Ecole Polytechnique Fédérale de Lausanne

Computational Science and Engineering

Master Project

---

iCeBOUND Project - Porting a Solar Energy Potential Algorithm on GPU

Author:
Seif Ben Bader

Supervisors:
Pr Jan Hesthaven
Pr Nabil Abdennadher

Assistants:
Dr John White
Dr Gilles Fourestey

Sciper Number: 186962

July 6, 2016

Contents

Acknowledgements

I would like to thank first of all Professor Jan Hesthaven for all his confidence and support. His sense of humor and serene attitude had always been very appreciated for all his lectures I attended. The way Professor Jan Hesthaven interprets teaching and relations with students are definitely to be considered as an example.

I thank Professor Nabil Abdennadher for having giving me the opportunity to work on this exciting project, leaving me full options and freedom of work. Our collaboration was to me very pleasant and instructive. Going to the Hepia was always a great moment with all the great atmosphere Professor Nabil Abdennadher managed to set.

I thank Doctor John White for all his support and supervision. I appreciated a lot our collaboration. His easy goingness and open mindness were very helping. I thank him also for the great effort in reading and correcting my work.

My biggest thanks goes to Doctor Gilles Fourestey. I thank him for all his time and devotion. I rarely had such a big support for a project and that definitely strengthened my motivation in doing better and better. I salute his talent, his passion and brilliant ideas. Our discussions were always productive. Doctor Gilles, hats off!

Last but not least, to that tiny family on the other side of the Mediterranean sea, I send all my love.

Abstract

This project deals with the assessment of solar energy potential for a town, in this case Geneva, Switzerland. It essentially treats the Sky Viewing Factor (SVF) calculation of each position in the town, as the core and heaviest part of the solar energy potential model. This algorithm, initially implemented and tested in Java over a long period of time, was ported during this project to the Compute Unified Device Architecture (CUDA) platform for GPUs. Furthermore, the Message Passing Interface (MPI) was included in order to allow for execution on multiple-GPUs, resulting in a time-performance advantage to the original Java version. The execution time was reduced from hundreds of minutes to few seconds. In addition, the application was deployed on the cloud and the performance-to-cost for execution on such a platform was evaluated. This results in a financial saving along with the reduction of execution time when comparing with the Java version. Finally, a proposition is made for a more efficient and elegant SVF calculation for a whole city in one computing task.

1 Introduction

Today cities of developed countries are increasingly digitized as 3D urban numerical models. However, the use of these models as a technological support for different applications related to the fields of environment, energy and urban planning, remain under-utilized. The assessment of solar potential (solar mapping), underlined here as solar energy potential is an application based on the computation of 3D urban numerical models. The main goal is to use this application for different stakeholders, such as:

1. Local collectivities, considering distinct needs and purposes;

2. Industrial and energy companies doing business at local collectivity scales.

The extraction of three-dimensional spatial indicators adjusted for this application is a research challenge. It is done using different methods of aggregation of data. Indeed, the use of these particular spatial indicators allows a more refined analysis of the urban fabric and guarantees that output results are adjusted to end-user needs.

During the iCeBOUND project, a Decision Support System (DSS) that leverages 3D digital urban data to facilitate environmental analyses in cities was developed by researchers at HES-SO (Hepia), aiming at providing decision makers with relevant indicators for city planning and energy management. This DSS consisted in developing an application written in Java dealing with the solar energy potential application, provided as a cloud service. In a search to improve the DSS speed, a study has been made to run on Graphical Processing Units (GPUs). This is the subject of this project. First, by trying to speed up the original application regardless of the cloud specifics, and ultimately by evaluating the best option in terms of time to cost for a deployment on such a platform.

The report will be organised as follows: Section 2 describes our use case, the solar potential application in its general aspects. It focuses on the end-user features that should be fulfilled. Section 3 deals with the "Shading Algorithm", the fundamental algorithm of the application. It will introduce two approaches, both of them suitable for GPU implementation. Section 4 is a further explanation of the solar radiation model and its heaviest part, the Sky View Factor (SVF). Section 5 is a small introduction to the GPU programming, in order to introduce some of the basic aspects and to explain what kind of algorithms are better suited for such a programming model. Section 6 will be about the GPU implementation of the Shading Algorithm, explaining in a first step why is it adaptable to the GPU approach, and showing some outputs. Section 7 treats of the GPU implementation for the SVF calculation. Section 8 move on to an implementation on multiple GPUs, detailing different parallelization strategies and running a performance analysis. Section 9 is about the cloud deployment and the time to cost analysis. We end up with suggesting an optimal solution designed for a straightforward calculation of the SVF of a whole city.

2 Solar potential

Recently increasing attention given to environmental issues in urban studies shows that city planners are convinced that cities play a leading role in controlling sustainability. Our use case, "Solar Energy Potential", consists of evaluating the potential of house and building roofs located in urban areas for producing solar energy (both thermal and photovoltaic). The creation of solar maps of cities allowing to accurately evaluate areas that can be used for the installation of renewable energy, such as solar panels on building roofs, is currently considered as highly relevant [10]. This provides an automated and dynamic solar radiation calculation in order to provide an accessible platform for many different stakeholders. The goal of the solar potential use case is to develop a decision support system that leverages 3D urban data to facilitate environmental analysis of cities or regions. In other words, this project aims at providing relevant ecological information for efficient solar panel installations in cities.

For this purpose, the solar energy potential is calculated by summing the estimated value of the solar radiation in kWh by hour, for every hour during a time span at a given place. To compute accurately the amount of solar energy that can be produced by a solar panel, two families of parameters have to be taken into account:

1. Natural parameters: weather, sky visibility and sun position;
2. 3D urban numeric data.

Weather is represented by a weather data file, which is a set of coefficients based on the previous years measurements of each month. This information allows to take roughly into account the typical weather of each month. The sky visibility or Sky View Factor (SVF) is calculated by a dedicated algorithm, which will be detailed later. Concretely speaking, solar radiation is computed using:

1. The location (in latitude);
2. The weather;
3. The surrounding relief and landscape: mountains, hills, etc;
4. 3D urban numeric data: buildings' structure
5. The distant and close hourly shading (at a given position);
6. The ratio of the visible sky (at a given position): the SVF.

The four first points are input data while the two last points are calculated thanks to the Shading Algorithm. This algorithm is observed to be irregular and time consuming.

The cities are digitized and divided into several sub-areas called tiles. Each tile is represented by a matrix that describes the targeted sub-area. Matrices are stored in a Geolocalized TIFF (GeoTIFF) format:

- The Digital Urban Surface Model (DUSM) that describes the elevation of the buildings on each point of a given tile. This file has a resolution of 0.5 m. It represents the structure of the city such as the buildings and the houses for calculating close shadings.
- The Digital Terrain Model (DTM) which describes the elevation of the ground on each point of a given tile. This file has a resolution of 0.5 m. It represents the terrain relief such as the mountains for calculating distant shading.
- The slope matrix which describes the slope of each point (value between 0 and 90°) that belongs to a roof of a given tile. A point with a slope of 0° belongs to a flat roof whereas a point with a slope of 90° belongs a facade. This file is used to understand more accurately the structure of the tile surface.
- The orient matrix which describes the orientation of each point (value between 0 and 360°) that belongs to a roof of a given tile. A point with an orientation of 0° is north oriented whereas a point east oriented has an orientation of 90°. That is, the orientation is growing in clockwise direction.

The solar radiation is the sum of three components: The direct solar radiation, the diffuse solar radiation and the reflected solar radiation. The direct radiation is directly proportional to the sun visibility. The reflected radiation depends on the ground and the nearby object reflection. The diffuse radiation is derived from the sky visibility. Figure 1 illustrates the three components of the solar radiation that are involved in the calculation of the solar energy potential.

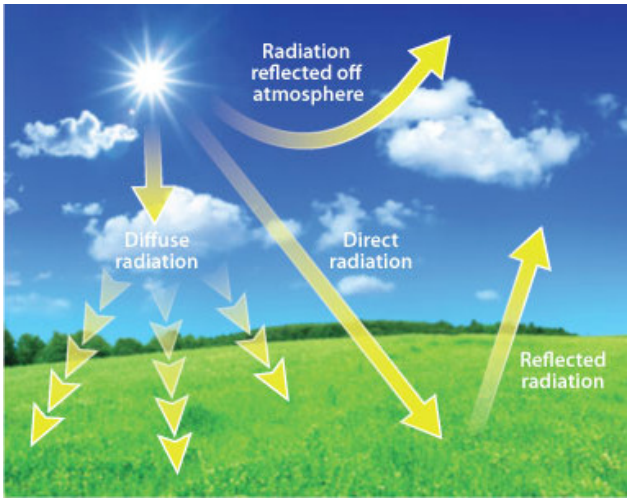

Figure 1: Three components of the solar radiation: direct, reflected and diffuse radiation.

The altitude and azimuth of the sun (i.e the sun position) is used to compute the direct solar radiation. The sky visibility (or Sky View Factor - SVF) is used to compute the diffuse radiation. SVF describes the amount of visible sky from a point. SVF is calculated by using a sky model. The sky model represents the sky as a cupola composed of different light sources (the more light sources are in the sky model, the more accurate will be the SVF). A light source is given by its azimuth and altitude component. For instance, a point located in an urban area has a SVF close to 0 whereas a point located in a rural area has a SVF close to 1. Figure 2 illustrates the above example.

The outcome of “Solar Energy Potential" is a solar map of the given tile which corresponds to the level of solar energy that can be produced by a solar panel on each point of the tile in kWh. Figure 3 shows the result of a yearly solar radiation study of the residential zone of Meyrin, Switzerland. Bright points are sunny areas that have a significant solar potential.

Figure 2: Illustration of the sky view factor in a rural and an urban environment.

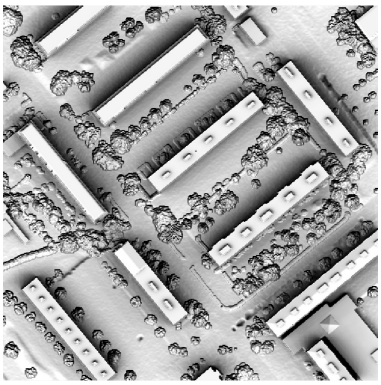

Figure 3: Yearly solar energy map of the residential zone of Meyrin, Switzerland.

In general, roofs are the most efficient place to put solar panels in order to generate solar energy. Nevertheless, when facades are south oriented, they also provide a good place to install a solar panel. Therefore, solar radiation maps can also be carried out for facades. This is considered as a further research interest, and is beyond the scope of this project.

3 Shading Algorithm

This section explains the Shading Algorithm used in the solar potential application. This algorithm calculates the shadow map of a given tile according to a given light source (sometimes referred to as as a sun) or signal source. This map contains Boolean values that represent the state of each point in the tile (0 for shaded, 1 otherwise) for a given light/signal source. Figure 5 shows three calculated shadow maps for an area of Geneva at 8 am, 2 pm and 5 pm.

The purpose of a Shading Algorithm is to discover, for each point P that belongs to an urban model (i.e DTM or DUSM), which other point P' is shading it according to a given light/signal source. Figure 4 shows

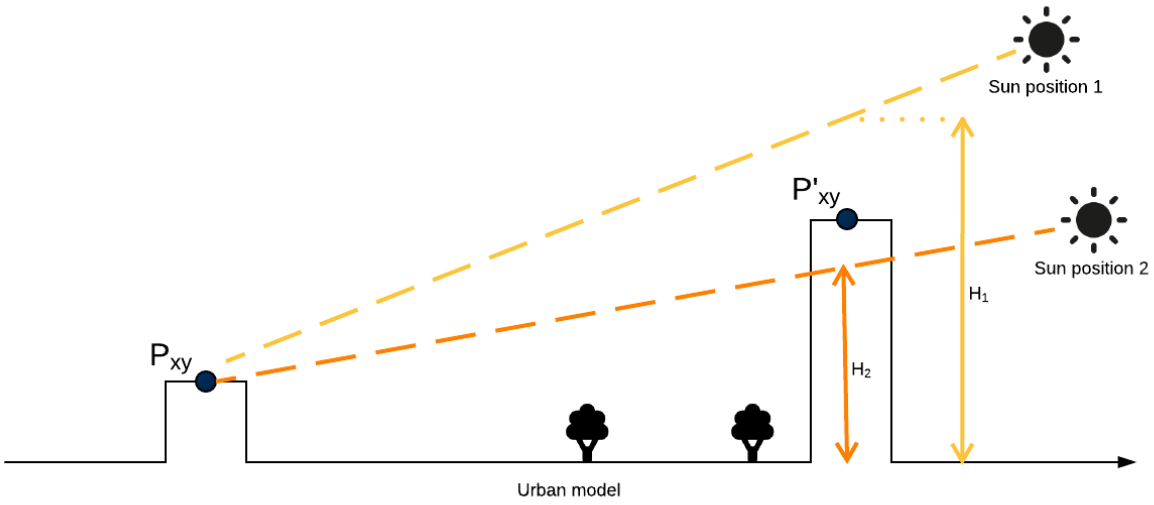

Figure 4: Principle of the Shading Algorithm.

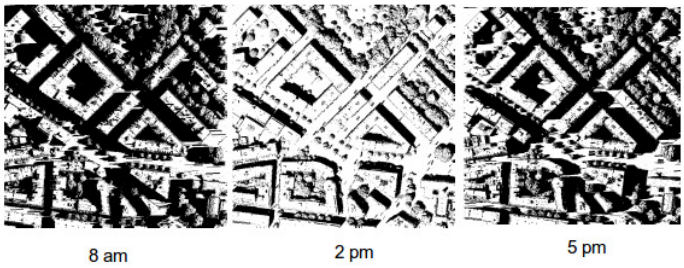

Figure 5: Calculated shadows for an area of Geneva at three times of the day.

the algorithm explained above. When the sun is in position 1, H1 is higher than P' and P is sunny. On the other hand, when the sun is in position 2, the height H2 is less than that of the building (point P'), P is shaded.

The Shading Algorithm uses three input data: The light source position, the DUSM or DTM and the latitude of the location of the given model. This algorithm is observed to be an irregular time consuming algorithm: the complexity of the algorithm depends on sun position. The output of this algorithm is a shadow map that contains the state of each point that belongs to a tile; either shaded or sunny.

Two paradigms are considered for implementing the Shading Algorithm: The "shadow receiver" and the "shadow emitter".

3.1 Shadow receiver approach

The principle of the "shadow receiver" paradigm is as follows: for a given light source and a given point P, the algorithm proceeds in the direction of the light source and checks if the encountered points (buildings, trees, obstacles) hide P (see Figure 4). Therefore, this algorithm is an iterative search, for each point that belongs to a tile, of an obstacle inside its neighbourhood that hides P [8].

For each point P_{xy} belonging to the tile, this point by point paradigm behaves as follows:

1. The direction of the light source beam is expressed as the light source beam vector $\begin{pmatrix} dx \\ dy \\ dz \end{pmatrix} = \begin{pmatrix} \sin(\phi) \\ \cos(\phi) \\ \tan(\gamma) \end{pmatrix}$, as we can see from Figure 6 given that ϕ is the azimuth and γ is the altitude.

2. The neighbour P' of a given point P at given step i is given by:
   $P'_{xy} = \begin{pmatrix} P_x + round(i * dx) \\ P_y + round(i * dy) \end{pmatrix}$
   The height of P'_{xy} (i.e P'_z) is given by its value in the urban model. i is an iterator that starts from 1 and ends either when P' is outside the tile or $P_z + i * dz$ is greater than the highest point of the tile or P'_z is greater than $P_z + i * dz$ (i.e the neighbor is shading P).

3. A point is shaded if at least one of his neighbour is higher than $P_z + i * dz$

3.2 Shadow emitter approach

The Shadow Emitter paradigm simply consists of calculating, for a given light source and for each point P_{xy} belonging to the tile, the list of points that are shaded by P_{xy}: $LIST_{xy}$.

A point P, belonging to the tile, is sunny (for a given light/signal source) if it does not belong to any lists $LIST_{xy}$. In other words, P is not shaded by any other point P_{xy}.

Unlike the first alternative, the algorithm proposed in this section calculates the shaded points through a given point P_{xy} and not the shadows that fall on P_{xy} from nearby points.

This algorithm can be divided into two steps:

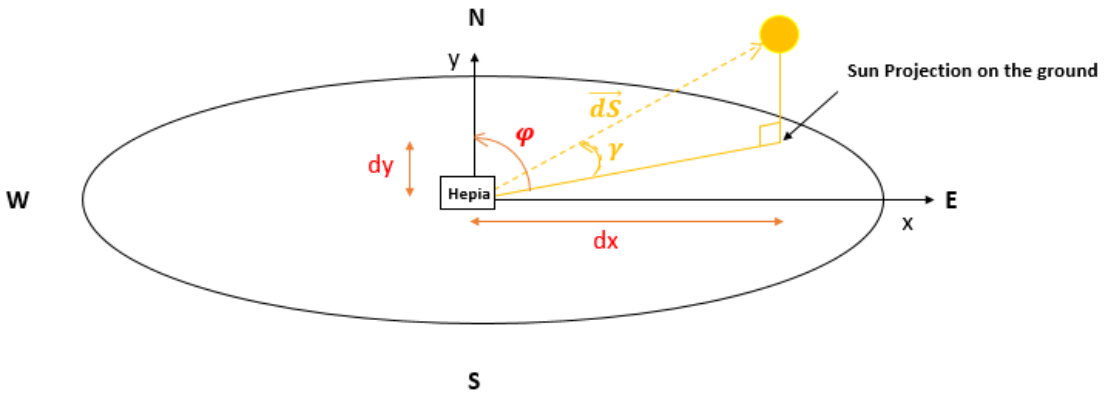

Figure 6: Light source beam direction

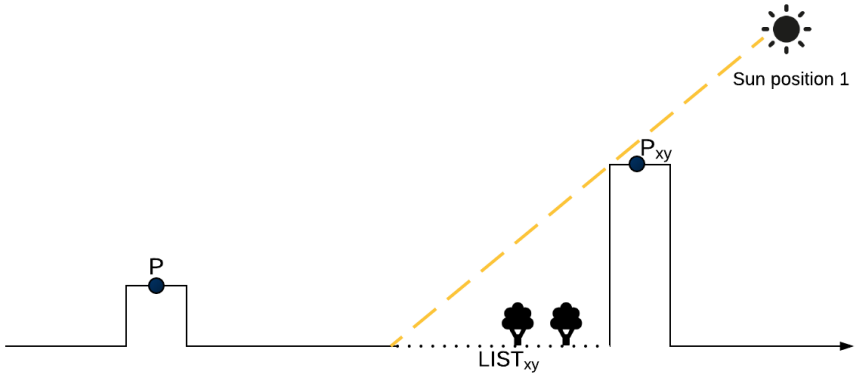

Figure 7: Principle of the shadow emitter paradigm.

1. “shaded area identification": This step consists of identifying, for each point that belongs to a tile, what points are shaded by it (i.e identifying the $LIST_{xy}$) in function of a given light/signal source. For this purpose, the algorithm follows the opposite light source beam direction towards the ground and checks if the encountered points (buildings, trees, obstacles) are shaded by P_{xy}, in this case they are added to $LIST_{xy}$ (see Figure 7). The encountered points are computed iteratively and are given by:

   $$P'_{xy} = \begin{pmatrix} P_x - round(i * dx) \\ P_y - round(i * dy) \end{pmatrix}$$

   dx, dy and dz are the light source beam components. The height of P'_{xy} is given by its value in the urban model. i is an iterator that starts from 1 and ends either when P'_{xy} is outside the tile or if the source beam reaches the ground. Therefore, P'_{xy} is shaded if it is lower than $P_z - i * dz$.

   At this step, each point knows the points that it shades.

2. "merging" : This step consists to identify the points that belongs to any $LIST_{xy}$ and those that are shaded in order to construct the shadow map.

Figure 8 shows an example of the three steps defined above. Notice that steps two and three can be done at the same time.

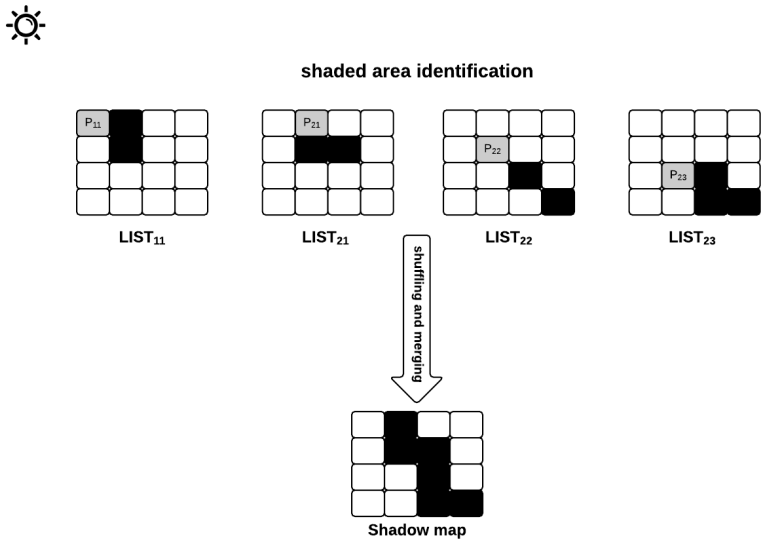

Figure 8: Example of steps in the shadow emitter paradigm. P_{11}, P_{21}, P_{22} and P_{23} are higher than the ground, all the other points do not produce any shadow (i.e $LIST_{xy}$).

4 Solar radiation and SVF

4.1 Solar radiation

The factor of solar radiation is a relevant information in case of thermal and photovoltaic, energy production. Hence, it becomes important to calculate an accurate estimation of the energetic production capability of roofs and facades. For this purpose, solar radiation maps are built from the input data specified in Section 2.

Solar radiation $SR_l(P)$ at a given point P depends on several factors, such as:

- Local weather;
- Latitude, slope and orientation of P (assuming that P is a roof point or a facade point);
- Sun position (denoted by l);
- How much shade falls on P from nearby trees and buildings.

As stated in Section 2, three types of radiation make up the total solar radiation: direct, reflected and diffuse. Formula 4.1 provides a simplified version:

$$SR_l(P) = I_{dir} * S_l(P) + I_{ref} + I_{dif} * (f(BPV(P), SVF(P)) + g(S_l(P))) \tag{4.1}$$

I_{dir}, I_{ref}, and I_{dif} are three constants which depend on weather data, latitude, slope and orientation of P. Functions f and g are respectively the circumsolar and the isotropic components of the diffuse radiation [11]. For a given sun position l, $S_l(P)$ is equal to 0 if P is shaded, 1 otherwise. The Sky View Factor (SVF) will be explained further.

Formula (4.1) is divided into three parts:

1. Direct radiation ($I_{dir} * S_l(P)$) is directly proportional to the sun visibility;
2. Radiation reflected from buildings and ground is given by I_{ref};

3. Diffuse radiation ($I_{dif} * (f(BPV(P), SVF(P)) + g(S_l(P)))$) is derived from the sky visibility, direct sun visibility and the best possible view, which is described in Section 4.2.

The value of the direct solar radiation is zero if P (belonging to a roof) is shaded. The Shading Algorithm is used to determine whether P is shaded or not.

The reflected solar radiation represents the amount of light reflected by the environment such as buildings or the ground. It has a small impact on the total solar radiation.

The diffuse solar radiation depends on the ratio of the sky visible at P. When P is located among several obstacles, its sky visibility is limited (close to 0). On the other hand, if P is an open point (located outside urban areas for example) its sky visibility is close to 1. The sky visibility is modeled by the SVF that has a value between 0 and 1 (Figure 2). To clarify, the SVF is the percentage of sky that can be seen from a specific point.

The solar radiation calculation is based on three distinct processes:

- The first one is the SVF calculation which involves hundreds of shadow processes (Shading Algorithm) for an accurate estimation of the SVF.
- The second process is the hourly shading calculation which involves one shadow process per hour during a given time span (i.e a year, a month) for establishing the direct sun visibility.
- The results of these two processes are the main components of the diffuse radiation and the direct radiation respectively. Therefore, the last process consists in creating the solar map using the previous results.

Figure 9 shows the functional diagram of this use-case.

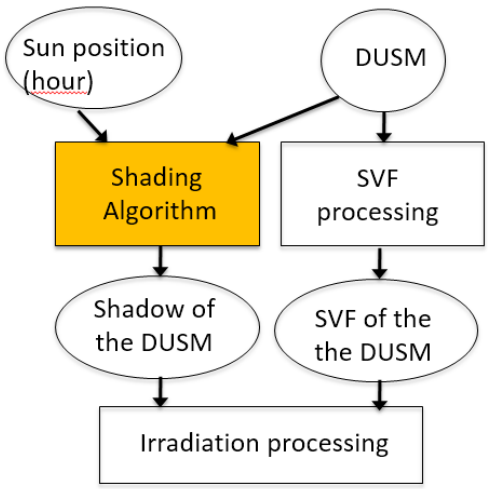

Figure 9: Functional diagram of the "Solar Energy Potential"; Sun position and DUSM are inputs of the shadow and SVF processing; The colored square is a part of the generic core engine of iCeBOUND which is involved the Solar Energy Potential.

The next section will explain the computation of the SVF and the input data that is involved in its computation.

4.2 SVF

As stated in the previous section, the sky view factor is the percentage of sky that can be seen from a specific point. By definition, each point of the DUSM has a corresponding SVF but in our case, only the roofs points are points of interest. The SVF at point P (SVF(P)) is calculated as follows [9]:

- The sky is modeled by a cupola having different suns (light source) uniformly placed on the cupola;
- The Shadow Algorithm is executed for each light source in the sky (cupola);
- SVF(P) is the average of all shadows calculated at point P.

The more light sources in the cupola, the more accurate is the SVF. In our case, the number of light sources in the sky is set to 400. Moreover, the SVF can be formalized by the following equation:

$$SVF(P) = \frac{1}{L} \sum_{l=1}^{L} S_l(P) \tag{4.2}$$

where l is the identifier of the light source, L is the number of light sources in the sky model. As a reminder, $S_l(P)$ denotes the shading state of the current point for the given sun (true = 0, false = 1).

Overall, the SVF is a suite of executions of the Shadow Algorithm which is irregular and time consuming. Therefore, SVF needs resources with high performance computing capabilities.

In addition, it is important to notice that the SVF can not be larger in value than the best possible view (BPV) which can be computed as follows:

$$BPV(P) = 1 - \frac{\beta(P)}{180} \tag{4.3}$$

where $\beta(P)$ is the slope of the point P This value is used in the solar radiation formula (described in Section 4.1) to reduce the impact of the slope on the isotropic component of the diffuse radiation.

The SVF process is done on every point of the targeted model and the output of this algorithm is a SVF map of the whole tile. Figure 10 shows an example of SVF map on an area of Geneva.

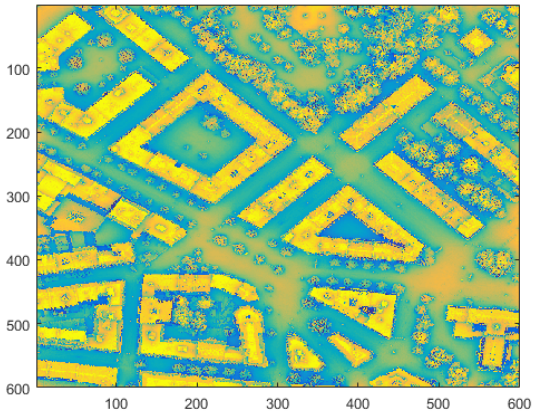

Figure 10: Output of the SVF algorithm

5 GPU programming concept

This section is dedicated to introduce GPU programming and present some of its main aspects. It explains why such a programming model can result in a speed up of applications, especially for problems exhibiting big data parallelism.

5.1 CUDA for GPU programming

GPU programming was made possible by Nvidia with the appearance of Compute Unified Device Architecture (CUDA). CUDA is a C++ language extension providing Application Program Interfaces (APIs) that facilitate the communication between the CPU and the GPU. The main goal is to execute computations on the GPU instead of the Central Processing Unit (CPU).
Since CUDA is compatible only with Nvidia graphic cards, Apple suggested that a unified language was developed in order to be able to program on any graphic card. This ended up with Khronos Group developing the OpenCL, that can communicate with any GPU [6]. Though, CUDA still remains very used and very popular. In this project, CUDA was used and we relied entirely on Nvidia GPUs.

5.2 Architecture of a GPU

An explanation of the GPU architecture is essential in understanding its programming model. The main components of a GPU consist in its SMXs (SM is an acronym for Streaming Multiprocessors), represented here in Figure 11. Most essentially, each SMX comes with a number of CUDA cores (sometimes referred to as Streaming Processors) and every CUDA core runs a thread, under the control of a thread execution manager that distributes the tasks among the different building blocks. Thus, the performance of a GPU mainly resides in its number of SMXs and CUDA cores. If we consider for example the Tesla K40 (a Kepler based GPU architecture, one of the recent CUDA-capable device architecture), we will find 15 SMXs, each of them made of 192 single precision CUDA cores and 64 double precision CUDA cores.

5.3 CUDA kernels and grid configuration

The main idea about the GPU programming is that the CPU, also called the "Host" in terms of GPU programming terminology, transfers the work to the GPU that spawns a thread per element for the same task, which are then executed. This model that consists in executing the same task with different threads simultaneously is referred to as Single Instruction Multiple Threads (SIMT) [20].
When a program is to be run on the GPU performing some function on some variables, memory has to be allocated on the "Device" (the GPU) and the variables need to be copied in this memory space. This is done through the CUDA APIs. In addition, the function itself needs to be adapted in order to execute on the GPU. This function, called a kernel function, has the particularity to address to a unique thread, i.e the operation to be performed simultaneously by multiple threads [1].
For instance, consider the implementation of a matrix addition function. A CPU version would loop over lines and columns, before adding the matrices elements corresponding to the current position. The code will therefore contain 2 loops. For the GPU kernel function, it will contain no loop as it is only concerned with a single output position. Indeed, every kernel is launched for a whole grid of threads which replaces the loops and the thread grid maps the matrices. CUDA APIs such as **threadIdx**, **blockIdx** and **blockDim** help keep track of the thread position in the grid, which can also be interpreted as the matrix position [2].

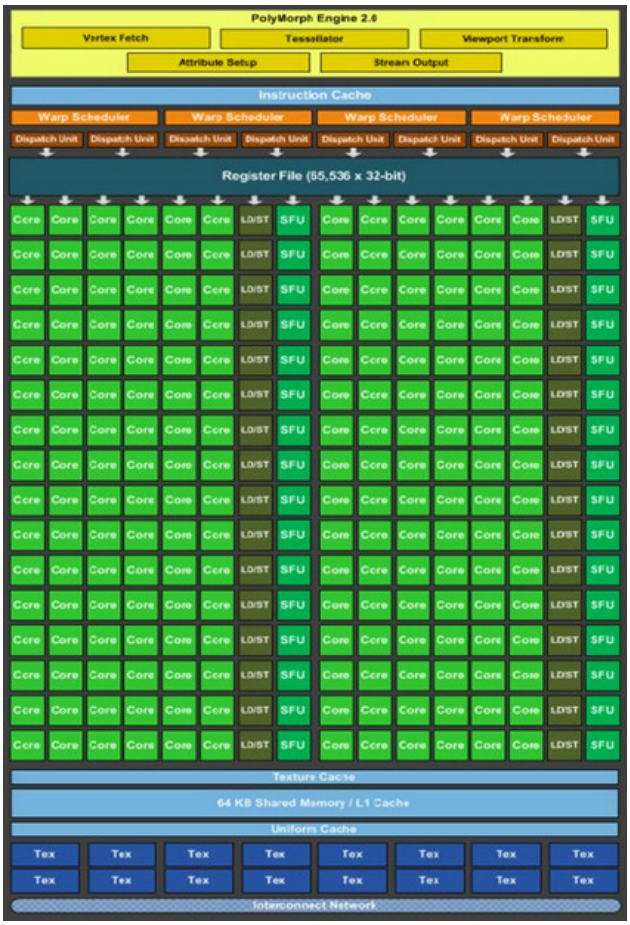

Figure 11: Overview of a Kepler SMX [3]

A grid is declared in a block fashion. These blocks containing threads, are then assigned to the GPU resources and executed in form of warps (group of 32 threads). In Figure 12, a 2D block grid matching with the matrix addition example is reported.
The kernel executed by the grid can be written in this way:

```
__global__ void Add(double** A, double** B, double** C)
{
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int ty = blockIdx.y * blockDim.y + threadIdx.y;
    C[tx][ty] = A[tx][ty] + B[tx][ty];
}
```

The kernel is then launched by specifying the grid dimension following this syntax:

```
Add<<<blockDim.x,blockDim.y>>>(A,B,C)
```

which fully determines the SIMT functioning of the CUDA programming model.
In addition, a programmer has to be aware that from a GPU to another, these parameters may vary:

- Maximum number of threads per SM;
- Maximum number of blocks assigned to an SM;

- Maximum number of threads per block;
- Maximum dimension size of a thread block;
- Maximum dimension size of a grid size.

Regarding that, a pre-analysis of the best grid configuration need to be done before running a program on a GPU. Indeed, some could allow more threads to work at a time than others when prevented by some of the limitations cited above [1].

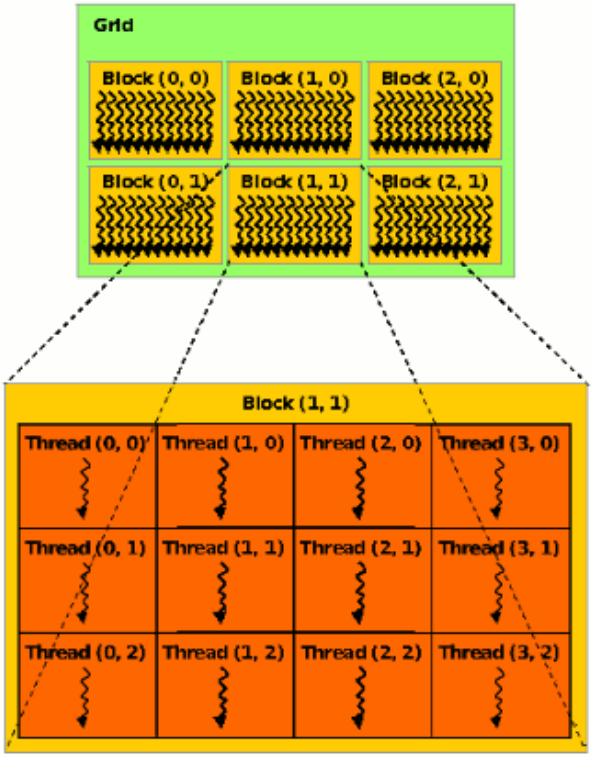

Figure 12: 2D grid for matrix addition kernel.

5.4 CUDA memory model

Another very important aspect in CUDA is the memory model. The Host communicates with the device through the Global Memory and the Texture Memory. These are shown at in Figure 13.
Global Memory is a Read/Write memory, whereas Texture Memory is Read Only memory. Therefore, Global Memory is the only way to produce output and communicate it to the Host. However, the Global Memory is an off-chip memory and exhibits more memory access latency. One should consider using the Texture Memory whenever a variable is to be read only, since it is cached on-chip. Other on-chip memories that might be worth using for reducing the memory access latencies, are the Shared Memory and the Registers. Shared Memory is at the block level, i.e all threads in a block have access to this memory. Variables that reside in Shared Memory can be accessed more efficiently, resulting in a high performance improvement. However, it necessitates that threads from the blocks highly communicate in order to be fully efficient, and is hence not adapted to every algorithm. It is also limited to 64Kb on the Kepler architecture, so it should be used with a special care. Registers are at the thread level, and consist basically of all the variables declared

at the kernel level, so that every thread will have its own copy, remembering that every thread will execute the same kernel.

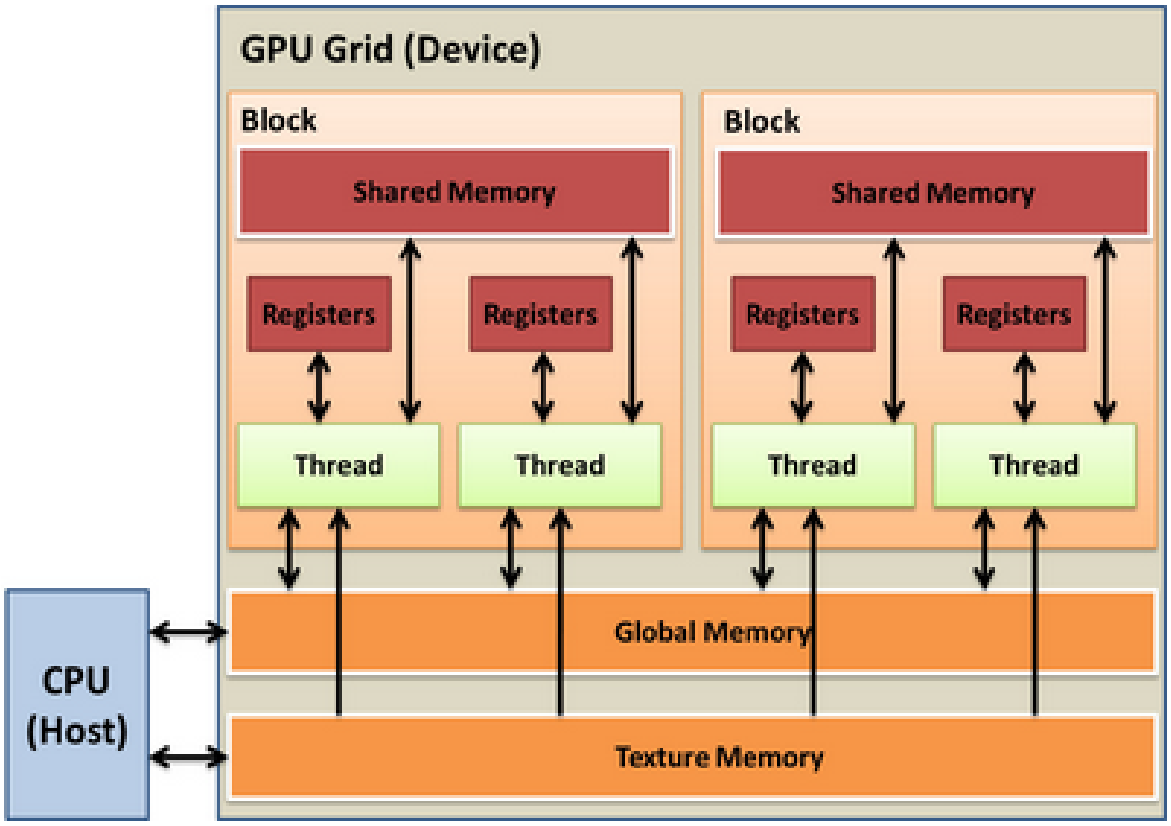

Figure 13: CUDA memories [15].

6 GPU implementation of the Shading Algorithm

Section 3 explained in detail both approaches for the Shading Algorithm. The receiver paradigm (c.f Section 3.1 page 5) acts in a straightforward manner, in the sense that every point in the tile is responsible for investigating whether it is shaded or not. The emitter paradigm (c.f Section 3.2 page 5) is an elegant formalism of the shade volume calculation performed in [8]. By shadow volumes is intended the neighboring points shaded by the original one. Both have advantages and drawbacks depending on the type of DUSM we are working on.

6.1 Advantages of the receiver approach

For an urban DUSM (a tile containing mostly buildings), the receiver approach is expected to work better since that every point in the tile is most probably to find out quickly whether it is shaded or not. Indeed, for high height values, few iterations would be sufficient for meeting the stopping condition, i.e the projected light beam becomes greater than the maximum DUSM height value. For small height values (points in streets for example), there is a big probability that we quickly meet an obstacle and end up being shaded, since we are surrounded with lot of buildings. In that case, the emitter algorithm would still have to complete the calculation of all these shadow volumes until they reach the floor.

6.2 Advantages of the emitter approach

For a rural DUSM (a tile containing few buildings), the emitter approach seems more adapted. The reason is that all these shadow volumes would be either small, or zero, which means that few iterations will be performed for almost every point in the tile. In that case, the receiver algorithm will perform much more iterations in order to meet the stopping condition, ending up with an increased execution time.

6.3 Receiver or emitter?

In the context of GPU programming, the memory management plays a primordial role. Performance can be influenced by the amount of memory passing back and forth, between the Host and the Device. For the emitter approach, one would have to use more memory at the Host level by declaring empty lists for every point in the tile. These need to be transferred to the Device, which will fill them and send them back to the Host. Furthermore, the Host code would have to perform an additional step that consists of merging all these lists. In addition to that, the major drawback of the receiver approach can be eliminated by setting a maximum iteration number, for which we assume that if the point is still not shaded at this point, then it never will. Considering all that, the receiver approach seems to be more adapted to GPUs. We will rely on this algorithm for further analysis.

6.4 Reference problems

The previous work related to the performance evaluation of the cloud-based algorithm version used the DUSM shown in Figure 14.
This tile represents a 3.4x3.4km^2 surface, in some part of Geneva with a resolution of 0.5m, hence a matrix of size 6800x6800. We will rely on this reference DUSM for all the comparisons with the previously established results, based on the Java implementation of the algorithm. Furthermore, and following the discussion in the previous subsection about the importance of the DUSM type, we are going to consider three additional DUSM's of urban,rural and hybrid kind. These are shown respectively in Figures 15, 16 and 17.

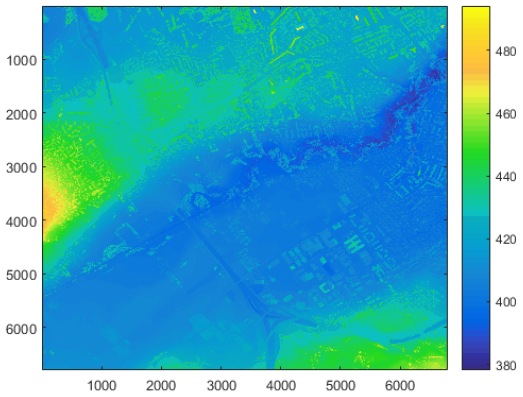

Figure 14: Reference DUSM.

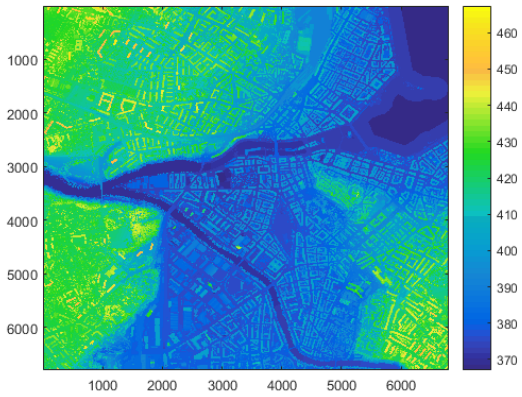

Figure 15: Urban DUSM.

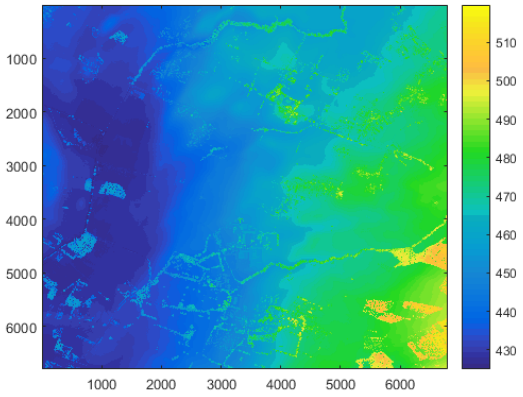

Figure 16: Rural DUSM.

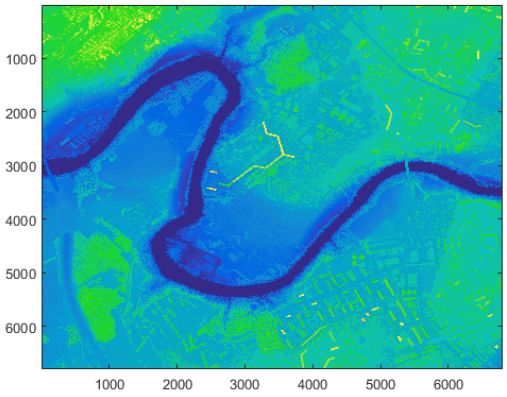

Figure 17: Hybrid DUSM.

6.5 Simulation outputs

We run the receiver version of the Shading Algorithm for the following parameters:

- Azimuth = 89°
- Altitude = 46°

We obtain the shadow map and iteration matrix in Figure 18.

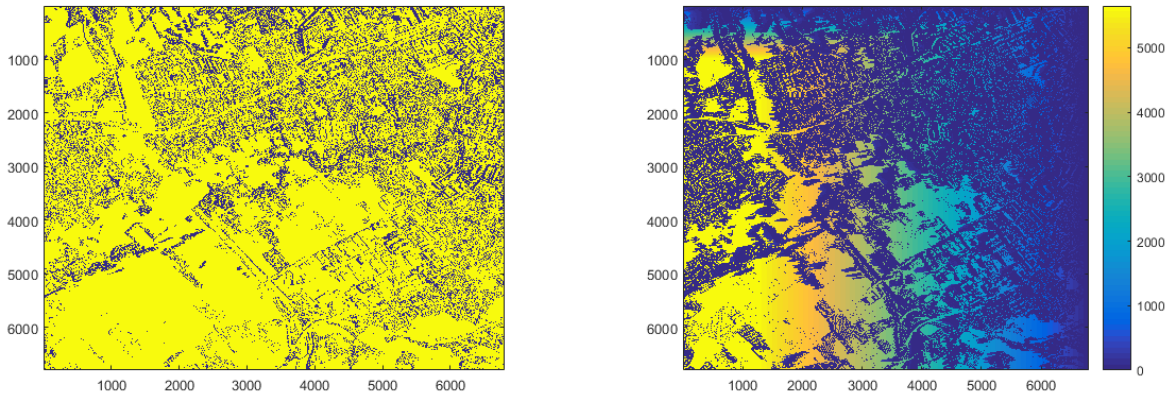

Figure 18: Shadow map (left, yellow = sunny, purple = shaded) and iteration matrix (right) for the Shading Algorithm on the reference DUSM.

It appears clearer now that the reference DUSM contains in its lower left corner a plain surface (rural part). This part looks mostly sunny, and one can see the difference of performed iterations in this zone, in comparison with other parts of the tile. Indeed, from the color map, the number of iterations in this area shows that we probably performed the iterative research until leaving the tile without meeting the stopping condition. The other locations exhibit instead a much lower number of iterations. This is in consistency with our expectations from the receiver approach of the Shading Algorithm.
In order to analyze the results more locally, we focus on different locations of the shadow map, shown in Figure 19.

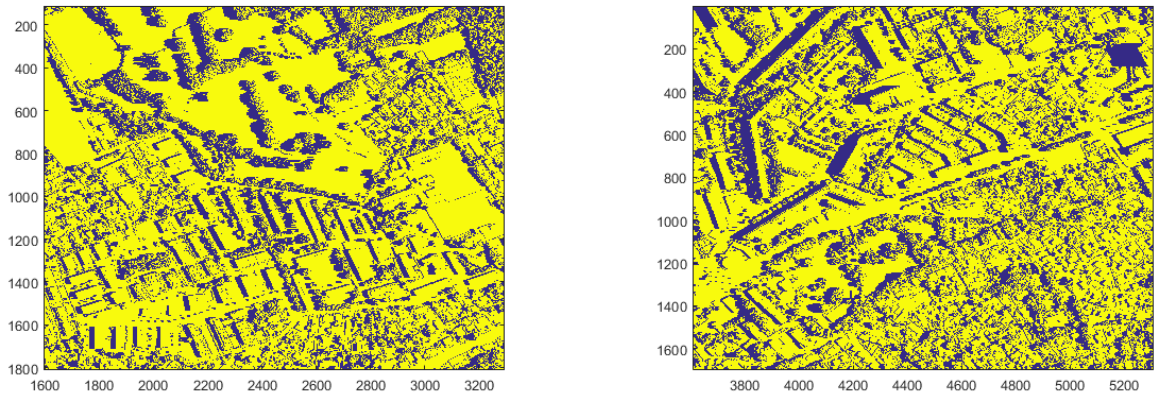

Figure 19: Focus on two different locations of the shadow map.

The azimuth parameter value corresponds to a sun position located in the east. The results are consistent with such a sun position, as the shadows lie on the left sides of the buildings and trees. The shadow maps for the Urban, Rural and Hybrid tiles are reported respectively in Figures 20, 21 and 22.

6.6 GPU Code implementation

Before the performance analysis, we dedicate this section to the explanation of the implementation aspects.

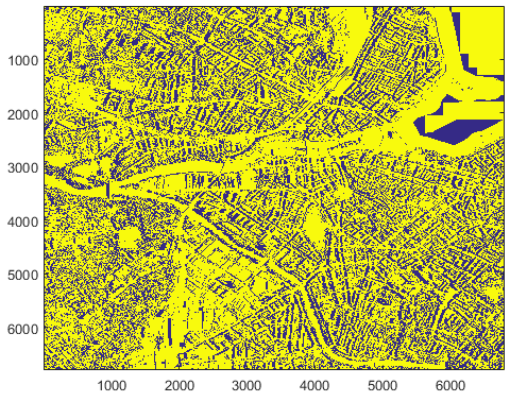

Figure 20: Shadow map for the Urban DUSM

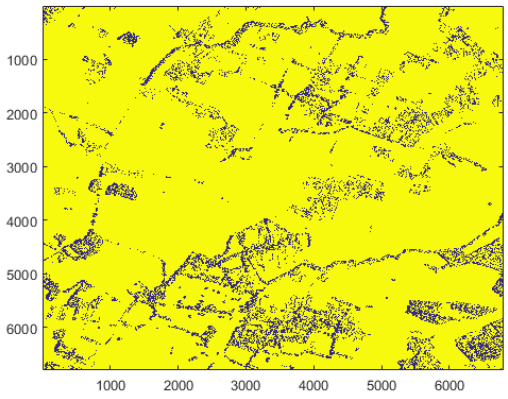

Figure 21: Shadow map for the Rural DUSM

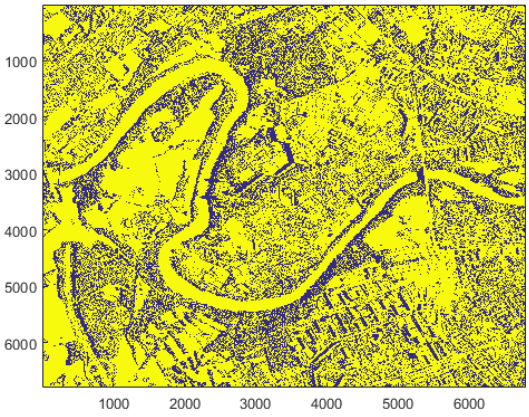

Figure 22: Shadow map for the Hybrid DUSM

6.6.1 Basic implementation

A straightforward CUDA program concept consists of sending all variables to the GPU via the Global Memory to perform the calculations. A basic implementation of the kernel function in this case, the one executed by all the threads, can be the following:

```
__global__ void pointShadowReceiver(double* shadowMap, double* DUSM, double maxDUSM, double dx, double dy
    , double dz, int Size)
{
        // Calculating thread identifier: position in the grid
        int Px = blockIdx.x * blockDim.x + threadIdx.x;
        int Py = blockIdx.y * blockDim.y + threadIdx.y;

        if (Px < Size && Py < Size)

        {
                // Starting the shadow process
                int Sunny = 1;
                int iter = 1;
                int Nx = Px + round(dx);
                int Ny = Py - round(dy);
                double sunBeamProj = dz;

```

```
17                  while (Nx >= 0 && Nx < Size && Ny >= 0 && Ny < Size && sunBeamProj < maxDUSM && Sunny !=
                        0)
18                  {
19                          Sunny = !(DUSM[Nx + Ny*Size] > sunBeamProj + DUSM[Px + Py*Size]);
20
21                          // Incrementing the iteration counter and the next neighboring points
22                          iter++;
23                          Nx = Px + round(iter*dx);
24                          Ny = Py - round(iter*dy);
25                          sunBeamProj = iter*dz;
26
27                  }
28                  shadowMap[Px + Py*Size] = Shade;
29          }
30  }
```

Line 1 is the kernel function declaration. Lines 4 and 5 are for getting the matrix position of the point of interest in the tile. **ThreadIdx** and **BlockIdx** are built-in CUDA APIs for getting the thread index and the block index in the grid set for executing the kernel. **BlockDim** gives the dimension of the grid in terms of blocks. Line 7 is a control instruction for not going beyond the tile's points. Indeed, the grid in general does not fit exactly with the tile size and a larger grid needs to be declared. This instruction is therefore essential in order to avoid segmentation faults. Line 11 is the variable **Sunny**, the output value for the considered point; 0 if the point is shaded and 1 if it is sunny. Line 12 declares the iteration variable to keep track of. Lines 13 and 14 computes the first neighbor point in direction of the sun, whereas Line 15 computes the light source beam projection at its level. Line 17 is the **while** instruction defining the stopping criterion. Algorithm stops whenever the neighbor gets outside of the tile, the light source beam becomes greater than the highest altitude in the tile, or simply when the point is determined as shaded. This is determined by the next line, Line 19. Line 22 increments the iteration variable. Lines 23,24 and 25 does the same than Lines 13,14 and 15 but for further iterations. Finally, Line 28 sets the output point value with **Sunny** in the variable **shadowMap**, naturally designating the final shadow map.

6.6.2 Pinned Memory

For the GPU to operate functions on data, the memory contents need to be sent from the CPU to the GPU [16]. When the CPU allocate Host Memory, this memory is pageable by default [18]. This type of memory cannot be read directly by the GPU, and the CUDA driver needs to allocate temporary Pinned Memory in which it copies the data, before sending it to the device (the GPU). This is is shown in Figure 23.

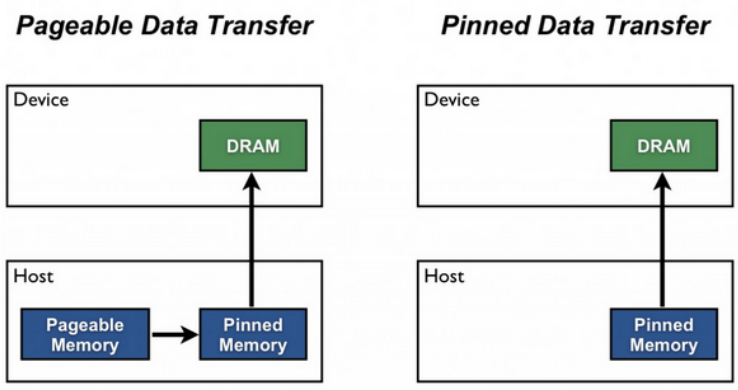

Figure 23: Pageable Memory conversion to Pinned Memory [16].

Pinned Memory is basically memory without swaps. This can be prevented by directly allocating the Host Memory as Pinned Memory. This is made possible by the CUDA API **CudaMallocHost()**, and allows for a higher transfer throughput [17]. These modifications are to be done on the Host code, the kernel function is unchanged.

6.6.3 Texture

As introduced in Section 5.4 page 12, the Texture Memory is a Read Only memory, cached on-chip. Therefore, it is suitable for variables that are read from only and should be theoretically reducing the memory access latencies in comparison with the Global Memory [19]. In our case, the variable DUSM fulfills this condition, and the following is the new modified kernel for such a purpose:

```
1  __global__ void pointShadowReceiverTex(float* shadowMap, cudaArray* DUSM, float maxDUSM, float dx, float
       dy, float dz, int Size)
2  {
3          // Calculating thread identifier: position in the grid
4          int Px = blockIdx.x * blockDim.x + threadIdx.x;
5          int Py = blockIdx.y * blockDim.y + threadIdx.y;
6
7          if (Px < Size && Py < Size)
8          {
9                  // Looping over the different sun positions
10                 // Starting the shadow process
11                 int Sunny = 1;
12                 int iter  = 1;
13                 int Nx    = Px - round(dx);
14                 int Ny    = Py + round(dy);
15                 float sunBeamProj = dz;
16                 float height_orig = tex2D(heightFieldTex, Px + 0.5f, Py + 0.5f);
17
18                 while (Nx >= 0 && Nx < Size && Ny >= 0 && Ny < Size && sunBeamProj < maxDUSM && Sunny !=
                       0 )
19
20                 {
21                         float height = tex2D(heightFieldTex, Nx + 0.5f, Ny + 0.5f);
22                         iter++;
23                         Sunny = !(height > sunBeamProj + height_orig);
24
25                         // Incrementing the iteration counter and the next neighboring points
26                         Nx = Px - round(iter*dx);
27                         Ny = Py + round(iter*dy);
28                         sunBeamProj = iter*dz;
29                 }
30
31                 shadowMap[Px + Py*Size] += Sunny;
32         }
33 }
```

Notice that the difference with the ancient kernel version reside in Lines 16 and 21. These are respectively storing the DUSM values of the original point **heightOrig** and the neighbor point **height** in the texture memory. The variable heightFieldTex needs to be declared as a 2D texture variable somewhere else in the Host code following this syntax:

```
texture<float, 2, cudaReadModeElementType> heightFieldTex;
```

The other modification is the use of single precision variables (floats) instead of double precision, since the texture can for now only be encoded in 32-bit variables (floats).

6.7 Execution times and performance analysis

In this section, we start by evaluating the optimisations detailed in Section 6.6, before analyzing the speed-up between the GPU version and the Java version. For this performance analysis, we relied on the Tesla K40 (c.f Section 5.2). We end up with analyzing the performance over different CUDA devices.

6.7.1 Pageable vs Pinned

As the execution time is most likely to depend on the sun altitude value rather than the azimuth, we fix the azimuth value and run the simulations for three different altitude values (in degrees):

- Azimuth = 89
- Altitude = 14, 46, 86

DUSM	Reference	Urban	Hybrid	Rural
Pageable (Alt = 14)	6.55	5.5	6.33	7.9
Pinned (Alt = 14)	4.29	3.25	4.06	5.5
Pageable (Alt = 46)	4.16	3.9	4.24	4.3
Pinned (Alt = 46)	1.9	1.7	1.8	2.02
Pageable (Alt = 86)	2.45	2.48	2.65	2.65
Pinned (Alt = 86)	0.22	0.22	0.22	0.22

Table 1: Execution times (in seconds) comparison for Pageable vs Pinned Memory for different altitude values

The execution times are given in seconds and concern only the kernel function. As expected, the Pinned Memory brought a clear speed up in every single simulation. The original execution times are however small, and we cannot state on a concise value for the speed up. It is though clearly a good option to consider Pinned Memory instead of Pageable Memory.

6.7.2 Texture vs Global Memory

We now run simulations of the same settings, for code with and without Texture Memory use. Both are using Pinned Memory, since it showed satisfaction in terms of execution times.

DUSM	Reference	Urban	Hybrid	Rural
Global Memory (Alt = 14)	4.29	3.25	4.06	5.5
Texture Memory (Alt = 14)	3.4	3	3.05	4.6
Global Memory (Alt = 46)	1.9	1.7	1.8	2.02
Texture Memory (Alt = 46)	1.3	1.18	1.25	1.19
Global Memory (Alt = 86)	0.22	0.22	0.22	0.22
Texture Memory (Alt = 86)	0.16	0.16	0.16	0.16

Table 2: Execution times (in seconds) comparison for Global vs Texture Memory for different altitude values.

As can be seen in Table 2, the execution times are reduced if the Texture Memory is included. However, the use of Texture Memory necessitates the replacement of the double precision floats by the single precision. Using the single precision floats can already speed up an application, since there are more single precision than double precision in the Tesla K40. A Texture Memory-free code using single precision was tried and showed similar results with that of the Texture Memory, demonstrating therefore that the speed up was due to the variable precision. For further comparisons, we will rely on the GPU code using the Pinned Memory. However, we will use a Texture Memory-free version, in order to stick with the original precision.

6.7.3 Java on CPU vs CUDA on GPU

The CUDA version is implemented following the previous explanations, and **is verified to show identical shadow maps outputs compared to the original Java version**. A comparison between the Java version implemented for CPUs and the CUDA version implemented for GPUs is shown in Table 3. The execution times are given in seconds and concern only the core part of the algorithm (reading input and saving output are excluded). An overall execution time comparison, based on averaging the execution times for the 10 altitude angles involved later in the SVF calculation (14,22,30,38,46,54,52,70,78 and 86), is provided in Table 4, providing a rough estimation about the speed up for each DUSM.

DUSM	Reference	Urban	Hybrid	Rural
Java (Alt = 14)	106	105	107	110
GPU (Alt = 14)	3.4	3	3.05	4.6
Speed up	x31	x35	x35	x24
Java (Alt = 46)	105	105	106	108
GPU (Alt = 46)	1.3	1.18	1.25	1.19
Speed up	x81	x89	x85	x91
Java (Alt = 86)	24	24	24	24
GPU (Alt = 86)	0.16	0.16	0.16	0.16
Speed up	x150	x150	x150	x150

Table 3: Execution times (in seconds) comparison for the Java and GPU versions

DUSM	Reference	Urban	Hybrid	Rural
Java	93	92	93.9	95
GPU	1.93	1.67	1.82	2.02
Speed up	x48	x55	x52	x47

Table 4: Execution times (in seconds) comparison for the Java and GPU versions

6.7.4 Execution on different CUDA devices

It is important to observe how the performance evolves whenever executing the algorithm on different CUDA devices. This was tested on the following Nvidia Graphic Cards, which are detailed here in terms of SMXs and CUDA cores:

- **GT 740M:** 2 SMX x 192 = 384 CUDA cores
- **GTX 960:** 8 SMX x 192 = 1536 CUDA cores
- **Tesla K20:** 13 SMX x 192 = 2496 CUDA cores
- **Tesla K40:** 15 SMX x 192 = 2880 CUDA cores

We run the simulation for the Rural DUSM with azimuth and altitude set (in degrees) to:

- Azimuth = 89
- Altitude = 14

The input and parameters were chosen in order to maximize the running time, hence allowing for a better comparison. In the Table 5, the execution times for each CUDA device cited above are reported in seconds.

Device	Execution time
GT 740M	65
GTX 960	7.7
Tesla K20	5.5
Tesla K40	4.6

Table 5: Execution times (in seconds) on different CUDA devices.

The GT 740M is a commercial (consumer) graphics card. GPUs with closer characteristics equip most of the computers one can find in the market. They are therefore not meant for the GPU programming and the performance on the GT 740M was tested by pure curiosity. The Figure 24 shows how the execution time decreases as the number of GPU cores increases for the three other Graphic Cards.

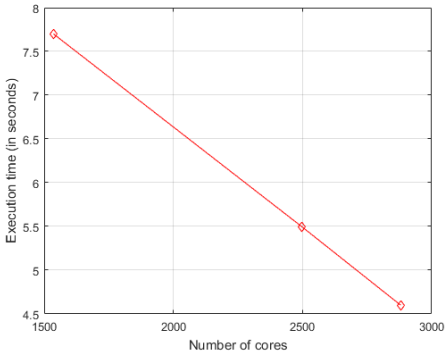

Figure 24: Execution time graph comparison for different CUDA devices.

Although it seems that the execution time is decreasing linearly, one cannot confidently state on this behaviour. Indeed, only three Graphic Cards were considered and a larger number of cards need to be taken into account before getting to any conclusion. However, that gives an idea about the general behaviour of the algorithm.

7 SVF implementation on a single GPU

As stated in Section 4, the SVF is the ratio of sky visibility. By assuming that the sky can be modeled as a cupola of suns, running the Shading Algorithm on each of these sun positions would indicate the sky visibility for each point in the tile. This implies that the SVF algorithm is exactly a Shading Algorithm, with an extra outer loop on the sun positions. Its implementation is therefore straightforward.

7.1 Simulation outputs

For modeling the sky, we use 400 sun positions (azimuths and altitudes) uniformly distributed over 40 azimuth and 10 altitude values. The SVF of the reference DUSM (Figure 14 page 15) is reported in Figure 25.

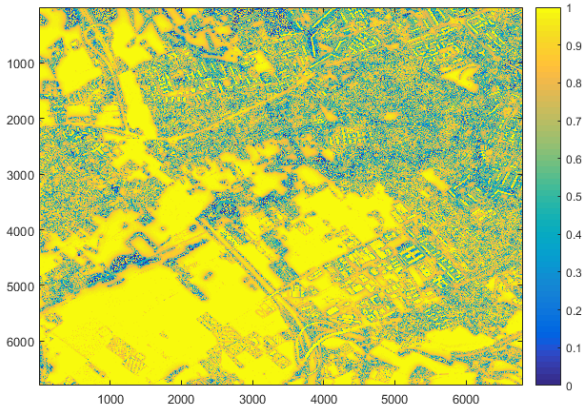

Figure 25: SVF of the reference DUSM.

The SVF of four different locations in Figure 25 is shown in Figures 26, 27, 28 and 29.

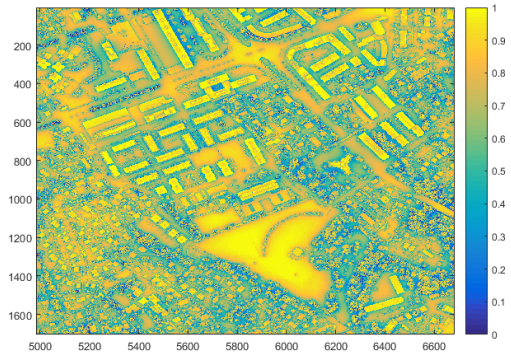

Figure 26: Urban zone.

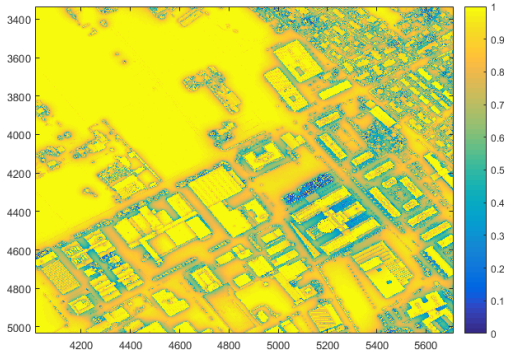

Figure 27: Hybrid zone.

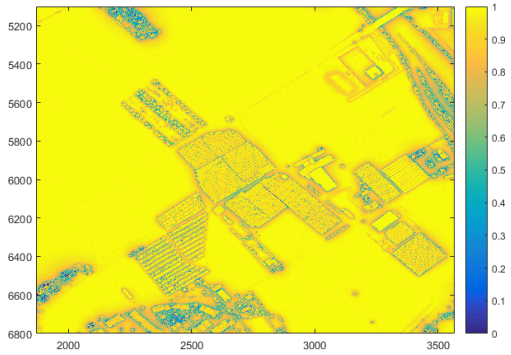

Figure 28: Rural zone.

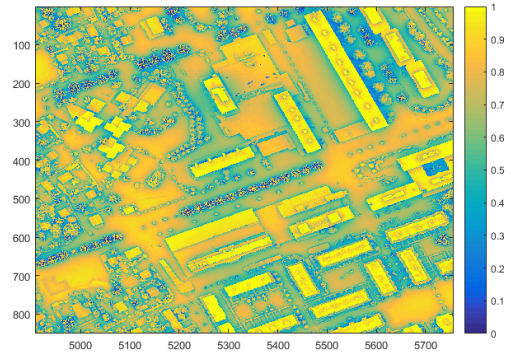

Figure 29: Street focus.

Figure 26 shows an urban part of the Reference tile. We notice that only the roofs and the empty region have a perfect sky visibility. The predominant components of this zone are the trees, and their sky visibilities are as we should expect them to be, i.e quite low. For the zone in Figure 28, we have a rural side of the Reference SVF. The sky visibility is almost perfect for the majority of the points in the zone. Figures 27 and 29 show how the sky visibility of streets in between buildings reduces since their walls mask the view.

We now report on the SVF of the Urban, Rural and Hybrid tiles.

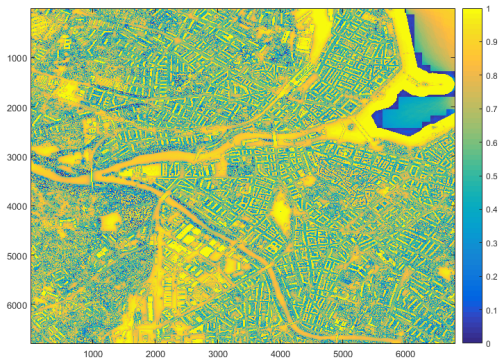

Figure 30: SVF of the Urban DUSM.

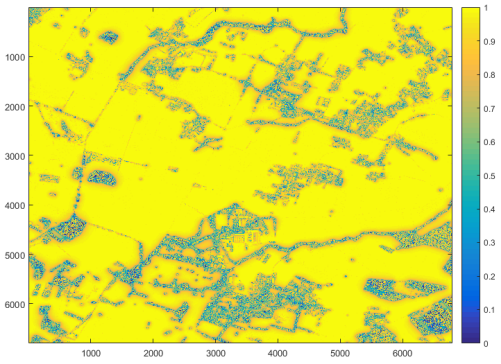

Figure 31: SVF of the Rural DUSM.

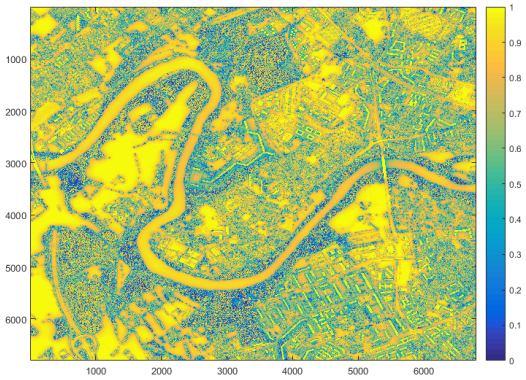

Figure 32: SVF of the Hybrid DUSM.

7.2 Execution time and performance analysis

Table 6 shows the execution times in minutes of the SVF algorithm on the four previously presented inputs: Reference, Urban, Hybrid and Rural DUSMs.

DUSM	**Reference**	**Urban**	**Hybrid**	**Rural**
Execution times	12m18	9m56	11m35	13m47

Table 6: Execution times (in minutes) of the SVF algorithm (GPU version)

Unfortunately, no such execution times were provided for the Java implementation. The execution time on a single CPU of a single Shading Algorithm (roughly 1/400 an SVF) was already time consuming and the algorithm was only executed on multiple CPUs. A more fair comparison will be established once the multiple GPUs version will be introduced and detailed (c.f Section 8). However, to give the reader an idea about the speed-up, the SVF calculation of the Reference DUSM took about 120 minutes on 40 vCPUs (virtual CPUs, CPUs in the cloud), whereas the single GPU version executed in about 12 minutes.

So far, we have only experimented the algorithm on four random inputs, yet exhibiting different characteristics in terms of building concentration. This was established as a comparison criterion between various DUSMs, hence separating them into 3 kinds; Urban, Rural and Hybrid. It appeared clear that the algorithm behaviour is DUSM type-dependent.
Civil engineers at Hepia divided the whole Geneva map into 55 tiles of 3x3 km^2, among which some were stated of a type or another. This is shown in Figure 33.

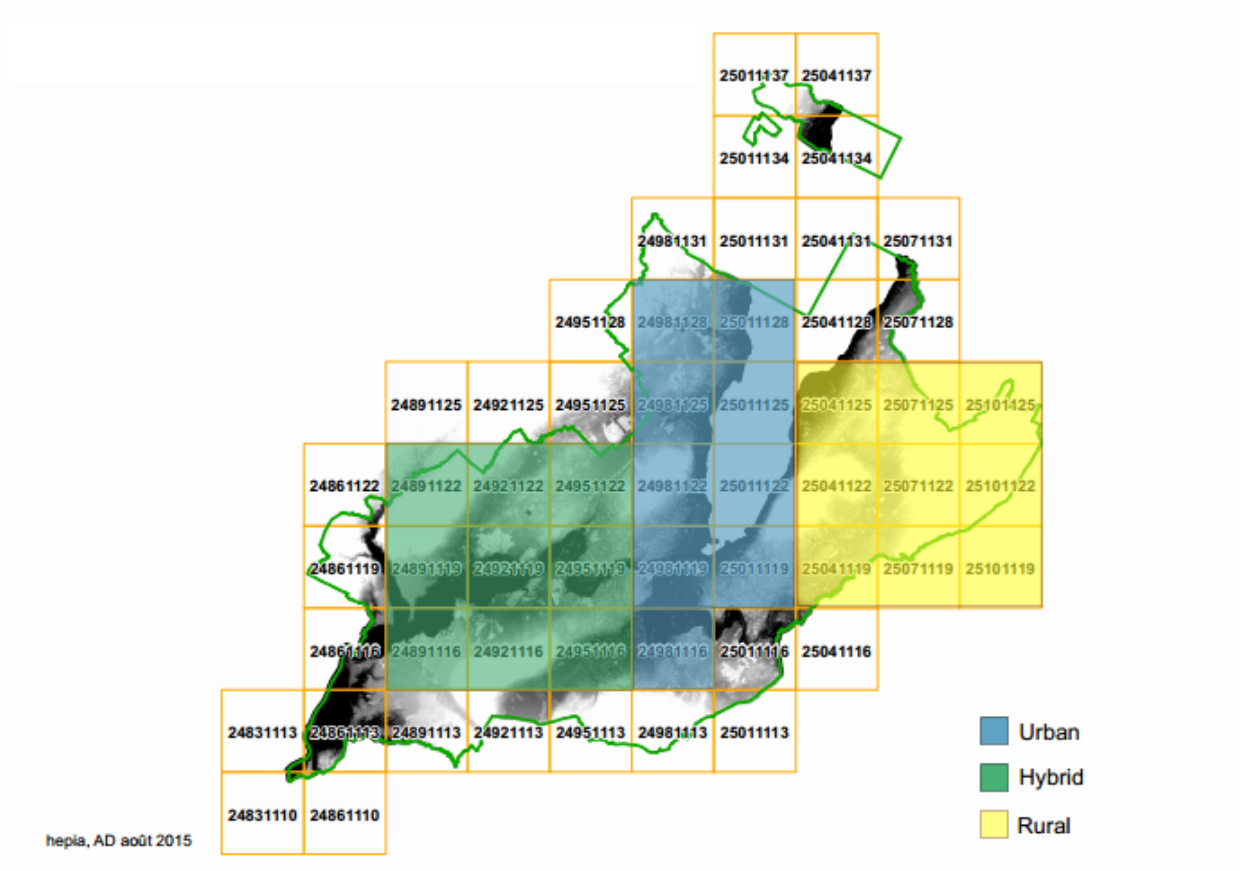

Figure 33: Geneva map in tiles of different type

Averaging the performance on multiple DUSMs of a kind is definitely more reliable in terms of performance indicator, than performing simulations on random DUSMs. We show in Table 7 the average execution times for the block DUSM of each type underlined in Figure 33.

DUSM	**Urban**	**Hybrid**	**Rural**
Average execution times	11m35	12m30	13m25

Table 7: Average execution times (in minutes) of the SVF algorithm on different types of DUSMs

We conclude the section with a final remark regarding the tile sizes. One might wonder why tiles of different sizes (3x3 km^2 and 3.4x3.4 km^2) were invoked. Actually, the smallest size was used for separating the whole map in small parts, as shown in Figure 33. The bigger one is the size of all the tested DUSM inputs, which include an overlap with the neighboring tiles, in order to allow for a correct calculation of SVF at the borders of the original tile.

8 SVF implementation on multiple GPUs

8.1 Cluster characteristics

When deploying an algorithm for execution on multiple GPUs, one needs to query about the available resources first; either the GPUs are connected to the same node (CPU) or to multiple ones. The word node refers to the cluster terminology, in which it represents a single machine. Therefore, a cluster is a collection of nodes or machines, which work can be assigned to through a principal node called the front-end node.
For instance, the Deneb cluster of EPFL is equipped with **16 Accelerated GPU nodes**, each one equipped with **4 Tesla K40 Nvidia cards**. That makes a total of **64 GPUs** altogether, connected to multiple nodes. Thus, communication needs to be established between these multiple nodes which will require the use of MPI (Message Passing Interface) [7].
During this project, the code implemented for the purpose of executing the solar energy potential application on multiple GPUs considered a similar cluster setting, i.e multiple GPUs across multiple nodes with MPI [5]. This represents a more general implementation, as it suffices to set the number of nodes to one whenever all the GPUs are connected to a single node. The simulation were performed on the Deneb cluster.

8.2 Parallelization strategies

8.2.1 Parallelization at the suns level

Since the SVF calculation consists in executing the Shading Algorithm for different light source positions (the 400 modeling the sky), a straightforward way of using multiple GPUs is to distribute the light sources among them. This simply means that every GPU will execute the Shading Algorithm for an equal portion of the light sources, hence dividing the number of light sources by the number of GPUs in use. This would naturally represent the perfect performance speed up, i.e the execution time being divided by the number of used GPUs.
For this parallelization strategy, it is crucial that the light source are fairly distributed among the GPUs. In fact, the Shading Algorithm execution time is light source position dependent. In particular, the smallest the light source altitude is, the longest will be the execution time. The contrary holds whenever the light source altitude is big.
Recall that the light source altitudes are uniformly distributed between 14° and 86° and have the following ten values: 14, 18, 22, 30, 38, 46, 54, 62, 70, 78 and 86. For each of these values, 40 light source positions with azimuth values uniformly distributed between 9° and 360° are being considered. We denote a sequence with 10 altitude values for an azimuth one a vertical sweep, and a sequence of 40 azimuth values for an altitude value a horizontal sweep. A sky model can be interpreted therefore as a collection of 40 vertical or 10 horizontal sweeps. The argumentation above suggests thus, that it is better to distribute the light sources in terms of vertical sweeps, rather than horizontal ones. Let us imagine we want to use 10 nodes of 4 GPUs each, for a total of 40 ones. We could think of distributing the light sources in a way that every GPU is in charge of one vertical sweep. This has the advantage to provide the GPUs with fairly comparable altitude angles, resulting in a close execution time between the GPUs.

8.2.2 Parallelization at the tile level

A classical parallelization strategy in MPI applications is to divide equal work among the nodes in relation with the final output, so that every node is responsible for a subset of it. In problems occurring under a matricial form, dividing the matrix in smaller matrices for which every node is responsible appears as a good idea. In our application, this strategy works fine whenever every node gets the full information (DUSM data) about the whole tile, in order to process the subpart it is concerned with. This is resumed in the following block diagram, whenever for instance 4 nodes with one GPU each are available.

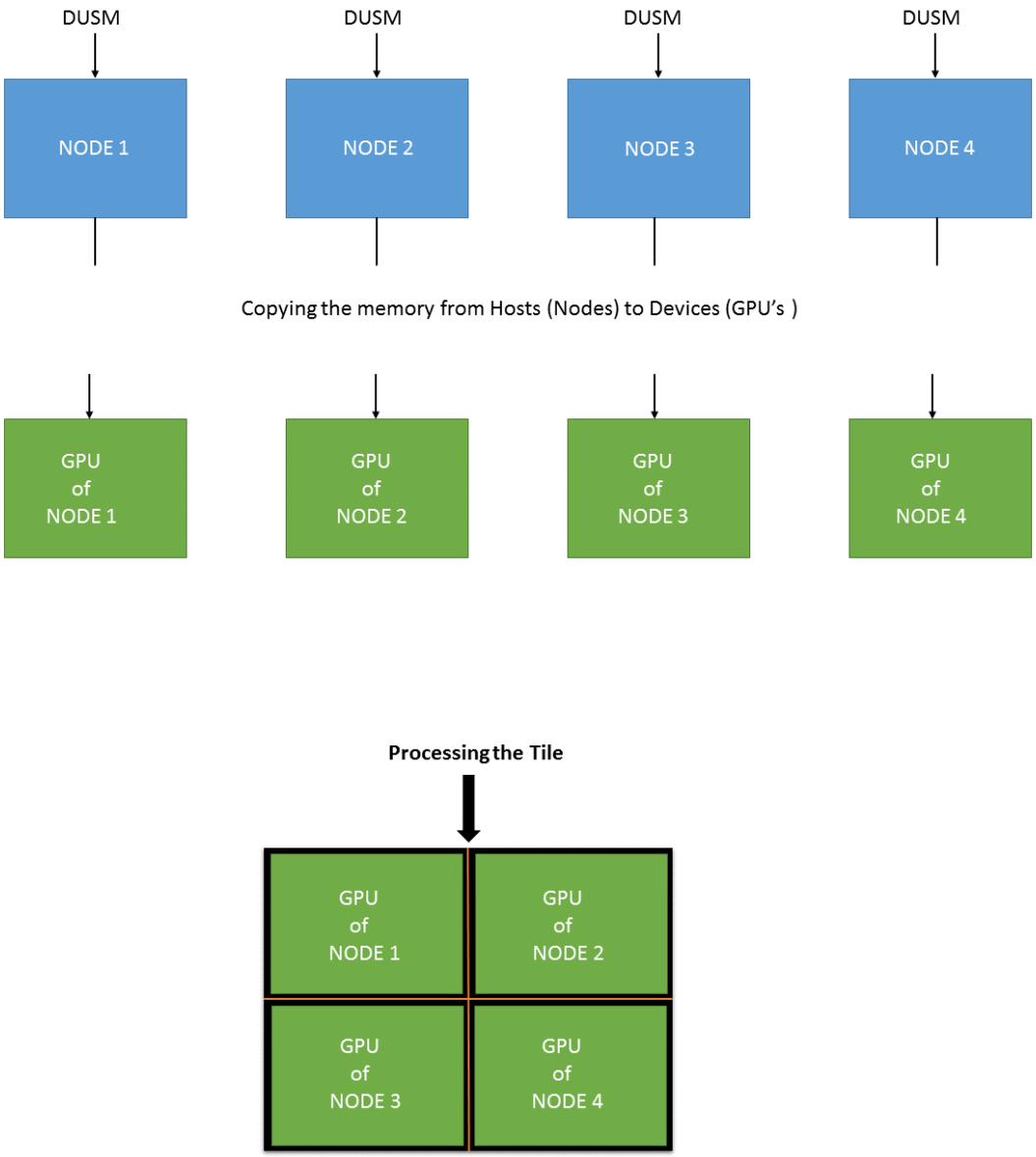

Figure 34: Block Diagram for parallelization at the tile level

8.2.3 Parallelization at both levels

Since the number of GPU nodes and GPUs per node is cluster-specific, both parallelization methods can be combined in order to produce a more flexible implementation allowing for more Nodes/GPUs combinations. The main goal is to adapt the use of multiple GPUs to any cluster.
As detailed in the previous section, we can think of dividing the tile in sub-tiles, so that each of these are processed separately. In addition, we allow multiple nodes to process the same sub-tile, with as many GPUs as each of these nodes contain.
Imagine for instance that we want to use 8 nodes containing 4 GPUs each. This setting is not perfectly adapted for the parallelization at the suns level, since 400 is not dividable by 32. We will end up with assigning more light sources to some GPUs, in addition to the hard task of finding a fair light sources repartition in terms of altitude-dependent time execution behaviour of the Shading Algorithm (c.f discussion Section 8.2.1).
As for the parallelization at the tile level, since only square division of the tile was implemented, the latter can be divided either in 4 or 9. Hence, either we divide the tile in four sub-tiles and end up having 4 idle nodes or we divide in 9 and wait for the first node that finishes to process the last sub-tile. Also, we will be able to use only one GPU in every node, instead of the 4 available ones.
Mixing parallelization strategies in this case, as described earlier, allows for an elegant way of using efficiently all the available resources. We could think of dividing the tile in 4 sub-tiles, each of which is to be processed by two nodes at the time. Every node is naturally putting all of its GPUs to the contribution, making a total of 8 GPUs per sub-tile. These 8 GPUs will further process 5 vertical sweeps (SVF is a total of 40 vertical sweeps), resulting in an easy and straightforward fair light sources repartition. This is shown is the block diagram Figure 35, assuming that $\text{GPU}_{i,j}$ is the j^{th} GPU of the Node i

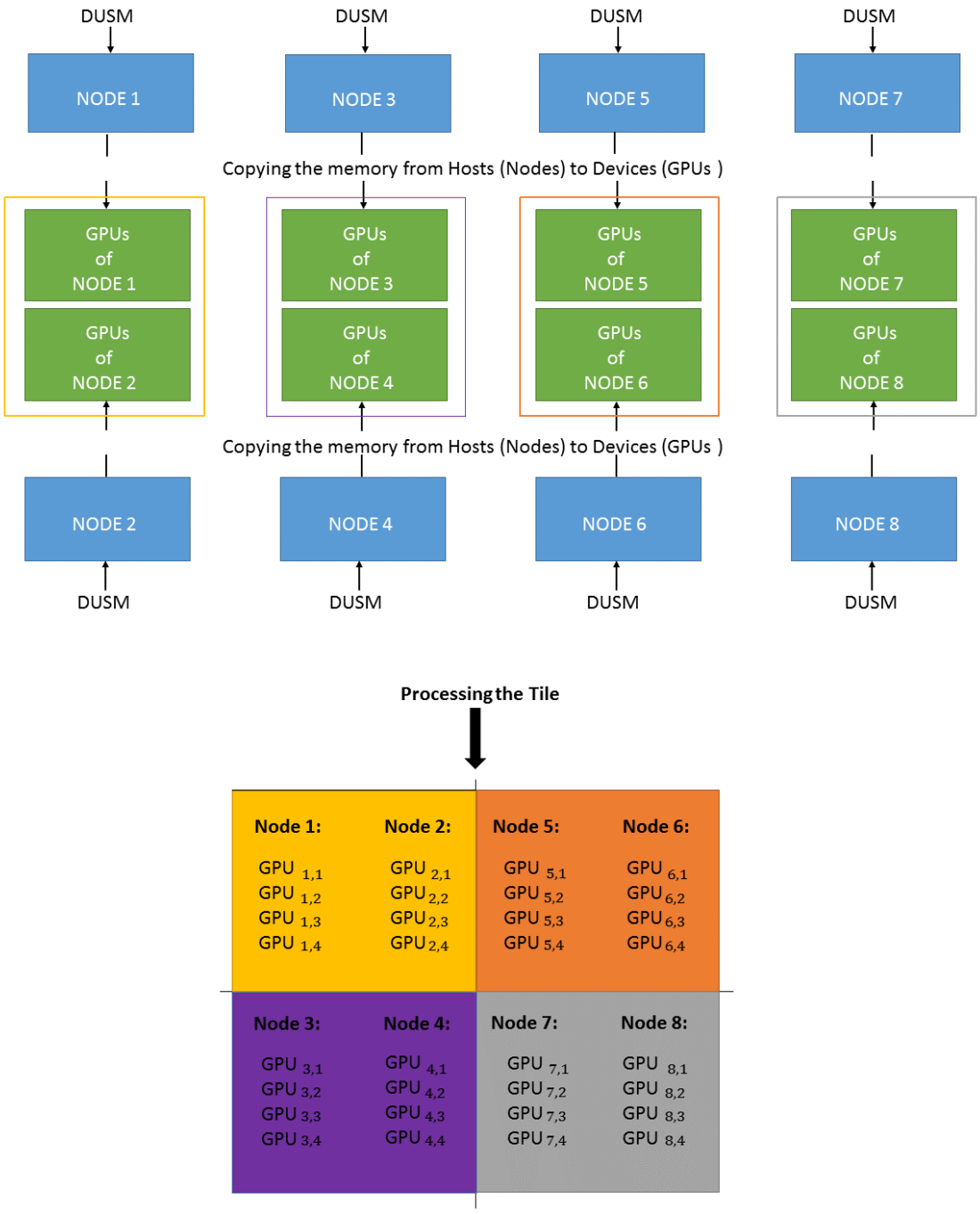

Figure 35: Block Diagram for parallelization at the tile and light sources levels

8.3 Execution times and performance analysis

8.3.1 CUDA on Multiple GPUs vs Java on Multiple CPUs

We compare the execution times of the Java and CUDA versions using multiple resources on the Reference DUSM. We rely on the parallelization at the suns level described in 8.2.1. Figure 36 shows the different time executions of both versions in **minutes**.

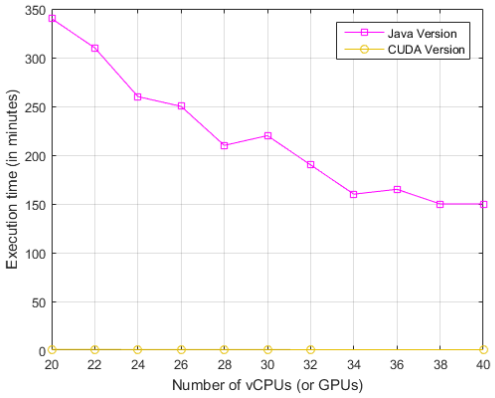

Figure 36: Java vs CUDA execution on multiple supports in **minutes**

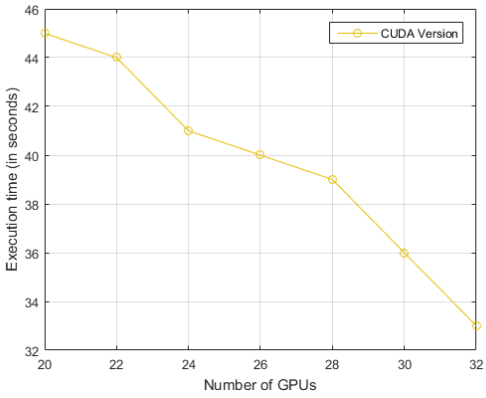

Figure 37: Zoom in the CUDA performance plot. Execution times are given in **seconds**

This graph shows that the GPU CUDA version is clearly faster. It would not be relevant speaking about a speed up, since the CUDA version is way faster, actually thousands of times. However, this is not really a surprise since the previous investigations showed that a single shadow map on a single GPU can be calculated up to 100 times faster. In Figure 37, only the CUDA performance plot of Figure 36 is shown, this time in **seconds**.

8.3.2 CUDA Parallelization strategies comparison

When multiple resources are being used for executing an algorithm, a perfect scaling is always a challenge. By a perfect scaling, we mean that one would always target the closest execution time to the initial one divided by the resources number.
We start with considering the parallelization at the suns level strategy. We run simulations for different number of GPUs on the usual DUSMs; Reference, Urban, Rural and Hybrid. The results are reported in Figure 38. The execution times are indicated in **seconds**.

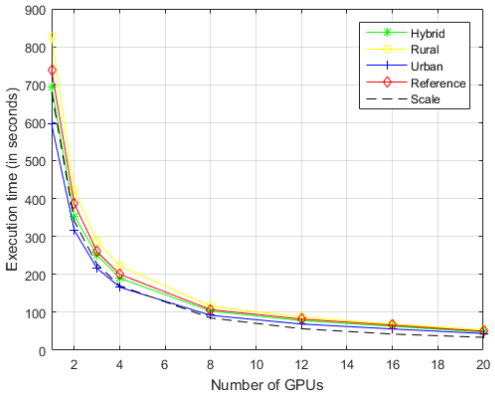

Figure 38: Parallelization at the sun level scaling the best performance

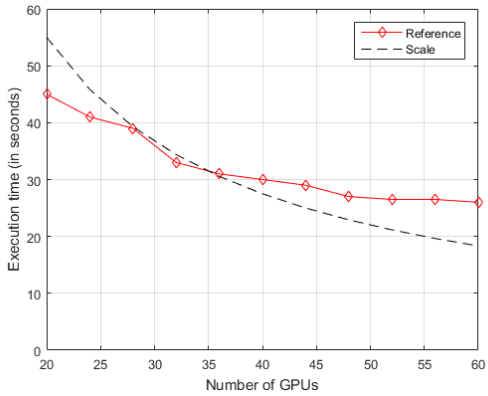

Figure 39: Scaling break up for bigger number of GPUs.

The black dashed line shows the perfect scaling of the performance divided by the number of GPUs used. The scaling holds until 40 GPUs, however when going over this value, the scaling is not anymore perfect. This is shown in Figure 39 for the reference DUSM and is in accordance with our previous analysis, which described the problem as an equal distribution of 40 plane sweeps. It is therefore normal that beyond this value, no equal distribution can be found, hence breaking the performance scaling. Furthermore, the importance of the sun distribution manner is shown in Figure 40.

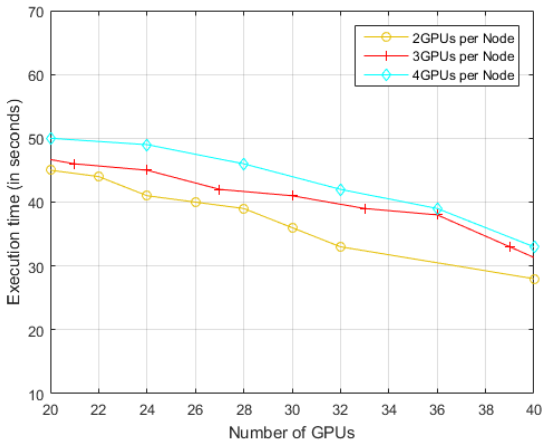

Figure 40: Performance difference according to the number of GPUs per nodes.

This graph represents the performance whenever a different number of GPUs per node is being used. Although the interpretation is not straightforward, from an implementation side, using more GPUs per Nodes complicates the distribution. Indeed, one has to see the repartition of suns as two consecutive repartitions; one according to the GPUs per nodes and also to the total number of nodes. For instance, if 2 GPUs per node are being used for a total of 10 nodes, one would need to to divide the set of suns in 2 subsets, before breaking each subset further into 10 or the other way round. It is hence more complicated with more GPUs per node if the suns are being provided in a uniform distribution way.

However, it is possible to overcome the scaling break up and sun distribution problem for a bigger amount of GPUs with the parallelization at the tile level. Theoretically, this is supposed to perform fine as long as the amount of work for each sub-tile stays close to each others. It seems then that its performance is

tile-specific. Comparison of execution times in **seconds** between multiple GPUs distributed over sub-tiles and over suns are shown in Figure 41.

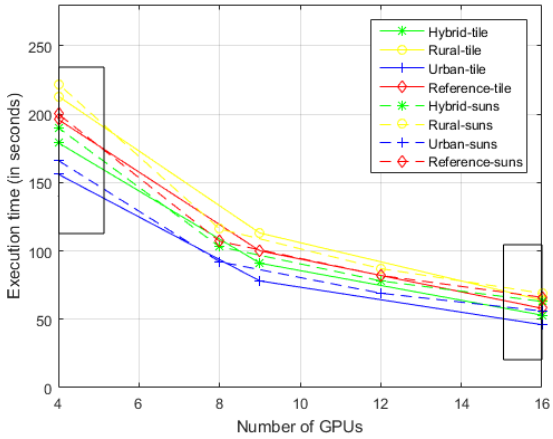

Figure 41: Comparison between parallelization at suns and tiles levels.

Since the matrix can only be divided in terms of number powers (4,9,16,25,..) and since the biggest amount of nodes available in the Deneb cluster of EPFL is 16, we can only run the simulation for 4,9 and 16 nodes (hence GPUs). Therefore, the most important features in Figure 41 are the results contained in the two black rectangles. These show that the performance induced by parallelization at tiles level is always slightly better for all the DUSMs (the lines are always lower than the dashed ones).
This does not show yet that we will recover the perfect scaling for a greater number of GPUs, but is a good sign on how this parallelization behaves. In order to investigate on whether the scaling will occur or not, we will have to start testing the parallelized version at both levels regarding the lack of available resources (16 GPU nodes in Deneb). We now allow each node to use more than a GPU and to perform sun distribution over these GPUs for each sub-tiles, as explained in Section 8.2.3.

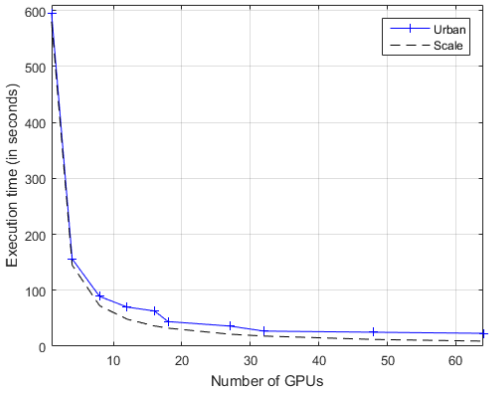

Figure 42: Parallelization at the sun level scaling the best performance.

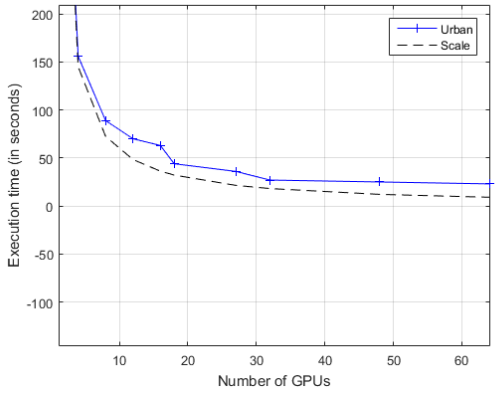

Figure 43: Zoom-Scaling even for bigger number of GPUs.

We can see now from Figure 43 that the performance scaling no longer breaks up. Parallelization at both levels described in Section 8.2.3 not only allows for multiple GPUs in a node, but also for multiple nodes in a sub-tile. This is meant to be a further parallelization degree of freedom, as that would allow for example to deploy more resources for an unexpectedly computationally demanding sub-tile.

9 GPU Cloud Deployment

The main initial objective of researchers at the Hepia, was to provide a cloud-based Decision Support System for the solar energy potential application. As for this project, the objective was not only speeding up the application but eventually reducing the processing cost on the cloud, according to the available machine configurations. Now that the application speed up appears clear from the previous investigations, we dedicate this section to the cloud deployment for the GPU solution and eventually a time to cost analysis. For this purpose, we will rely on the Amazon Web Services (AWS) public cloud infrastructure.

9.1 GPU instances on AWS

In the cloud, jobs are executed through instances. These represent virtual machines (or servers) made available by the cloud service provider, for a given price per hour. This price might be based on the machine power (the processor and its clock rate), the memory or the number of cores among some of the instance characteristics. It is therefore important for the user to investigate on the most suitable instances for their application, in the sense that one should not be picking an instance with a huge amount of memory if that is not necessary for the task they want to fulfill.
For GPU applications, no much of a choice is offered on AWS. The use of GPUs on the cloud being not very popular so far, there are only two available instances. These are detailed in the Table 8.

Model	**GPUs**	**vCPU**	**Mem(GB)**	**SSD Storage(GB)**	**Price Per Hour($)**
g2.2xlarge	1	8	15	1x60	0.65
g2.8xlarge	4	32	60	2x120	2.6

Table 8: GPU instances characteristics on AWS.

The most important feature in these is naturally the number of GPUs. The instance g2.2xlarge comes with a single GPU, whereas the g2.8xlarge has 4 GPUs. One can notice that not only the GPUs number, but every other feature scales to 1 to 4. Most importantly, this is also the case in terms of price which means that no savings can occur from using the instance with multiple GPUs. It is hence considered only for a matter of performance. These instances are equipped with NVIDIA GPUs containing up to 1,536 CUDA cores and 4 GB of memory. These are not state of the art GPUs (a Kepler K40 GPU equipping the Deneb cluster of EPFL has 2880 cores). However, we should expect these to be adequate for our purposes. Notice that these can execute only in single precision.

9.2 Performance on AWS

The calculation of the SVF for the Reference, Urban, Hybrid and Rural DUSMs had been performed on AWS and the time executions are reported in Table 9. The Table 9 includes also the time execution of the algorithm on the Deneb cluster for the single precision version. The instance g2.2xlarge was chosen for the task. We expect the g2.8xlarge to be 4 times faster, as the performance scaling holds with the algorithm.

DUSM	Reference	Urban	Hybrid	Rural
AWS	17m15	14m07	16m15	19m05
Deneb	10m33	8m35	10m07	11m52

Table 9: Execution times (**in minutes**) of the SVF algorithm in **Single precision** on the AWS cloud.

9.3 Costs comparison on AWS

Figure 44 shows the execution costs in function of the number of CPUs used for the calculation of the Reference SVF with the **Java** version.

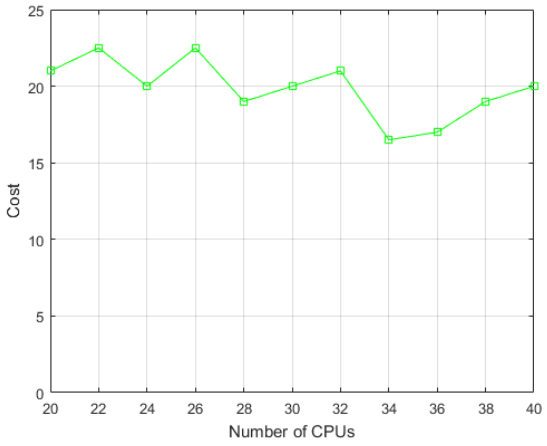

Figure 44: Costs for the Reference SVF calculation of a single tile provided by the **Java** solution.

The graph demonstrates an irregular sawtooth pattern. This pattern is an artefact of the Amazon pricing model which charges by whole hours or a portion thereof. For example, a calculation that takes 2 hours and 1 minute is charged for 3 hours. Therefore, the cost of using more vCPUs will increase until the wall-time drops to the next lowest full hour. The lowest price for the SVF calculation is given by 16.5 $ when the number of CPUs is 34.
For the GPU cloud solution, things are simpler. Table 9 shows that in all the cases (all the considered DUSM inputs), the execution is completed in less than an hour. Therefore, the price for calculating the SVF is given by the price of a g2.2xlarge instance, i.e 0.65 $.
Let us now do further rough estimation. Table 9 showed that the calculation of the Rural SVF takes 19 minutes and 5 seconds. Furthermore, Tables 6 and 7 shown on pages 24 and 25 respectively, suggest that the Rural DUSM considered takes more time to process than the average of rural DUSMs, these DUSMs being the more consuming ones. Considering this, let us assume that the Rural DUSM is one of the most time-consuming tiles. Using the GPU on the cloud for one hour would roughly allow for the calculation of 3 SVFs. Given that we have a total of 55 tiles in our use case, that implies that we will need a bit more than 18 hours. That would make a total price of $19 \times 0.65 = 12.35$ $ for the SVF calculation of the whole city. We would advise for such a task to use the g2.8xlarge instance with 4 GPUs which reduces the time execution by 4. Thus the whole-city SVF calculation should take no more than 5 hours.

10 Optimal solution for unlimited resources

Originally, civil engineers at Hepia divided the Geneva map into tiles for many obvious reasons. One of these is for catching the shape of the map as it will not necessarily be in a perfect rectangular form. Memory issues can also explain such a division, as well as algorithm execution times among many other reasons. However, this tile division comes with a dilemma. Either the precision at the borders will be of a poor quality (since the neighbors are located in other tiles), or we would have to use overlapping DUSM inputs. In the latter case, this will cause a task redundancy and necessitate an additional cropping step of every final output. This, added to the fact that we are processing bigger DUSMs, hence enlarging the computational cost.
From another hand, we were only concerned with computing the SVF of a tile so far. Whereas, the main final objective in relation with the Solar Energy application is to calculate the SVF of a whole city.
In this section, a method dedicated for overcoming the problem at the borders and outputting the whole SVF "in one shot" is suggested. It is meant to be an invitation for a further improvement of the whole application and was not implemented during this project.

10.1 Concept description

Let us imagine we have a city for which the partitioning produced N tiles. An example could be our use case, Geneva, for which we show a similar configuration in Figure 45

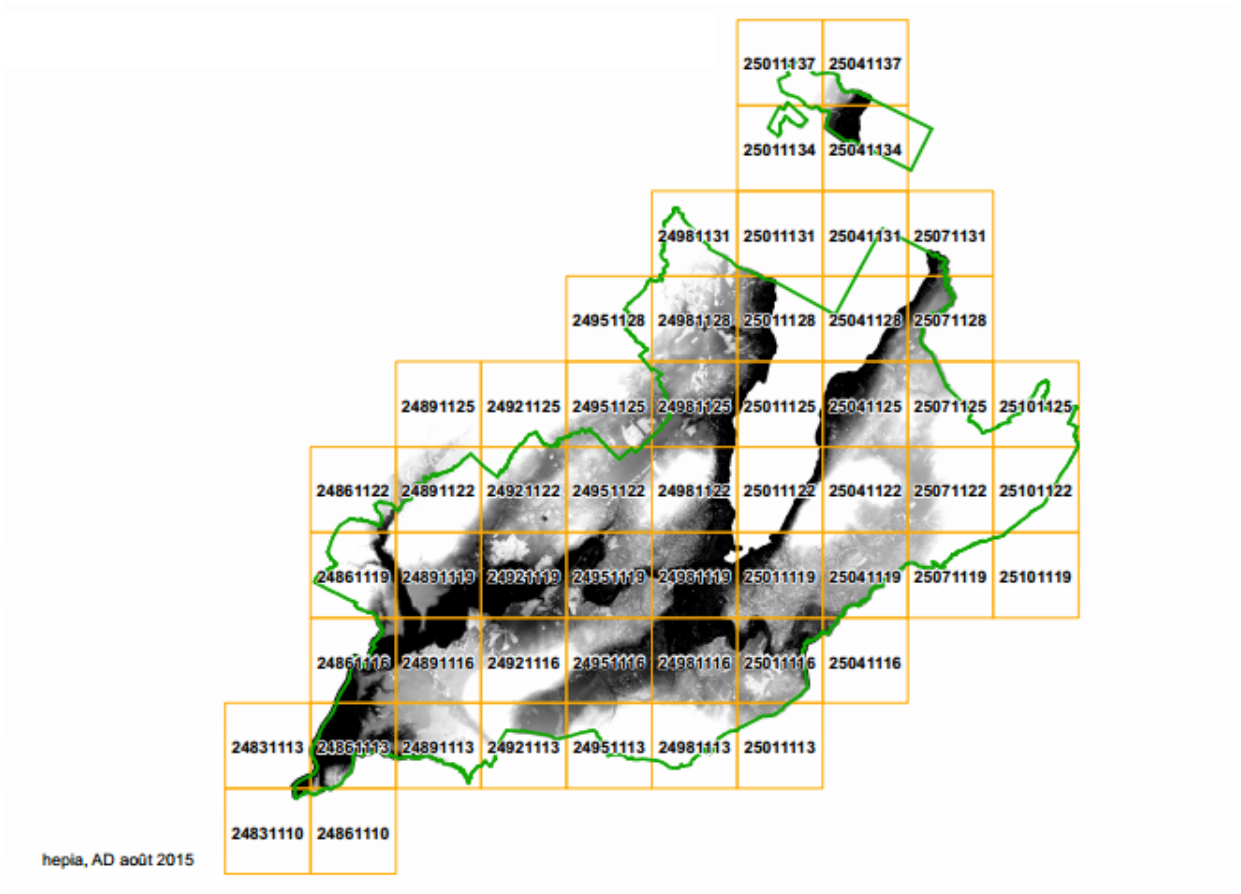

Figure 45: Partitioning of a city (Geneva) in tiles.

Assume in addition we have as many GPU nodes as tiles. The idea is that every GPU is responsible for

the SVF of a single tile, which in a way is similar to the parallelization at the tile level demonstrated in Section 8.2.2. The only difference is that every GPU in the latter used to have the whole DUSM information, and no problem was encountered whenever points outside the sub-tile were needed. Indeed, since the problem now is much bigger and much more memory demanding, we cannot afford to send all the tiles information to all the GPUs. As a consequence, every GPU gets only the information of the tile it is concerned with. Naturally, that raises the question on how to deal with points that go beyond a certain tile. One way could be to send these to the GPUs in charge of the neighboring tiles, so that these make the calculation and send back the result. MPI could be used for this purpose.
Notice that not necessarily the GPUs have to belong to different nodes. It might be even easier if these could belong to one single node, as CUDA APIs allow memory transfers between different GPUs without passing through the Host. That would be indeed the perfect scenario.

10.2 Geneva SVF

The concept presented above was not implemented. However, we proceeded to the calculation of the whole Geneva SVF with the old routine. We launched the calculation on 55 GPUs at the same time, each of which computing the SVF of a single tile. The outputs were then cropped to fit with the original tile sizes. The calculation took 20 minutes 55 seconds as a whole, the duration of the most time-consuming SVF (tile 25011113 in Figure 45). We are glad to present in Figure 46 the SVF of the beautiful city of Geneva, and its wonderful lake.

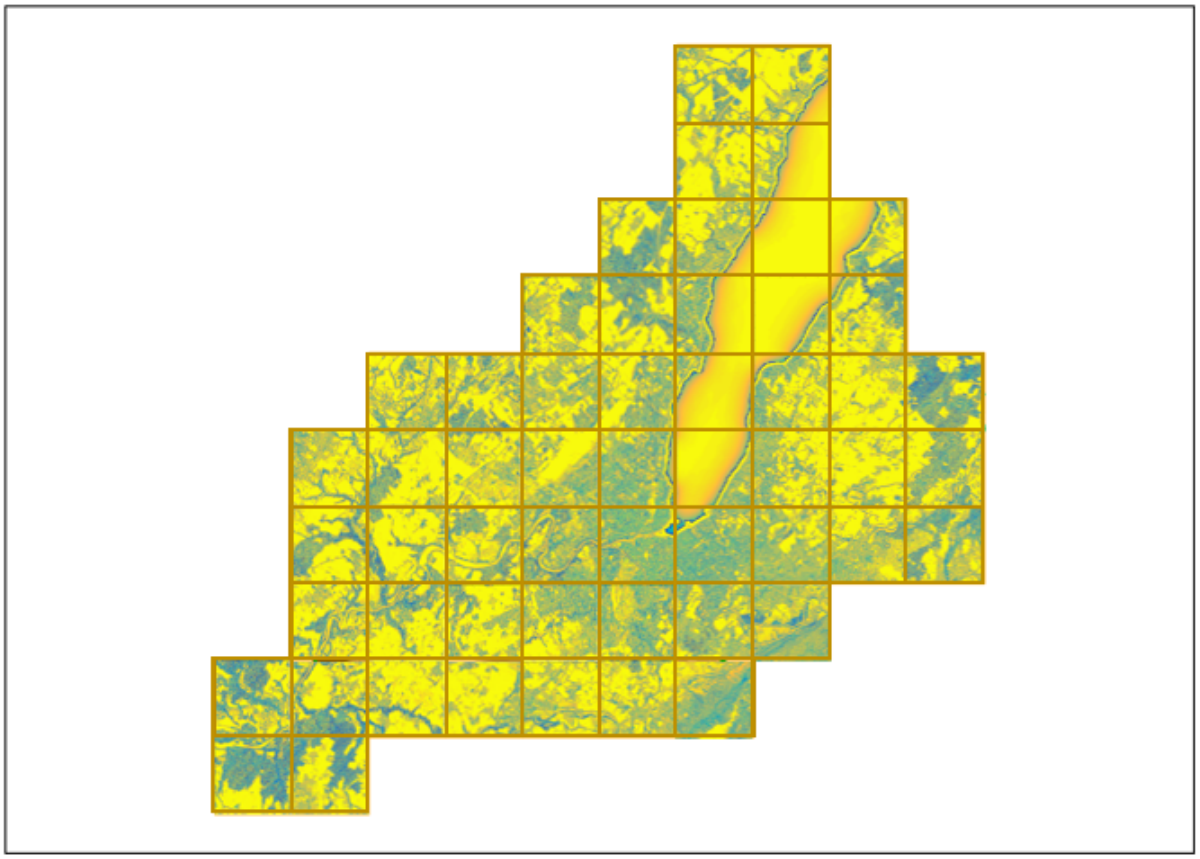

Figure 46: The whole Geneva SVF.

11 Conclusion

The solar energy potential application is not the only application based on processing 3D city numerical models. Other applications may include for example the estimation of satellite visibility, as it might be necessary to determine whether a position is receiving satellite signal, and whether this signal is of a good quality or not. In a way, this is very similar to the Shading Algorithm where the sun is replaced by the satellite or satellites. Therefore, the same algorithm can be used for investigating whether a position gets a good satellite signal, if the latter is a percentage of the received ones among all the diffused ones.
In the case of the solar energy potential application, and regarding the final output, the hotspots that show the best places where to install photovoltaic panels for maximum yield is not the only information that can be interpreted. Farmers might be interested in getting to know which locations in their field improves that chances of growing certain crops. A farmer could further separate their plantations, based on this information, in order to avoid that a contamination destroys the whole crop.
Benefits from using 3D city numerical models for different purposes can be tremendous and are worth exploring for coping with our new world challenges. The GPU portability of the solar energy potential application showed not only for this application, but in general, that it is possible to treat such a huge amount of information in a very short period of time. It has been demonstrated that there is no need to spend large amounts of money for processing on computing resources treatment and the financial aspect should not represent a barrier. This might encourage cities to proceed and digitize their surface data in a standard 3D model (e.g. DUSM), and achieve the potential benefits.

References

[1] D.B. Kirk, W.W. Hwu, *Programming Massively Parallel Processors* , Elsevier Inc., 2nd edition, 2013.

[2] J. Cheng, M. Grossman, T. McKercher, *Professional CUDA C Programming*, John Wiley & Sons Inc., 2014.

[3] Nvidia Corporation, *CUDA C Programming Guide*, 2010.

[4] P.N. Glaskowsky, *NVIDIA's Fermi: The First Complete GPU Computing Architecture* , Nvidia Corporation, 2009.

[5] M. Sourouri, T. Gillberg, S. B. Baden, X. Cai, *Effective Multi-GPU Communication Using Multiple CUDA Streams and Threads, 20th IEEE International Conference on Parallel and Distributed Systems, 2014.*

[6] M. Scarpino, OpenCL In Action*, Manning Publications, 2011.*

[7] W. Gropp, E. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface*, The MIT Press, 1994.*

[8] C. Ratti, S. Di Sabatino, R. Britter, Urban texture analysis with image processing techniques: winds and dispersion*, Theoretical and Applied vol. 84, no. 1, pp. 77-90, 2005.*

[9] P. Redweik, C. Catita, M. Brito, Solar energy potential on roofs and facades in an urban landscape*, Solar Energy, vol. 97, pp. 332-341, 2013.*

[10] T. Santos, N. Gomes, S. Freire, M. Brito, L. Santos, J.A. Tenedorio, Applications of solar mapping in the urban environment*, Applied Geography, vol. 51, pp. 48-57, 2006.*

[11] M. Iqbal, An introduction to Solar Radiation*, Academic Press, 1983.*

[12] Prabhat, Q. Koziol, High Performance Parallel I/O*, Chapman and Hall/CRC, 2014.*

[13] C. Cooper, GPU Computing with CUDA, Lecture 3 - Efficient Shared Memory Use*, Boston University, 2011. http://www.bu.edu/pasi/files/2011/07/Lecture31.pdf*

[14] D. Negrut, CUDA Optimization: Tiling as a Design Pattern in CUDA Vector Reduction Example*, University of Wisconsin, 2012. http://sbel.wisc.edu/Courses/ME964/2012/Lectures/lecture0313.pdf*

[15] Y. Wang, M. Olano, M. Gobbert, W. Griffin, Parallel Computing for Long-Time Simulations of Calcium Waves in a Heart Cell*, Proceedings in Applied Mathematics and Mechanics, 2012.*

[16] Nvidia blog, https://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/

[17] https://www.cs.virginia.edu/m̃wb7w/cuda_support/pinned_tradeoff.html

[18] What is pinned memory in CUDA, http://jasonjuang.blogspot.ch/2015/06/what-is-pinned-memory-in-cuda.html

[19] Texture Memory In Cuda, http://cuda-programming.blogspot.ch/2013/02/texture-memory-in-cuda-what-is-texture.html

[20] Parallelism in NVIDIA GPUs, http://yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html