLAGS	-O3	-O3 -march=native	-O3 -march=native
Dependences	Version		
HDF5	1.8.12	1.8.12	1.8.15
PNETCDF	1.7.0	1.7.0	1.6.0
NETCDF	4.3.3.1	4.3.3.1	4.3.3.1
SZIP	2.1	2.1	2.1

Table 3.2: WARIS-Transport Compilation details.

	Minotauro	**Hulk**
CPU compiler	g++-4.9.3	g++-4.9.3
GPU compiler	nvcc-7.5.17	nvcc-7.5.17
CUDA Runtime	7.5	7.5
CXXFLAGS	-O3 -march=native	-O3 -march=native
Dependences	Version	
BLAS	1.0	3.6.0
LAPACK	3.3.1	3.6.0
BOOST	1.57	1.58
cryptopp	5.6.2	6.0.0
jsoncpp	1.7.2	1.7.2
wjelement	1.0	1.0

Table 3.3: PELE Compilation details.

To measure results in WARIS-Transport we used the internal timer that WARIS Framework uses that reports the total executing time of the application. Then, as only one execution in the CPU can take hours we decided to only run 4 executions of each WARIS-Transport implementation and then report the mean of those executions.

In PELE to report the execution time we used */usr/bin/time -f=%e* to calculate the elapsed time of each execution and then calculate the average execution time.

Chapter 4

WARIS-Transport

4.1 Introduction

WARIS-Transport is a high-performance implementation (written in C) of a well-known industry standard application for volcanic ash transport simulation named FALL3D which is already in production in the Buenos Aires VAAC [7].

In 2010 the Eyajfajallajokull volcano (in Iceland) erupted causing a major disruption of the European airspace: more than the 48% of the flights were canceled provoking 1.3 billion euros cost to airline industry.

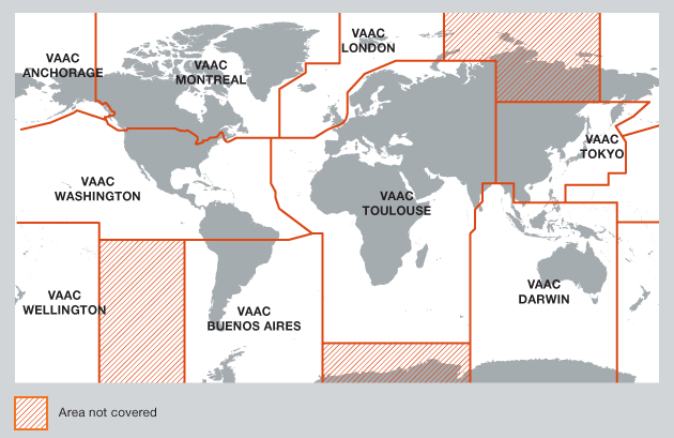

Figure 4.1: Volcanic Ash Advisory Centers [4].

The goal of WARIS-Transport is to do forecasting of atmospheric transport and deposition of volcanic ashes. This type of application is mainly used by the 11 Volcanic Ash Advisory centers (see Figure 4.1), that are in charge of monitoring volcanic eruptions and help airliners redirect their flights to avoid the ash clouds of volcanic eruptions.

WARIS-Transport sits on top of the WARIS framework, a Barcelona Supercomputing Center (BSC) in-house multi-purpose framework focused at solving scientific problems using Finite Differences (FD).

WARIS-Transport can be classified as an Earth Simulation application that does Computational Fluid Dynamics (CFD) computations using FD on structured meshes.

4.2 Application Structure

Algorithm 1 shows the control flow of typical FD applications using structured meshes. The first step is to divide the computational domain into smaller chunks that will be allocated to each CPU. Time is discretized in Δ steps. For each step, input data is read (from disk or memory, etc..) and *injected* into the structured mesh (line 4). If necessary this data is pre-processed (line 5). Then boundary conditions have to be applied to ensure correctness (line 6). For all discretized points a computation called *stencil* is performed (line 8). This step is the most computationally expensive of the whole algorithm. If the domain has been distributed using MPI, in line 10 overlap points between neighbor domains (from different MPI ranks) are exchanged otherwise communication is done via shared memory. To finalize (if necessary) data is post processed and written to disk.

Algorithm 1 General control flow structure of a generic FD code.

```
1:  procedure WARIS
2:      Structured mesh Domain decomposition
3:      for t = time_start to time_end in Δt steps do          ▷ Loop over time
4:          Read input for t
5:          Pre-process input
6:          Apply boundary conditions
7:          for all discretized points do
8:              Stencil computation
9:          end for
10:         Exchange overlap points between neighbour domains
11:         Post-process output
12:         Write output for t
13:     end for
14: end procedure
```

4.3 The WARIS Framework

WARIS [5] is an in-house BSC framework aimed at solving finite differences problems. The goal of this framework is to reuse as much as possible code from different FDM problems and optimizations, and to, facilitate the adaptation to new hardware and devices.

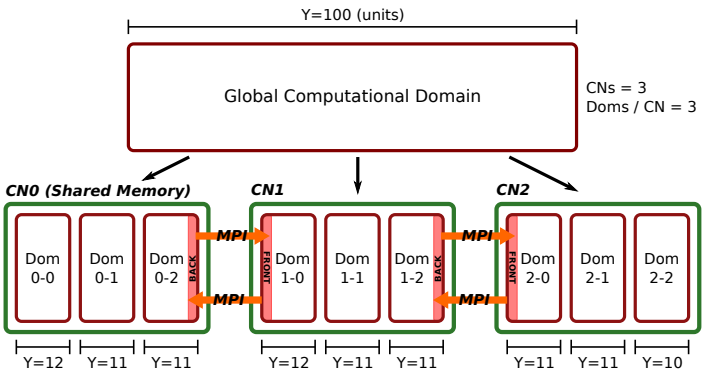

Figure 4.2: WARIS computational domain distribution with MPI+pthreads [5]. (CN= Computational Node)

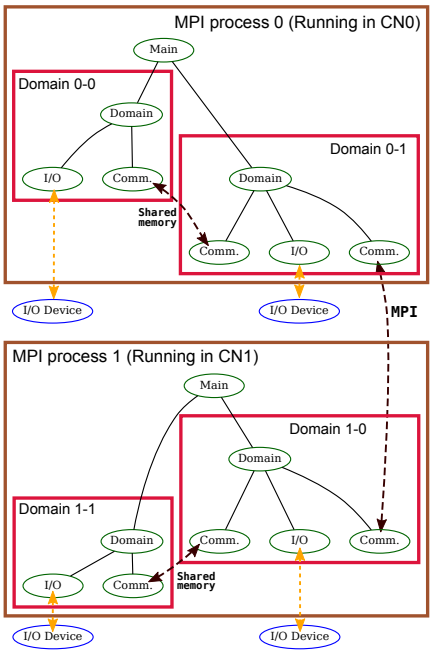

Figure 4.3: WARIS communication between nodes [5].

The WARIS framework uses a combination of MPI to distribute work between nodes, and posix threads to fully occupy a compute node. In Figure 4.2 we see an example problem of 100 units decomposed in three MPI ranks with three threads each, for a total of nine threads working on the whole problem. The WARIS framework tries to exploit the SMT capabilities of CPUs by creating multiple pthreads per MPI rank, eventually oversubscribing the node. Each MPI rank has a master thread that controls the shared memory domains (see Figure 4.3), and each shared memory domain contains a compute thread, two communication threads and a IO thread.

4.3.1 The stencil computation

Stencil computations are the computations that perform the time propagation in FD simulations.

Typical stencils work with one, two or three dimensional grids. In each time-step, the stencil updates all grid elements using the neighboring elements as a pattern. This computational pattern, is called *the stencil* and largely depends on the shape of the PDE formula. (e.g., in Algorithm 2 we can observe a two dimensional 5 point stencil)

Algorithm 2 A stencil example: 2D 5p-stencil.

1: **procedure** STENCIL(unew, u)
2: **for** $i = halo\_size$ to $sizex - halo\_size$ **do**
3: **for** $j = halo\_size$ to $sizey - halo\_size$ **do**
4: $unew_{i,j} = u_{i,j} + u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}$
5: **end for**
6: **end for**
7: **end procedure**

In Figure 4.5, we can observe the data dependences of the stencil shown in Algorithm 2 using a 3x6 grid. In this case, the 5-point stencil has one data dependence up, down, right and left, therefore the grid has some exterior elements called *halos* that are maintained to fulfill data dependences.

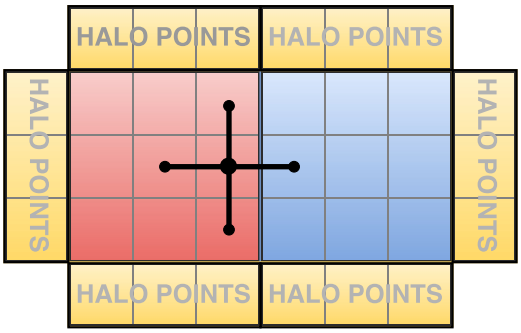

Figure 4.4: A 2D 5p-stencil computation.

4.3.2 Parallel stencil decomposition

Parallel stencil implementations using MPI (like the one used in WARIS-Transport) introduce the problem of managing data dependences between neighboring ranks. Those neighboring points are usually called *halos*, and have to be communicated between MPI ranks at each step to ensure correct execution. In Figure 4.4 we can observe an example of a 2D stencil parallelization using two MPI ranks. Firstly, each MPI rank computes his own piece of halos, then while MPI halo communication is ongoing the rest of the stencil is compute achieving an overlap in time between MPI communication and execution.

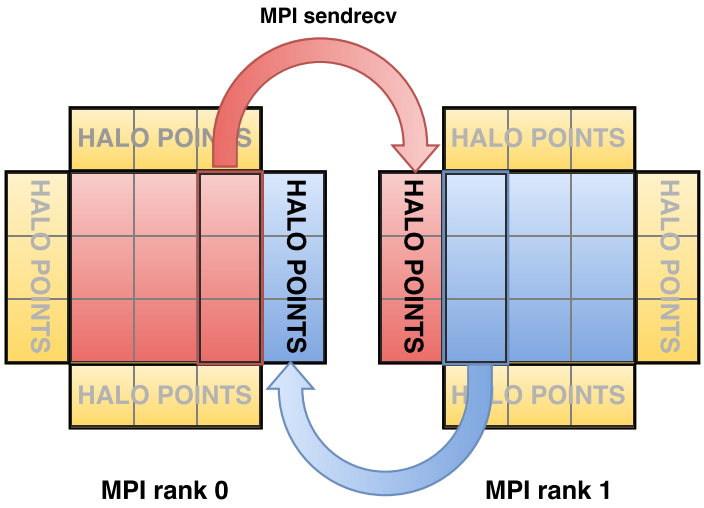

Figure 4.5: Halo exchange between MPI ranks

WARIS-Transport solves a general transport equation, also called General Advection Diffusion Reaction Equation in a cartesian (x,y,z) grid using FD (for a more in-depth discussion of WARIS-Transport equations see Appendix A).

Therefore, the WARIS framework follows the typical FD algorithm but with specific computations related to the problem at hand. In Algorithm 3 (a more detailed version of Algorithm 1), we can observe the general algorithm of WARIS-Transport. Basically, the application starts reading meteorological (line 4) and configuration data (line 2 and 3) and time is discretized (line 4). Then, the main loop starts. As long as the the eruption is ongoing, source injection data is read from meteorological database (line 14) and the term is scaled by a factor (line 15). After that (and if there is the need to update the meteorological information) meteorological data is read from the meteorological database (in NetCDF format) and several

meteorological computations are launched (from line 18 to 29). Then, boundary conditions are applied and the advection-diffusion-sedimentation function is launched for each particle class in line 32. To finalize, several post-process calculations are done (lines 35 to 38).

Algorithm 3 WARIS-Transport Algorithm [8].

1: **procedure** WARIS-TRANSPORT
2: Read particle properties from granulometry file
3: Read time and configuration variables from input file
4: Read meteo database and grid data from NetCDF file
5: Compute grid increments
6: Compute time-lag between the time origin of the database and the time origin of the input data
7: Allocate memory arrays
8: Read topography from meteo file
9: Compute scaling factor
10: Read output strategy
11: **for** $t = time_{start}$ to $time_{end}$ in Δt steps **do** ▷ Loop over time
12: **if** $source\_time \leqslant t$ **then**
13: **for** $p = particle_0$ to $particle_n$ **do**
14: Read the source term from source file (S^{np})
15: Scale source term (S_*^{np})
16: **end for**
17: **end if**
18: **if** $meteo\_time <= time$ **then**
19: Read meteorological variables from the meteo file. ($u_{x,y,z}, T, p, etc.$)
20: Update meteo_time
21: Initialize boundary conditions
22: Compute horizontal and vertical diffusion coefficients ($K_{x,y,z}$)
23: **for** $p = particle_0$ to $particle_n$ **do**
24: Compute the terminal settling velocity (V_{sj})
25: **end for**
26: Compute scaled quantities
27: Calculate the critical time step (Δt).
28: AllReduce of Δt
29: **end if**
30: Set the boundary conditions (C)
31: **for** $p = particle_0$ to $particle_n$ **do**
32: Calculate ADS stencil (move in time) (C^{t+1}) ▷ Most expensive part
33: Exchange overlap points between neighbor domains
34: **end for**
35: Compute ground accumulation of ash (C_{accu})
36: Compute mass lost at boundaries
37: AllReduce of mass lost
38: Mass balance correction
39: Output results
40: End time step. Swap input (C^t) and output matrices (C^{t+1})
41: **end for**
42: **end procedure**

Another remarkable point of WARIS-Transport is that particles of different granularities are discretized in different standardized particle sizes (named *particulate matter* or PM). Therefore, stencil computations

working with particles of different granularities are independent of each other, enabling concurrent stencil computations.

4.4 Initial profiling

To assert the total execution time of each part shown in Algorithm 3 we profiled WARIS-Transport using *gprof*. The initial profiling (for one MPI process) is shown in Table 4.1. The application was compiled with -O3 and with AVX enabled (meaning it's not a naive sequential implementation). The dataset used is from the Cordón-Caulle volcano with dimensions 121x121x30 and 8 particle types.

%	Time (s)	#calls	Name
79.81	6778.74	54648	device_maingrid_adv
6.62	561.89	236012920	device_gama
5.79	491.71	6831	device_mass
2.49	211.12	236012920	device_vsettling
2.31	196.41	40986	device_abc_zgnc
2.04	172.89	6831	device_divcorr
0.44	37.21	6831	device_groundaccum
0.25	20.95	65	device_meteo_vsettling
0.10	8.07	66	device_prepare_io
0.04	3.75	65	device_stabdt
0.03	2.78	65	device_meteo_scale
0.02	1.97	390	device_abc_mirror
0.02	1.28	528	device_cutflevel
0.01	1.16	65	device_meteo_precompute
0.01	0.85	390	device_permvolume
0.01	0.48	65	device_meteo_setvdrydep
0.01	0.47	147507893	device_fEIDX
0.01	0.46	147507897	device_fMIIDX
0.00	0.38	65	device_meteo_divu
0.00	0.31	1	eapd_transport_read_meteo
0.00	0.13	65	device_meteo_hdiffusion
0.00	0.13	65	device_meteo_vdiffusion
0.00	0.10	29501615	device_fFIDX
0.00	0.08	6831	device_src
0.00	0.08	1	device_init_fields
0.00	0.08	29501615	device_fIDX
0.00	0.04		device_fMIDX
0.00	0.03	65	device_meteo_terrvel
0.00	0.02	20493	eap_phase_proc
0.00	0.01	6831	device_maingrid
0.00	0.01	6831	eapd_transport_prepare_meteo
0.00	0.01	6831	eapd_transport_src
0.00	0.01	65	eapd_transport_read_source
0.00	0.01	11	eapd_transport_meteo

Table 4.1: WARIS-Transport CPU profile using *gprof* in Minotauro (O3+AVX) and the Caulle-0.25-8bin 121x121x30 dataset.

As expected, the ADR calculation in function *device_maingrid_adv* is the most expensive part of the

application, with 79.81% of the execution time. The second most expensive function is the aggregate of *device_meteo_vsettling* with their children, *device_vsettling* and *device_gamma* with an aggregated time of 9.36%. Other important functions are the absorbing boundary conditions (*device_abc*) with a 2.31% and *device_divcorr* wiht a 2.04%.

4.5 The Advection Diffussion Reaction

As we have seen in section 4.4, the most important part in terms of execution time is the advection-diffusion reaction function (ADR).

The ADR function computes a 3D stencil over a Cartesian grid.

ADR computes the transported quantity (C) in a Δt step, the algorithm reads the fluid velocities in each component ($V(x,y,z)$, $V(x,y,z)$ and $W(x,y,z)$), the turbulent diffusion coefficients ($K_x(x,y,z)$, $K_y(x,y,z)$, $K_z(x,y,z)$) and the source term ($S(x,y,z)$).

The overall ADR stencil computation is equivalent to a 3D 13-point stencil for C (Figure 4.6) plus a 3D 12-point stencil for W (Figure 4.7) plus a 3D 7-point stencil for each K (Figure 4.8).

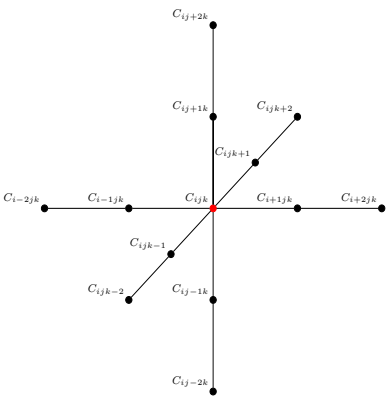

Figure 4.6: C 13-p stencil

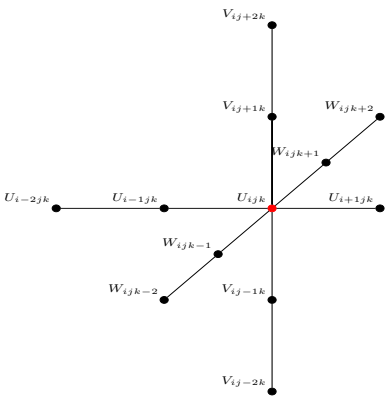

Figure 4.7: $U/V/W$ 12-p stencil

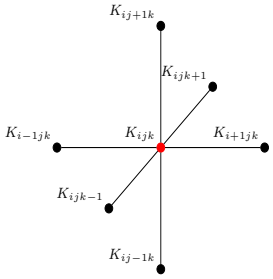

Figure 4.8: K 7-p stencil

The arithmetic intensity is the ratio of floating operations (in FLOPs) done in a piece of data versus the cost it takes to bring all necessary data through the memory hierarchy (in bytes). Applications with lower arithmetic intensity (measured in FLOPs/byte) will be more prone to be limited by the memory subsystem (therefore called memory-bound), than applications with high arithmetic intensity, that will be limited by the amount of computational power available in the hardware (typically the number of CUDA cores in the case of NVIDIA GPUs).

In fact, we can estimate if a naive ADR implementation would be compute bound or memory bound. Knowing that the ratio of input:output points is 32:1, and this is 128 bytes/point in single precision, and a naive implementation of ADR would have 250 floating point operations (FLOPs) per output point). We can compute the arithmetic intensity of a naive ADR kernel as $\frac{250 FLOPs}{128 bytes} = 1.95 \frac{FLOPs}{byte}$ which puts ADR in the upper band of stencils (stencils typically have 0.1~1.0 FLOPs/byte).

A Roofline Performance Model [9] is a visually intuitive way of bounding the performance achieved by a code in a certain architecture. In Figure 4.9 we can observe the roofline model of a Tesla K40, where the dashed blue line represents the theoretical performance limit using global memory (288 GB/s), the solid green line represents the theoretical performance limit using only shared memory (aggregated

bandwidth of 2860.8 GB/s). The vertical black line represents the arithmetic intensity projection of the ADR kernel.

In Tesla K40 the peak performance in single precision is 5040 GFLOPs, therefore kernels with less than $\frac{5{,}04x10^{12}FLOP/s}{3{,}09x10^{11}byte/s} \approx 16\frac{FLOPs}{byte}$ arithmetic intensity will be memory bound. Since $1.95 < 16$, we can estimate that our naive implementation of the ADR kernel would be memory-bound. Therefore a good memory management is key to achieve peak performance in ADR computations.

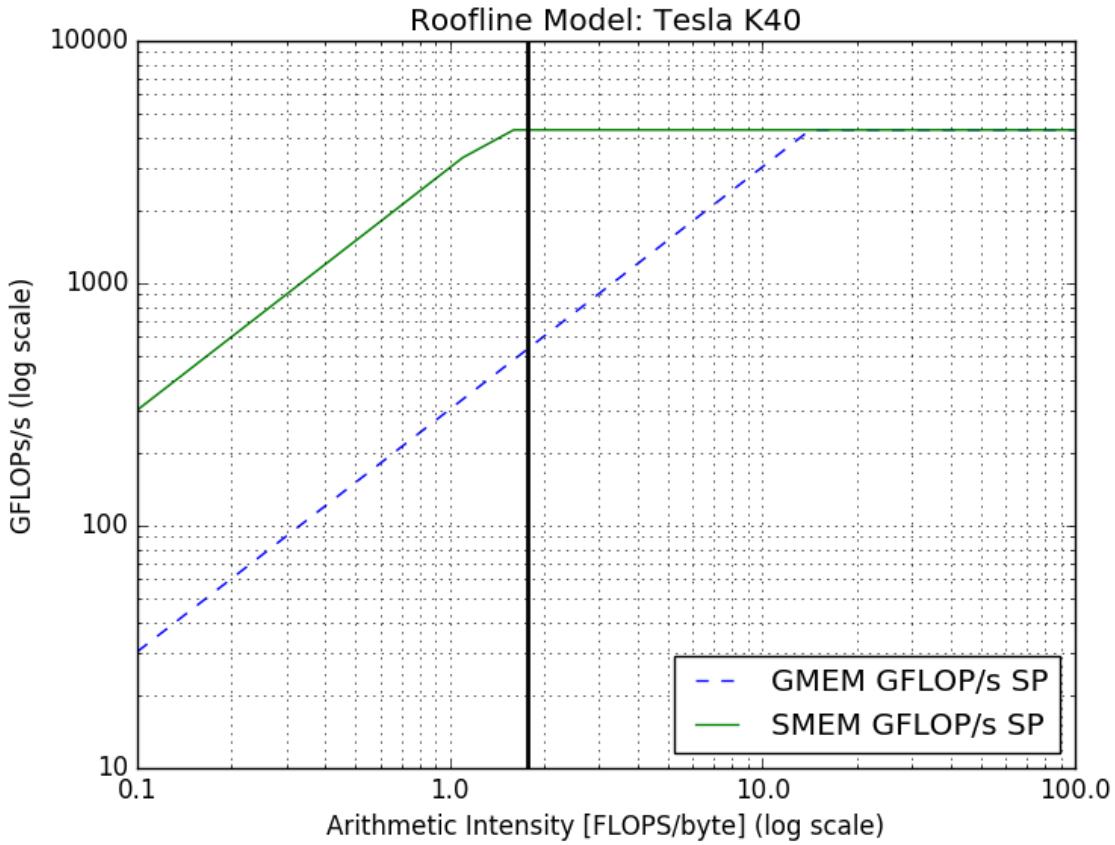

Figure 4.9: Roofline Model of NVIDIA Tesla K40 with WARIS-Transport

4.6 CUDA Implementation

In this section we present the CUDA implementation we have done. Based on the profiling previously done, we have ported to CUDA the ADR and vsettling kernels. Then, after some post-analysis we decided to reduce the number of memory copies between CPU & GPU by moving all remaining computations to the GPU.

At the end we end up with all major computations implemented in the GPU, improving the overall execution time. One MPI rank per GPU was used, therefore getting a multi-GPU implementation scaling the number of MPI ranks with the benefit that the WARIS framework is in charge of distributing the correct data between neighbors (GPUs).

Implementing just the ADR computation in the GPU and executing the rest in the CPU had two problems from the performance point of view:

1. Overheads. Although we accelerate the ADR kernel, CPU to GPU and GPU to CPU copies have to be performed to maintain the data coherent across the CPU & GPU domain. This is necessary to ensure that functions using the results from the ADR kernel have correct inputs.

2. Ideally (without counting the previous overheads) the achieved performance is bounded by the amount of parallelized task. Applying Amdahl's law, if we were to accelerate ADR by an ∞ factor, we would get a maximum speedup of:

$$Speedup_{ADR-ideal} = lim_{s->\infty} \frac{1}{(1 - 0.7981) + \frac{0.7981}{s}} = \frac{1}{1 - 0.7981} = 4.95x$$

Therefore, we decided to accelerate not only ADR, but all major non-IO computations with GPU kernels, in part, to minimize data movement from CPU & GPU and to enable greater speedups in this application.

Some computations are independent and can be done in parallel thus we used multiple CUDA streams to enable concurrent execution in the GPU.

4.6.1 ADR Kernel

To accelerate the ADR stencil with CUDA some of the techniques from Micikevicius et. al. [10] were used. All threads in a thread-block cooperate to compute the stencil using several techniques like register tiling, where data is loaded from global memory into registers, and is moved from register to register until is no longer needed, and shared memory to increase overall performance.

The ADR kernel uses a 2D thread-block decomposition mapped into the XZ plane.

The first optimization we used is to declare all read only pointers as $const\_\_restrict\_\_$, helping the compiler to further optimize the code. In Kepler or later architectures, this optimization enables reading from the texture pipeline (and the texture cache) without having to use texture objects. In memory-bound kernels, optimizations like this one are very important because it improves the overall memory bandwidth available to every thread from GPU main memory since you are exploiting the L1-L2 bus for read-write data and the Texture Cache-L2 bus for read-only data (see 2.3).

In a stencil, each point is visited by several threads, then to reuse data between threads we used shared memory. But when loading from global memory to shared memory we have to face the problem of loading the stencil neighboring points (or halos), being the two major solutions:

Given a 2D thread block of (x, y) threads:

A) Allocate a shared memory region of $(x + 2 * halo\_size, y + 2 * halo\_size)$ elements and then make the exterior threads in a thread-block load halos from global memory to shared memory. Then the stencil computations is done entirely with data resident in shared memory.

B) Allocate a shared memory region of (x, y) elements and then read stencil halos from global memory while inner points are read from shared memory.

In our implementation the second scheme is used because it's simpler since we don't have to load those points into shared memory.

Another key point when writing CUDA kernels is that general performance highly depends of the number of thread blocks in flight in the SM. This is measured with a metric called *occupancy*. The

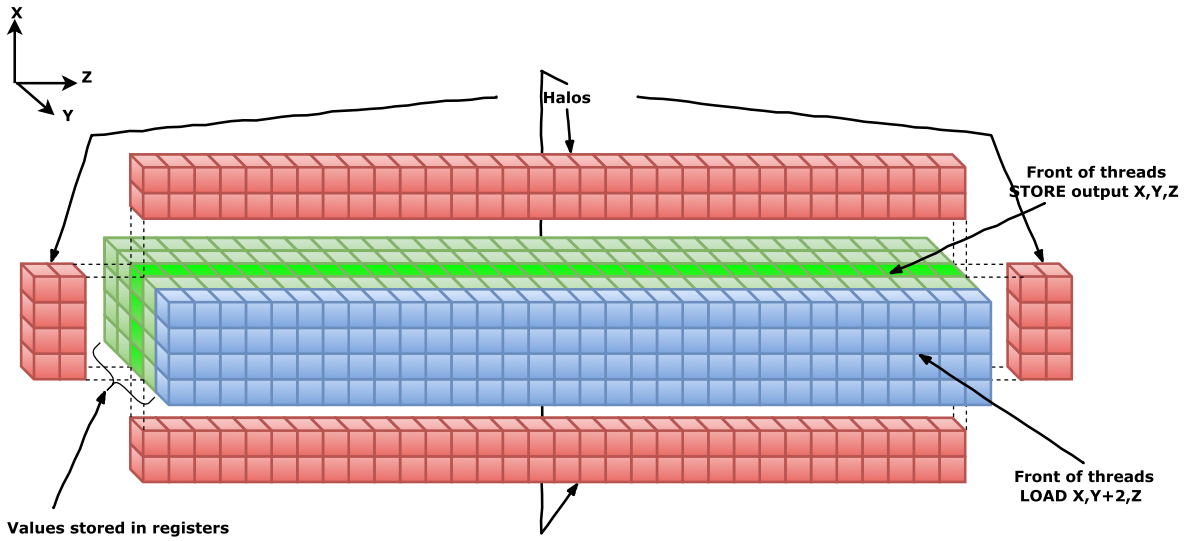

Figure 4.10: ADR Kernel: Thread block with front of threads that read from gmem and store in registers.

maximum number of thread blocks in a SM is a function of different SM resources like the register usage, shared memory usage and block size shape because those resources are shared between all threads in a SM.

Therefore optimized the number of registers used in this kernel, including not using register tiling for all input volumes, for instance, we use register tiling for C, K and Rho but volumes U, V and W are loaded directly from global memory to save registers. Because sometimes small changes in register usage (for instance from 33 to 32) means that instead of 8 resident thread blocks, 16 thread blocks can be scheduled, improving occupancy and performance.

In Figure 4.10 we can observe a graphic representation of all data required to compute a slice of Y. We make use of colors to help understand the Figure. At each iteration of Y, the front of threads (blue color) loads input values of two Y slices forward ($Y + 2$) from global memory into registers.

Then each thread performs a register tiling and store X, Y, Z data into shared memory.

Finally threads in the same warp (Z dimensions) exchange data using *shuffle intrinsics*. If no neighbor is available then halos are read directly from global memory. In Y dimensions, data is read from shared memory. Again if no neighbor is available, data is read directly from global memory.

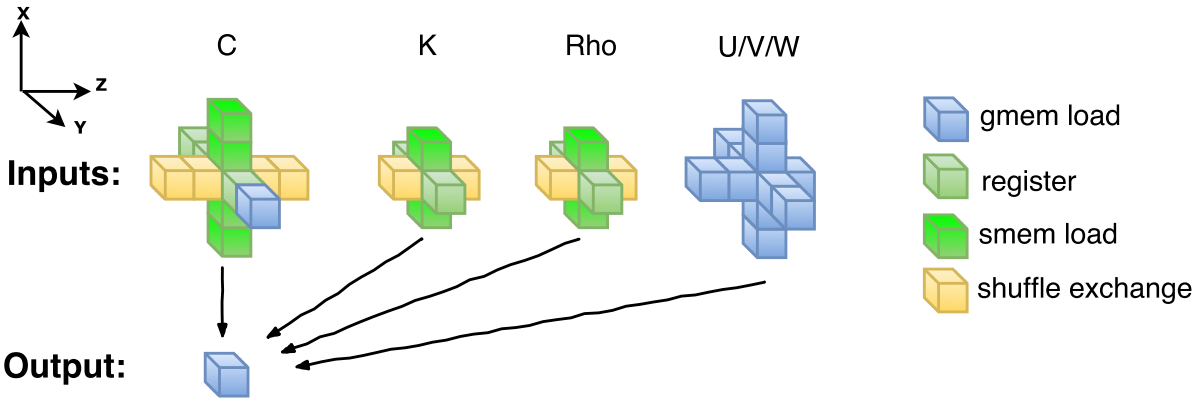

Figure 4.11: ADR Kernel: Example of Thread (i, j) computing output point $C_{x,y,z}$.

In figure 4.11 we can observe a more detailed view for a thread. Inputs for volumes C, K, Ro and

$U/V/W$ are read to produce a unique output point of C. Given the amount of 3D volumes this kernel has to load from memory, we were forced to load $U/V/W$ volumes from global memory to not use too much registers and not to hurt performance.

4.6.2 Vsettling Kernel

The second most important function in WARIS-Transport is *device_gama* + *device_vsettling* that accounts for a total of $6.62\% + 2.49\% + 0.25\% = 9.36\%$ of the original application. This function calculates the particle settling velocity v_s where particles settle down at their terminal velocity. For calculating v_s this formula is used:

$$v_s = \sqrt{\frac{4g(\rho p - \rho\alpha)d}{3C_d\rho\alpha}} \tag{4.1}$$

Where ρ_a and ρ_α denote air and particle density, respectively, d is the particle-equivalent diameter, and C_d is the drag coefficient calculated using the GANSER [11] formula.

Unfortunately in WARIS-Transport, the GANSER formula is implemented as an iterative process that tries to converge to the best possible v_s. This is not optimal in GPU code, because introduces *thread divergence*[1] because some threads converge before than others. But to maintain correctness between CPU & GPU the same algorithm had to be implemented in the GPU.

Therefore the vsetting kernel was implemented using one thread per particle, where each thread has to iterate to find the best v_s for the particle assigned to it. As an optimization, read-only data was declared as $const * \_\_restrict\_\_$ to let the compiler use texture cache while writes go to the L2 cache.

4.6.3 Other Kernels

To minimize CPU&GPU copies copies all non-IO computations were implemented as CUDA Kernels in the GPU. This includes functions present in the main loop such as *device_mass*, *device_divcorr*, *device_groundaccum*, all meteorologic functions *device_meteo_*** and some parts of *device_prepare_io*.

4.7 Results

The datasets used in this section are shown in Table 4.2. The Caulle-0.25 dataset is the same used in the initial profiling and is used again for the GPU profiling in section 4.7.1. The Caulle-0.05 dataset has bigger resolution and is used fro the CPU vs GPU evaluation in section 4.7.2.

Dataset Name	Size	#particles	# Total nodes
Caulle-0.25	121 x 121 x 30	8	3.5M
Caulle-0.05	601 x 601 x 64	8	184.9M

Table 4.2: Volcano Datasets.

4.7.1 Final profiling with nvprof

In Table 4.3 we can observe the profiling of all GPU kernels using the Caulle-0.25-8bin 121x121x30 dataset, the same used on the initial profiling. We should warn to the reader that this profiling only

[1] In SIMT programs we say that we have thread divergence when not all threads in a warp take the same branch condition (in an if/else statement), then each part has to be replayed with the correspondent threads.

%	Total Time	#Calls	Kernel Avg. Time	Name
67.06	101.386s	54648	1855.30 us	device_maingrid_adv_kernel
10.66	16.1217s	520	31003.00 us	device_meteo_vsettling_kernel
4.75	7.18898s	54648	131.55 us	device_divcorr_kernel
3.75	5.67294s	54648	103.81 us	device_abc_zgnc_top_kernel
3.35	5.05788s	54648	92.55 us	device_abc_zgnc_bottom_kernel
3.17	4.79063s	54648	87.66 us	device_mass_voldepmass_kernel
2.14	3.23239s	54648	59.14 us	device_abc_zgnc_right_kernel
1.17	1.76681s	54648	32.33 us	device_abc_zgnc_left_kernel
0.90	1.36253s	3120	436.71 us	device_abc_mirror_kernel
0.67	1.01662s	54648	18.60 us	device_groundaccum_kernel
0.52	792.51ms	54648	14.50 us	device_mass_outmass_bottom_kernel
0.32	476.80ms	54648	8.72 us	device_abc_zgnc_front_kernel
0.26	386.79ms	54648	7.07 us	device_abc_zgnc_back_kernel
0.24	363.79ms	520	699.59 us	device_stabdt_kernel
0.19	282.81ms	54648	5.17 us	device_mass_outmass_leftright_kernel
0.13	202.23ms	54648	3.70 us	device_src_kernel
0.13	197.19ms	164141	1.20 us	[CUDA memset]
0.13	191.89ms	62328	3.07 us	[CUDA memcpy HtoD]
0.10	157.02ms	520	301.95 us	device_meteo_scale_part2_kernel
0.10	156.96ms	198	792.72 us	device_io_accum_c2_idx_kernel
0.10	155.48ms	528	294.46 us	device_cutflevel_kernel
0.05	76.511ms	21416	3.57 us	[CUDA memcpy DtoH]
0.04	58.662ms	66	888.81 us	device_io_accum_c2_idx_kernel
0.02	29.548ms	264	111.92 us	device_io_accumz_kernel
0.01	21.758ms	65	334.74 us	device_meteo_precompute_kernel
0.01	18.899ms	520	36.34 us	device_meteo_vdrydep_kernel
0.01	15.101ms	65	232.32 us	device_meteo_scale_part1_kernel
0.01	7.7082ms	65	118.59 us	device_meteo_divu_kernel
0.00	3.3987ms	462	7.35 us	device_io_thick_b_kernel
0.00	2.5924ms	65	39.88 us	device_meteo_terrvel_kernel
0.00	1.4124ms	65	21.72 us	device_meteo_vdiffusion_const_kernel
0.00	1.3494ms	65	20.76 us	device_meteo_hdiffusion_const_kernel
0.00	484.98us	66	7.34 us	device_io_thick_a_kernel
0.00	325.15us	66	4.92 us	device_io_thick_c_kernel
0.00	261.27us	65	4.01 us	device_meteo_prate_kernel

Table 4.3: WARIS-Transport GPU profile with *nvprof* using 1 K80 in Minotauro. Caulle-0.25-8bin 121x121x64 dataset.

accounts for the execution time of kernels in the GPU and therefore is not representative of the time spent in the CPU.

We can see that *device_maingrid_adv_kernel* is responsible for the 67.06% of executing time in the GPU, a little bit less than the original 78.81% in Table 4.1. One possible explanation to this change is that our ADR kernel is more optimized than other kernels and therefore the relative weight of ADR is diminished.

Another important thing to remark is how huge device_meteo_vsettling_kernel is with respect to the other kernels; only 520 calls and accounts for 10% of the exec time. Another important thing we achieved is that memory coherency operations (shown as *CUDA memcpy HtoD* and *CUDA mempcy DtoH*)

represents less than a 0.18% of GPU executing time.

We also calculated the speedups achieved for each CUDA kernel (see table 4.4) compared to their original implementation (compiled with *gcc* and flags -O3 and AVX vectorization enabled).

Seq. Time (s)	Kernel Time (s)	Kernel Speedup	Name
6778.74	101.38	66.86	device_maingrid_adv
793.96	16.12	49.25	device_meteo_vsettling
491.71	5.86	83.85	device_mass
196.41	16.57	11.85	device_abc_zgnc
172.89	7.18	24.07	device_divcorr
37.21	1.01	36.62	device_groundaccum

Table 4.4: Speedups of most important kernels using 1 K80 in Minotauro. Caulle-0.25-8bin 121x121x64 dataset.

4.7.2 CPU vs GPU Implementation

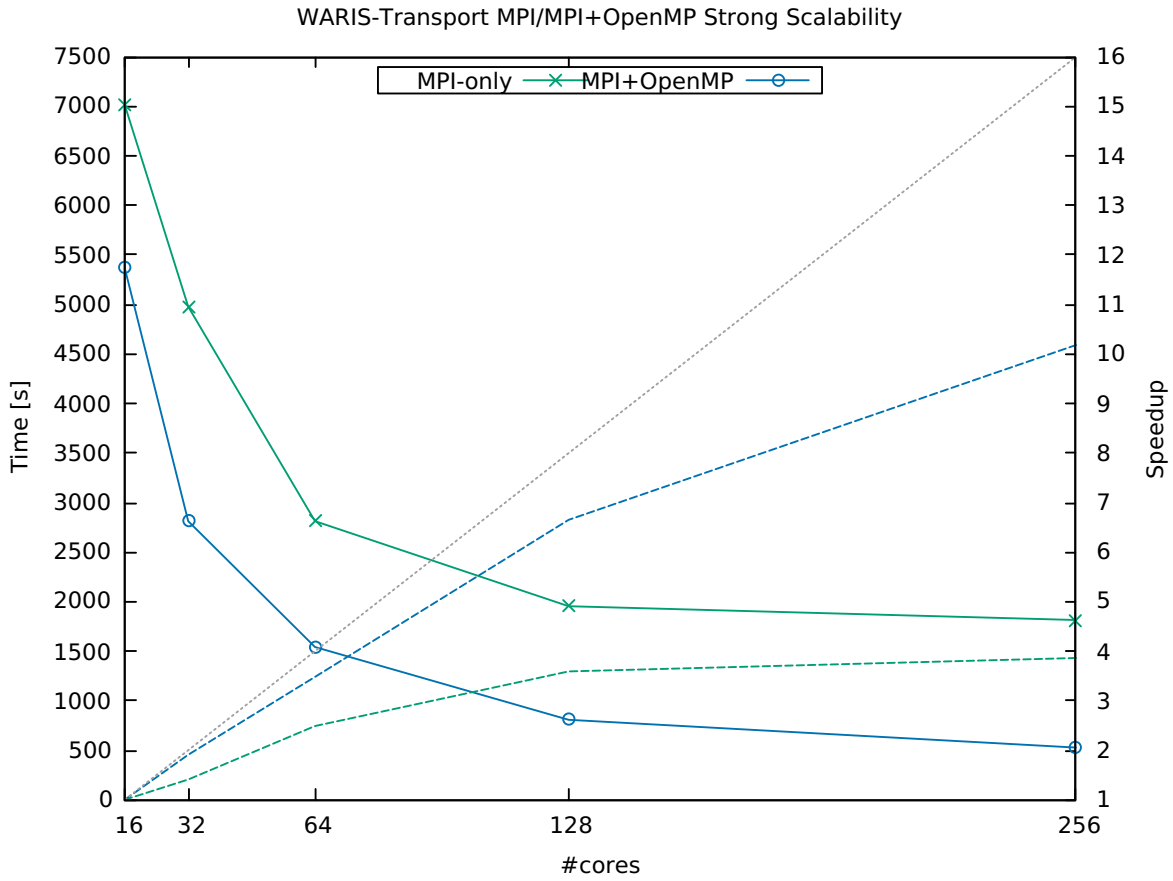

Figure 4.12: WARIS-Transport Strong Scalability for MPI+AVX (1 MPI rank/core) and MPI+OpenMP+AVX (1 MPI rank/node, 1 thread/core) implementations in MareNostrum. Caulle-0.05-8bin dataset (601x601x64 x 8).

In Figure 4.12 we show the execution time (solid lines) and speedups (dashed lines) of the two CPU implementations. The biggest dataset available from the Cordón-Caulle volcano (601x601x64 x 8). Since a WARIS-Transport sequential execution with the biggest dataset in MareNostrum was expected to last +24h, we considered a dual-socket MareNostrum node (two Intel Xeon CPU E5649, 8+8 cores) as the speedup baseline.

#cores	Marenostrum	
	Intel Xeon CPU E5649 @ 2.53GHz	
	MPI-only	MPI+OpenMP
	Execution Time (Speedup)	
16	7015.67 (1.00)	5368.19 (1.00)
32	4978.05 (1.41)	2812.86 (1.91)
64	2812.71 (2.49)	1541.76 (3.48)
128	1954.16 (3.59)	806.81 (6.65)
256	1815.24 (3.86)	527.38 (10.18)

Table 4.5: WARIS-Transport Execution Time for MPI+AVX (1 MPI rank/core) and MPI+OpenMP+AVX (1 MPI rank/node, 1 thread/core) implementations in MareNostrum. Caulle-0.05-8bin dataset (601x601x64 x 8).

All implementations have been compiled with AVX vectorization (intrinsics). The first implementation only uses MPI with 1 MPI rank/core to scale from a two socket node (16 cores) to 16 dual-socket nodes (256 cores). The second implementation uses MPI+OpenMP (+AVX) to exploit all parallelism available. In this case we use only 1 MPI rank/node and 1 OpenMP thread/core. For completeness, in Table 4.5 we show the exact execution time and speedup achieved. We believe that the performance difference is due to the fact that WARIS framework creates 5 POSIX threads per MPI rank (Master, IO, CommUp, CommDown and Computation). Then the MPI-only implementation suffers from over-subscription (more threads than cores).

We can see that the MPI-only implementation scales slowly, and the parallel efficiency (rated as $Par_{eff} = \frac{Speedup}{Speedup_{ideal}}$) falls to 70% with just 32 cores (2 nodes). We also see that the MPI+OpenMP scales better since with 16 nodes (256 cores) we still see execution time improvements but the parallel efficiency falls to 63,61%.

#GPUs	Hulk	Minotauro
	Tesla K40	Tesla K80
	MPI+GPU	
	Execution Time (Speedup)	
1	2917.76 (1.00)	4192.43 (1.00)
2	1632.70 (1.79)	1999.26 (2.09)
3	1027.88 (2.84)	1492.30 (2.81)
4	809.44 (3.61)	1026.86 (4.08)

Table 4.6: WARIS-Transport Execution Time for MPI+AVX (1 MPI rank/core) and MPI+OpenMP+AVX (1 MPI rank/node, 1 thread/core) implementations in MareNostrum. Caulle-0.05-8bin dataset (601x601x64 x 8).

In Figure 4.13 we show the execution time (solid lines) and speedups (dashed lines) of the same GPU implementation in Hulk and Minotauro. The GPU implementation uses 1 GPU/MPI rank. Since Hulk is limited to 4 GPUs, we only used 1 Minotauro node (that contains 4 Tesla K80). Again, we provide a Table 4.6 with the exact execution time and speedup achieved. We observe that Hulk with a K40 (GK110B) is faster than Minotauro with a K80 (GK210), in part this could be explained by the fact that a GK210 has 13 SMs while a GK110B has 15 SMs (15% performance). In Hulk, the GPU implementation scales up to 4 Tesla K40 with a parallel efficiency that stays at an acceptable rate of 90% with 4 GPUs. In Minotauro, the GPU implementation also scales up to 4 Tesla K80. We observe superlinear speedup that can be an effect of GPU caches.

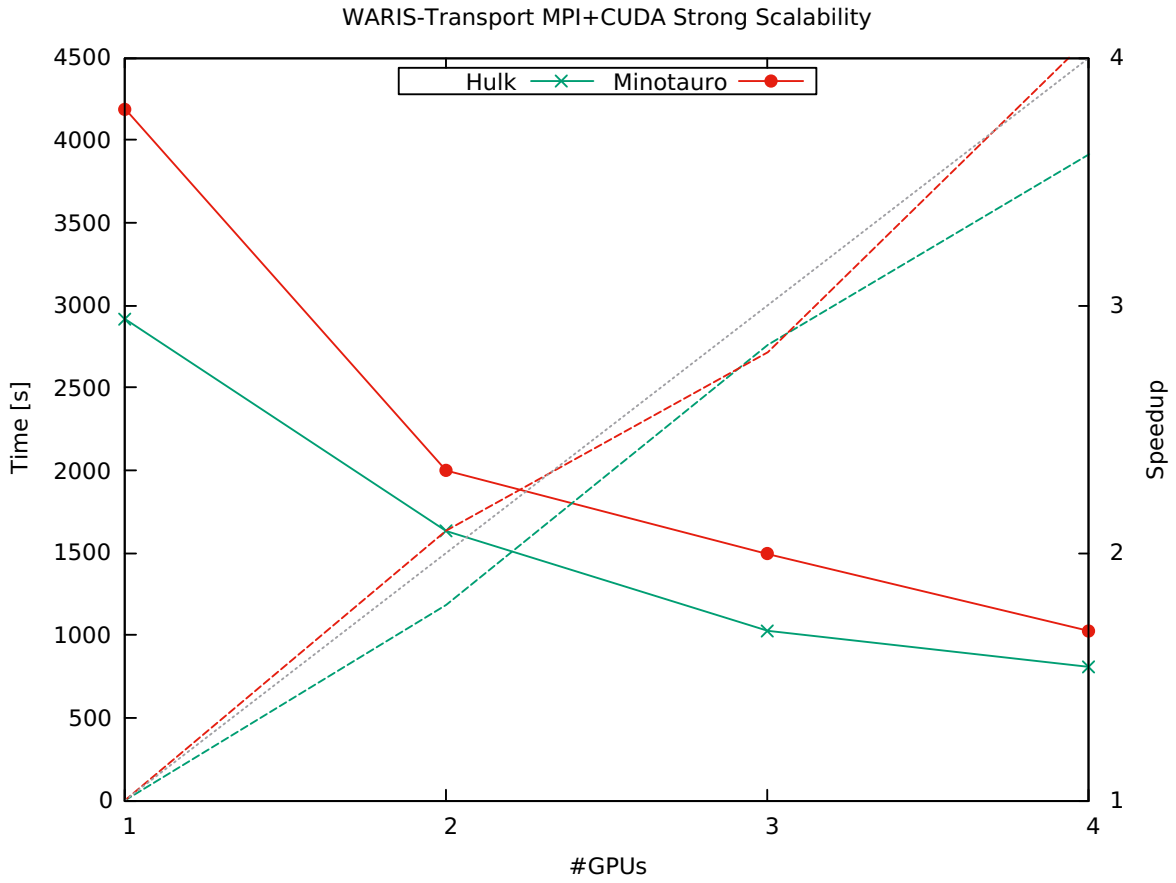

Figure 4.13: WARIS-Transport Strong Scalability for MPI+CUDA (1 MPI rank/GPU) implementation in Hulk and Minotauro. Caulle-0.05-8bin dataset (601x601x64 x 8).

If we compare the CPU and GPU implementations from Table 4.5 and Table 4.6 we can observe that an execution with 4 Tesla K40 (809.44 seconds) runs as fast as 8 MareNostrum III nodes (2x Intel Xeon CPU E5649 @ 2.53GHz) (806.81 seconds).

Chapter 5

PELE: Protein Energy Landscape Exploration

5.1 Introduction

Macro-molecular understanding of proteins is a complex task that requires long simulation times. The most widely used method is Molecular dynamics (MD), which models the molecular interactions at the atom level. MD uses empirical approximations of forces that are later integrated in time to obtain the movement of the macro-molecule. Examples of MD applications are AMBER [12], NAMD [13] or GROMACS [14].

The problem of MD simulations is that they are sensitive to the problem size and to the total time of the propagation.

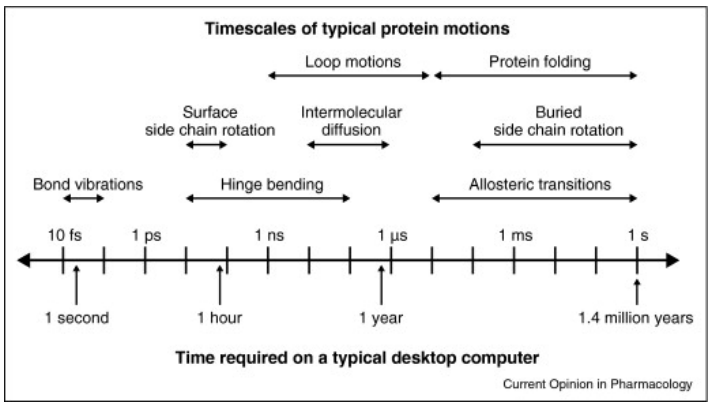

Figure 5.1: Timescales of typical protein motions [6].

In Figure 5.1 we can see the timescales required to do MD simulations. Feasible simulations are in the order of nanoseconds per day, making simulations of the order of milliseconds expensive and simulations on the scale of seconds almost impossible. To overcome this problem, researchers use statistical sampling (Monte Carlo) to get rid of the integration in time of MD applications and randomize the protein movements, achieving faster simulations.

Protein Energy Landscape Exploration (PELE [15]) is a bio-informatics application that simulates the conformation of a protein-drug interaction using a Monte Carlo algorithm. PELE generates and propagates changes in a protein-ligand system by generating a series of protein conformation positions

with similar minimum energy which are combined into a trajectory by the researcher. The objective of PELE is to accelerate molecular conformation to be able to simulate difficult protein phenomenons that are hard to simulate in classical MD. To that purpose, PELE uses Monte Carlo, energy minimization algorithms and rotamer libraries for further energy minimization.

5.2 Application Structure

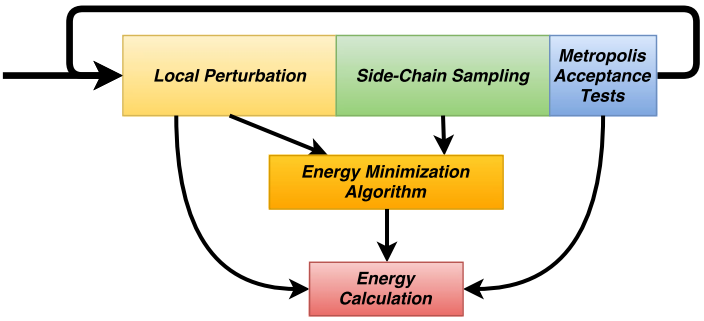

Figure 5.2: PELE algorithm for one Monte Carlo step.

PELE is composed by three main phases: *Local perturbation*, *Side-Chain sampling* and *Metropolis Acceptance tests* (see Figure 5.2). All parts of the application call a *Energy calculation* through the *Energy Minimization Algorithm* (later explained in section 5.2.2) or directly. The *Energy Calculation Algorithm* is where the PELE energy formula is calculated and is explained in later section 5.3.

Since each Monte Carlo step is independent, PELE parallelization is embarrassingly parallel thus the objective of this work is to decrease the simulation time of one step because the actual execution time is too slow to achieve a semi-real time simulation. MPI is used to distribute each simulation step to each process, using a *master-worker* pattern where the master process (MPI rank 0) controls the execution of the other workers (MPI ranks different from 0). PELE simulation orders are read from a control file, which contains one or multiple Protein Database Files (PDB) to be simulated, the number of Monte Carlo steps to be simulated (PELE steps) and other important parameters for the simulation. Then, each worker simulates one or multiple PELE steps from one or multiple PDBs. To finalize, all successful PELE steps (those that generate new molecular conformations that pass the Metropolis acceptance test) are communicated to the master and the program finishes.

5.2.1 Local Perturbation

In the Local Perturbation phase, the ligand is moved to random positions, each movement constituting a *trial*. Then the conformation energy of this new position is calculated and after eliminating impossible conformations, the *trial* is considered a candidate or discarded. At the end of Local Perturbation we obtain the *trial* with minimum energy from all candidates.

Then, as a second step in Local Perturbation, the conformation energy is minimized. PELE supports three types of energy minimization algorithms: conjugate gradient minimizer (CG), Steepest-Descent Minimizer (SD) and a Truncated-Newton minimizer (TN) with Multi-scale [16]. Usually TN is used as the energy minimization algorithm.

5.2.2 Side-Chain Sampling

After the ligands have been perturbed and the energy minimization process has found a conformation with minimum energy, The *Side-Chain Sampling* step begins. The Side-Chain phase replaces high-energy rotamers by another low-energy rotamers using a library of rotamers to further minimize the conformation energy. Then another TN minimizer pass is performed.

Multi-scale Truncated Newton Minimization Algorithm

To reduce computational complexity, PELE implements a Multi-scale Truncated Newton Minimization Algorithm [16]. The Truncated Newton (TN) minimizes the objective function (PELE formula) by searching new points of local minima in the direction of maximum slope (gradient).

Also, the Multi-scale algorithm divides the molecular interactions into short and long-range components. The short-range forces contain all covalent interactions and non-bonded interactions between atoms separated by small distances, and the long-range interactions include all other non-bonded interactions. In addition, PELE adds another level of work division where interactions are classified according to whether van-der-waals forces have to be computed or not.

The performance of the Multi-scale algorithm depends on the frequency in which each short and long range force is updated. Therefore, in the current implementation short-range interactions are calculated more frequently than the long-range ones.

5.2.3 Metropolis Acceptance Test

Finally, the final conformation is accepted or rejected following the Monte Carlo Metropolis acceptance test. Given an energy delta $\Delta E = E_{end} - E_{start}$) the PELE step will be accepted if one of the two conditions are met: If there is a decrease in energy ($\Delta E < 0$), or if at a given temperature T the inverse exponential shown in equation 5.1 (where K_B is a Boltzmann constant) is less than a random number R with a $[0, 1]$ range.

$$\begin{gathered} \Delta E < 0 \\ exp(-\Delta E / K_B T) < R \end{gathered} \tag{5.1}$$

Then the PELE step results are written to disc and another PELE step will be simulated until all steps are simulated.

5.3 PELE Energy formula

The most computationally expensive part of PELE is the energy calculation. It is called from almost every part of the application and it constitutes the main acceleration goal of our CUDA implementation. The general energy function in PELE has the form:

$$E = E_{\text{MM}} + \Delta G_{\text{solv}} + E_{\text{constraints}} \tag{5.2}$$

The first term is the molecular mechanics component, and corresponds to the energy of the system in vacuum. Depending on the force field chosen for the simulation, this term will have different functional forms and parameters.

The second term is the solvation energy for the system, and depends on the solvation model chosen (in vacuum studies this term is null).

The third term depends on the constraints fixed by the user (for example, to keep the ligand at the perturbation target position). We won't consider this term as relevant since the PELE specification says that, at most, this term has a complexity linear with the number of atoms ($O(n)$).

The molecular mechanics term is further divided in bonded and non-bonded energy:

$$E_{\text{MM}} = E_{\text{bnd}} + E_{\text{nb}} \tag{5.3}$$

These terms can be further subdivided as:

$$\begin{aligned} E_{\text{bnd}} &= E_{\text{bond}} + E_{\text{angle}} + E_{\text{torsion}} + E_{\text{improper torsion}} + E_{1-4} \\ &= \sum_{\text{bonds}} K_r(r - r_{\text{eq}})^2 + \sum_{\text{angles}} K_\theta(\theta - \theta_{\text{eq}})^2 + \sum_{\text{angle}\, i} \sum_{j} k^i_{j,1}[1 + k_{j,2}\cos(n_j\phi_i)] \end{aligned} \tag{5.4}$$

$$\begin{aligned} E_{\text{nb}} &= E_{\text{vdw}} + E_{\text{ele}} \\ &= \sum_i \sum_{\substack{j \\ i<j}} \left[\text{scale}_{\text{coulomb}} \frac{q^s_i q^s_j}{\varepsilon_{\text{in}(ij)} r_{ij}} + \epsilon^s_i \epsilon^s_j \left(\left(\frac{\sigma^s_{ij}}{r_{ij}} \right)^{12} - \left(\frac{\sigma^s_{ij}}{r_{ij}} \right)^{6} \right) \right] f_{ij} \end{aligned} \tag{5.5}$$

The bonded part (equation 5.4) includes the *bond* (a summation of all bond interactions in the molecule), *angle* (a summation of all angles interactions formed by two consecutive bonds in the molecule), torsion and improper torsion angles (a summation of up to 4 terms, each a function of a cosine related to the torsion angle θ_i), and 1-4 interactions terms. The non-bonded energy (equation 5.5), is the summation of all atom pair interactions and includes two terms: the *van-der-waals* energy and the *electrostatic* energy. When using the multi-scale energy algorithm, and for efficiency reasons, it is calculated only for pairs of neighboring atoms. These neighbors atoms are grouped as short-distance neighbors and long-distance neighbors, and each group is itself divided into two groups, depending on whether all non-bonded interactions are calculated, or only the van-der-Waals forces.

Finally, the solvation energy (equation 5.6 which uses an implicit Surface-Generalized Born consists of one term for the electrostatic polar solvation energy regarding the same interactions as in the non-bonding term. The non-polar solvation *self* energy is the complement of the polar solvation energy and is calculated for every atom. The solvation penalty depends on the model used, and has polynomial complexity. The last term is the non-polar solvation energy term, which has contributions from all energies.

$$\begin{aligned} \Delta G_{\text{solv}} =& \Delta G_{\text{pol}} + \Delta G_{\text{npol-self}} + \Delta G_{\text{penalty}} + \Delta G_{\text{npol-solvation}} \\ =& \sum_i \sum_{\substack{j \\ i<j}} \left(\frac{1}{\varepsilon_{\text{in}(ij)}} - \frac{e^{-\kappa f_{\text{GB}}}}{\varepsilon_{\text{solv}}} \right) \frac{q^s_i q^s_j}{f_{\text{GB}}} \\ &+ \frac{1}{2} \sum_i \left(\frac{1}{\varepsilon_{\text{in}(ij)}} - \frac{e^{-\kappa \alpha_i}}{\varepsilon_{\text{solv}}} \right) \frac{(q^s_i)^2}{f_{\alpha_i}} \\ &+ \Delta G_{\text{solv,penalty}} \\ &+ \sum_i C_{\gamma,i}\text{SASA}_i + C_{\alpha,i} S(a/\alpha_i) \end{aligned} \tag{5.6}$$

Therefore, the expanded PELE energy formula is:

$$
\begin{aligned}
E = & \sum_{\text{bonds}} K_r(r - r_{\text{eq}})^2 + \sum_{\text{angles}} K_\theta(\theta - \theta_{\text{eq}})^2 + \sum_{\text{angle}\, i} \sum_{j} k^i_{j,1}[1 + k_{j,2}\cos(n_j\phi_i)] + \\
& \sum_i \sum_{\substack{j \\ i<j}} \left[\text{scale}_{\text{coulomb}} \frac{q^s_i q^s_j}{\varepsilon_{\text{in}(ij)} r_{ij}} + \epsilon^s_i \epsilon^s_j \left(\left(\frac{\sigma^s_{ij}}{r_{ij}} \right)^{12} - \left(\frac{\sigma^s_{ij}}{r_{ij}} \right)^{6} \right) \right] f_{ij} + \\
& \sum_i \sum_{\substack{j \\ i<j}} \left(\frac{1}{\varepsilon_{\text{in}(ij)}} - \frac{e^{-\kappa f_{\text{GB}}}}{\varepsilon_{\text{solv}}} \right) \frac{q^s_i q^s_j}{f_{\text{GB}}} + \frac{1}{2} \sum_i \left(\frac{1}{\varepsilon_{\text{in}(ii)}} - \frac{e^{-\kappa \alpha_i}}{\varepsilon_{\text{solv}}} \right) \frac{(q^s_i)^2}{f_{\alpha_i}} + \\
& \Delta G_{\text{solv,penalty}} + \sum_i C_{\gamma,i} \text{SASA}_i + C_{\alpha,i} S(a/\alpha_i) + \\
& E_{\text{constraints}}
\end{aligned}
\tag{5.7}
$$

5.4 Initial Profiling

Initial profiling for PELE was done with *gprof*. This application happens to be much bigger than WARIS-Transport, and the fact that it's a non regular, object-oriented C++ application complicates reading gprof output. To overcome this problem we used a graphical representation of gprof (as a callgraph heat map) to detect the most important part of the application in terms of execution time (shown in Figure 5.3). Red color means high aggregated execution time (main function) and blue means low execution time in comparison to the total execution time. Another kind of information not visible unless it's scaled up is the function names, the number of calls and the aggregated time per function, which constitute useful information for finding hot spots in the code.

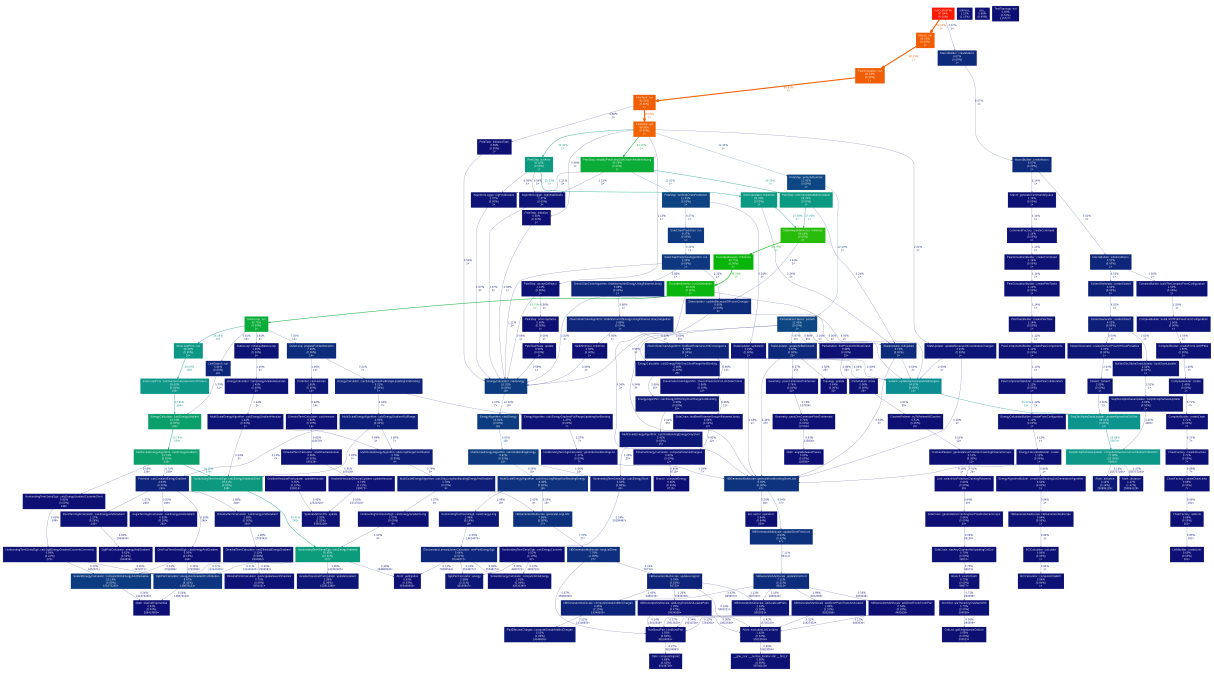

Figure 5.3: Gprof Callgraph (O3)

Table 5.1 shows a summary of our findings using Figure 5.3. We can observe that 37.6% of the execution time in PELE is spent computing the non-bonding interaction term (fourth, fifth and sixth terms in equation 5.7). The other 27,96% is the non-polar solvation energy (eight term in equation 5.7),

%	Name
37.58	Non-bonding interactions
27.96	Update alphas
2.13	Dihedral Term
1.27	Bond Term
0.93	Angle Term
30,13	Others

Table 5.1: PELE CPU profile using *gprof* in Hulk and Global dataset.

calculated in the *update_alphas* function. In *Others* we include all computations that are not part of the energy calculation, like list creations, etc. Others occupies a large portion of execution time in which non-bonding list generation occupies a 16.15% of the total execution time and other pieces contribute for less than <1% being Others a pool of functions difficult to optimize.

5.5 CUDA Implementation

In this section we present our CUDA implementation of PELE. Firstly we will discuss the details of the accelerated parts, then we will discuss how we have improved GPU programmability in PELE.

In Table 5.2 we can observe the CPU functions that have been accelerated with GPUs and their weight in the original CPU implementation.

%	Name
5.89	NonBondingTermSerialSgb::calcEnergy
30.48	NonBondingTermSerialSgb::calcEnergyGradient
0.52	NonBondingTermSerialSgb::calcEnergyCoulomb
0.69	NonBondingTermSerialSgb::calcEnergyCoulombGradient
27.96	SeqObcAlphaSasaUpdater::updateAlphas

Table 5.2: PELE Accelerated functions.

5.5.1 Non-bonding Energy

In section 5.2.2 we explained that PELE divides interactions between long and short interactions, and that a second division is performed to avoid computing van-der-waals forces in long distances. This is translated into a double layer where functions like *calcEnergyLong* and *calcEnergyShort* construct a list of interactions and then call one of the four versions of calcEnergy that are implemented: *calcEnergy* (coulomb + van-der-walls + SGB), *calcEnergyGradient* (coulomb + van-der-waals + SGB + gradient), *calcEnergyCoulomb* (coulomb + SGB) and *calcEnergyCoulombGradient* (coulomb + SGB + gradient).

In Algorithm 4 we can observe the *calcEnergy* and *calcEnergyGradient* pseudocode. CalcEnergy function has as inputs a vector of interactions, the coordinates of all atoms, and some dielectric constants. In the other hand calcEnergyGradient also adds the gradient vector to that list. For every pair of interactions, electrostatic, van-der-waals and SGB is calculated. In calcEnergyGradient the gradient vector is updated (line 10).

The first step to accelerate a function with CUDA is to adapt the CPU data structures. As seen in Figure 5.4, the first problem we encountered is that lists of interactions inherit from *std::pair* to hold a pair of

Algorithm 4 PELE - calcEnergy sequential

1: **procedure** CALCENERGY(vector<NonBondPair>, coords, dielectrics)
2: van-der-waals = 0
3: electrostatic = 0
4: SGB = 0
5: **for** every pair i, j in vector<NonBondPair> **do**
6: Atom a = pair.first
7: Atom b = pair.second
8: electrostatic += calcElectrostatics(a, b) ▷ $\text{scale}_{\text{coulomb}} \frac{q_i^s q_j^s}{\varepsilon_{\text{in}(ij)} r_{ij}}$
9: van-der-waals += calcVanderWaals(a, b) ▷ $\epsilon_i^s \epsilon_j^s \left(\left(\frac{\sigma_{ij}^s}{r_{ij}} \right)^{12} - \left(\frac{\sigma_{ij}^s}{r_{ij}} \right)^{6} \right)$
10: SGB += calcSGB(a, b) ▷ $\left(\frac{1}{\varepsilon_{\text{in}(ij)}} - \frac{e^{-\kappa f_{\text{GB}}}}{\varepsilon_{\text{solv}}} \right) \frac{q_i^s q_j^s}{f_{\text{GB}}}$
11: **end for**
12: **end procedure**

procedure CALCENERGYGRADIENT(vector<NonBondPair>, coords, grad, dielectrics)
2: van-der-waals = 0
electrostatic = 0
4: SGB = 0
for every pair i, j in vector<NonBondPair> **do**
6: Atom a = pair.first
Atom b = pair.second
8: electrostatic += calcElectrostatics(a, b) ▷ $\text{scale}_{\text{coulomb}} \frac{q_i^s q_j^s}{\varepsilon_{\text{in}(ij)} r_{ij}}$
van-der-waals += calcVanderWaals(a, b) ▷ $\epsilon_i^s \epsilon_j^s \left(\left(\frac{\sigma_{ij}^s}{r_{ij}} \right)^{12} - \left(\frac{\sigma_{ij}^s}{r_{ij}} \right)^{6} \right)$
10: SGB += calcSGBandGradient(a, b, grad) ▷ $\left(\frac{1}{\varepsilon_{\text{in}(ij)}} - \frac{e^{-\kappa f_{\text{GB}}}}{\varepsilon_{\text{solv}}} \right) \frac{q_i^s q_j^s}{f_{\text{GB}}}$ + gradient
end for
12: **end procedure**

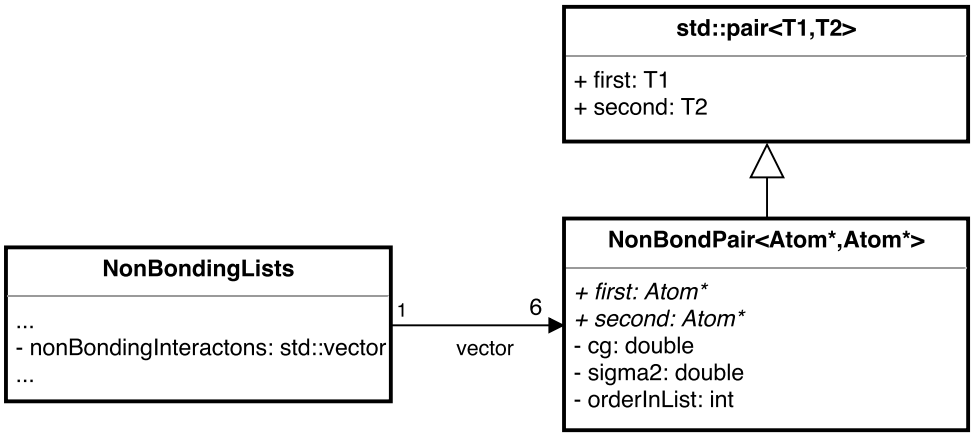

Figure 5.4: PELE Array of Structures(AOS)

Atom*. Unfortunately most of the STL is not GPU callable because it lacks the __device__ specifiers. The second problem is that accessing atom fields through pointer chasing is not good to achieve good memory performance in CUDA since it is crucial to perform coalesced memory accesses. Therefore we had to adapt the Array of Structs (AOS) *std::vector<NonBondPair>* into a new Structure of Arrays (SOA) class. This new class called NonBondPairsSOA, implements the same API that a std::vector of NonBondPair

but internally holds each atom field required in a separate std::vector. Therefore NonBondPairsSOA class members are valid in CPU & GPU.

Algorithm 5 PELE - calcEnergy Kernels

```
1:  procedure CALCENERGYKERNELGRADIENT(NonBondPairsSOA, coords, grad, atoms, dielectrics)
2:      for atom_pair i in NonBondPairsSOA do                                  ▷ GRID
3:          van-der-waals = calculateVDW(i.first, i.second)
4:          electrostatic = calculateCoulomb(i.first, i.second)
5:          SGB = calculateSGB(i.first, i.second, delta)
6:          updateGradient(grad, delta)
7:
8:          vdwTot = Reduction(van-der-waals)
9:          elecTot = Reduction(electrostatic)
10:         sgbTot = Reduction(SGB)
11:
12:         if threadIdx.x == 0 then
13:             atomicAdd(vdwTot)
14:             atomicAdd(elecTot)
15:             atomicAdd(sgbTot)
16:         end if
17:     end for
18: end procedure
```

Another problem was the use of CPU data structures like *dielectric* constants, that are implemented using virtual methods to achieve polymorphism which are forbidden in CUDA.

Algorithm 6 PELE - updateGradient CPU

```
1: procedure UPDATEGRADIENT(gradient, atomAidx, atomBidx, delta)
2:     grad[3*atomAidx+0] += delta.x
3:     grad[3*atomAidx+1] += delta.y
4:     grad[3*atomAidx+2] += delta.z
5:
6:     grad[3*atomBidx+0] -= delta.x
7:     grad[3*atomBidx+1] -= delta.y
8:     grad[3*atomBidx+2] -= delta.z
9: end procedure
```

In Algorithm 5 we can observe the pseudo-code of the actual CUDA kernel implementation. Each CUDA thread is in charge of calculating one interaction. The CUDA implementation defines a 1D thread-block of 128 threads, and a 1D grid of $\lceil \frac{nblistsize}{n_{threads/tb}} \rceil$. Each thread calculates van-der-waals, coulomb and the Surfafe-Generalized Born (SGB) energies. Then 3 thread block wide reductions are launch to accumulate all van-der-waals, coulomb and SGB energies. And the last thread stores this results into global memory using atomics.

CalcEnergy, if necessary, also updates the gradient. In the CPU implementation (Algorithm 6) the contributions are added and subtracted using regular += and -= operators, but in CUDA we had to use atomic operations to avoid race conditions.

Algorithm 7 PELE - updateGradient

```
1: procedure UPDATEGRADIENT(gradient, atomAidx, atomBidx, delta)
2:     atomicAdd(&grad[3*atomAidx+0], delta.x)
3:     atomicAdd(&grad[3*atomAidx+1], delta.y
4:     atomicAdd(&grad[3*atomAidx+2], delta.z
5:
6:     atomicSub(&grad[3*atomBidx+0], delta.x)
7:     atomicSub(&grad[3*atomBidx+1], delta.y)
8:     atomicSub(&grad[3*atomBidx+2], delta.z)
9: end procedure
```

Gradient update with Atomics

In Algorithm 7 we can observe that *atomcAdd* and *atomicSub* are used because the access pattern to global memory is not regular, see Figure 5.5.

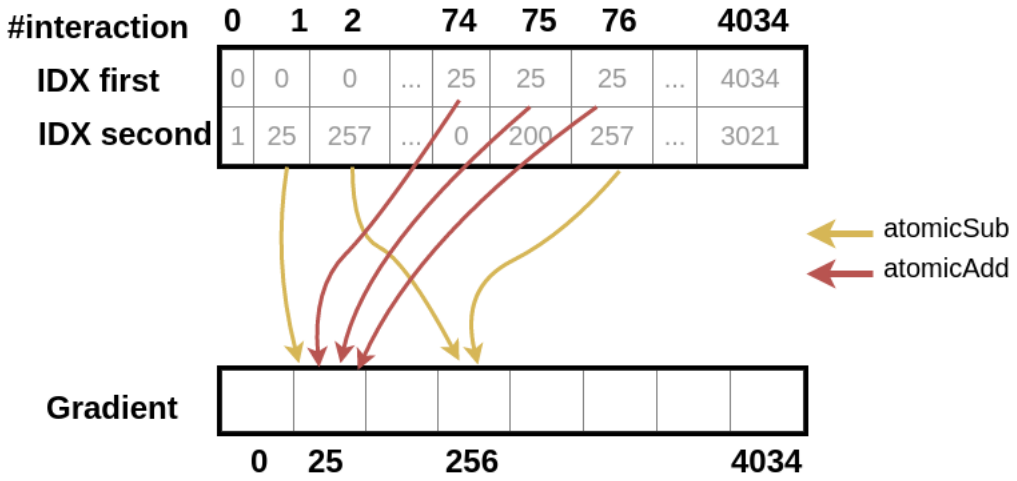

Figure 5.5: Gradient update with atomics

Also, since PELE uses double precision, and CUDA does not implement a double precision *atomicAdd/atomicSub*, we had to implement double precision atomics using *atomicCAS*.

```
__device__ inline double atomicAdd(double* address, double val) {
    unsigned long long int* address_as_ull = (unsigned long long int*)address;
    unsigned long long int  oldval         = *address_as_ull;
    unsigned long long int  assumed, newval;

    do {
        assumed = oldval;
        newval  = __double_as_longlong(__longlong_as_double(assumed) + val);
        oldval  = atomicCAS(address_as_ull, assumed, newval);
                  // *address == assumed ? newval :  assumed
    } while (assumed != oldval);

    return __longlong_as_double(oldval);
}
```

Gradient update without Atomics

Later we saw that double precision atomics had a performance hit and decided to implement the gradient updates without atomics. One possibility would have been to implement the gradient update in CPU, but we decided to still use the GPU to do that operation using multiple kernels to reorder the data an regularize the access patterns. To increase programmability and reuse software, we decided to use a CUDA library named CUB [17]. CUB is mainly used to write reusable components for every layer of the CUDA programming layer. In our case we used some *Device-wide primitives* like sorting algorithms, reductions, parallel-prefix-sum, etc...

Figure 5.6 shows an example with a list of 8 elements of the post-processing required to update the gradient vector without atomics.

1. All index and values are sorted with cub::DeviceRadixSort::SortPairs in ascending order.
2. Count the number of unique index we have with cub::DeviceRunLenghtEncode::Encode.
3. Parallel-prefix sum to get the starting points and end points (offsets) of every index.
4. Segmented reduce is launched with cub::DeviecSegmentedReduce::Sum where each thread is in charge of accumulating values from range $inclusive - scan[threadIdx.x]$ to $exclusive - scan[threadIdx.x]$.
5. Another kernel takes a vector of contributions, and adds/subtracts this contributions to gradient.

5.5.2 Update Alphas

In equation 5.6 we described the non-polar solvation energy as one term of the PELE formula. As a prerequisite for calculating this term, PELE has to calculate the α coefficients for all atoms. This is done in the *updateAlphas* function.

The main reason update alphas is one of the most expensive parts is because each atom a_i has to visit all atoms a_j for a total complexity of $O(n^2)$. The fact that is the second most expensive function of PELE and that there is enough parallelism available ($O(n^2)$) made it a perfect candidate to be accelerated.

In Algorithm 8 we can observe the sequential implementation of the update alphas calculation. Each atom i calculates the contributions of all other atoms and updates his alpha coefficient.

To parallelize this function we had two possibilities. The first possibility is to exploit $O(n)$ parallelism by implementing the outer loop (line 5) using a 1D grid where each thread is in charge of, sequentially compute all contributions of other atoms over his atom (line 9). This straightforward approach, has the benefit that is simple but when N is small enough, no sufficient thread blocks are generated e.g., with 1024 threads per block and 4096 atoms three are only 4 thread blocks in flight which incurs in SM unbalancing (only 4 occupied SMs over 15 in a K40). Also to hide memory latencies it is crucial to have multiple thread-blocks per SM, in which case having few thread blocks in flight don't help at all.

Then the second possibility was to exploit the $O(n^2)$ parallelism by launching n thread-blocks with a thread-collaborative scheme were all threads in a thread-block work together to compute contributions of atom $blockIdx.x$.

In Figure 5.7 we can observe how the n^2 iteration space is distributed using one CUDA thread block per atom i. The left Figure represents the iteration space where atom i has contributions from all other

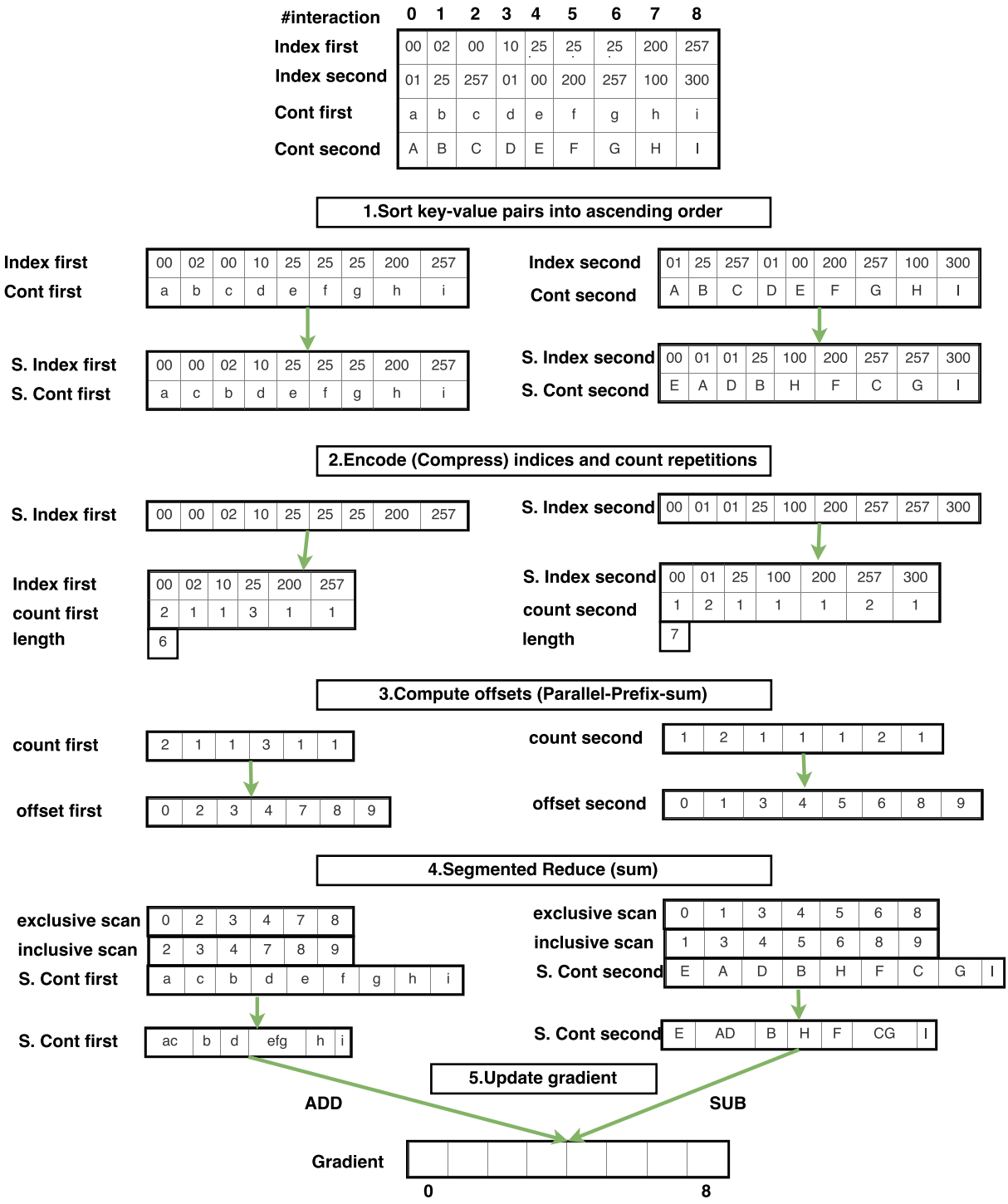

Figure 5.6: Gradient update without atomics

atoms N. Gray squares means that i - i interactions are not possible and therefore forbidden. The right Figure represents the CUDA iteration space distribution where each *row* is a thread block (TB) in charge of atom $blockIdx.x$ and each column is an atom $[0, N-1)$.

In Algorithm 9 we can observe the implementation of the update alphas. All threads in the thread block load the same $atom_i = atoms[blockIdx.x]$, and then they enter a collaborative loop where each thread loads a different $atom_j = atoms[threadId.x]$. Each thread computes his i to j contribution and stores his private value in a register. If the number of atoms N is larger than the number of threads,

Algorithm 8 PELE - OBC Update Alphas Sequential

```
1:  procedure UPDATEALPHAS
2:      ALPHA = 1.0
3:      BETA = 0.8
4:      GAMMA = 4.85
5:      for atom i in atomSet do
6:          sum = 0
7:          for atom j in atomSet do
8:              if i != j then
9:                  compute_contribution(i, j)
10:             end if
11:         end for
12:         α_i = r_i − tanh(ALPHA ∗ sum − BETA ∗ sum^2 + GAMMA ∗ sum^3)
13:     end for
14: end procedure
```

Algorithm 9 PELE - OBC Update Alphas Kernels

```
1:  procedure COMPUTECONTRIBUTIONSALLATOMSKERNEL(atoms, coords, bornRadiiOffset, HCT)
2:      for do                                          ▷ Grid of N thread blocks
3:          atom_i = atoms[blockIdx.x]      ▷ all threads in a thread block share the same i atom
4:          for atom j = blockIdx.x; j < N j+= blockDim.x do
5:              if i != j then
6:                  compute_contribution(i, j)
7:              end if
8:          end for
9:          store partial result in a vector
10:     end for
11: end procedure
    procedure UPDATEALPHASKERNEL(atoms, partialContributions)
2:      for do                                ▷ Grid of N/thread − block − size thread blocks
            sum = partialContributions[i]
4:          α_i = r_i − tanh(ALPHA ∗ sum − BETA ∗ sum^2 + GAMMA ∗ sum^3)
        end for
6:  end procedure
```

each thread picks a new atom $threadIdx.x + iter * BLOCK\_SIZE$, calculates the contribution and accumulates the partial result. As a final step, all threads in the thread block perform a reduction of all partial values into a unique value per thread block (the accumulation of all forces of atoms over atom i) and is stored in a temporal array.

Then, a second kernel reads all temporal contributions and computes all α coefficients. Using one thread per output value (complexity $O(n)$).

5.6 Improving GPU programmability using CUDA Unified Memory

PELE is a complex application of 109k lines of C++ code and +5 years of development. Common GPU development approaches, like the one used in WARIS-Transport, consists in porting function by function until a reasonable amount of the application is in the GPU, minimizing CPU/GPU memory copies and

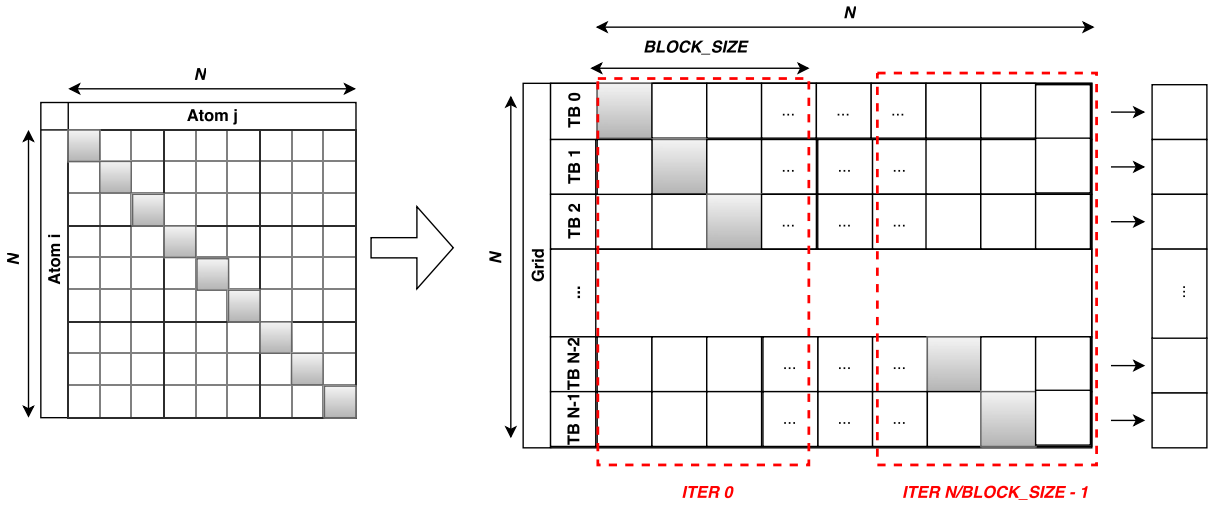

Figure 5.7: Update Alphas Kernel 1 - $O(n^2)$ iteration space distribution with CUDA

increasing data locality in the GPU. But in PELE not all parts of the application can be efficiently executed on to the GPU making memory management even more important.

In order to improve the GPU programmability we have chosen a CUDA automatic memory manager called Unified Memory (UM). Using this manager allows us to share data structures between CPU&GPU to simplifiy the code complexity. Also we chose to use the *Resource Acquisition is Initialization* (RAII) pattern to tame the complexity of managing UM-enabled allocations.

RAII ensures correct deallocation of resources and consists in wrapping raw resources (allocated objects or allocated memory, etc...) with containers. The language guarantees that the object destructor will always be invoked when control flow leaves the scope (e.g., because of a return or an exception). One example of a container that follows the RAII idiom is the *std::vector*, when execution leaves the enveloping braces, the vector is deallocated: Here we can see an example of RAII using the *std::vector* container:

```
{
    vector<type> container(size) //Constructor allocates memory
    ...
}                    //Destructor deallocates memory when falls out of scope
```

Then, to achieve a RAII with CUDA we used STL containers allocated with CUDA UM. The way to achieve this is to provide a C++ class that implements two methods (*allocate* and *deallocate*) that are defined by the STL:

```
class cuda_allocator {
    pointer allocate(size_type n)              { /* call cudaMallocManaged */ }
    void deallocate(pointer ptr, size_type n) { /* call cudaFree          */ }
}
```

Then we can pass our allocator as a template parameter of any STL container to make it UM-aware:

```
vector<type, cuda_allocator> v(size);
```

The STL container itself is not callable inside a kernel because it lacks the *__device__* specifier that each device callable function has to have. But the contents from the container are effectively addressable from the GPU. Therefore, we have to pass the internal pointer that *std::vector* holds and pass it to the CUDA kernel:

```
void main() {
    vector<type, cuda_allocator> v(1000);
    kernel<<<1,1>>>(v.data());           /* C++11 way of &v[0] */
    cudaDevieSynchronize();
}

__global__ void kernel(pointer data) { *data = 1; } /* valid */
```

5.6.1 Semi-automatic Memory Manager

CUDA UM is a great tool to improve programmability in CUDA programs, but performance-wise is still far from optimal due to excessive movement of data from device to host and from host to device. The problem is that the GPU has no ability to detect which data has been modified and, therefore after a kernel finishes all data is copied back to the CPU.

To improve performance, we implemented a Semi-Automatic Memory Manager (called SAMM for brevity) with lazy GPU to CPU memory copies and an C++ allocator to enable STL structures to be SAMM-aware. SAMM manages the memory coherency of allocations between the CPU and the GPU; that is, is in charge of performing CPU & GPU memory copies when necessary. It implements a per-allocation *hash table* that stores the useful information of the memory allocation (what we call *chunk*). Each memory chunk is composed of two internal allocations, one for the CPU and another for the GPU, the page protections (*RW*, *RO* or *NONE*) and the allocation size in bytes.

The protocol to control CPU & GPU memory coherency is shown in Figure 5.8. We define the Ownership as CPU or GPU and the possible data locations as CPU, GPU or CPU+GPU (i.e., replicated). We provide an API to explicitly declare the ownership of the data (*toGPU* or *toCPU*), and to declare which access types will be performed (*Read*, *Write*, *ReadWrite* or *FullWrite*) and to get the GPU pointer to pass to the CUDA kernel. In Write accesses we considered unsafe to just generate a GPU to CPU copy because kernels might only partially touch a buffer (requiring a CPU to GPU copy previous to the kernel launch plus the GPU to CPU copy after). We also provide the FullWrite access type when we know in advance that the kernel will write the whole buffer. This is useful when the GPU is in charge of initializing data (e.g., memset operations) since we can assume data will not be partially written, and therefore avoid the previous CPU to GPU copy. The memory protocol performs lazy movement of data from GPU to CPU when moving data ownership to the CPU, data is not moved to it until it is accessed by the CPU code; see state CPU/GPU-only. CPU accesses are detected by changing the memory protection bits of allocations managed by SAMM (using *mprotect*), which the OS signals through a page fault signal handler.

When a STL container calls to *samm_allocator::allocate*, memory is allocated (in both CPU & GPU) and inserted to the hash table. With initial *CPU/CPU-only* state (see Figure 5.8).

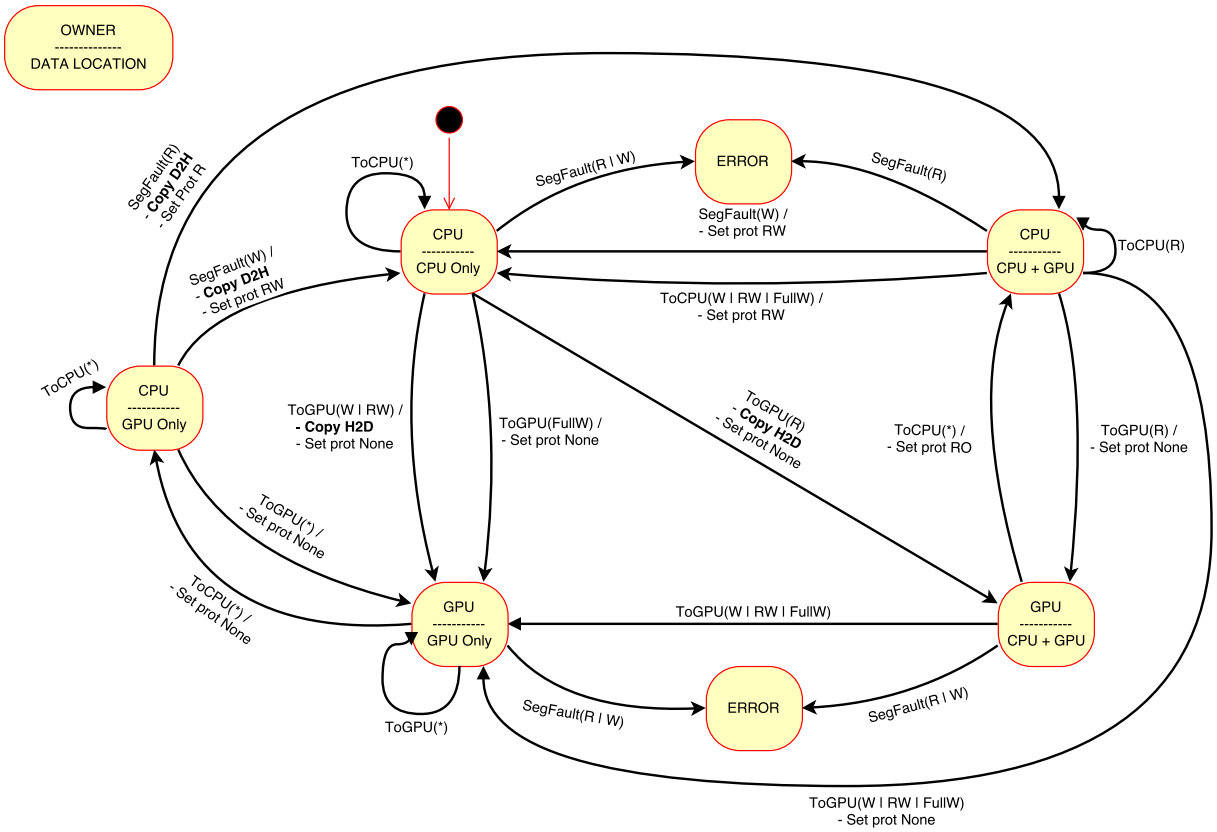

Figure 5.8: SAMM memory coherence states.

```
std::vector<double, samm_allocator<double>> data(1000); //object allocation
```

Following the example from Figure 5.9, when the programmer wants to call a kernel, the kernel call has to be *"decorated"* with two functions, *toGPU* executed before the kernel and *toCPU* executed after the kernel. The programmer specifies that ownership is moved from CPU to GPU and the type of access that the GPU will perform is a Read with *toGPU*:

```
void* gpu_ptr = toGPU(cpu_ptr, mem_acces_type::Read);
```

Then the kernel is called with the device pointer:

```
kernel<<<grid,block>>>(gpu_ptr);
```

After calling the kernel, we return the ownership of the data to the CPU and we specify that the CPU will be able to perform ReadWrite operations. Since our data manager performs lazy movement of data from GPU to CPU, the data will remain in the GPU while the ownership is in CPU. To protect invalid accesses in the CPU, our data manager sets the per-allocation memory protections to *PROT_NONE* and a segmentation handler is installed.

```
toCPU(gpu_ptr, mem_access_type::ReadWrite);
```

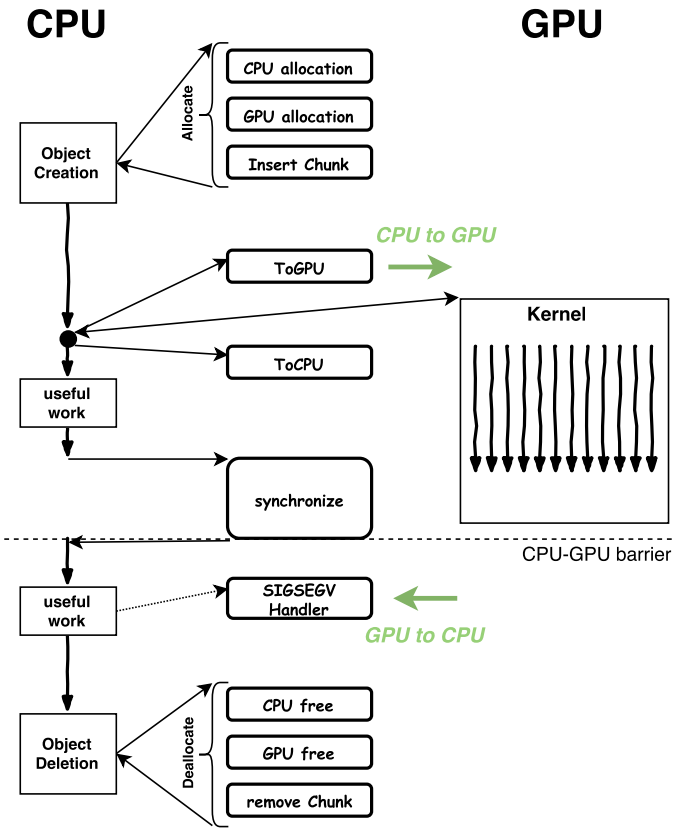

Figure 5.9: SAMM use case.

To finalize, the user has to make sure that all kernel executions in the GPU are finished.

```
cudaDeviceSynchronize()
```

Then, when the CPU reads or writes the content of our data a *SIGSEGV* is thrown:

```
int result = data[0]; //SIGSEGV
```

Our handler will see that the faulting *addr* is from one of the chunks allocated with our allocator. Then a GPU to CPU copy is performed and the page protections are changed to *PROT_READ* if it was a read operation or *PROT_READ | PROT_WRITE* if it was a write operation. Finally, the faulting operation is replayed, this time successfully.

To finalize, when the container goes out of scope, the destructor will call the allocator's *deallocate* function and this will free the CPU and GPU allocations and remove the chunk from the global hash table.

5.7 Results

To stress different parts of the application we have available two datasets, *Local* and *Global* of 4357 atoms each. They are modified versions of the same protein (PDB 4UDA) with different ligand positions taken

from [18]. Local dataset has a ligand that is moved in a small region of space (therefore named local) that spends more time in the Local Perturbation phase and the Global dataset the ligand is moved to further distances (therefore named global) that stresses the Multi-scale Truncated Newton minimizer.

Since the objective of this development was to accelerate the execution time one PELE step, we compare the original sequential implementation (one CPU core) versus our GPU implementation with one GPU (strong scaling).

Firstly, we will analyze the performance obtained kernel by kernel, comparing all implementations. Then we will show the performance of the whole application.

5.7.1 GPU Accelerated Parts

In this section we compare the performance of the original CPU implementation of *calcEnergy*, *calcEnergyGradien* and *UpdateAlphas* versus the GPU implementation using the two available memory managers in Minotauro. This way we can compare the performance of SAMM vs UM. All execution times for the GPU include CPU to GPU copies, Kernel execution and GPU to CPU copies.

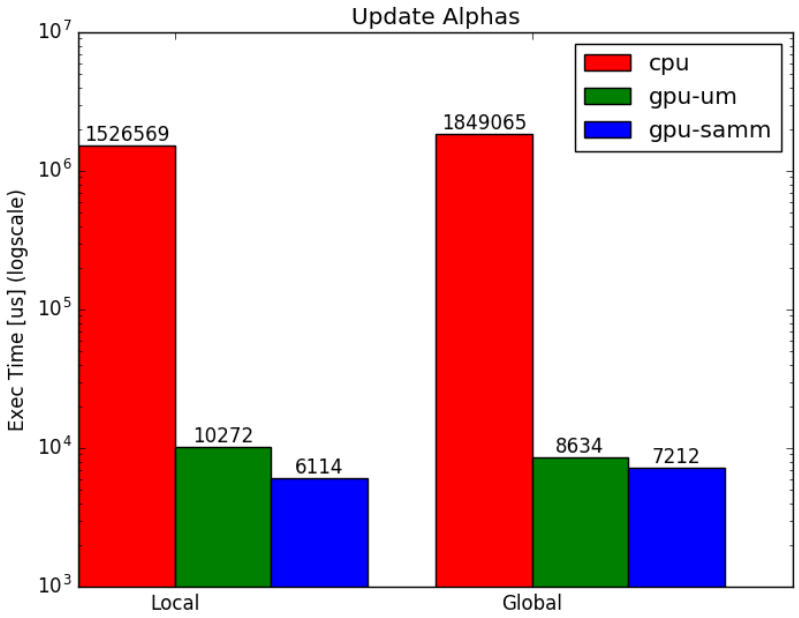

Figure 5.10: UpdateAlphas in Minotauro (K80).

First, we evaluate the performance of the *UpdateAlphas* function. In Figure 5.10 we see a comparison between the *UpdateAlphas* CPU implementation, the GPU implementation with UM and the GPU implementation with SAMM. Note that the Time scale is logaritmic, measured in microseconds. We see that the execution time went from almost 1.5s and 1.8s in Local and Global datasets respectively to 6.1ms and 7.2ms. That is equivalent of 249x and 256x times faster. This is fairly normal because the CPU implementation is a sequential implementation.

Figure 5.11 shows the performance evaluation of the calcEnergy function. We compare between the kernel implementation that use atomics and the implementation that does not use atomics. We also evaluate the effect of using UM or SAMM. Therefore we show 5 implementations, CPU, GPU+UM+Atomics, GPU+UM+NoAtomics, GPU+SAMM+Atomics and GPU+SAMM+NoAtomics. Since the calcEnergy

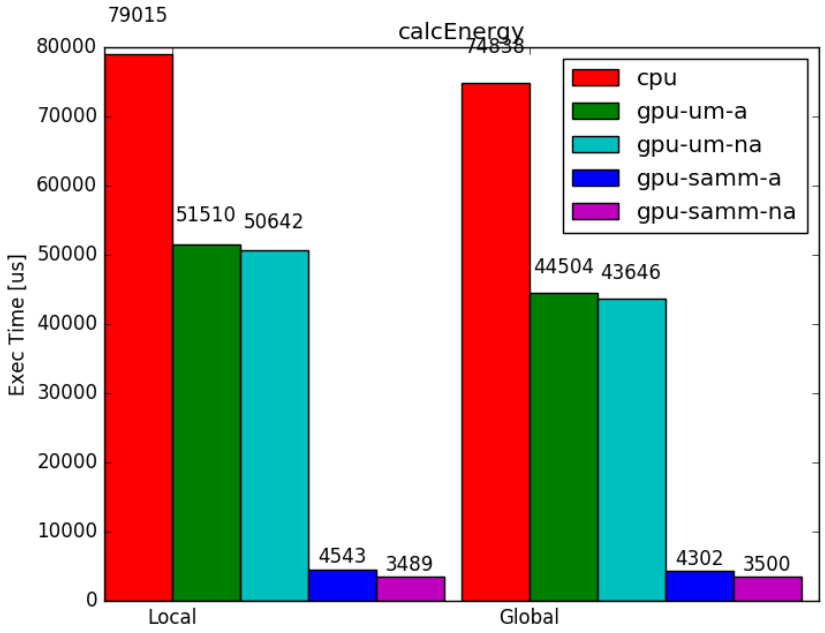

Figure 5.11: calcEnergy in Minotauro (K80).

kernel does not have to compute the gradient (we only substitute two atomic operations to obtain the electrostatic energy and the lennard jones energy) the effect of having or not atomics is reduced. We see that an improvement between the CPU implementation (79ms) with respect GPU with UM and without atomics (50ms and 1.58x speedup) and with respect GPU with SAMM (3.4ms and 22x speedup). Similar behavior is observed for the Global dataset. We also see great improvements in terms of memory management when SAMM is used instead of UM (1.58x to 22x), this is due SAMM moving less data and performing lazy copies from GPU to CPU.

In Figure 5.12 we can see a comparison of all *calcEnergyGradient* implementations including UM or SAMM with and without atomics. We see that in the *calcEnergy* function SAMM outperforms UM. We also see that the implementation without atomics is 2.8x and 2.6x times faster than the one with 64 bit atomics (in the case of gpu-samm-a vs gpu-samm-na with Local and Global datasets respectively). In total, the maximum speedup achieved by SAMM without atomics is 14.5x and 13.8x for Local and Global respectively.

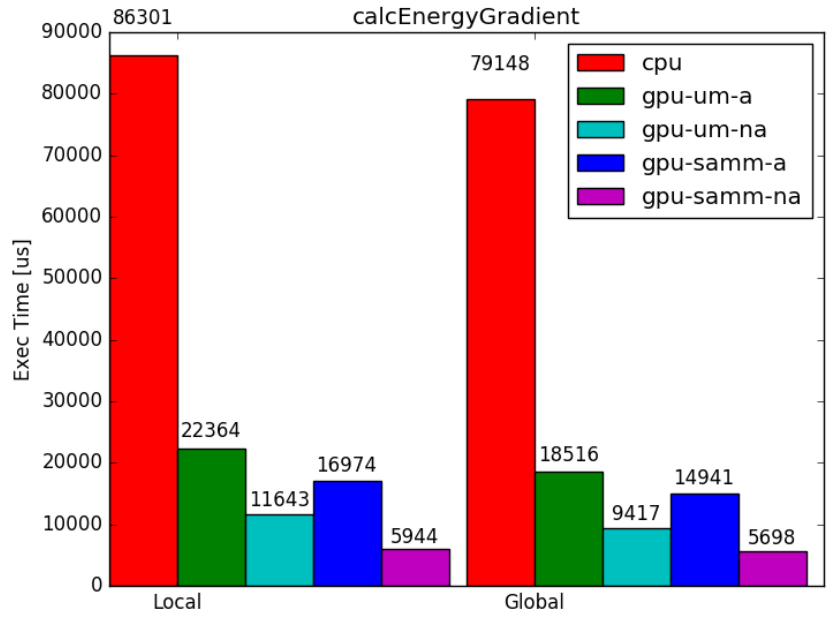

Figure 5.12: calcEnergyGradient in Minotauro (K80).

5.7.2 Global Results

In this section we compare the overall performance of PELE using the CPU and GPU implementations in Hulk and Minotauro.

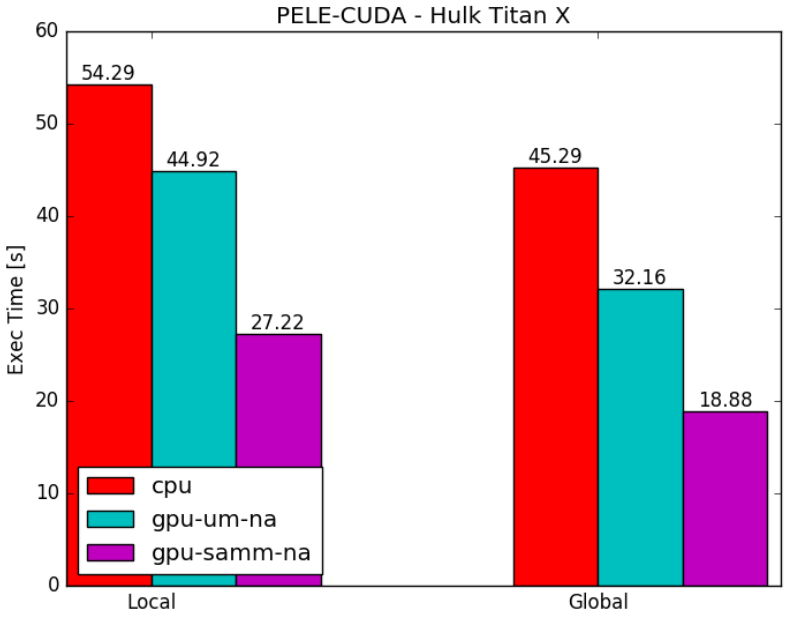

Figure 5.13: Execution Time of PELE-CUDA in Hulk (Titan X).

In Figure 5.13 we can see the execution time for the UM and SAMM implementations (without atomics) and in Figure 5.14 we can observe the speedups achieved. We can see that in Hulk, the SAMM

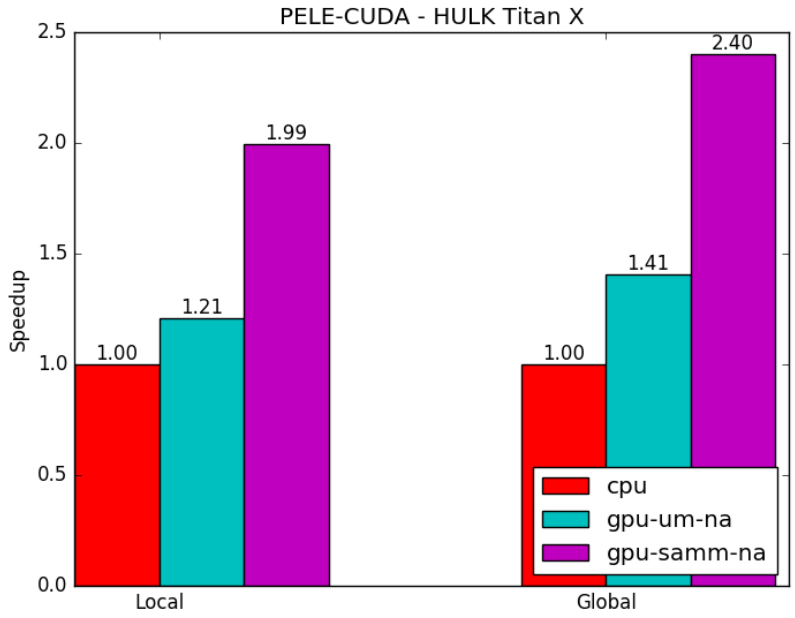

Figure 5.14: Speedups of PELE-CUDA in Hulk (Titan X).

implementation outperforms the UM implementation. Since a Titan X only has 200 GFLOPs of theoretical peak performance in 64-bit floating point we also measured the performance in Minotauro.

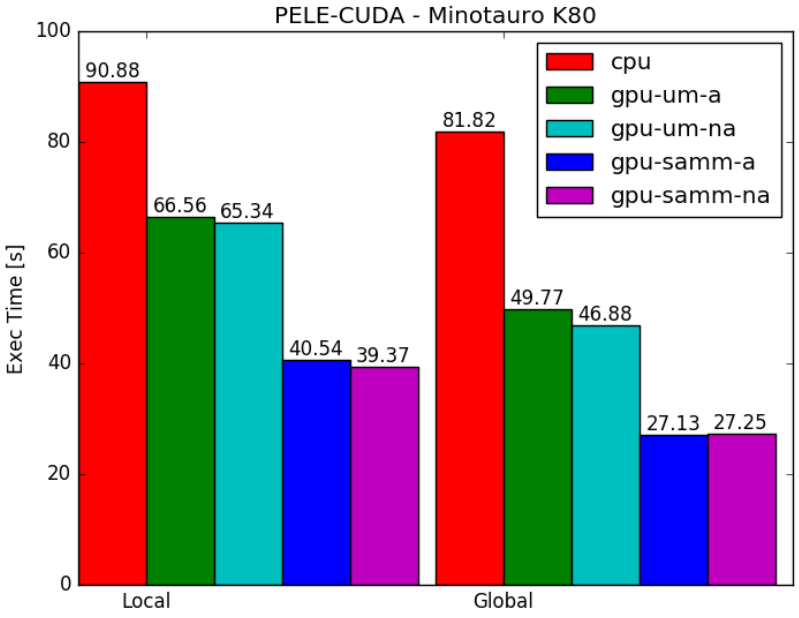

Figure 5.15: Execution Time of PELE-CUDA in Minotauro (K80).

In Figure 5.15 we can see the PELE execution times for all implementations in Minotauro (K80), and Figure 5.16 shows the corresponding speedups. Once more we see that our SAMM implementation outperforms the UM implementation. The reason we got moderate speedups and not big improvements like in WARIS-Transport is because we only managed to parallelize the 65-70% of the application,

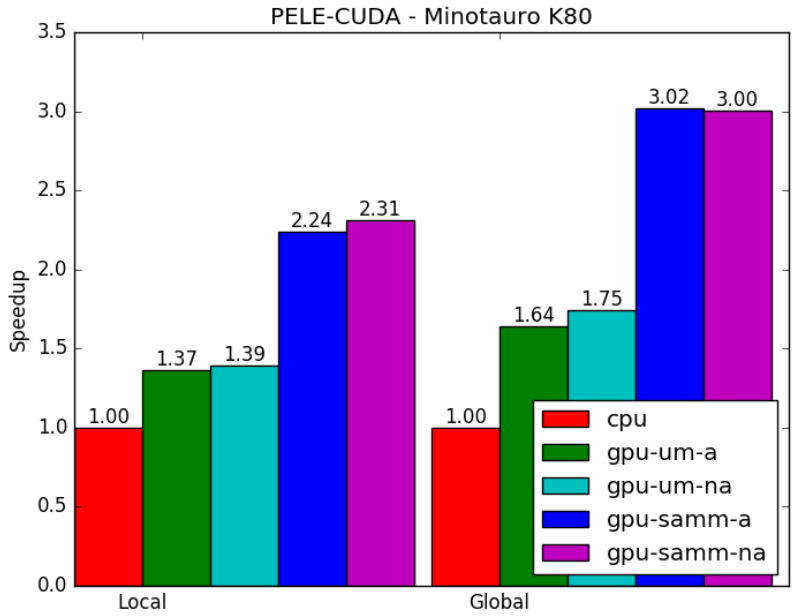

Figure 5.16: Speedups of PELE-CUDA in Minotauro (K80).

therefore we are limited by Amdahl-law.

Chapter 6

Conclusions & Future Work

6.1 Conclusions

In this thesis we have presented two approaches of how to develop GPU applications.

The first approach consists in moving all computations to the GPU to maintain data locality in the GPU and minimize CPU/GPU copies.

The second approach is used when it is not possible to move all parts of an application to the GPU. Usually in partially-accelerated GPU applications CPU & GPU memory coherence becomes a performance issue. Then to minimize data movement between CPU & GPU and to ease GPU programmability we propose to use an automatic memory manager like CUDA Unified Memory (UM). If performance is a concern, we propose a solution like our Semi-Automatic Memory Manager (SAMM). In the future, GPU architectures will implement better support for UM and thus we expect performance of applications developed with UM to increase in the future. In addition, we have shown that we can unify CPU and GPU data structures using UM or SAMM (e.g., in *std::vector*)

We have shown that finite differences applications are good candidates to be accelerated with GPUs. We have shown that WARIS-Transport scales up to 4 GPUs in Hulk and in Minotauro. Our 4-GPU implementation (in Hulk) is able to execute 8.67x and 6.61x faster than the MPI+AVX and the MPI+OpenMP+AVX implementations using one MareNostrum node (2x Intel Xeon E5649 with 8+8 cores). Also the same 4 GPU execution (Hulk) is able to perform similarly than 8 MareNostrum nodes (128 cores) running the most optimized version of WARIS-Transport.

We have shown PELE as an example of complex application that cannot be offloaded entirely to the GPU. To overcome the programmability problem of maintaining CPU & GPU coherence we proposed a memory manager like CUDA UM. Results in Hulk and Minotauro show that PELE with UM is faster than the sequential implementation. Later we saw the opportunity to improve performance by introducing a new Semi-Automatic Memory Manager (SAMM). We saw that SAMM is able to perform faster than the CUDA UM, in Hulk and also in Minotauro. To finalize, we also showed the effect on performance of 64 bit atomics in the calcEnergyGradient functions, obtaining performance speedups with Local and Global datasets using both memory managers. Hopefully in future architectures like *Pascal*, GPUs will support 64 bit atomics by hardware.

Work like this [19] show a possible implementation of how to hide our pair of functions *toCPU* and *toGPU* from the programmer. It consists in implementing a variadic template function that acts as a wrapper for the *toCPU/toGPU* functions and kernel call. But the difficult part that cannot be solved with

this approach is how to specify the type of accesses that will be performed in the subsequent section of code in a generic way. Languages like OmpSs [20] provide pragmas that contain this information, because the programmer already specifies which type of access will be performed (e.g., pragma *in* would be access-type *Read*, *out* would be *FullWrite* and *inout* could be *ReadWrite*) Then, with proper compiler support, our SAMM memory manager could be automated using pragmas.

6.2 Future Work

In WARIS-Transport we think that there is still room for improvement. The AVX kernel is already optimized, but *vsettling* implementation should be optimized. Currently in some kernels and with some data sizes, some accesses are not coalesced. Also to achieve maximum performance all data that is going to be accessed with a 2D or 3D stride should be padded using *cudaMalloc3D*. WARIS-Transport performance is very dependent of the amount of work per GPU, then it would be interesting to measure weak scalability with more than one node. Currently, only GPU per MPI rank is used to facilitate the multi-GPU development. This comes with benefits, like obtaining a multi-GPU implementation from day zero. And with some drawbacks like slow GPU-to-GPU memory copies (GPU to CPU plus CPU to CPU plus CPU to GPU copy). To solve this problem we have two possibilities. One is to use a CUDA-Aware MPI implementation where MPI calls detect if a pointer is a GPU pointer and therefore automatically trigger a GPU-to-GPU copy without CPU intervention. Another possibility would be to enable more than one GPU per MPI rank and perform GPU-to-GPU ourselves.

PELE right now has a problem with non-bonding list generation (they take almost 30% of the application). Porting this part to the GPU is a tough task because is a complex algorithm with a lot of corner cases that don't allow the GPU to fully exploit it's potential. Maybe with a simpler list creation, we could offload it to the GPU and achieve better performance. Another thing to explore would be to replace the current lists for some structure that expresses spatial locality to avoid the overhead of recalculating the non-bonding lists. After this, more energy terms could be accelerated to increase GPU data locality.

Bibliography

[1] NVIDIA. Pascal p100 gpu block diagram. http://core0.staticworld.net/images/article/2016/04/pascal-100-gpu-block-diagram-100654508-orig.png. Accessed: 2016-4-5.

[2] NVIDIA. Pascal sm. https://cms-images.idgesg.net/images/article/2016/04/pascal-100-gpu-streaming-multiprocessor-100654507-orig.png. Accessed: 2016-4-5.

[3] NVIDIA. *CUDA Programming Guide 7.5*, 2016.

[4] Boeing. Safe, efficient flight operations in regions of volcanic activity. http://www.boeing.com/commercial/aeromagazine/articles/2011_q3/2/img/. Accessed: 2016-4-02.

[5] R. de la Cruz, M. Hanzich, A. Folch, G. Houzeaux, and Cela J.M. *Numerical Mathematics and Advanced Applications - ENUMATH 2013: Proceedings of ENUMATH 2013, the 10th European Conference on Numerical Mathematics and Advanced Applications, Lausanne, August 2013*, chapter Unveiling WARIS Code, a Parallel and Multi-purpose FDM Framework, pages 591–599. Springer International Publishing, Cham, 2015.

[6] Matthew C Zwier and Lillian T Chong. Reaching biological timescales with all-atom molecular dynamics simulations. *Current Opinion in Pharmacology*, 10(6):745 – 752, 2010. Endocrine and metabolic diseases/New technologies - the importance of protein dynamics.

[7] A. Folch, M. Suaya, A. Costa, and J. Viramonte. The FALL3D Ash Cloud Dispersion Model and its Implementation at the Buenos Aires VAAC. *AGU Fall Meeting Abstracts*, December 2009.

[8] Raúl De la Cruz. *Leveraging performance of 3D finite difference schemes in large scientific computing simulations*. PhD thesis, Universitat Politècnica de Catalunya. Departament d'Arquitectura de Computadors, http://hdl.handle.net/10803/325139, 11 2015.

[9] S.Williams. The roofline model: A pedagogical tool for auto-tuning kernels on multicore architectures. *Hot Chips*, 2008.

[10] Paulius Micikevicius. 3d finite difference computation on gpus using cuda. In *Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 79–84, New York, NY, USA, 2009. ACM.

[11] G Ganser. A rational approach to drag prediction of spherical and nonspherical particles. *Powder Technology*, 77:143–152, 1993.

[12] D.A. Case, T.E. Cheatham III, T. Darden, H. Gohlke, R. Luo, K.M. Merz Jr., A. Onufriev, C. Simmerling, B. Wang, and R.J. Woods. The amber biomolecular simulation programs. *Journal of Computational Chemistry*, 26(16):1668–1688, 2005. cited By 3138.

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

[14] S. Pronk, S. Páll, R. Schulz, P. Larsson, P. Bjelkmar, R. Apostolov, M.R. Shirts, J.C. Smith, P.M. Kasson, D. Van Der Spoel, B. Hess, and E. Lindahl. Gromacs 4.5: A high-throughput and highly parallel open source molecular simulation toolkit. *Bioinformatics*, 29(7):845–854, 2013. cited By 1148.

[15] Kenneth W. Borrelli, Andreas Vitalis, Raul Alcantara, and Victor Guallar. Protein energy landscape exploration. a novel monte carlo based technique. *Journal of Chemical Theory and Computation*, 1(6):1304–1311, 2005.

[16] † Kai Zhu, † Michael R. Shirts, † Richard A. Friesner, , and ‡ Matthew P.Jacobson*. Multiscale optimization of a truncated newton minimization algorithm and application to proteins and protein-ligand complexes. *Journal of Chemical Theory and Computation*, 3(2):640–648, 2007. PMID: 26637042.

[17] NVIDIA Duane Merrill. Cub library. `https://nvlabs.github.io/cub/`. Accessed: 2016-4-14.

[18] Ligand binding mechanism in steroid receptors: From conserved plasticity to differential evolutionary constraints. *Structure*, 23:2280–2290, 2015.

[19] NVIDIA Mark Harris. C++11 in cuda: Variadic templates. `https://devblogs.nvidia.com/parallelforall/cplusplus-11-in-cuda-variadic-templates/`. Accessed: 2016-5-17.

[20] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: A proposal for pprogramming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[21] NVIDIA. *Kepler GK110 Whitepaper*, 2012.

[22] NVIDIA. *TESLA K40 GPU Active Accelerator - Board Specification*, November 2013.

Appendix A

Appendix

A.1 WARIS-Transport Governing Equations

$$\frac{\partial C}{\partial t} + \frac{\partial}{\partial x}UC + \frac{\partial}{\partial y}VC + \frac{\partial}{\partial z}WC = \frac{\partial}{\partial x}(K_x\frac{\partial C}{\partial x}) + \frac{\partial}{\partial y}(K_y\frac{\partial C}{\partial y}) + \frac{\partial}{\partial z}(K_z\frac{\partial C}{\partial z}) + S \tag{A.1}$$

Equation A.1 is the basic representation of a General transport equation. Given C is the transported quantity. First term ($\frac{\partial C}{\partial t}$) is the change in concentration in time at x,y,z. Terms 2,3 and 4 ($\frac{\partial}{\partial x}UC + \frac{\partial}{\partial y}VC + \frac{\partial}{\partial z}WC$) are changes due to wind advection, and terms to the right of the equation are changes due wind diffusion.

The computational domain consists of a 3D brick of n_x, n_y, n_z points. Boundary conditions assume free flow in the computational domain. This is important to avoid absorption or reflection from these boundaries. In boundary nodes different outgoing and incoming flux conditions are imposed following equation A.2.

$$x = x_0 \begin{cases} C_0jk = 0 & \text{if } U_1jk >= 0 \\ C_0jk = C_1jk & \text{if } U_1jk < 0 \end{cases} \tag{A.2}$$

The solving algorithm considers a Lax-Wendorff scheme for advection (second order accurate in space and time) with a slope-limiter minmod function whereas the diffusive terms are evaluated using a central difference scheme accounting for variable turbulent diffusivity tensor.

Given a point of indexes (i, j, k) in a Cartesian grid, and a value in step n of C^n_{ijk}, the value at $t = t + \Delta t$ is:

$$C_{ijk} = C_{ijk} + \Delta t S_{ijk} + f(C, U, V, W, K_x, K_y, K_z) \tag{A.3}$$