
**João Pedro Martins Rogeiro**

Licenciado em Engenharia Informática

# Geometry Based Visualization with OpenCL

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador : Professor Doutor Fernando Birra, Professor Auxiliar,
Universidade Nova de Lisboa

Júri:

| | |
|---|---|
| Presidente: | Doutor Pedro Abílio Duarte de Medeiros |
| Arguente: | Doutor João Madeiras Pereira |
| Vogal: | Doutor Fernando Pedro Reino da Silva Birra |


FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

**December, 2011**

**Geometry Based Visualization with OpenCL**

# Abstract

This work targets the design and implementation of an isosurface extraction solution capable of handling large datasets. The Marching Cubes algorithm is the method used to extract the isosurfaces. These are graphical representations of points with a constant value (e.g. matter density) within volumetric datasets. A very useful approach to visualize particular regions of such data.

One of the major goals of this work is to get a significant performance improvement, compared to the currently available CPU solutions. The OpenCL framework is used to accelerate the solution. This framework is an open standard for parallel programming of heterogeneous systems recently proposed. Unlike previous programming frameworks for GPUs such as CUDA, with OpenCL the workload can be distributed among CPUs, GPUs, DSPs, and other similar microprocessors.

**Keywords:** Isosurface Extraction, Volume Visualization, Marching Cubes, OpenCL

# Resumo

Este trabalho tem como finalidade o desenho e implementação de uma solução de extracção de isosuperfícies capaz de lidar com grandes conjuntos de dados. O algoritmo Marching Cubes é o método utilizado para extrair as isosurperfícies. Estas surperfícies são representações gráficas de pontos com valor constante (p.e. densidade da matéria) dentro de um conjunto de dados volumétrico. Uma abordagem muito útil para visualizar certas regiões desses dados.

Um dos grandes objectivos deste trabalho é conseguir um aumento de velocidade significativo, comparado com as soluções de CPU actualmente disponíveis. A framework OpenCL é usada para acelerar a solução. Esta framework é um norma aberta de programação paralela para dispositivos heterogéneos recentemente proposta. Ao contrário de anteriores frameworks de programação para GPUs como CUDA, com OpenCL a carga de trabalho pode ser distribuída por por vários CPUs, GPUs, DSPs e outros microprocessadores idênticos.

**Palavras-chave:** Extracção de Isosuperfícies, Visualização de Volumes, Marching Cubes, OpenCL

# Contents

# List of Figures

# List of Tables

# Listings

# 1

# Introduction

Marching Cubes (MC) [LC87] is one of the most common algorithms for indirect volume rendering[1] (IVR) of volumetric datasets, it allows the creation of three-dimensional (3D) models of constant value. This can be very useful in many scientific areas, such as medical imaging, geophysical surveying, physics and computational geometry, but some of these applications require huge data sets to produce reliable models. A major problem is that the number of elements grows to the power of three with respect to sample density, and the massive amounts of data puts hard requirements on processing power and memory bandwidth, even harder if the visualization is interactive or dynamic.

In the past, one way of handling problems with massive amounts of data was to use a distributed version[2] of the marching cubes algorithm [Mac92]. This approach could use supercomputers or a cluster of computers to speed-up the process, but it yield scale problems due to typical bottlenecks of distributed approaches, like latency and limited bandwidth. More processing units implied more network bandwidth to feed them and a central unit powerful enough to receive the results. Ultimately, to speed-up the process the use of higher frequency CPUs was required, because the overhead of adding more processing units wouldn't pay-off its costs.

Eventually the increase of frequency stalled and the solution found by the computer industry was to couple more processors in a single physical package. Although the shift

[1] Indirect Rendering techniques involve rendering of an intermediate structure, such as an isosurface, that has been extracted from the data.

[2] In distributed computing, a distributed version divides the problem into many tasks, each of which is solved by one node.

toward multi-core CPU architectures has created a strong potential for highly efficient solutions, at least compared with distributed ones, they still lacked the resources for high demanding usages.

Graphics processing units (GPUs) have always been based on a many-core design and their overall performance has continued to increase at a much higher rate than the traditional multi-core CPUs, shown in figure 1.1. Performance increase led to units more programmable, capable of producing complex graphics effects, through the use of shaders[3]. This also allowed their usage in fields besides graphics applications.

There has been a lot of research on volume data processing using GPUs. This field involves huge computational tasks with challenging memory bandwidth requirements, building on massive parallelism. Such workloads have the potential to preform much better using massive parallel processing devices such as the GPUs, than the highly sophisticated serial processing offered by the CPUs.

But despite all previous facts, programing GPUs was a lot more complicated and much more restricted than any multi-core CPU architecture.

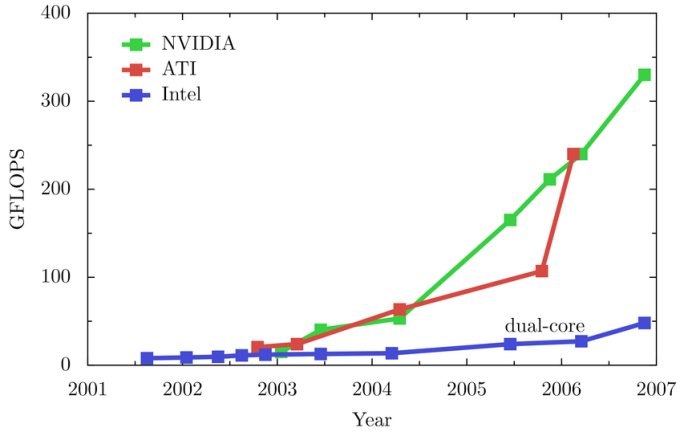



**Figure 1.1:** Enlarging peak performance gap between GPUs and CPUs.

With the advent of General-Purpose computing on Graphics Processing Units (GPGPU) came a new degree of freedom to implement algorithms capable of exploiting such processing power. This would turn GPUs into open architectures, much like the regular CPUs but with tremendous parallel-processing power. To enable this new computing paradigm there are several frameworks, CUDA [NBGS08] introduced by NVIDIA and OpenCL [mun10] proposed by Khronos Group being the most popular ones.

We will be focusing on OpenCL, a new and open standard for task-parallel and data-parallel heterogeneous computing on a variety of modern CPUs, GPUs, DSPs, and other microprocessor designs. This is mainly due to the limited range of hardware devices

[3] A shader is a set of software instructions, which is used primarily to calculate rendering effects on graphics hardware with a high degree of flexibility.

supported by other frameworks, they are either limited to a single microprocessor family or don't support heterogeneous computing. For example, something developed with CUDA can only run on NVIDIA devices.

OpenCL provides easy-to-use abstractions and a set of programming APIs based on past successes with CUDA and other programming frameworks. Even if OpenCL can't completely hide significant differences in hardware architecture, it does guarantee portability and correctness. This makes it much easier to port OpenCL programs for different architectures, beginning with a generic version and then tweak each of them independently.

OpenCL can be a significant help in implementing an accelerated version of the Marching Cubes algorithm. At least compared with graphics-based programing standards, although much more verbose than similar CPU versions. Besides the advantages from the programming point of view, OpenCL also provides a clean and simple way to manage the hardware resources. A solution that can distribute its workload across several devices has the potential for great performance improvements.

## 1.1 Motivation

Besides the technical viability, this implementation also seeks its practical use. Its need comes from a scientific project[4] that expects to build a set of GPU-accelerated tools. These tools perform heavy computational tasks that without GPU assistance would be too much impractical or require considerable CPU processing power.

Providing a visualization solution like this one using CPU power makes little to no sense. If the final results are presented by the GPU and it's possible to produce such results using also the GPU, why not do the whole process there?

In situations where it's practical to use CPU power, the resources can be very expensive if similar performances are expected. Using GPU assistance can be viewed as just a way of building a solution that otherwise would be more expensive.

[4]Project developed by the Departamento de Informática in collaboration with the Departamento de Materiais

# 2

# Related Work

In this chapter we provide an introduction to the components used in our solution. Section 2.1 begins with a brief introduction to what are the goals and requirements of the marching cubes algorithm. Then the fundamental stages in the standard algorithm are explained. Ending with some of the relevant approaches to our solution. Section 2.2 also begins with an introduction, including a brief history of GPGPU and the components of OpenCL. After that, an overview of OpenCL architecture is supplied, described following four models.

## 2.1 Marching Cubes

Marching Cubes [LC87] is a computer graphics algorithm, published in the 1987 SIGGRAPH proceedings by Lorensen and Cline, with the purpose of modeling 3D medical data, typically, produced by scanners such as computed tomography (CT) or magnetic resonance (MR). The output of the algorithm is a polygonal mesh representing points of a constant value (e.g. pressure, temperature, velocity, density) within a volume.

### 2.1.1 Introduction

Usually we know in advance the shape of the body we want to model, but sometimes that is not the case, therefore, in some of those cases, we can use the marching cubes algorithm to build a close model. To achieve this there are a couple of requirements which are necessary to meet. An *isovalue* that will determine whether a given point is "inside" or "outside" of the *isosurface*[1]. And a volumetric data set, or a function capable of

[1]Since the algorithm can also work with open surfaces, the terms inside and outside aren't strictly correct, they are used for simplicity purposes.

produce such data set, representing the desired model. This data must be arranged as a regularly structured grid of 3D points, $P(x, y, z)$.

### 2.1.2 Algorithm

The algorithm begins by identifying which is the relevant data to build the isosurface, according to the isovalue specified by the user. This is done analyzing all logical cubes, also known as *voxels*[2], formed by two adjacent slices of data, each corner has a value, four at each slice, like in figure 2.1. Each of these values are compared against a threshold, the isovalue, to determine if it belongs inside or outside the surface, values that exceeds (or equals) the isovalue belong inside. Only cubes with values inside and outside are relevant, this means that the surface cuts (intersects) the cube somewhere.

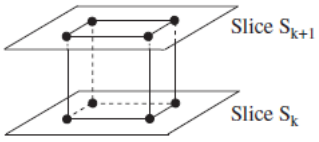



**Figure 2.1:** Illustration of a logical cube (voxel) formed by two adjacent slices of data.

Since cubes have eight corners and each corner has two possible states, inside and outside, there are $2^8 = 256$ possible combinations of edge intersections. In order to simplify the process of determining edge intersections it's used a look-up table (built offline) with all possible combinations, which is indexed in such way that each corner has a distinct weight, a power of two. For example, if a cube has corner 1 and 3 inside, the correspondent index would be $1^2 + 3^2 = 10$. In the original MC implementation [LC87] the 256 combinations were reduced to 15, shown in figure 2.2, by using symmetry properties like rotation and reflection (complementarity), shown in figure 2.3.

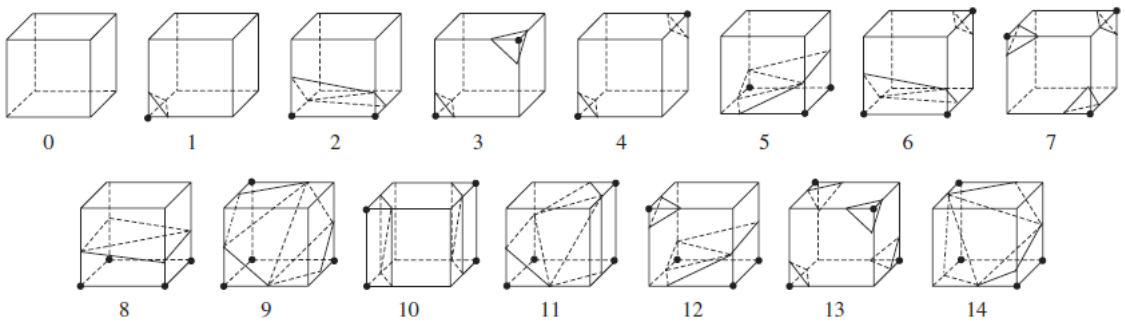



**Figure 2.2:** Illustration of the 15 basic intersection topologies.

Based on the index previously calculated it's determined which edge topology each relevant cube has, using an edge table (also built offline). Since this table only provides

[2] A voxel is a volume element, representing a value on a regular grid in 3D space. This is analogous to a pixel, which represents 2D image data in a bitmap.

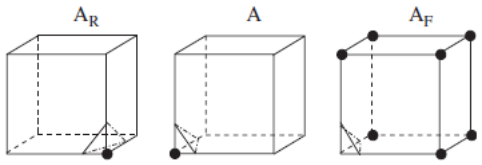



**Figure 2.3:** Illustration of reflective (A with Af) and rotational (A with Ar) symmetries.

the intersection topology it's necessary to calculate the actual intersection based on the value of each corner via linear interpolation. All vertex information needed to build the patches (triangles) that compose the surface is generated in this step.

With all vertex information generated the only thing missing it's calculating normals for each triangular patch, for smooth rendering purposes. The original algorithm [LC87] calculates a unit normal at each cube vertex using central differences and interpolating the normal for each triangle vertex. Another way [NY06] of accomplish this after the facets have been created is to average the normals of all the faces that share a triangle vertex.

To finish the process it's necessary to render all vertex information, a collection of triangular patches across all relevant cubes forming the triangular mesh that defines the isosurface. Moreover, the rendering process can also use normal information to produce Gouraud-shaded models.

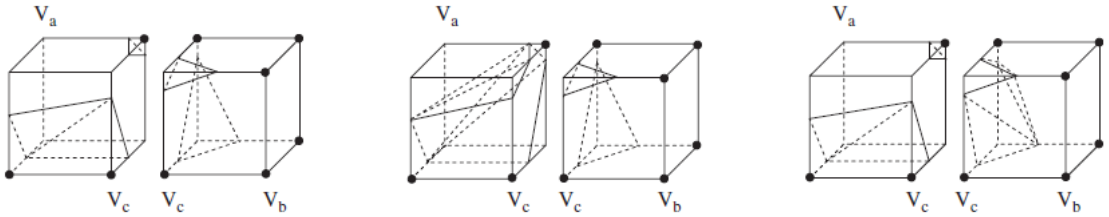
### 2.1.3 Challenges

There are two main challenges while using marching cubes algorithm, the first is correctness and consistency, the second is efficiency and performance. End-user understanding of a data set is positively impacted if the extracted isosurface is both correct and topologically consistent. An efficient algorithm can also have impact on user experience, specially if the output is a dynamic model and not a static one.

An isosurface is correct if it accurately matches the behaviour of a known function (or some assumed interpolant) that describes the phenomenon sampled in the data set. If each component of an isosurface is continuous then it is topologically consistent. It's possible for an isosurface to have a consistent topology and not be correct.

When marching cubes was initially introduced the problems of topologically consistency weren't address. Only later it was discovered that some of the basic intersection topologies could be facetized in multiple ways. The consistency problems arise when two adjacent cubes share faces that can be intersected in multiple ways and the default intersections are inconsistent, generating face ambiguity, like in figure 2.4(a). The unresolved ambiguity produces a hole in the isosurface, which will lead to a topological inconsistency. The resolution of such problems requires variation of the intersection in

one or both cubes, like in figure 2.4(b) and figure 2.4(c).




(a) isosurface with holes from an ambiguity in a face shared by two cubes

(b) one alternate facetization that yields a topologically consistent isosurface for the cubes

(c) another alternate facetization that yields a topologically consistent isosurface for the two cubes

**Figure 2.4:** Illustration of face ambiguity and resolutions.

Since the use of reflective symmetries is what causes face ambiguity, one simple way of resolving the problem directly is to use a look-up table that don't exploit such symmetries, like in figure 2.5.

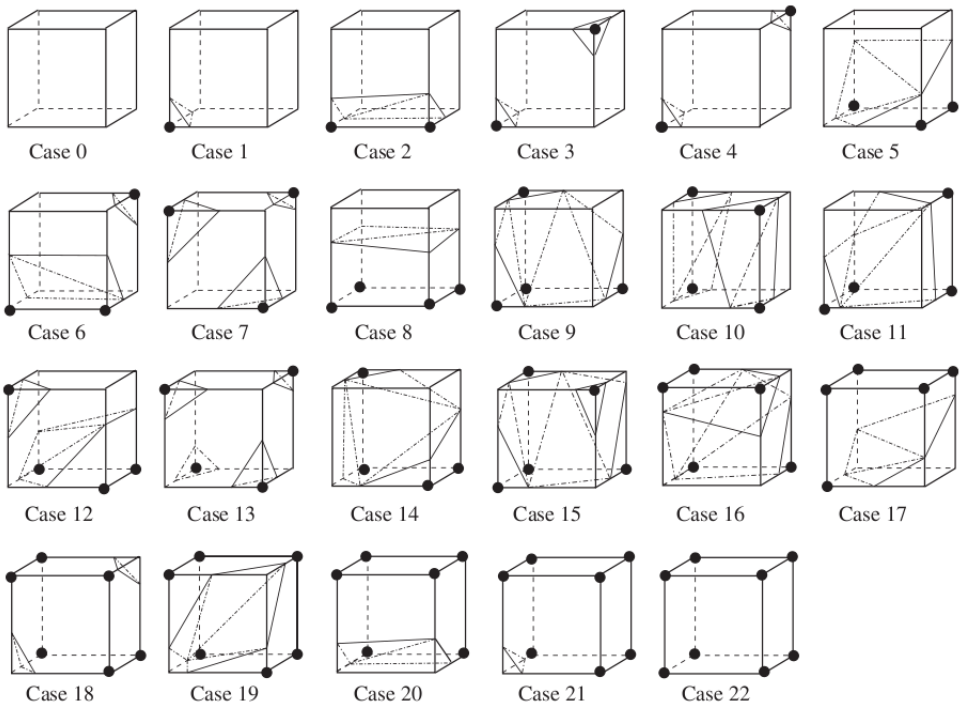
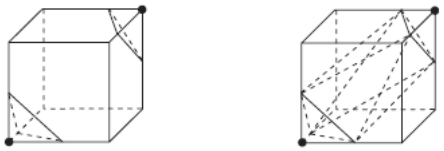



**Figure 2.5:** Illustration the 23 intersection topologies exploiting only rotation.

The facetization of a cube that hasn't ambiguous faces can still have internal ambiguity, this kind of ambiguity doesn't cause inconsistency but can yield an incorrect isosurface, as shown in figure 2.6.

Efficiency is very important in graphics algorithms, its iterative nature provide great potential for optimizations, which can lead to better performance. Marching cubes allows a few enhancements that can save a lot of processing time and consumed memory.



(a) disjoint facetizations (b) linking facetization

**Figure 2.6:** Illustration of internal ambiguity (two facetizations of the case 4).

Employing computation avoidance and parallelization techniques can improve the algorithm performance.

Beginning with computation avoidance, one way of reducing processing cycles is to avoid unnecessary operations on non-relevant (empty) cells, which in general represents up to 70%. Although this technique can't be used in all steps, since generating a correct isosurface requires each cell to be visited at least once to determine if the cell is relevant or not, it can avoid many unnecessary computations. Another enhancement is the possibility of sharing all common data, from the twelve edges of a cube only three need processing since the rest was already processed or will be, with the exceptions of boundaries.

Parallelization presents an interesting path for improving performance. In theory a completely parallel algorithm can lead to unlimited speed-ups, with the necessary resources. Like most graphics algorithms, marching cubes exhibits a degree of intrinsic parallelism (e.g., cube faces not shared with previously visited cubes can be processed independently), its parallelization offers the potential for performance improvement.

Sometimes there are commitments between efficiency and performance that need to be done. A parallelized solution of marching cubes requires splitting data to be distributed through the processing resources, and each portion of data has boundaries that are shared with adjacent portions. This means that some data is duplicated, wasting memory, and some work is also duplicated, wasting processing cycles. Typically, this kind of commitments have a spot where efficiency and performance are better combined.

### 2.1.4 Implementations

Since our solution will be implemented with OpenCL we will be focusing our attention on solutions of the Marching Cubes algorithm based on stream processors like GPUs. This kind of solutions has been a topic of intensive research in recent years. Mainly because MC algorithm is particularly suited for parallelization and sometimes the required processing power is huge. But even in solutions based on GPUs, which have massive processing power and memory bandwidth, the employment of computation avoidance strategies are really important. Unfortunately, this strategies greatly increase the

complexity of such solutions, mostly due to programming functionality being graphics-based.

Prior to the introduction of geometry shaders (GS), GPUs completely lacked functionality to create custom primitives directly. Consequently, geometry had to be instantiated by the CPU or be prepared as a vertex buffer object (VBO). Therefore, a significant part of the work needed to be done in the CPU. Or a fixed number of triangles had to be assumed for each MC cell, wasting unnecessary resources.

A very common approach while using vertex shader (VS) or fragment shader (FS) solutions is to exploit a generalization of the MC algorithm, the marching tetrahedra (MT) algorithm [Elv92]. It has some advantages due to the reduced amount of redundant triangle geometry, since MT never requires more than two triangles per tetrahedron. In addition, inspecting only four corners is enough to determine the configuration of the tetrahedron, reducing the amount of inputs. MT has also the advantage of being easily adapted to unstructured grids. Pascucci *et al.* [Pas04] and Klein *et al.* [KSE04] are a couple of examples that explored this approach.

Even though MT being very common there are also approaches that used the MC algorithm. Goetz *et al.* [KW05] used the CPU to classify MC cells and only then were passed to the GPU to process the rest of the algorithm. A similar approach was followed by Johansson *et al.* [JC06] where a kd-tree were used to cull empty regions. In both situations was noted that this pre-processing on the CPU limits the speed of the algorithm.

When using hardware with shader model (SM) 4 capabilities the GS stage can produce and also discard geometry on the fly. This is very useful since most methods based on previous hardware generations produce isosurfaces that are cluttered with degenerate geometry. Degenerated geometry can yield poor performance because of unnecessary computations, which can represent a significant percentage. To produce a compact sequence of triangles additional post-processing is required, such as stream compaction.

An interesting approach was followed by Dyken *et al.* [DZTS08] where they reformulated MC as a data compaction and expansion process. This reformulation was base on Histogram Pyramid (HP) algorithm [ZTTS06], previously only used in GPU data compaction. The implementation was suitable for any graphics hardware with at least SM 3 capabilities, using OpenGL 2.0 or a comparable graphics APIs. The entire process was computed on the GPU and at the same time could produce a compact sequence of isosurface triangles, resulting in a highly efficient and interactive MC implementation. At the time, they claimed the performance crown of GPU-based isosurface extraction solutions. They also created a CUDA version but performed worst than the SM 3 implementation, due to the usage of an earlier framework version that lacked some functionality to avoid

unnecessary copy of data.

Another implementation of MC in CUDA is presented by Nagel [Nag08]. This approach has very specific properties, it's able to handle data sets so large that cannot be entirely loaded in the working computer's main memory. These data sets can be of two kinds, a single volumetric grid that is too large to fit in memory and many temporal volumetric grids that individually fit in memory, but combined don't. Because of these restrictions the process is force to compute only data set's portions at a time in the GPU and then send them back to the CPU. After that, vertex information is compressed to make rendering possible. Thanks to this method the author claims that this solution can handle data sets as large as 77GB.

The NVIDIA GPU Computing SDK also provides an MC implementation in CUDA C source form[3]. Because of that it will be provided a deeper presentation than previously done with other presented solutions. Although, trying to omit certain implementation details or optional features for simplicity sake. Even without any kind of introduction to NVIDIA's CUDA framework shouldn't be too difficult to understand the implementation principles. Besides, most framework architecture details are common to OpenCL, examined in the section 2.2.

The application follows a typical structure of a GPGPU approach. It's composed of two parts, the main program that executes on the CPU and functions that execute on the GPU.

The main program executed on the CPU do initializations, memory allocations and CUDA device coordination. All code that is executed only once or just a few times, work where serial processors are good at. The source code file is `marchingCubes.cpp` and most relevant functionality is in function `computeIsosurface()`.

The functions executed on the GPU do all operations that require large amounts of iterations and can be parallelized. These programs represent, or at least should represent, all the heavy work and most of the processing cycles.
The source code file is `marchingCubes_kernel.cu`.

The application has three functions that run on the GPU, each with a different task:

1. classification of voxels topology (`classifyVoxel`)
2. separation of occupied voxels from empty ones (`compactVoxels`)
3. calculation of triangle's vertices and normals (`generateTriangles`)

Additionally, there's also a CUDPP library[4] function that runs on the GPU, preforming scan operations, also known as perfix sum operations. All these functions will be further explained in the algorithm context.

[3] http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html#marchingCubes
[4] http://code.google.com/p/cudpp/

One relevant part of the MC algorithm is the look-up table, which in this case is built offline and stored like a texture. In this solution there's also a vertex table that covers all possible cube topologies, also built offline and stored like a texture. Since the data being processed is arranged in a 3D form, the processing elements are also organized in such way. All other bits of initialization, memory allocation, and other operations which aren't relevant to the MC algorithm will be ignored.

This MC algorithm begins by classifying all voxels in the data set, this operation is done by the first GPU function. The goal of this evaluation is to populate two lists, one with the numbers of vertices per voxel, the other indicating whether a voxel is occupied. Each thread processes one voxel at a time and regardless its organization all vertex are verified, even if some vertices are shared with other voxels.

After classification an scan operation (also known as prefix-sum) is applied on the binary list that contains all occupied and empty voxels. A naive version of this operation is able to sum $n$ elements of a list in $log_2(n)$ steps, illustrated in figure 2.7. Along with the sum, it also produces a new list where all ones are replaced by unique and sequential values, as shown in table 2.1.

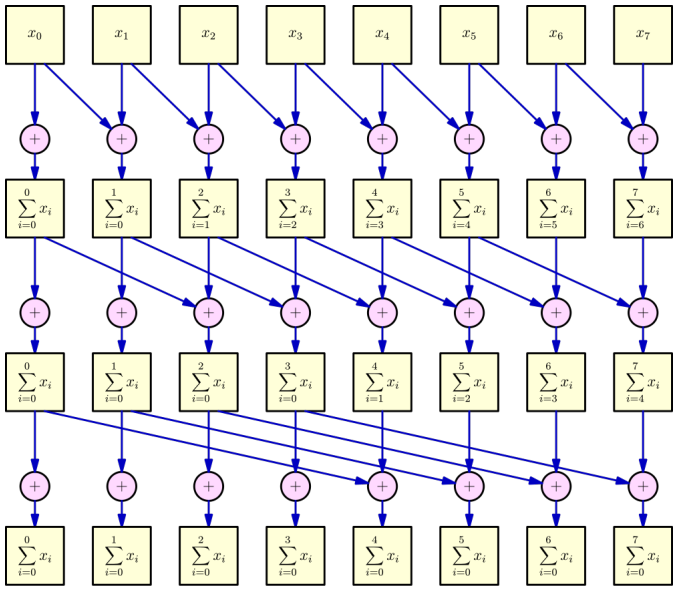


**Figure 2.7:** Illustration of a naive scan applied to an eight-element list in $log_2(8) = 3$ steps.

The previous operation is particularly useful in the next stage. To be able to produce an isosurface efficiently it's necessary to filter out all voxels that aren't useful, the empty ones. Taking advantage of the list produced in the previous stage, which enumerated all occupied voxels, a new list containing only useful voxels is created. This operation is also called compaction and is done by the second GPU function.

| **1** | 0 | 0 | **1** | **1** | 0 | **1** | **1** |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 2 | 1 | 1 | 2 |
| 1 | 1 | 1 | 2 | 2 | 2 | 3 | 3 |
| **1** | 1 | 1 | **2** | **3** | 3 | **4** | **5** |

**Table 2.1:** Illustration of a naive scan applied to a binary list, resulting in a unique and sequential values.

After compaction another scan operation is applied, this time on the list containing the vertices for each voxel. The operation results in its sum and a list that contains the vertices index for each voxel. The principle is the same as the above but with integers, shown in table 2.2.

| **3** | 0 | 0 | **6** | **3** | 0 | **9** | **12** |
|---|---|---|---|---|---|---|---|
| 3 | 3 | 0 | 6 | 9 | 3 | 9 | 21 |
| 3 | 3 | 3 | 9 | 9 | 9 | 18 | 24 |
| **3** | 3 | 3 | **9** | **12** | 12 | **21** | **33** |

**Table 2.2:** Illustration of a naive prefix sum applied to a integer list, resulting in the sum of all previous values.

All previous stages produced the necessary information for starting the triangle generation at this stage, it corresponds to the third GPU function. Vertices are interpolated to find out where the actual edge intersection occurred, producing more detail. This information is stored in shared memory instead of local one, which is claimed to be faster. After that, it starts organizing vertices in triangles, and calculating their surface normals. The final result is the population of two lists, one with vertex information, the other with normal information. The information of these two lists are indexed based on the list produced in the previous stage, creating a compact stream of isosurface triangles. Since there's a list of occupied voxels it's possible to run this operation only on them, discarding all irrelevant ones.

The final stage is rendering all triangle information to create the isosurface. Additionally, the triangle surface normal information is used to enable shading.

Besides some brute force operations, like the verification of voxels topology, all heavy work is done in GPU, ensuring good performance and even interactivity. One detail that is also pointed in the source code is that, it's possible to combine the both scan operations into just one. Because the information related with each voxel shares the same index in both lists.

There's a detail that is not completely true but was used for keep the presentation

easy to understand. The scan operation used was inclusive when in reality the exclusive is used, it fits computer programming needs better. In the exclusive prefix sum, the first element in the result list is the identity element (0 for add operation) and the last element of the operand list is not used. More detailed information about these operations with CUDA is available in [HSO07].

One thing that all the above solutions have in common is that all preform better than CPU approaches. Some with marginally speed-ups others with huge ones, depending on the chosen approach. But most of them required at least moderate implementation efforts since working with graphics-based APIs, isn't an easy task. In this field, GPGPU APIs provide easier ways to implement such solutions, in theory it requires the same efforts that parallel-CPU solutions.

## 2.2 OpenCL

OpenCL (Open Computing Language) is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors, giving software developers portable and efficient access to the power of these heterogeneous processing platforms.

### 2.2.1 Introduction

The internal architecture of early GPUs were too tied to graphics programming standards such as DirectX and OpenGL. The tight relationship ensured an overall good performance, but limited the possibilities to what was provided by such graphics standards. Eventually, the GPU makers introduced customizable processing elements in the GPUs pipelines to overcome this limitation. These elements were able to run specific programs, called shaders.

Custom processing elements evolved over time, replacing more fixed function stages in the GPU pipeline. Eventually all these elements were unified, being able to provide functionality that were previously offered by different elements. The change was also reflected in the shaders, that went from assembly to high-level languages, capable of offering programs with advanced functionality. This would allow the creation of stunning applications, like video games, and the beginning of what is today known as GPGPU.

Before the introduction of GPGPU standard APIs this kind of processing was already done, it could be accomplished using OpenGL and DirectX APIs, but that was a really tough job.

When Nvidia introduced its GPGPU implementation, known as CUDA, the task of creating something capable of running on a GPU became much easier. This framework provides an easy API, which isn't graphics-based like OpenGL or DirectX, coupled with C language (with some restrictions and Nvidia extensions) for writing kernels, the actual functions that execute on GPU devices.

OpenCL born from efforts of Apple while developing the Grand Central framework, its goal was to make multi-threaded parallel programming easier, later they realized it could be mapped nicely to the GPGPU problem domain. Then they wrapped up their Grand Central API into an API specification and, supported by Nvidia, AMD and Intel, released it to the Khronos Group as OpenCL.

OpenCL is the first open standard capable of producing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, and other processors. This flexibility is defined in its specification [mun10], made up of three main parts: the language specification, platform layer API and runtime API.

The language specification describes the syntax and programming interface for developing kernels. The language used is based on subset of ISO C99, due to its prevalence and familiarity in the developer community. To guarantee consistent results across different platforms, a well-defined IEEE 754 numerical accuracy is defined for all floating point operations. One of the weakest points in CUDA is not being able to guarantee such accuracy, very important in scientific fields. There's also some functions that assist the manipulation of such numerical formats. The developer has the option of pre-compiling their OpenCL kernels or letting that operation be preformed in runtime. Compilation in runtime guarantees that kernel programs can run on devices that don't even exist.

The platform layer API offers access to functions that query the OpenCL system, providing all kinds of useful information. Based on such information, developers can then choose the fittest compute devices to properly run their workload. It is at this layer that contexts and queues for operations such as job submission and data transfer requests are created.

The runtime API is responsible for managing the resources in the OpenCL system. It allows performing operations defined in contexts and queues.

### 2.2.2 Architecture

To help describe OpenCL architecture, it's presented a hierarchy of models based on OpenCL specification [mun10]:

- Platform Model
- Memory Model
- Execution Model
- Programming Model

#### 2.2.2.1 Platform Model

Every OpenCL environment has one *host*, where the software to control the devices is executed, typically a CPU. The host is connected to, at least, one *computational device*, which can be a CPU, GPU, or another accelerator. Each device contain one or more *compute units* (processor cores). And these units are themselves composed of one or more single-instruction multiple-data (SIMD) *processing elements*. This model is shown in figure 2.8.

#### 2.2.2.2 Execution Model

OpenCL programs are composed of two parts, a *host program* that executes on the host and *kernels* that execute on computing devices. The host program controls the device(s) and typically executes serial code that wouldn't take advantage of computing devices' parallelism. Kernels are parallel portions of code that can be accelerated in computing

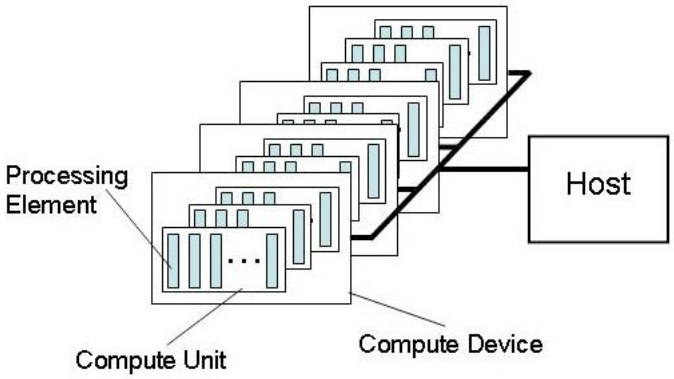



**Figure 2.8:** Illustration of the platform model.

devices.

The key property of OpenCL execution model is defined by how kernels execute. Each kernel submitted for execution gets an index space, which is mapped to all instances. A kernel instance is called *work-item* and is identified by its index in the corresponding index space. The kernel code executed is the same in all work-items but each one of them has a different identification (ID), allowing different execution pathways.

Each kernel submission is applied to a group of work-items, called *work-group*. These groups provide a simple yet powerful organization and like work-items they are also identified by an index. Since work-items IDs are only unique within a work-group it's necessary to use their global ID or a combination of their work-item and work-group IDs to identify them globally. The work-items in a given work-group execute concurrently on the processing elements of a single compute unit.

The space index used to identify work-items or work-groups is called *NDRange*. A NDRange is an N-dimensional index space, where N can be one, two or three. Figure 2.9 is an example of a two-dimensional (2D) arrangement of work-groups and work-items.

After learning how working units are organized it's time to know how to submit requests to them. OpenCL defines a *context* that provides resources to enable the execution of the kernels. The context includes the following resources:

**Devices** The collection of OpenCL devices to be used by the host.

**Kernels** The OpenCL functions that run on OpenCL devices.

**Program Objects** The program source and executable that implement the kernels.

**Memory Objects** A set of memory objects visible to the host and the OpenCL devices.

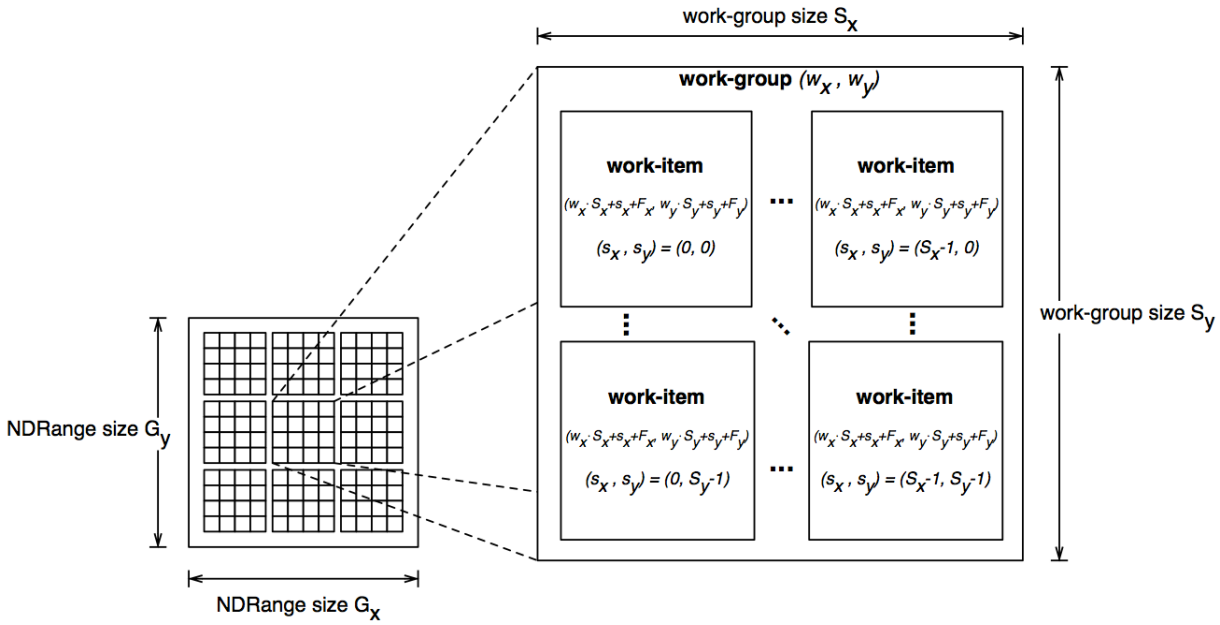



**Figure 2.9:** Illustration of a two-dimensional (2D) arrangement of work-groups and work-items.

OpenCL API provides functionality to create and manipulate the context. To coordinate the execution of the kernels there's a facility called *command-queue*. This queue schedules commands onto the devices within the context. These commands include the following functionality:

**Kernel execution commands** Execute a kernel on the processing elements of a device.

**Memory commands** Transfer data to, from, or between memory objects, or map and unmap memory objects from the host address space.

**Synchronization commands** Constrain the order of execution of commands.

The commands queued are then executed asynchronously between the host and the device. The order of execution depends on the mode, which can be *in-order* or *out-of-order*.

In-order execution guarantees the submission order. It fallows the logic first in first out (FIFO), in other words, a prior command on the queue completes before the following command begins. This serializes the execution order of commands in a queue.

Out-of-order execution doesn't enforce any particular order, following commands don't have to wait for the current to finish execution. Any order constrains must be enforced by the programmer through explicit synchronization commands.

Kernel execution and memory commands submitted to a queue generate event objects. These are used to control execution between commands and to coordinate execution between the host and devices.

It is possible to associate multiple queues with a single context. These queues run concurrently and independently with no explicit mechanisms within OpenCL to synchronize between them.

#### 2.2.2.3 Memory Model

OpenCL defines four different memory regions that are available to the kernels. These memory regions differ mainly in size, latency and access capabilities. Much like the levels of cache in a CPU, with the exception that in OpenCL they are explicitly manipulated, providing great potential for optimizations. Smaller regions are the fastest and also closest to the working units, and vice-versa. A conceptual OpenCL device memory model is shown in figure 2.10. Table 2.3 resumes memory allocations and access capabilities of different regions from the device and the host.

**Global Memory** This memory region permits read/write access to all work-items in all work-groups. Work-items can read from or write to any element of a memory object. Reads and writes to global memory may be cached depending on the capabilities of the device.

**Constant Memory** This memory region permits read access to all work-items in all work-groups. Work-items can read from or write to any element of a memory object. Reads and writes to constant memory may be cached depending on the capabilities of the device.

**Local Memory** A memory region local to a work-group. This memory region can be used to allocate variables that are shared by all work-items in that work-group. It may be implemented as dedicated regions of memory on the OpenCL device. Alternatively, the local memory region may be mapped onto sections of the global memory.

**Private Memory** A region of memory private to a work-item. Variables defined in one work-item's private memory are not visible to another work-item.

| | Global | Constant | Local | Private |
|---|---|---|---|---|
| Host | Dynamic allocation<br>Read / Write access | Dynamic allocation<br>Read / Write access | Dynamic allocation<br>No access | No allocation<br>No access |
| Device | No allocation<br>Read / Write access | Static allocation<br>Read-only access | Static allocation<br>Read / Write access | Static allocation<br>Read / Write access |

**Table 2.3:** Memory allocations and access capabilities.

Memory models of the host and OpenCL devices are almost independent, that's because the host is outside of the OpenCL scope. However they interact when explicitly

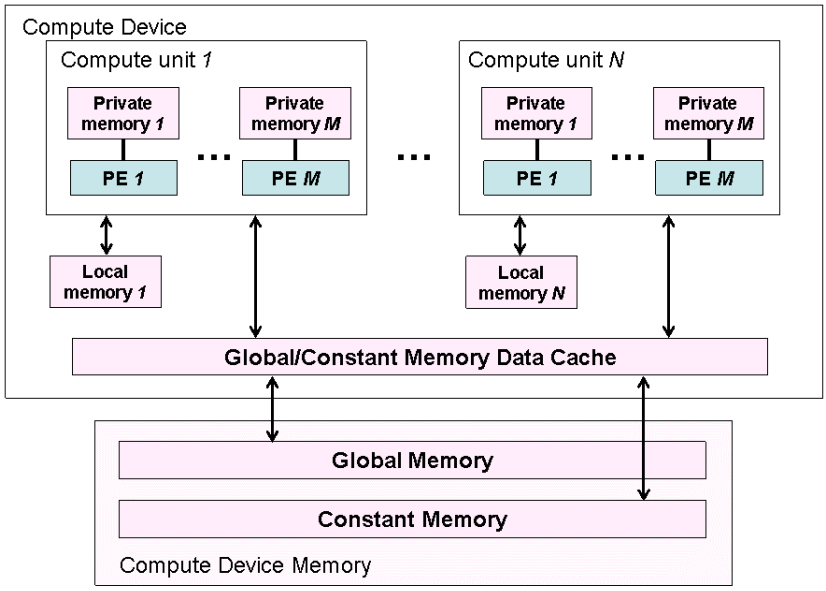



**Figure 2.10:** Illustration of a conceptual OpenCL device memory model.

copying data or mapping and unmapping regions of a memory object. These two operation are accomplished using the command queuing mechanism explained previously, in sub-sub section 2.2.2.2.

Explicit copy of data between the host and a device may be a blocking or non-blocking operation. Blocking operation function calls return only after the host can be safely reused, without the risk of tempering the data being transferred. Non-blocking operation function calls return as soon as the command is enqueued.

OpenCL uses a relaxed consistency memory model, for example, the state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times.

Within a work-item memory has load / store consistency. Local memory is consistent across work-items in a single work-group at a work-group barrier. Global memory is consistent across work-items in a single work-group at a work-group barrier, but there are no guarantees of memory consistency between different work-groups executing a kernel.

#### 2.2.2.4 Programming Model

OpenCL API explicitly supports *data parallel* and *task parallel* programming models. The primary supported model in the design of OpenCL is data parallelism.

In the data parallel programming model, the same kernel is passed through the command queue to be executed simultaneously across the compute units or processing elements. The index space associated with the OpenCL execution model defines the work-items and how the data maps onto the work-items.

For the task parallel programming model, different kernels are passed through the command queue to be executed on different compute units or processing elements. The parallelism can be express by enqueuing multiple tasks.

The figure 2.11 shows the different between the two models. Supposing that a program is composed by four independent tasks that operate on a data set to produce a result. While it's possible to process those four tasks simultaneously it's also possible to divide the data in four equal parts and apply the combination of the four tasks to them simultaneously.

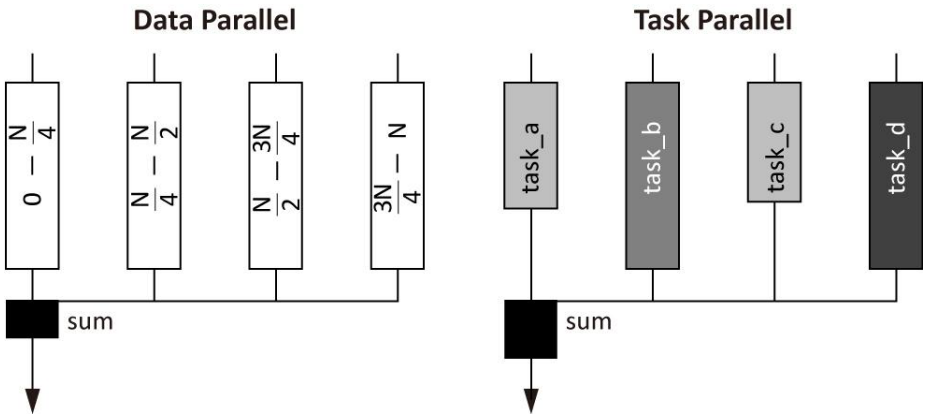



**Figure 2.11:** Illustration of the difference between data parallel and task parallel programming models.

There are two domains of synchronization in OpenCL, work-items in a single work-group and commands enqueued to command-queue(s) in a single context.

Synchronization between work-items in a single work-group can be accomplished using *barriers*. All the work-items of a work-group must execute the barrier before any are allowed to continue execution beyond the barrier. Note that the work-group barrier must be executed by all work-items of a work-group or by none of them, to avoid dead-locks. There is no mechanism for synchronization between work-groups.

Synchronization between commands in command-queues can be accomplished using *barriers* or *events*.

Command-queue barriers ensure that all previous queued commands have finished and any memory changes are visible to posterior queued commands before they begin executing. This barrier can only be used to synchronize between commands in a single command-queue.

Events result from OpenCL API function calls that enqueue commands. A posterior command waiting for certain event can guarantee that updates to memory objects are visible before the command begins execution.

# 3

# Implementation

## 3.1 Introduction

The goal is to implement an (indirect) volume rendering solution using the Marching Cubes algorithm, taking advantage of the OpenCL framework. This framework allows to accelerate the execution of the algorithm, using heterogeneous processing devices, in its heavy paths. Which means that besides the traditional CPUs it's possible to use GPUs and other kinds of (parallel) processors, like the IBM Cell, to explore their high parallel processing power. Also, this framework is device/vendor independent (contrary to CUDA), it provides portability between different families of devices. The implementation tries to be as efficient and parallel as possible, allowing it to be fast and scale up well in multi-processor devices. Its generic design tries to promote extensibility and readability over small tweaks.

## 3.2 Summary

This implementation bundles two important components, one focused on the Marching Cubes algorithm (mc* prefixed modules) and the other addressing OpenCL implementation details (cl* prefixed modules). The algorithm component has two fundamental modules, one in charge of decompose and distribute the work for the available processing devices (mcDispatcher) and the other containing a Marching Cubes algorithm's implementation (mcCore). The OpenCL component part offers extended functionality (clScan) and low-level API abstraction (clHelper).

The execution starts by loading the dataset to main memory, then the mcDispatcher module defines work units (independent subsets of the dataset) and distributes them

for the available processing devices, following a method similar to round-robin. This division of the dataset in smaller pieces allows big volumes of data to be processed, in addition to enable the usage of more than one device in parallel. These work units are computed by the mcCore module, which can work in parallel by (at least) as many instances as available devices, each producing a fraction of the output until all of them are finished. The mcCore implementation of the Marching Cubes algorithm takes as input an isovalue and a dataset and outputs the corresponding 3D model (geometry and shading). This module is divided in three phases (classification, compaction and generation), closely corresponding the three OpenCL kernel functions used to speed up the algorithm.

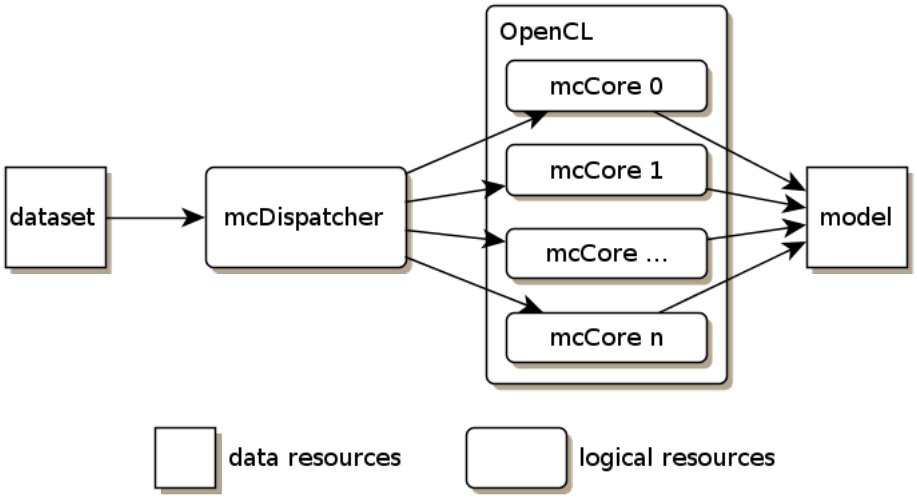



**Figure 3.1:** Overview of the execution work-flow (to keep it simple only Marching Cube modules are visible, most OpenCL modules usage is done inside mcCore module).

## 3.3 Host Modules

### 3.3.1 mcDispatcher

This module divides the work into smaller pieces, called work units, and dispatches them to be processed. Each work unit produces a memory entry (a fraction of the final result) that make the output list. The division of work brings some advantages, related to a higher division granularity, but there are also some associated costs.

One important advantage is the possibility of processing high quantities of data (big datasets). The memory usage during a regular execution of the algorithm (mcCore module) reaches several times the input (dataset) size, without the division it would be impossible to process an input with more than a small fraction of the device's memory.

Another important advantage is the distribution of work by multiple devices, this allows to aggregate their processing power and memory capacity to some extent. Since each work unit is pretty much independent from each other, it becomes easy to distribute and process them among different devices. To handle multiple devices in a practical

way, each has an independent execution path (using threads) and it's responsible for requesting work every time it becomes unoccupied and there's available work units yet to be processed.

The associated costs of duplicate data across boundaries of two consecutive work items are seen as overhead. Since datasets and work units are made of slices (like a stack of paper), being one slice (a sheet) the smallest unit of data, each division accounts for (at least) one slice of overhead. In addition, the process of dispatching may have to account for exceptions, boundaries that didn't result from division (the first and last slice) require especial treatment. This exceptional treatment only happens when the duplicated data goes further behind the boundary, in such case the overhead grows up one slice, in both extremities (exception made to the the first and last slice), for each extra level of duplicity. Another technical detail that influences the overhead is the work unit size, it must be defined considering aspects like overhead ratio, runtime algorithm's memory requirements, fair distribution of work across multiple devices, among others.

Besides the resulting overhead there's also the extra technical work of coordinate multiple concurrent paths of execution (threads). This coordination make use of locks, a mutual exclusion method, to ensure that each work unit (or some part of it) is assigned only to one device. A central variable is used to hold the progress of work, it points to the first slice of the remaining undone work. Whenever a work unit is requested this variable is updated to reflect the current progress of work, in a complete exclusive way.

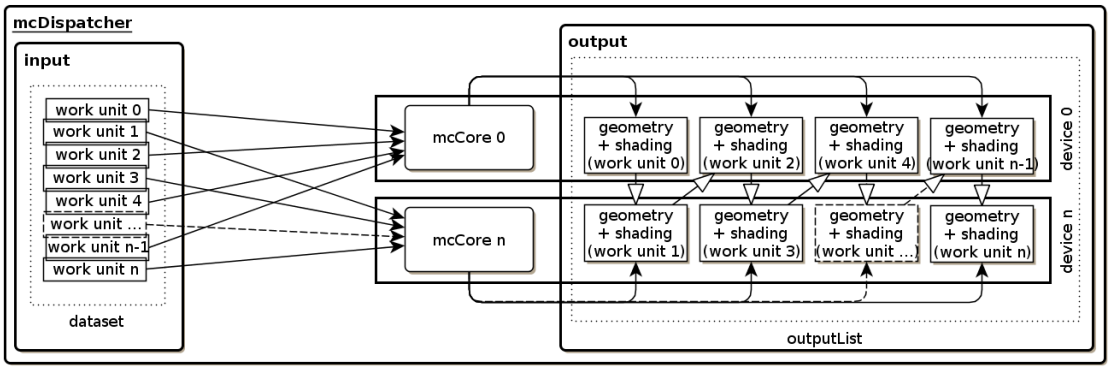



**Figure 3.2:** mcDispatcher module execution work-flow.

#### 3.3.1.1 Work Unit Size

Choosing a good work unit size can help minimize the side effects of the work division. But this choice has to have into account several important factors, such as the device memory size and processing power, the usage of multiple devices, the overall performance, among other things. In this context, small work units are sets having just a few slices while big work units have many slices.

As previously mentioned, during a regular execution of the algorithm (mcCore module) the memory usage reaches several times the input (work unit) size, which can become a problem if the input size is too big. In extreme cases can be impossible to run a single execution the algorithm, in less extreme cases there's still the problem of the limited amount of space to store other work units results (algorithm runtime needs + other work units results). Another issue with big work units is the unbalanced distribution of work when using multiple devices, specially relevant on the last work units. Some devices may become idle whereas others stay running for a significant period of time.

While using big work units leads to unwanted consequences, using too small units also has big disadvantages. Small work units means more divisions, more duplicated data, more overhead and a bigger performance hit. This happens because duplication is done for complete slices, so even if each slice is quite big (its area) but the work unit has just a few of them the overhead is high.

Device processing power can also assist the decision, specially useful when there are multiple devices and they yield different performances. Although useful in certain situations this subject wasn't deeply studied and because of that it doesn't help in the decision making process.

To see how the work unit size affects the algorithm's execution some tests were conducted using two different sized datasets, $256^3$ and $512^3$ samples, while different work unit sizes are being used. To focus the attention on this subject and keep things as simple as possible no additional details will be given about the input or the output, they are not important here and are further analysed in chapter 4. This results help choose a good work unit size in ideal conditions.

Tables 3.1 and 3.3 present details about the execution, the columns *size* and *percentage* are pretty much self explanatory and the column *count* has the quantity of work units resulted from the corresponding work unit size. Tables 3.2 and 3.4 present results about the execution, the columns *time* have the total time spent, the columns *worst* have the speedup (in percentage) from the worst case (which is the first), the columns *previous* have the speedup (also in percentage) from the previous case (hence the non existing result in the first case) and the column *speedup* has the speedup between the devices. Figures 3.3, 3.4 and 3.5 are charts based on the results from table 3.2 ($256^3$ samples dataset) and figures 3.6, 3.7 and 3.8 are based on the table 3.4 ($512^3$ samples dataset).

After analysing the results there's a clear trend, independently of the dataset size (either the number of slices or their size) the work units with **30** slices are the ones which best reflect a good choice without compromising the factors previously mentioned. This conclusion can be best backed up by the figures 3.5 and 3.8, in which it's possible to see what are the gains from each step. In both cases it's clear that the gains are significant until that mark, becoming marginal after that, this makes the work unit size around 30 slices the best choice. Another conclusion that can be taken from this results (and also from what was already mentioned) is that the work unit size based on the number of

| work unit | | | overhead | | |
|---|---|---|---|---|---|
| size(slices) | size(MB) | count | size(slices) | size(MB) | percentage |
| 10 | 2,5 | 29 | 84 | 21,0 | 24,71% |
| 20 | 5,0 | 14 | 39 | 9,8 | 13,22% |
| 30 | 7,5 | 9 | 24 | 6,0 | 8,57% |
| 40 | 10,0 | 7 | 18 | 4,5 | 6,57% |
| 50 | 12,5 | 6 | 15 | 3,8 | 5,54% |
| 60 | 15,0 | 5 | 12 | 3,0 | 4,48% |
| 70 | 17,5 | 4 | 9 | 2,3 | 3,40% |
| 80 | 20,0 | 4 | 9 | 2,3 | 3,40% |
| 90 | 22,5 | 3 | 6 | 1,5 | 2,29% |
| 100 | 25,0 | 3 | 6 | 1,5 | 2,29% |

**Table 3.1:** Details about processing a $256^3$ samples dataset using different work unit sizes.

| work unit | Quadro FX3800 | | | Tesla C2050 | | | |
|---|---|---|---|---|---|---|---|
| size(slices) | time(ms) | worst | previous | time(ms) | worst | previous | speedup |
| 10 | 200 | - | - | 193 | - | - | 1,04 |
| 20 | 170 | 15,00% | 15,00% | 143 | 25,91% | 25,91% | 1,19 |
| 30 | 146 | 27,00% | 14,12% | 123 | 36,27% | 13,99% | 1,19 |
| 40 | 143 | 28,50% | 2,05% | 116 | 39,90% | 5,69% | 1,23 |
| 50 | 140 | 30,00% | 2,10% | 116 | 39,90% | 0,00% | 1,21 |
| 60 | 136 | 32,00% | 2,86% | 110 | 43,01% | 5,17% | 1,24 |
| 70 | 133 | 33,50% | 2,21% | 110 | 43,01% | 0,00% | 1,21 |
| 80 | 133 | 33,50% | 0,00% | 106 | 45,08% | 3,64% | 1,25 |
| 90 | 130 | 35,00% | 2,26% | 103 | 46,63% | 2,83% | 1,26 |
| 100 | 130 | 35,00% | 0,00% | 103 | 46,63% | 0,00% | 1,26 |

**Table 3.2:** Results about processing a $256^3$ samples dataset using different work unit sizes.

| work unit | | | overhead | | |
|---|---|---|---|---|---|
| size(slices) | size(MB) | count | size(slices) | size(MB) | percentage |
| 10 | 10 | 57 | 168 | 168 | 24,71% |
| 20 | 20 | 27 | 78 | 78 | 13,22% |
| 30 | 30 | 18 | 51 | 51 | 9,06% |
| 40 | 40 | 14 | 39 | 39 | 7,08% |
| 50 | 50 | 11 | 30 | 30 | 5,54% |
| 60 | 60 | 9 | 24 | 24 | 4,48% |
| 70 | 70 | 8 | 21 | 21 | 3,94% |
| 80 | 80 | 7 | 18 | 18 | 3,40% |
| 90 | 90 | 6 | 15 | 15 | 2,85% |
| 100 | 100 | 6 | 15 | 15 | 2,85% |

**Table 3.3:** Details about processing a $512^3$ samples dataset using different work unit sizes.

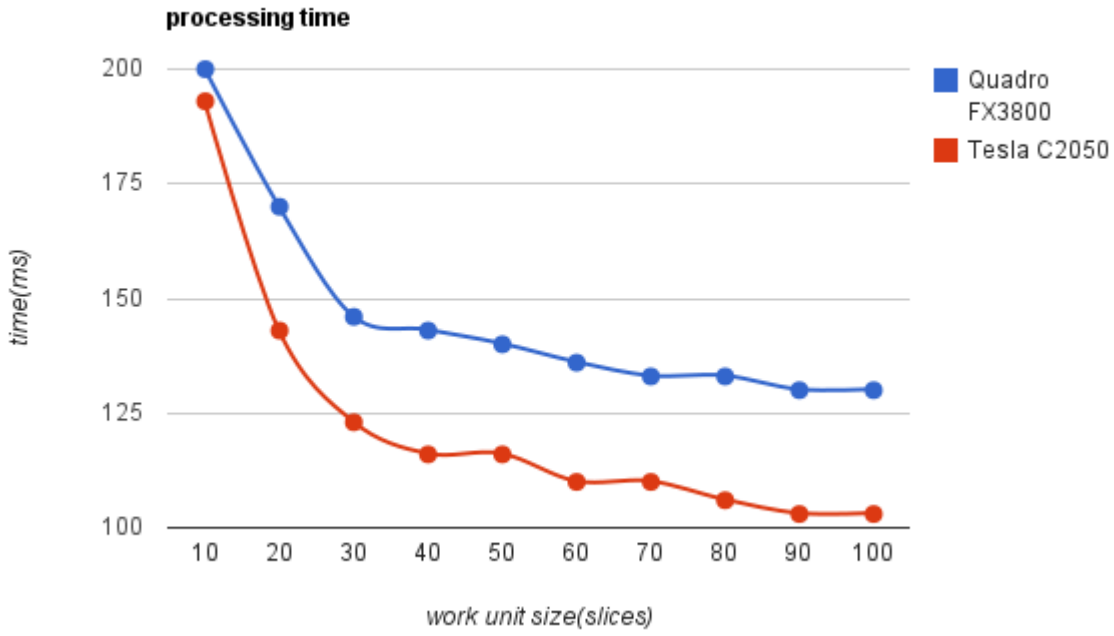



**Figure 3.3:** Processing time of a $256^3$ samples dataset using different work unit sizes.


**Figure 3.4:** Processing speedup from worst case of a $256^3$ samples dataset using different work unit sizes.


**Figure 3.5:** Processing speedup from previous case of a $256^3$ samples dataset using different work unit sizes.

| work unit | Quadro FX3800 | | | Tesla C2050 | | | |
|---|---|---|---|---|---|---|---|
| size(slices) | time(ms) | worst | previous | time(ms) | worst | previous | speedup |
| 10 | 1063 | - | - | 883 | - | - | 1,20 |
| 20 | 946 | 11,01% | 11,01% | 756 | 14,38% | 14,38% | 1,25 |
| 30 | 916 | 13,83% | 3,17% | 716 | 18,91% | 5,29% | 1,28 |
| 40 | 903 | 15,05% | 1,42% | 696 | 21,18% | 2,79% | 1,30 |
| 50 | 890 | 16,27% | 1,44% | 680 | 22,99% | 2,30% | 1,31 |
| 60 | 883 | 16,93% | 0,79% | 670 | 24,12% | 1,47% | 1,32 |
| 70 | 880 | 17,22% | 0,34% | 670 | 24,12% | 0,00% | 1,31 |
| 80 | 880 | 17,22% | 0,00% | 663 | 24,92% | 1,04% | 1,33 |
| 90 | 876 | 17,59% | 0,45% | 660 | 25,25% | 0,45% | 1,33 |
| 100 | 876 | 17,59% | 0,00% | 660 | 25,25% | 0,00% | 1,33 |

**Table 3.4:** Results about processing a $512^3$ samples dataset using different work unit sizes.

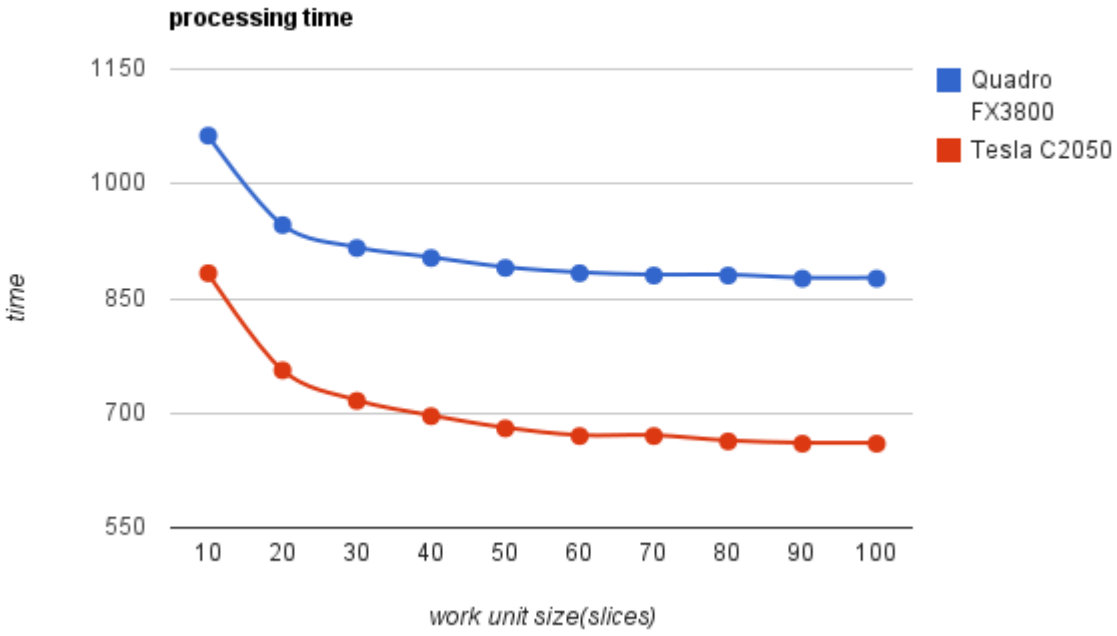



**Figure 3.6:** Processing time of a $512^3$ samples dataset using different work unit sizes.

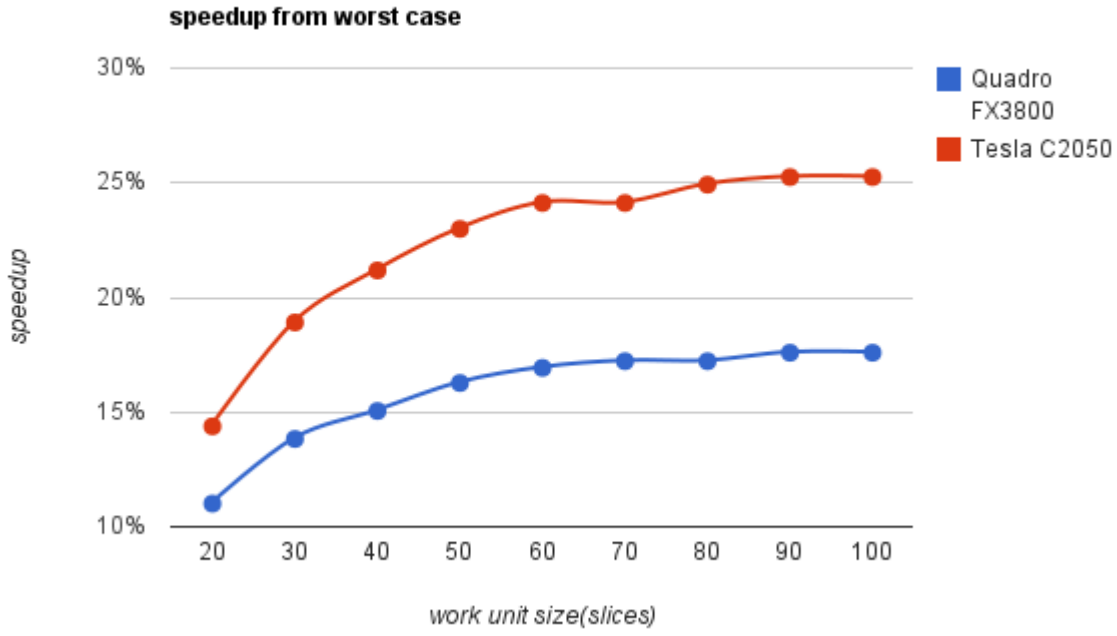



**Figure 3.7:** Processing speedup from worst case of a $512^3$ samples dataset using different work unit sizes.


**Figure 3.8:** Processing speedup from previous case of a $512^3$ samples dataset using different work unit sizes.

slices it's a better approach than using the storage space.

Even with 30 being a good choice there are some situations where this size is not practical, specially when dealing with big slices (its area). Since it's important to keep a moderate usage of memory because of the runtime needs and the results from other work units, the storage space approach is also used to limit extreme cases. This implies an enforcement of maximum and minimum limits in the memory usage, based on the device memory and reference dataset sizes. Besides device memory, dataset sizes are also used because they provide real numbers. As reference datasets, the small one has $128^3$ samples (8MB) and the big one has $1024^3$ samples (4GB). The minimum work unit size is set to 4MB, which allows the small dataset to be divided in 2 work units (each with 64 slices). The maximum work unit size is set to the smaller size between 128MB or $device\ memory\ size/30$. The first one allows the big dataset to have the ideal work unit size (32 slices), whereas the second means that the runtime needs of the algorithm are limited to about $1/5$ of the device's memory size, being the remaining used to store other work units output results. So, it's used the work unit size of 32 slices (around 30 and power of two) if it's within the enforced limits, else the necessary math are preformed to meet them.

### 3.3.2 mcCore

As previously mentioned, this module contains the Marching Cubes algorithm implementation and it's divided in three phases. In fact, the first two phases are just an efficiency requirement due to parallel implementation constrains, that is, the third (and last) phase do all the necessary work to build a model from the input dataset. The first two phases are just responsible for analyse and discard all the irrelevant data, generating a compacted version of the input dataset. This way the third phase will only process data that matters, and since it's the most expensive, the two previous phases are very important in the whole process. Previous implementations of the algorithm using OpenGL without Geometry Shaders suffer a great performance hit because weren't able to discard useless data, this would consume too much memory and processing time, render them inefficient and unable to scale up.

All critical paths of the algorithm are executed making use of OpenCL, through kernel functions. The host code provides the glue for setting up the execution of the algorithm, making a pipeline with the three phases.

The execution (and the first phase) begins by uploading the input dataset data to the device memory (in 3D texture form) and creating the two necessary memory buffers to output the results from the mcClassification kernel. These memory buffers, filled by the execution of the kernel, contain the analysis of the dataset (according to the chosen isovalue). For each voxel, one buffer indicates if the voxel is relevant (occupied) or not and the other the voxel's number of vertices (or triangles). The relevance buffer may seem redundant, since it is possible to deduce it from the buffer with the number of triangles (it's only relevant if it has at least one triangle), but is necessary for the next phase. The execution of the kernel ends this phase.

The second phase is all about using the results from the previous phase and discarding all the data (and further work) that isn't relevant. This translates into a new buffer containing just the indexes of voxels that will contribute to the output, a compacted index buffer produced by mcCompation kernel. But before executing the kernel it's necessary to preform an (exclusive) scan on the buffer holding voxels' relevance. Since this buffer has binary data the result is a buffer with an unique index, sequentially ordered, for all relevant voxels. Scanning an array also provides the sum of all its entries, in this case the number of relevant voxels, useful to determine the compacted buffer size and the resources used by the mcGeneration kernel. Also, if the result of this sum is zero the work is done, meaning that aren't any relevant voxel and that the execution of the algorithm is complete. For the remaining cases the next step is to create the compacted buffer previously mentioned and fill it by executing the mcCompation kernel. After the kernel execution there are two buffers no longer necessary, the voxels' relevance and its scanned version. These are released and replaced by the compacted version, which ends

the second phase.

The third and final phase is responsible for producing the model's geometry and the corresponding shading data. The result are two buffers, one containing groups of three vertices (triangles) and the other their normals, filled by the mcGeneration kernel. But before executing it there's one thing missing, a scanned version of the buffer with the amount of vertices per voxel (produced in the first phase). The result is a buffer with unique indexes (or positions) for all relevant voxels, but this time not sequentially ordered (non-binary array). Each index is the sum of the vertices from all previous entries. This fulfils exactly the requested needs because it allows to output all data in independent positions (addresses) and without any holes, similar to what was done in the previous phase. And once again, the sum of all entries, preformed by the scan, is useful to determine the output buffers size. After being scanned the triangles buffer is released since it's no longer needed. Now it's possible to execute mcGeneration kernel, which fills the output buffers. At this point, all remaining buffers, except the last two (geometry and shading buffers), are released and the algorithm is finished.

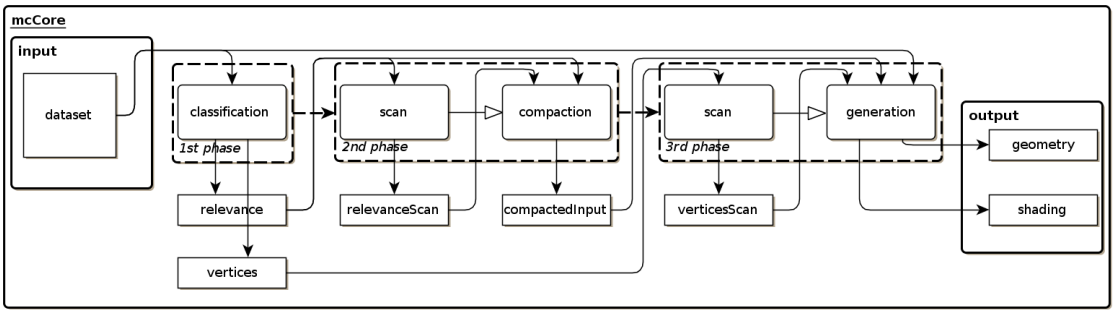



**Figure 3.9:** mcCore module execution work-flow.

#### 3.3.2.1 Memory Usage

As shown in figure 3.10 and already mentioned, the memory usage while executing mcCore module reaches several times the input size. The least heavy execution uses 4x the input size and happens when there's no relevant voxel. The most expensive execution uses at least 4x times the input size plus the compacted buffer size, or 2x times the input size plus the compacted buffer size plus 2x output buffer size, if it's bigger than the previous. Since the compacted buffer and the output buffers don't have a fixed size, it's not possible to predict *a priori* which of the two has the most expensive usage. It's only possible to predict that the memory usage is at least 4x time the input size plus the compacted buffer size (which most of the times is small, almost neglectable).

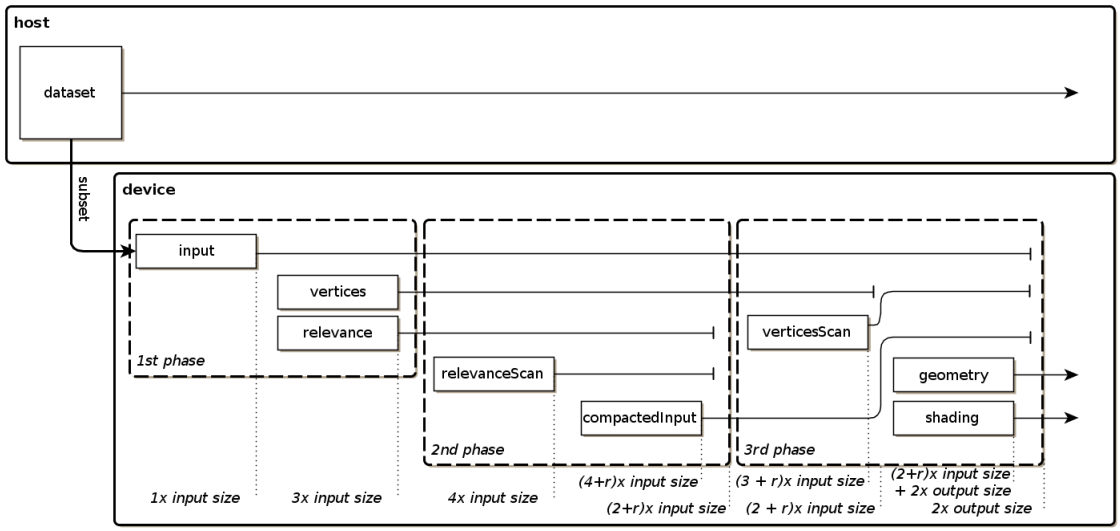



**Figure 3.10:** mcCore module memory usage.

#### 3.3.2.2 Data Format

This module takes as input a dataset and an isovalue, both of which are floating-point type (*cl_float*). The dataset is a simple array, implicitly arranged in a three dimensional form (volume's data), and the isovalue is a single value. The output can be released in one of two forms, VBOs[1] or OpenCL memory buffers, these are also floating-point type (*cl_float4*) and are stored in device memory. The difference between the two is that only memory buffers can be retrieved by the host and only VBOs can be used directly by OpenGL.

#### 3.3.2.3 CPU Implementation

A serial implementation that resembles the OpenCL one but skipping the unnecessary tasks, like compaction and all involved steps. Such implementation has the purpose of being the base reference for a serial version, it's very important to see how the OpenCL implementation preforms against it. The followed approach was to build something that could provide the same results while making it very simple and clean. Although it has no special tweak, the execution path is very strait forward which make it very fast.

The results aren't uploaded to the GPU, which isn't completely fair when compared with the OpenCL implementation.

### 3.3.3 clScan

This module provides the *scan* operation, a fundamental functionality in this implementation, also known as *prefix sum* or *reduction*. The operation outcome is the sum of all values from a vector while arranging it in a very useful way, performed in parallel, which makes such operation a very important piece of parallel computation.

[1] Vertex Buffer Objects

Just to recap, each value in the output vector results from summing all previous values in the input vector (e.g.: $out[2] = in[0] + in[1]$). This operation has two variants, *inclusive* [2] and *exclusive* [3], the former sums all previous values including it self while the latter doesn't, only includes previous values. While there are two variants, only the exclusive one is relevant in this case because its values can be used as proper indexes.

The output values can be used to uniquely address their corresponding [4] input ones, that is, for each non-null input value there's a corresponding unique output one. These values can be used as unique addresses to define independent and self-contained chunks of data on shared resources, allowing multiple processor units to operate simultaneously on it.

In this implementation scan is used to know the output size and the output indexes of dynamic sized buffers. The first case uses the scan operation on a binary vector, which counts (summing just ones and zeros can be thought as counting) how many elements (or components) are non-null and at the same time produces a vector containing unique and sequential indexes just in the corresponding non-null elements positions. These results are used to create a vector containing only non-null data based on the previous binary input vector. The second case uses scan on an integer vector, which sums all its elements and indexes (not sequentially) the non-null ones. Such results are used to create buffers and define independent chunks of data within them.

This module reuses the `OpenCL Parallel Prefix Sum (aka Scan) Example` [5] from Apple, adapted to fit the needs of the implementation. Instead of creating it from scratch the solution was forked from Apple's example, which provides features like non-power of two sizes, memory bank conflict avoidance and usage of local memory to store partial sums.

### 3.3.4 clHelper

This module provides some abstractions to the low-level OpenCL API. Tasks such as error reporting, resource and program handling, resource information, profiling and some others are offered as simple functions. Such tasks would require several steps using the OpenCL API. These functions try to be as generic as possible although some of them make some compromises oriented towards this implementation. This abstracted API was created with usability in mind, properties like similar behaviour and ease of use show that.

Although errors can be ignored, all functions handle and forward OpenCL errors, which can be very useful to know how things go in the OpenCL runtime while still using these abstractions.

[2] e.g: $out[1] = in[0] + in[1]$
[3] e.g.: $out[1] = in[0]$
[4] the same position in both vectors
[5] http://developer.apple.com/library/mac/#samplecode/OpenCL_Parallel_Prefix_Sum_Example

Some functions return the result directly instead of changing the arguments' referenced value, a very common approach in the OpenCL API. Tasks returning data in native types can be greatly simplified, reducing to just a function call what would require several lines of code using the OpenCL API. Profiling, resource information and error reporting functions benefit most with such approach.

Management of resources allows to initialize and free OpenCL resources in one step each, tasks that can be highly automatized. This is mostly because these resources are the minimum requisites to preform any useful work using OpenCL. Besides the common steps to initialize such resources, like platform and device selection and assignment of command queues to devices, the context creation also handles interoperation between OpenCL and OpenGL. Platforms can be selected based on their name, devices can be selected based on a provided list and their type. The creation returns a (struct) object that comprises the necessary OpenCL (opaque) objects, while the first is used by some functions offered by this module it's still possible to have direct access to OpenCL using the second ones. Besides the necessary OpenCL objects there's also some useful information, which avoids querying OpenCL to know things like the maximum work-group size and available global and local memory.

Compilation and management of binaries are handled in a simple and automated manner. It's possible to compile programs' sources into binaries and store them on disk for future use. This avoids compiling always from source, which shrinks the initialization process time.

Simpler functionality such as resources information, error reporting, profiling events and creating shared buffers (GL-CL) are also provided.

## 3.4 OpenCL

### 3.4.1 Kernels

#### 3.4.1.1 mcClassification

This kernel is responsible for analyse and classify all voxels from the input dataset. This work depends on the isovalue, it influences the interpretation. The result is the output of two arrays – one containing relevance (occupied or empty, binary values) the other vertices, for each voxel.

Since the output size is fixed (an input voxel corresponds to one entry in each array), even for the entry size, there's no need to calculate the output positions in advance. They are mapped linearly, which can be (independently) calculated only at the moment they are needed. *mcClassification* is the only kernel that outputs arrays with a predictable size, later kernels will have to employ some technique to know the output size and positions.

Each work-item is responsible for one voxel, which means that are launched as many work-items as voxels, a direct relation between the input size, the amount of work-items and also the output size (1:1:2). This implementation arranges the work-items in a 3D

grid, it follows closely the input dataset form (3D texture) to avoid unnecessary conversions between different arrangements (1D to 3D). Although useful it's not strictly required, instead of calculating only the output position (on 1D arrays) it would be possible to use an 1D arrangement of work-items and calculate the necessary coordinates (on a 3D texture) instead. But since most operations are related to the input it's much preferable to use 3D arrangement of work-items. Later kernels will not follow this approach because in those situations it would be counter productive, further details about them will be given ahead.

The execution can be started as soon as the dataset and the isovalue are available (and the output arrays are created). Each work-item gets its own coordinates, the base to determine the voxel's corners coordinates used to read the corresponding values from the input texture. Together with the isovalue these corner values are used to determine the voxel's combination, which is then used to know how many vertices (or triangles) the voxel will produce. To know how many vertices a combination produces there are a table with all possible combinations (256) that are looked up, this table was built offline. Since the output is done to an unidimensional array, it's necessary to convert a 3D position to an 1D position. The 1D position is then used to write the amount of vertices and its binary variant (relevance - has vertices or not), to each array.

**Kernel 1** mcClassification pseudo-code

**input:** dataset, isoValue
**output:** occupied[], vertices[]

$sizes \leftarrow getSizes()$
$coordinates \leftarrow getCoordinates()$
$outputPosition \leftarrow getPosition(coodinates, sizes)$

$corners[8] \leftarrow getVoxelCornersValuesFromDataset(dataset, coordinates)$
$combination \leftarrow calculateVoxelCombination(isoValue, corners)$
$verticesCount \leftarrow getVoxelVerticesCount(combination)$

$vertices[outputPosition] \leftarrow verticesCount$
**if** $verticesCount$ **then**
  $occupied[outputPosition] \leftarrow OCCUPIED\_VOXEL$
**else**
  $occupied[outputPosition] \leftarrow EMPTY\_VOXEL$
**end if**

#### 3.4.1.2 mcCompaction

This is the simplest and also the lightest of all kernels, its goal is to create an array which contains only the indexes of non-empty entries from an array of binary values. Implementing the above functionality sequentially it's as simple as read each value from the input array and if it's not zero add its index to the output array. Now, when there's more

than one thread of execution problems arise, some data paths may suffer from concurrency problems, in particular the output positions. To solve such problems it's common to use a mutual exclusion method to protect the affected data, but such technique doesn't fit well under our model of independent data, it greatly hits parallel performance. A better solution to this problem can be achieved with an extra array, holding the positions for all output data, which can be obtained by a scanned version of the input array. Note that this result can only be accomplished in arrays with binary values, which is the case (when needed, convert non-binary arrays to binary ones can be easily done). With the positions (addresses) for all output entries obtained, each work-item will be in charge of processing one entry, and since all entries are independent it's possible to process all of them in a truly parallel way.

Once again there's a direct relation between the input data size and the amount of work-items launched (1:1), but not with the output data size. In contrast with the previous kernel, this doesn't arrange work-items in a 3D grid, it prefers a flat 1D arrangement because all data arrays involved (the inputs and the output) are unidimensional.

Each work-item begins by getting its index, which uses to determine the entry it's in charge with and where it can write the results (through the scanned array). Then, it only has to check the corresponding input value to decide if it has to write its index to the output array or not.

This kernel doesn't perform compaction *per se*, it only discards data considered irrelevant, producing (in general) a much smaller array, hence the name.

**Kernel 2** mcCompaction pseudo-code

**input:** values[], scannedValues[]
**output:** compacted[]

$position \leftarrow getPosition()$
$compactedPosition \leftarrow scannedValues[position]$

**if** $values[position]$ **then**
  $compacted[compactedPosition] \leftarrow position$
**end if**

#### 3.4.1.3 mcGeneration

The last kernel is responsible for producing all model's data (geometry and shading). It takes as input a dataset, an isovalue, a scanned version of the array containing the amount of vertices each voxel produces and an array holding only the indexes of the voxels which produce vertices (relevant ones), referred to as compacted. And outputs two arrays, these contain geometry (vertices) and shading (normals), respectively.

The scanned array will be used in the same manner as in the previous kernel, this time providing output positions (memory addresses) for the resulting output data (vertices and normals). This array is necessary because not all voxels produce the same amount

of vertices, and it's not desirable to assume the maximum for all of them. And, since they are processed in an independent way (to allow efficient parallelization) the position where they will write the results must be known in advance.

As previously said, this kernel will only work on relevant voxels (listed in the compacted array), each of them are processed by one work-item. Once more there's a direct relation between an input size, this time the compacted array, and the work-items launched (1:1). This kernel (like the first one) has data with different arrangements, the dataset are 3D while all other input and output buffers are in a 1D form. Since voxel attribution is based on the compacted array (1D), work-items will also follow the unidimensional arrangement.

The execution begins by determining its own work-item index, which is then used to get the position of one relevant voxel from the compacted array. This voxel position is converted to coordinates (1D to 3D) and the same steps used in the mcClassification are replicated here up to determining the voxel combination (exception made to the voxel's corners, that besides their values also hold their coordinates). This may seem redundant because it's possible to save also the combination and get it at this point, instead of preforming all the necessary steps to determine it again. But the voxel's corners values, used to determine the combination, are also used in further steps, which means this values would still need to be fetched. So, the only work that could be saved was the calculations to determine the combination, and since it's faster to preform them than get it from an array, the redundant work are excused.

After determining the combination it's calculated all vertices along the edges (formed by the corners). This vertices are determined using linear interpolation, applied between any two corners that form an edge, using their values as weights. All potential vertices (12) are calculated, even if they are outside the edge's limits, because it avoids branching, unnecessary complexity and cause no side effects on the next steps.

The next step is to get the amount of vertices produced by the voxel (the same way as in the first phase), an array listing which edges (their vertices) are used to build the triangles (looking up in a table containing the edges of all combinations, also built offline) and the output position for the whole voxel (through the scanned array).

At this point it's now possible to execute the final step, produce the vertices and its respective normals. This step consists of a block that loops as many times as the number of vertices to produce, while the individual output position is incremented and the edge array is iterated. Each loop produces one vertex and its normal and store them in the geometry and shading arrays, respectively. The vertices are fetched from the vertices array calculated in previous steps, using the current iteration of the edge array as the index. Its normals are calculated using the central difference method.

**Kernel 3** mcGeneration pseudo-code

**input:** dataset, isoValue, scanned[], compacted[]
**output:** geometry[], shading[]

$rawPosition \leftarrow getPosition()$
$position \leftarrow compacted[rawPosition]$
$coordinates \leftarrow getCoordinatesFromPosition(position, dataset.sizes)$

$corners[8] \leftarrow getVoxelCornersValuesFromDataset(dataset, coordinates)$
$combination \leftarrow calculateVoxelCombination(isoValue, corners)$
$vertices[12] \leftarrow calculateVoxelVertices(isovalue, corners)$

$verticesCount \leftarrow getVoxelVerticesCount(combination)$
$voxelEdges[] \leftarrow getVoxelEdges(combination)$
$voxelOutputPosition \leftarrow scanned[position]$

**for** $v \leftarrow 0 : verticesCount$ **do**
  $edge \leftarrow voxelEdges[v]$
  $outputPosition \leftarrow voxelOutputPosition + v$

  $geometry[outputPosition] \leftarrow vertices[edge]$
  $shading[outputPosition] \leftarrow calculateNormal(dataset, vertices[edge])$
**end for**

### 3.4.2 Enhancements

The next subsections detail some enhancements to the original algorithm implementation presented previously in this chapter. These can be performance or usability enhancements.

#### 3.4.2.1 Local Work-Group Size

Setting a good local work-group size can be very important and in some situations tweaking this value can yield some performance improvements. This happens because kernel executions always define an local work-group size, even if it's not explicitly defined. In cases where the global work size isn't multiple of good numbers, like powers of two, this can be very bad, performing much worst than technically could. It's also possible that some local work-group sizes, even if defined as a power of two, can hurt performance instead of improve it. To solve such problems, local work-group size isn't explicitly defined (where isn't necessary) but the global work size is *ceiled* to a multiple of the maximum kernel local work-group size. This way the drivers can figure what's best, while in some situations doesn't reflect the best possible performance it's future-proof.

#### 3.4.2.2 Images

In some situations using images to store data (better known as textures in CUDA or OpenGL) instead of plain buffers has big advantages. This advantages are presented as performance enhancements and also as higher-level objects in the kernel code (functionalities and usability) compared to buffers. In this implementation, one particular case that can benefit greatly from using data in image form is the input dataset.

This data is read-only (reading and writing to the same texture within a kernel isn't supported) and its 3D structure is closely mapped to the execution arrangement of the work-items, which benefits from the spatial locality of their access patterns. Contrary to buffers, images are always cached (optimized for 2D spatial locality), most of the time reading from a texture costs only a cache read and not a read from global memory, much more expensive. The combination of the data access pattern and the texture cache optimization is a great fit, achieving better performance than buffers (even if they are cached - following their 1D structure). Images are used to avoid uncoalesced loads from global memory, which has big negative performance consequences.

Besides reading performance, images also offer some functionality within a kernel which buffers simply don't, because they lower-lever entities. Handling buffers structured in 3D (or 2D) is a manual job, that is, it's necessary some extra code to provide the functionality. Instead, images already have built-in functions that provide very useful functionality, which are supported in hardware by dedicated units (texture units). Images feature filtering which can turn discrete data into continuous data (interpolating contiguous values) and addressing modes which handle boundaries in a very simple way (clamp, repeat or limit values behind boundaries). In this implementation, calculating vertices normals relies heavily in interpolating values from the input dataset, using images avoids extra code and even enhance performance because the interpolation is preformed by dedicated hardware units (a win-win situation). Also, not worry with boundaries avoids some corner-cases code, which sometimes means branching (performance hit) due to condition instructions.

#### 3.4.2.3 Pinned Memory

Pinned memory (or page-locked) transfers achieve the highest bandwidth between the host and the device. This kind of memory is needed to preform DMA transfers, operations that are not preformed by the CPU and so have their limitations. In this case, the limitation relies in the fact that during these kind of transfers page-faults are not supported.

There are some side effects of using this tweak. Since the allocation and fill operations that are too slow, using this memory as a dynamic buffer isn't viable. To use it as a static buffer it's necessary preload all data needed during execution. But due to this memory being also allocated on the device memory the costs of using it are pretty high.

#### 3.4.2.4 Local Memory

Local memory (called shared memory in the CUDA world) is typically used to share data between work-items in the same work-group. This can also be accomplished using global memory but at a much lower speed, even in devices with great global memory bandwidth available because local memory is located on-chip [6], which makes it much faster (up to 100x under ideal conditions).

In this case local memory is not used to share data in a work-group, but to avoid using global memory to store work-item's own private data (automatic variables). Each work-item have a private memory space in which can hold its private data, this memory is very fast (at least as fast as local memory) but unfortunately very small, global memory is used to mask the limited space of private memory (just like swap). When a kernel needs too much private data some of it is allocated in global memory, which will probably hurt performance. To avoid such penalty the next best thing is used, local memory, which can be as fast as private memory under ideal conditions. Such conditions are bank-conflict free memory accesses, which can be obtained using the right offsets for each work-item.

This way of using local memory can be viewed as a form of caching, because the data is brought to a closer and faster memory instead of storing and loading it from global memory (which can be cached in certain devices).

#### 3.4.2.5 Prefetch

Prefetch in this context refers to fetch data before its real need, so that when becomes necessary there's already available. Since it's possible to upload data to the device in the background while executing kernels, this enhancement uploads the input data that will be used by the next work unit while the current one is being processed.

In theory this tweak would allow to hide the input data upload time completely from all work units except the first one. Completely hidden because the work unit's execution time is bigger than the fetch time. If the execution time became lower than the fetch one then the upload wont be completely hidden, only partially, corresponding to the execution time.

#### 3.4.2.6 Object Identification

There are situations, specially with big datasets, where the geometry model has so much details or is so big that some kind of identification would be very useful to distinguish different parts. This feature tries to do that using color to tell apart geometry regions with a different identification.

This implementation doesn't perform any kind of identification *per se*, it only differentiates geometry extents with different colors and transparency. This identification is accomplished using an extra dataset that has a tag for every voxel formed from the

[6] explain on-chip e off-chip

value's dataset. The two datasets must have identical sizes, the value's dataset must be bigger by 1 unit in each dimension, because for every voxel must exist a corresponding identification tag. And for every different tag there's a matching entry in the color table (like the vertices or triangles table). These entries are made of 4 channels, each with 8-bit integers, corresponding to the RGBA color model.

Another way of identifying particular zones, without producing the entire model, is clipping (discarding some geometry). Without this feature the output geometry would always produce the entire input dataset's model. This way, besides tagging different portions of the geometry is also possible to discard some of them using a particular tag, the *background tag*.

This feature tries to be as efficient as possible, without introducing too many changes and costs. There's an increase in memory consumption on the input, as an extra dataset, and also on the output, as the extra color data, more $1/4$ and $1/8$ respectively (producing the entire model). Also the classification an generation kernels saw an increase in execution time, attributed almost exclusively to the extra memory accesses, less than 10% (again, producing the entire model). Now classification kernel must have into account not only voxel's combination but also the its identification tag (in this particular case, the background tag) to know what should discard.

**Kernel 4** mcClassification pseudo-code with object identification

**input:** values, isoValue, valuesID
**output:** occupied[], vertices[]

$sizes \leftarrow getSizes()$
$coordinates \leftarrow getCoordinates()$
$outputPosition \leftarrow getPosition(coodinates, sizes)$

**if** $getVoxelID(valuesID, coordinates)$ **then**
  $corners[8] \leftarrow getVoxelCornersValue(values, coordinates)$
  $combination \leftarrow calculateVoxelCombination(isoValue, corners)$
  $verticesCount \leftarrow getVoxelVerticesCount(combination)$

  $vertices[outputPosition] \leftarrow verticesCount$
  **if** $verticesCount$ **then**
    $occupied[outputPosition] \leftarrow OCCUPIED\_VOXEL$
  **else**
    $occupied[outputPosition] \leftarrow EMPTY\_VOXEL$
  **end if**
**else**
  $vertices[outputPosition] \leftarrow 0$
  $occupied[outputPosition] \leftarrow EMPTY\_VOXEL$
**end if**

In addition to the geometry and shading buffers already produced by the generation kernel, a color buffer must also be produced. Each vertex has a color, associated with its

voxel identification tag.

**Kernel 5** mcGeneration pseudo-code with object identification

**input:** values, isoValue, scanned[], compacted[], valuesID
**output:** geometry[], shading[], color[]

$rawPosition \leftarrow getPosition()$
$position \leftarrow compacted[rawPosition]$
$coordinates \leftarrow getCoordinatesFromPosition(position, values.sizes)$

$corners[8] \leftarrow getVoxelCornersValues(values, coordinates)$
$combination \leftarrow calculateVoxelCombination(isoValue, corners)$
$vertices[12] \leftarrow calculateVoxelVertices(isovalue, corners)$

$verticesCount \leftarrow getVoxelVerticesCount(combination)$
$voxelEdges[] \leftarrow getVoxelEdges(combination)$
$verticesColor \leftarrow getVoxelID(valuesID, coordinates)$
$voxelOutputPosition \leftarrow scanned[position]$

**for** $v \leftarrow 0 : verticesCount$ **do**
  $edge \leftarrow voxelEdges[v]$
  $outputPosition \leftarrow voxelOutputPosition + v$

  $geometry[outputPosition] \leftarrow vertices[edge]$
  $shading[outputPosition] \leftarrow calculateNormal(dataset, vertices[edge])$
  $color[outputPosition] \leftarrow verticesColor$
**end for**

# 4

# Results Analysis

This chapter provides results achieved by the implementation described in chapter 3 and further enhancements described in subsection 3.4.2. These results are split in the **single device** (4.1) and **multiple devices** (4.2) sections. Enhancements used in a single device execution can also be applied to a multiple devices execution.

All units used in the next tables and figures are specified in the table 4.1, allowing to present uncluttered and uniform results while saving precious space in some long tables. Uniformity is specially useful when comparing results directly, assuming they belong to the same category.

The datasets used to preform the tests in this chapter are detailed in the table 4.2 and shown in the chapter 7 of the appendix. The column **output triangles** reflects only the geometry, but the **output size** column already accounts for all generated data (geometry plus shading).

The hardware used run the tests in this chapter is detailed in table 4.3 and 4.4.

The timing results presented in this chapter were obtained repeating each test 3 times and averaging the total time, after a warm up cycle. Timings concerning overall executions are measured by host timers, with a microsecond resolution, and rounded to milliseconds. Whereas kernel timings are measured by the device, which reports nanosecond resolution timestamps, and rounded to 1/10 of the millisecond. Device timings were retrieved using OpenCL profiling API and accounting only the time spent executing, the enqueue and submit times were considered host time. Sometimes, due to some limitation (typically memory), there's no available result.

| category | unit |
|---|---|
| time | millisecond (ms) |
| size | mebibyte (MiB) |
| speedup | how many times (x) |
| usage | percentage (%) |
| enhance | percentage (%) |
| cost | percentage (%) |

**Table 4.1:** Units used in the test results.

| input | | | | output | |
|---|---|---|---|---|---|
| dataset | samples | size | isovalue | triangles | size |
| skull | 256 * 256 * 256 | 64 | 50 | 1.528.872 | 139 |
| engine | 256 * 256 * 128 | 32 | 80 | 593.620 | 54 |
| aneurysm | 256 * 256 * 256 | 64 | 50 | 251.012 | 22 |
| sphere | 256 * 256 * 256 | 64 | - | 377.000 | 34 |
| 8 skulls | 512 * 512 * 512 | 512 | 70 | 6.419.964 | 578 |
| unknownHISO | 1024 * 1024 * 476 | 1904 | 45 | 6.260.182 | 573 |
| unknownLISO | 1024 * 1024 * 476 | 1904 | 30 | 16.097.484 | 1473 |

**Table 4.2:** Datasets used to perform the tests.

| details | System 1 | System 2 |
|---|---|---|
| operating system | Ubuntu 10.04 | |
| processor | Xeon E5506 (4x 2.13Ghz) | |
| memory | 12GB DDR3 | |
| opencl devices | Quadro FX3800<br>2x Tesla C1060 | Quadro FX3800<br>Tesla C2050 |
| device drivers | 285.05.09 | 285.05.15 |

**Table 4.3:** Systems used to preform the tests.

| details | Quadro FX3800 | Tesla C1060 | Tesla C2050 |
|---|---|---|---|
| compute units | 24 | 30 | 48 |
| processing elements | 192 | 240 | 448 |
| processor clock(mhz) | 1204 | 1296 | 1147 |
| memory interface(bits) | 256 | 512 | 384 |
| memory speed(mhz) | 1600 | 1600 | 3000 |
| memory bandwidth(GB/s) | 51,2 | 102,4 | 144 |
| global memory size(GiB) | 1 | 4 | 3 |
| opencl profile | 1.0 | 1.0 | 1.1 |
| compute capability | 1.3 | 1.3 | 2.0 |
| power consumption(watts) | 108 | 188 | 238 |

**Table 4.4:** Devices used to preform the tests.

## 4.1 Single Device

These results were achieved executing the algorithm just like it's described in chapter 3, without any of the enhancements described in subsection 3.4.2. Such results are used as the base reference for comparing with other modified versions of the algorithm, either by performance or usability enhancements.

The table 4.5 and figures 4.1 and 4.2 show overall execution timings of all datasets running on the CPU and on OpenCL devices. Besides timings, there's also the achieved speedup between CPU and OpenCL versions. Although figures 4.1 and 4.2 contain the same kind of information they are split to provide a good readability.

| input | CPU | Quadro FX3800 | | Tesla C1060 | | Tesla C2050 | |
|---|---|---|---|---|---|---|---|
| dataset | time | time | speedup | time | speedup | time | speedup |
| skull | 1248 | 151 | 8,3x | 105 | 11,9x | 85 | 14,7x |
| engine | 526 | 68 | 7,7x | 49 | 10,7x | 39 | 13,5x |
| aneurysm | 455 | 111 | 4,1x | 84 | 5,4x | 68 | 6,7x |
| sphere | 546 | 115 | 4,7x | 85 | 6,4x | 68 | 8,0x |
| 8 skulls | 6319 | 884 | 7,1x | 597 | 10,6x | 511 | 12,4x |
| unknownHISO | 12956 | 2824 | 4,6x | 2009 | 6,4x | 1678 | 7,7x |
| unknownLISO | 18862 | - | - | 2166 | 8,7x | 1776 | 10,6x |

**Table 4.5:** Algorithm performance running on different devices and with different datasets.

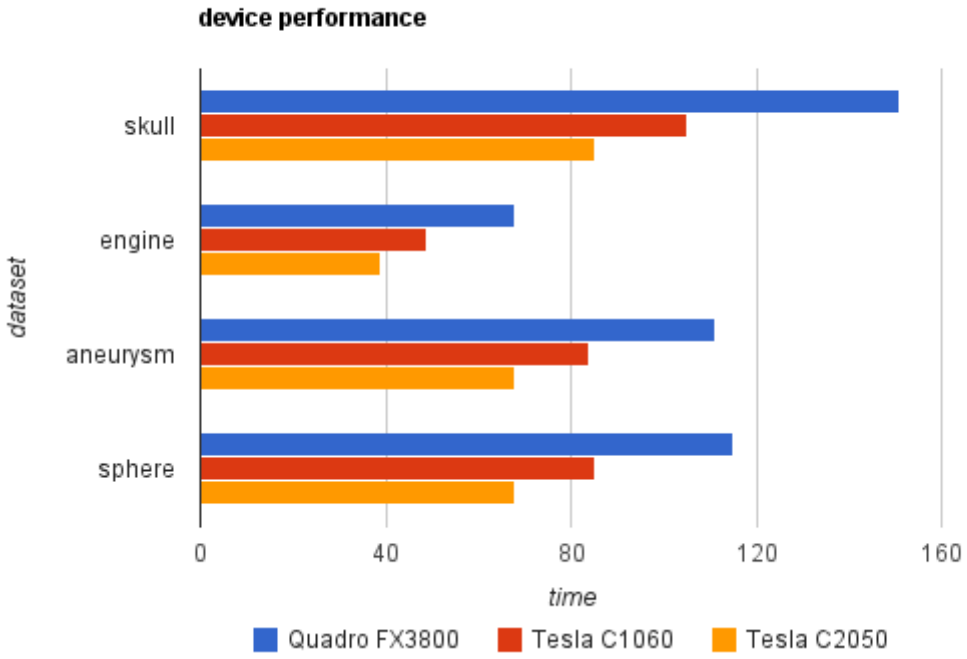



**Figure 4.1:** Algorithm performance running on different devices and with different (small) datasets.

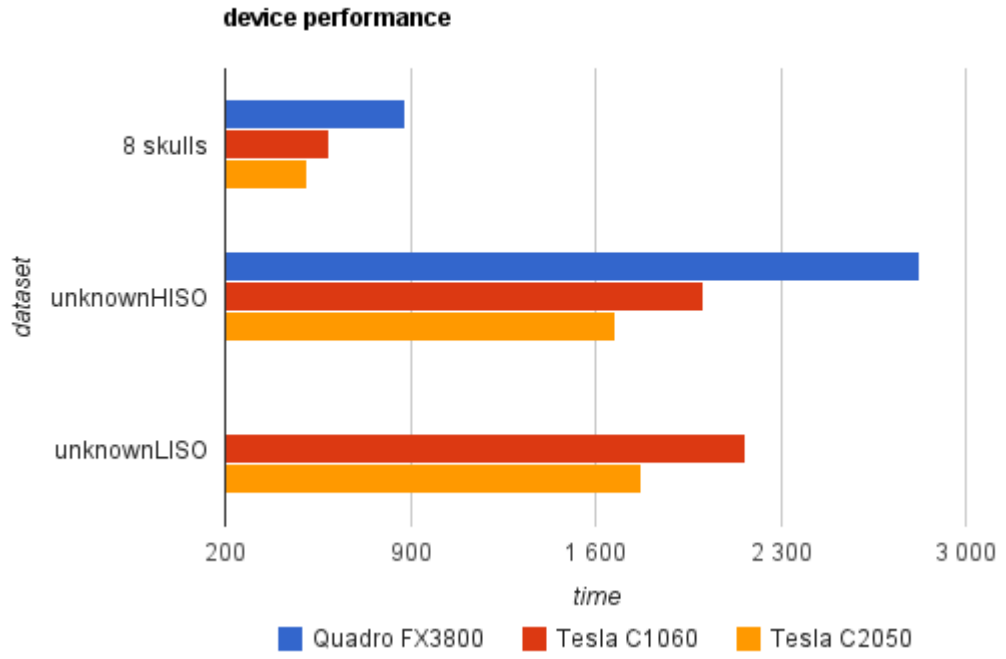



**Figure 4.2:** Algorithm performance running on different devices and with different (big) datasets.

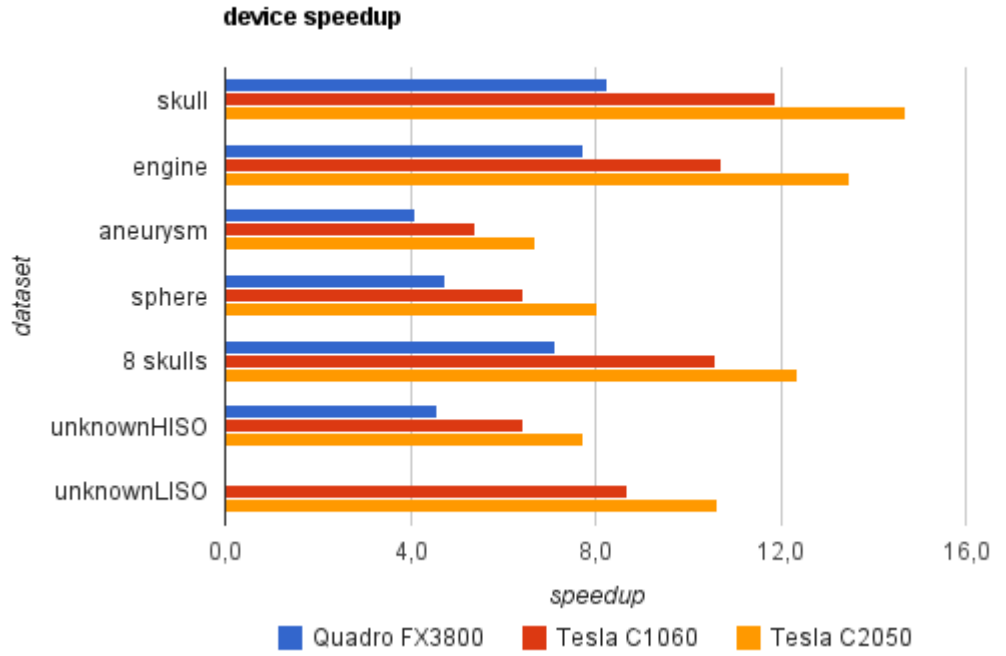



**Figure 4.3:** Algorithm speedup running on different devices and with different datasets.

One conclusion that can be quickly spotted is the CPU performance being more affected by the output data size than the input data size. While on the OpenCL side the input data size weights considerably more. This difference can be easily saw on tests which used the datasets **skull**, **engine**, **aneurysm** and **sphere**. The explanation for this difference relies on the different approach followed by the CPU (serial) and the OpenCL implementations (parallel). As previously said, the OpenCL implementation copies the input data to its memory and preforms several steps to classify and discard non-relevant data, which accounts for most of the execution time. On the other hand, the CPU implementation doesn't have to preform most intermediate classification steps, which shifts must execution time weight to the generation process.

Another easy trend to spot is the speedup being bigger on datasets which produce more output data. This is true either between CPU and OpenCL devices as between powerful and less-powerful OpenCL devices. Because the generation phase has more computation work (processor-bound) than the other steps (or phases, for that matter), the more powerful devices are able to have higher speedups.

The tables 4.6, 4.7 and 4.8 and figures 4.5, 4.6 and 4.7 provide a good insight of the algorithm execution. While tables 4.6, 4.7 and 4.8 show processing times for each individual component (including total), figures 4.5, 4.6 and 4.7 present a good graphical representation of them. Also, figure 4.4 present a clean representation of the device usage among different datasets.

All columns from tables 4.6, 4.7 and 4.8 contain algorithm components timings, exception made to **usage**. Usage represents the percentage of the execution time spent in the device. The remaining is spent on the host, typically enqueueing commands to *command queues* and then submitting them to the *devices*.

| input | Quadro FX3800 | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| dataset | fetch | class. | scan | comp. | scan | gen. | sum | *usage* |
| skull | 18,3 | 30,5 | 16,9 | 3,6 | 16,9 | 42,6 | 128,7 | 85% |
| engine | 8,3 | 15,1 | 8,5 | 1,7 | 8,1 | 15,7 | 57,5 | 84% |
| aneurysm | 16,8 | 30,5 | 16,9 | 3,1 | 16,3 | 7,3 | 90,9 | 82% |
| sphere | 18,4 | 30,5 | 16,9 | 3,2 | 16,3 | 9,4 | 94,7 | 82% |
| 8 skulls | 147,7 | 221,9 | 124,9 | 25,6 | 124,8 | 191,0 | 836,0 | 95% |
| unknownHISO | 732,9 | 682,1 | 465,8 | 94,1 | 465,8 | 200,3 | 2641,1 | 94% |

**Table 4.6:** Algorithm components performance running on a Quadro FX3800 device with different datasets.

The first thing that catches the attention is the **usage** results (or the **host** times, which represent the same thing presented in different form), synonymous of efficiency. While on smaller datasets this efficiency is similar among all devices, around 70-80%, it increases when executing bigger datasets. The worst device in terms of efficiency is the **Tesla C2050** and although improving on bigger datasets it can't be as efficient as the others. Further investigation revealed that although the **Tesla C2050** has bigger host timings (enqueueing

| input | Tesla C1060 | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| dataset | fetch | class. | scan | comp. | scan | gen. | sum | *usage* |
| skull | 16,8 | 15,7 | 12,0 | 1,8 | 12,0 | 21,7 | 80,0 | 76% |
| engine | 8,4 | 7,8 | 6,1 | 0,8 | 5,8 | 7,7 | 36,6 | 75% |
| aneurysm | 16,7 | 15,5 | 12,0 | 1,6 | 11,6 | 3,9 | 61,4 | 73% |
| sphere | 16,8 | 15,5 | 12,0 | 1,6 | 11,6 | 4,9 | 62,5 | 74% |
| 8 skulls | 137,2 | 125,1 | 86,6 | 13,1 | 86,6 | 97,1 | 545,8 | 91% |
| unknownHISO | 511,2 | 663,2 | 315,7 | 45,3 | 315,7 | 100,6 | 1951,7 | 97% |
| unknownLISO | 509,6 | 662,9 | 315,7 | 46,2 | 315,7 | 256,7 | 2106,9 | 97% |

**Table 4.7:** Algorithm components performance running on a Tesla C1060 device with different datasets.

| input | Tesla C2050 | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| dataset | fetch | class. | scan | comp. | scan | gen. | sum | *usage* |
| skull | 18,1 | 8,5 | 9,4 | 1,8 | 9,4 | 14,6 | 61,8 | 73% |
| engine | 9,0 | 4,3 | 4,7 | 0,9 | 4,5 | 4,8 | 28,1 | 72% |
| aneurysm | 17,9 | 8,5 | 9,3 | 1,7 | 9,0 | 2,9 | 49,5 | 73% |
| sphere | 16,7 | 8,5 | 9,3 | 1,7 | 9,0 | 3,0 | 48,3 | 71% |
| 8 skulls | 149,9 | 70,2 | 69,6 | 13,6 | 69,7 | 59,6 | 432,6 | 85% |
| unknownHISO | 516,8 | 318,4 | 255,3 | 49,9 | 255,3 | 69,0 | 1464,7 | 87% |
| unknownLISO | 515,0 | 318,4 | 255,2 | 50,0 | 255,3 | 177,5 | 1571,4 | 88% |

**Table 4.8:** Algorithm components performance running on a Tesla C2050 device with different datasets.

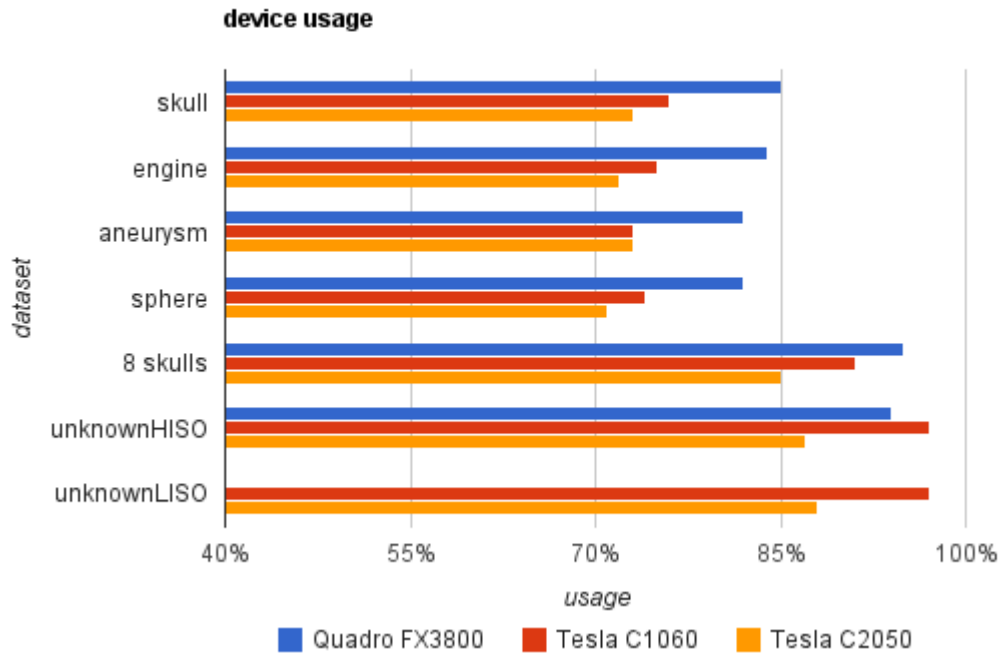



**Figure 4.4:** Algorithm device usage on different devices and with different datasets.

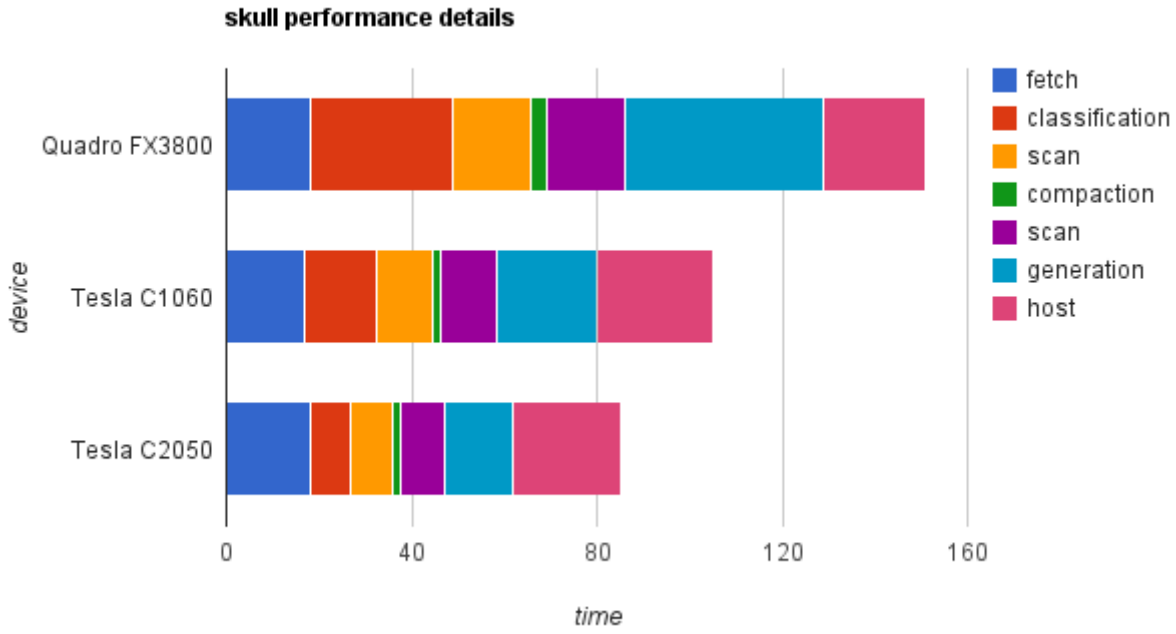



**Figure 4.5:** Algorithm components performance on different devices using the *skull* dataset.

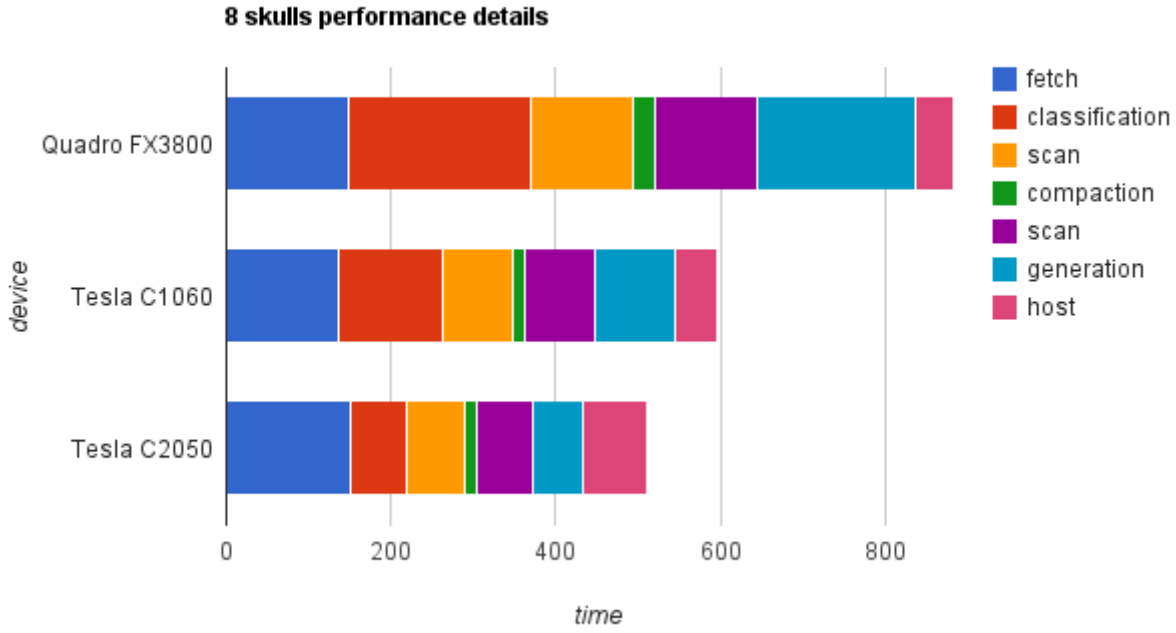



**Figure 4.6:** Algorithm's components performance on different devices using the *8 skulls* dataset.

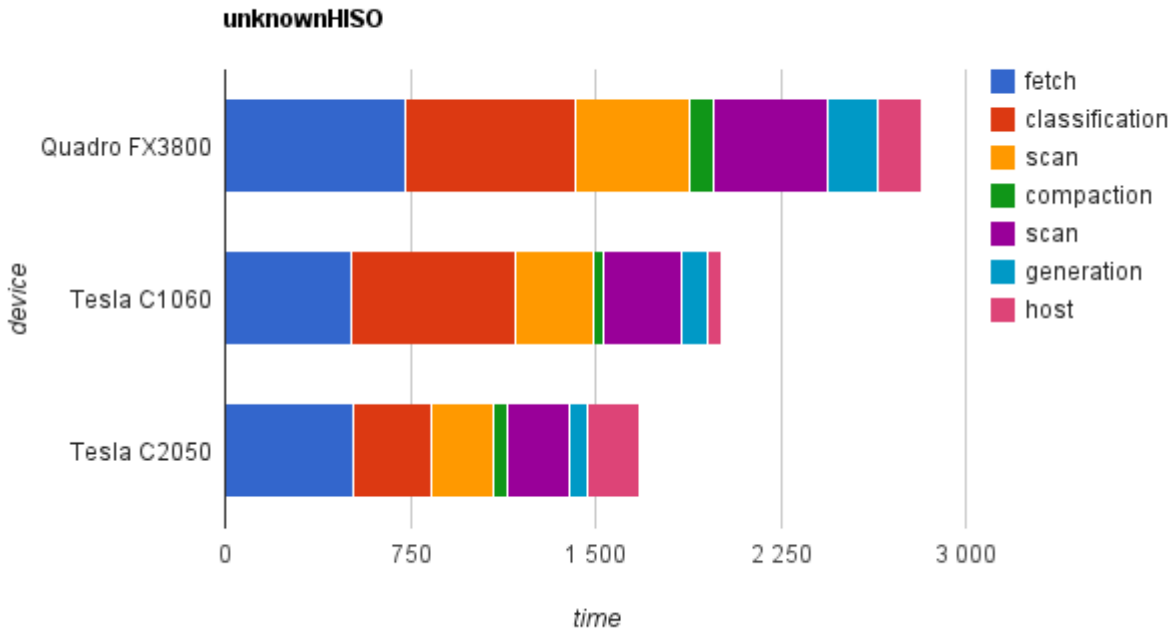



**Figure 4.7:** Algorithm's components performance on different devices using the *unknownHISO* dataset.

and submitting), using OpenCL profile 1.1 these only get event worst.

The compaction component being the simplest is also the lightest, accounting for only a small fraction of the total time. And the stranger results are there, where the **Tesla C1060** outperforms the **Tesla C2050**.

### 4.1.1 Performance Enhancements

In the next subsections 4.1.1.1, 4.1.1.2 and 4.1.1.3 only three datasets are used, since these represent the all use cases. These are, bigger output than input data size, similar input and output data sizes and smaller output that input data, respectively. Each subsection presents results applying performance enhancements individually, explained in subsection 3.4.2. The goal is to show if there's any performance improvement or not, and if there is how much it is. All **enhance** columns present results which compare them with the results from table 4.5.

#### 4.1.1.1 Pinned Memory

The tests performed in this subsection uses the enhancement detailed in 3.4.2.3. Improvements are expected only on the fetch component, presented in table 4.10.

All devices benefit from this enhancement, although some more than others. The **Tesla C2050** can achieve 6GB/s of transfer bandwidth.

| input | Quadro FX3800 | | Tesla C1060 | | Tesla C2050 | |
|---|---|---|---|---|---|---|
| dataset | time | enhance | time | enhance | time | enhance |
| skull | 145 | 4% | 99 | 6% | 76 | 12% |
| 8 skulls | - | - | 563 | 6% | 453 | 13% |
| unknownHISO | - | - | 1883 | 7% | 1494 | 12% |

**Table 4.9:** Algorithm performance using pinned memory.

| input | Quadro FX3800 | | Tesla C1060 | | Tesla C2050 | |
|---|---|---|---|---|---|---|
| dataset | time | enhance | time | enhance | time | enhance |
| skull | 15,4 | 19% | 14,7 | 14% | 13,1 | 38% |
| 8 skulls | - | - | 108,8 | 26% | 98,4 | 52% |
| unknownHISO | - | - | 394,5 | 30% | 368,5 | 40% |

**Table 4.10:** Fetch component performance using pinned memory.

#### 4.1.1.2 Local Memory

The tests performed in this subsection uses the enhancement detailed in 3.4.2.4. Improvements are expected only on the generation component, presented in table 4.12.

| input | Quadro FX3800 | | Tesla C1060 | | Tesla C2050 | |
|---|---|---|---|---|---|---|
| dataset | time | enhance | time | enhance | time | enhance |
| skull | 149 | 1% | 114 | -8% | 78 | 9% |
| 8 skulls | 853 | 4% | 640 | -7% | 480 | 6% |
| unknownHISO | 2740 | 3% | 2095 | -4% | 1621 | 4% |

**Table 4.11:** Algorithm performance using local memory.

Although **Quadro FX3800** has seen its performance improved it can't be compared to the **Tesla C2050** improvement. The **Tesla C1060** has an inexplicable performance hit.

#### 4.1.1.3 Prefetch

The tests performed in this subsection uses the enhancement detailed in 3.4.2.5. Improvements are expected only on the usage, presented in table 4.14. Here usage can be higher than 100% because components aren't executed serially, like before, there's parallelism between fetch and execution. Figures 4.8 and 4.9 show the difference between using and not using such enhancement.

While on **Tesla C2050** the improvement was almost perfect, on the other devices there isn't any kind of improvement. On the **Tesla C1060** it even hurt performance due to the lower bandwidth achieved while fetching the data.

Figure 4.10 shows what happens when using OpenCL profile **1.0**. The **Tesla C2050** is the only device supporting such profile that's why it's the only one reflecting improvements. Using Tesla C2050 with profile 1.0 also yields results similar to figure 4.10.

| input | Quadro FX3800 | | Tesla C1060 | | Tesla C2050 | |
|---|---|---|---|---|---|---|
| dataset | time | enhance | time | enhance | time | enhance |
| skull | 41,1 | 4% | 30,9 | -30% | 8,6 | 70% |
| 8 skulls | 168,6 | 13% | 126,2 | -23% | 34,0 | 75% |
| unknownHISO | 187,7 | 7% | 137,9 | -27% | 38,2 | 81% |

**Table 4.12:** Generation component performance using local memory.

| input | Quadro FX3800 | | Tesla C1060 | | Tesla C2050 | |
|---|---|---|---|---|---|---|
| dataset | time | enhance | time | enhance | time | enhance |
| skull | 139 | 9% | 111 | -5% | 79 | 8% |
| 8 skulls | 865 | 2% | 657 | -9% | 445 | 15% |
| unknownHISO | 2960 | -5% | 2176 | -8% | 1360 | 23% |

**Table 4.13:** Algorithm performance using prefetch.

| input | Quadro FX3800 | | Tesla C1060 | | Tesla C2050 | |
|---|---|---|---|---|---|---|
| dataset | usage | enhance | usage | enhance | usage | enhance |
| skull | 85 | 1% | 78 | 3% | 80 | 10% |
| 8 skulls | 95 | 0% | 92 | 1% | 97 | 15% |
| unknownHISO | 94 | -1% | 98 | 0% | 108 | 24% |

**Table 4.14:** Device usage using prefetch.

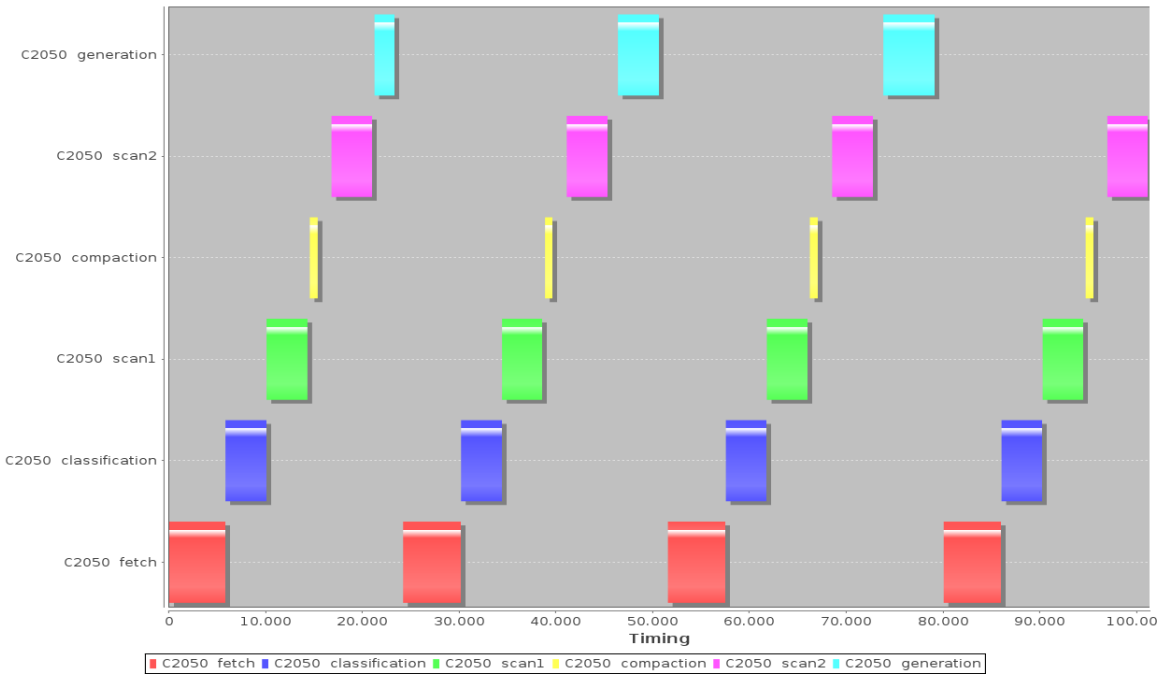



**Figure 4.8:** Without prefetch.

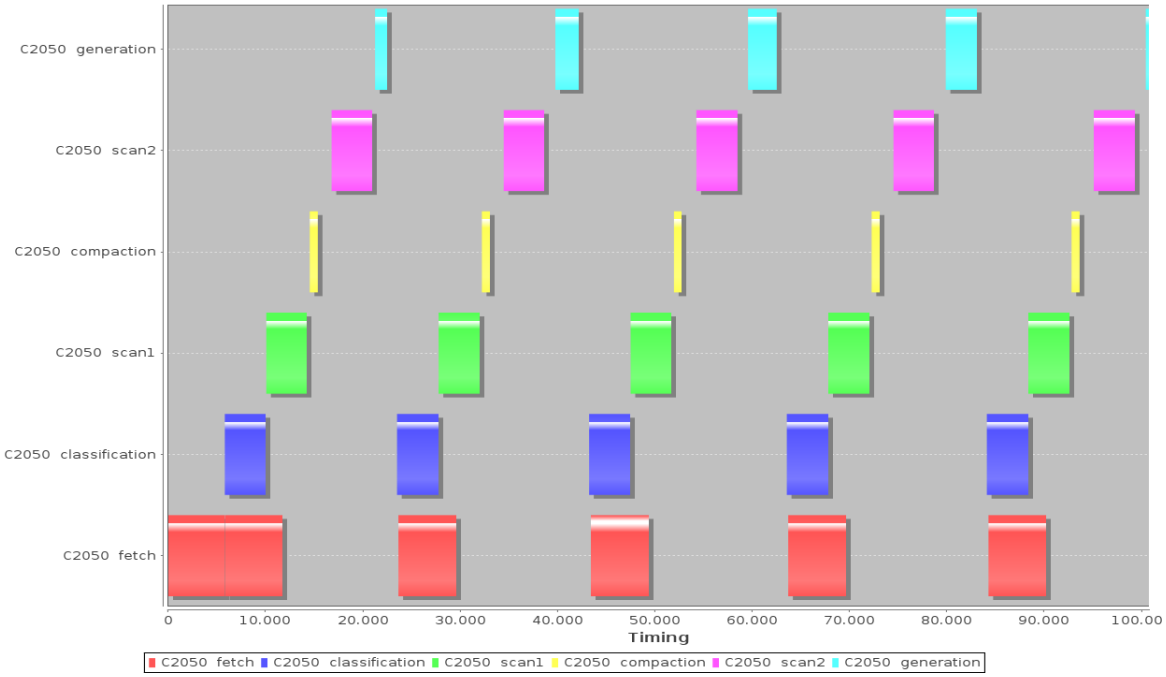



**Figure 4.9:** With Prefetch.

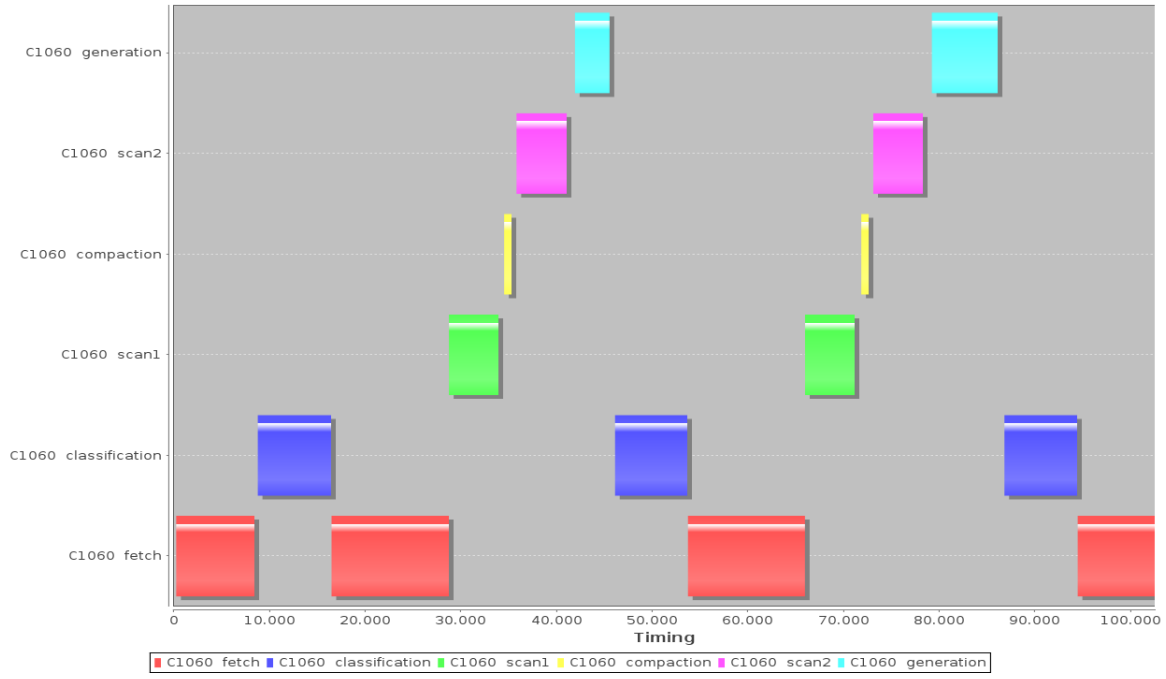



**Figure 4.10:** Prefetch on OpenCL 1.0.

#### 4.1.1.4 Combined Enhancements

The tests performed in this subsection uses the previous enhancements combined. Since not all devices improve performance when using some enhancements, only the ones that improve performance are turned on. In some cases it's also possible that some enhancement has to be turned off due to some device limitation, like memory. That's the case for the **Quadro FX3800** on **8 skulls** and **unknownHISO** datasets and the Tesla C2050 on the **unknownHISO** dataset.

| input | Quadro FX3800 | | Tesla C1060 | | Tesla C2050 | |
|---|---|---|---|---|---|---|
| dataset | time | enhance | time | enhance | time | enhance |
| skull | 144 | 5% | 99 | 6% | 60 | 42% |
| engine | 65 | 5% | 46 | 7% | 28 | 39% |
| aneurysm | 107 | 4% | 78 | 8% | 47 | 45% |
| sphere | 108 | 6% | 79 | 8% | 48 | 42% |
| 8 skulls | 853 | 4% | 563 | 6% | 337 | 52% |
| unknownHISO | 2740 | 3% | 1883 | 7% | 1115 | 50% |
| unknownLISO | - | - | 2042 | 6% | 1445 | 23% |

**Table 4.15:** Algorithm enhanced performance running on different devices and with different datasets.

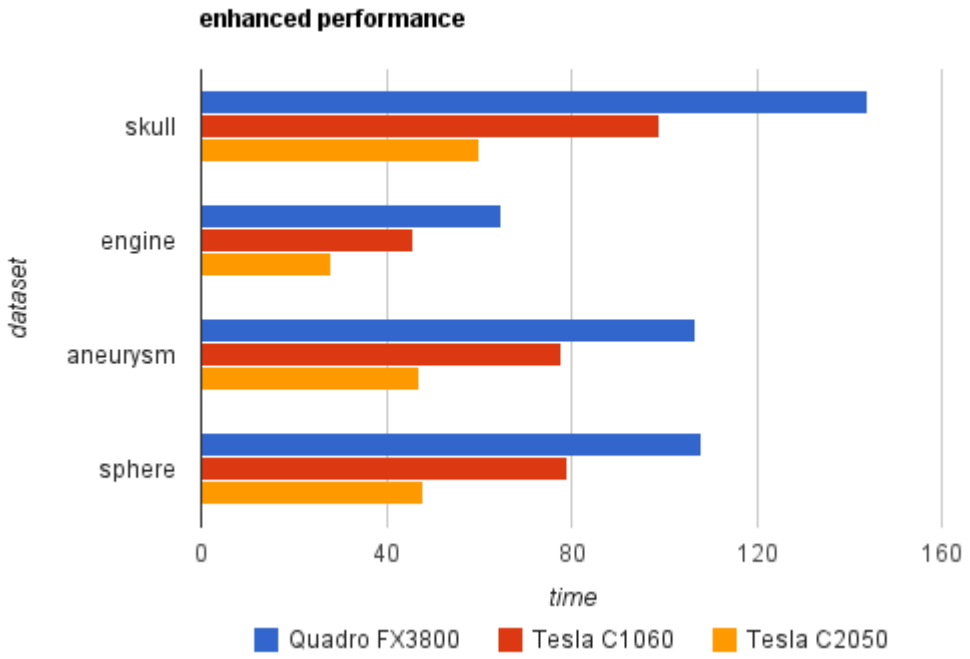



**Figure 4.11:** Algorithm components time on different devices using skull dataset.

As we can see, all devices have performance improvements but the **Quadro FX3800** and the **Tesla C1060** don't event reach the two digits. On the other hand, the **Tesla C2050** which was already the best device have great performance improvements, it event

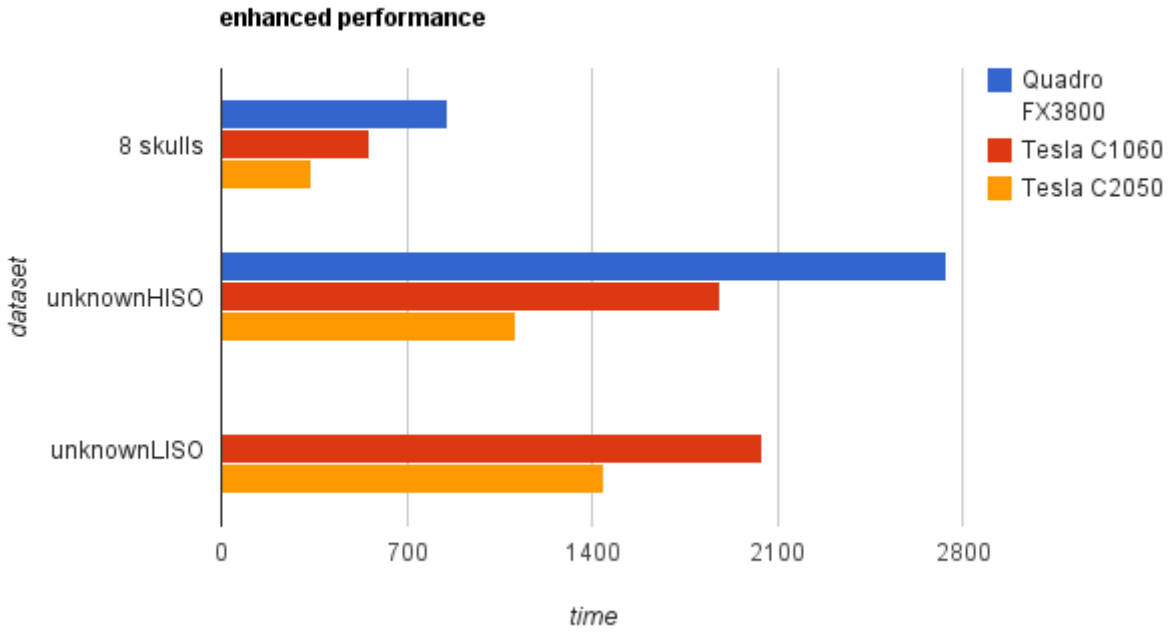



**Figure 4.12:** Algorithm components time on different devices using 8 skulls dataset.

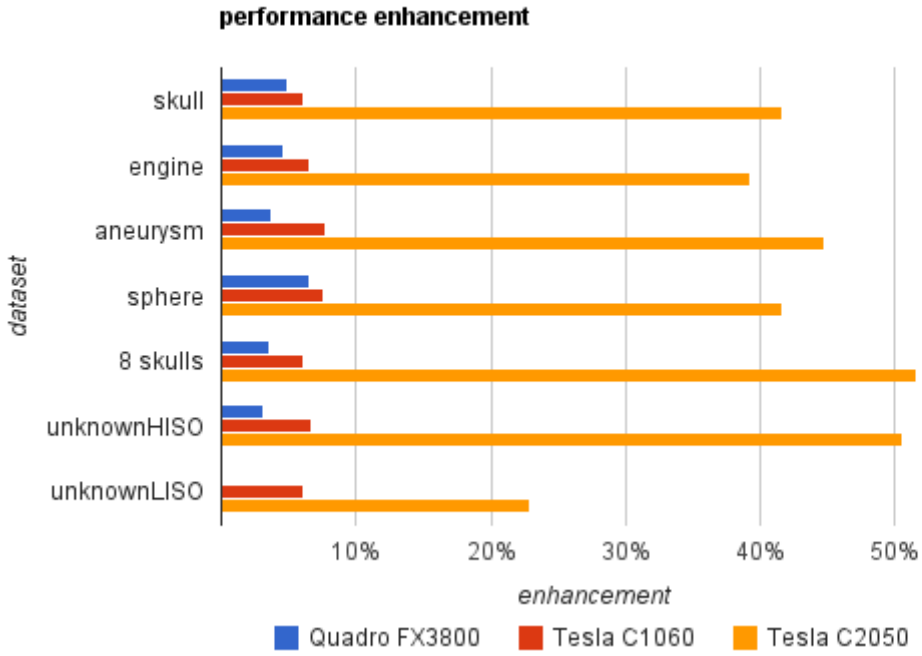



**Figure 4.13:** Algorithm components time on different devices using unknownHISO dataset.

| input | Quadro FX3800 | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| dataset | fetch | class. | scan | comp. | scan | gen. | sum | *usage* |
| skull | 15,4 | 30,4 | 16,9 | 3,6 | 16,9 | 41,1 | 124,3 | 86% |
| engine | 7,7 | 15 | 8,5 | 1,7 | 8,1 | 14,1 | 55,1 | 85% |
| aneurysm | 15,3 | 30,4 | 16,9 | 3,1 | 16,3 | 7,6 | 89,6 | 84% |
| sphere | 15,3 | 30,4 | 16,9 | 3,2 | 16,3 | 8,5 | 90,6 | 84% |
| 8 skulls | 139,2 | 221,8 | 124,9 | 25,6 | 124,8 | 168,6 | 804,9 | 94% |
| unknownHISO | 658,4 | 682,3 | 466,7 | 94,1 | 465,7 | 187,7 | 2554,9 | 93% |
| unknownLISO | - | - | - | - | - | - | - | - |

**Table 4.16:** Algorithm's components performance running on a Quadro FX3800 device with different datasets.

| input | Tesla C1060 | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| dataset | fetch | class. | scan | comp. | scan | gen. | sum | *usage* |
| skull | 14,7 | 15,5 | 12 | 1,8 | 12 | 21,7 | 77,7 | 78% |
| engine | 7,3 | 7,7 | 6,1 | 0,9 | 5,8 | 7,9 | 35,7 | 78% |
| aneurysm | 14,6 | 15,4 | 12 | 1,6 | 11,6 | 4 | 59,2 | 76% |
| sphere | 14,6 | 15,4 | 12 | 1,6 | 11,6 | 4,8 | 60 | 76% |
| 8 skulls | 108,8 | 124,8 | 86,6 | 13,1 | 86,6 | 97,7 | 517,6 | 92% |
| unknownHISO | 394,5 | 662,5 | 315,7 | 45,3 | 315,7 | 100,3 | 1834 | 97% |
| unknownLISO | 394,1 | 663 | 315,7 | 46,3 | 315,7 | 256,9 | 1991,7 | 98% |

**Table 4.17:** Algorithm's components performance running on a Tesla C1060 device with different datasets.

| input | Tesla C2050 | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| dataset | fetch | class. | scan | comp. | scan | gen. | sum | *usage* |
| skull | 13,1 | 8,5 | 9,4 | 1,8 | 9,4 | 8,6 | 50,8 | 85% |
| engine | 6,6 | 4,2 | 4,7 | 0,9 | 4,5 | 2,9 | 23,8 | 85% |
| aneurysm | 13,1 | 8,5 | 9,4 | 1,7 | 9 | 1,8 | 43,5 | 93% |
| sphere | 13,1 | 8,5 | 9,4 | 1,7 | 9 | 2 | 43,7 | 91% |
| 8 skulls | 98,2 | 70,2 | 69,6 | 13,6 | 69,6 | 34,1 | 355,3 | 105% |
| unknownHISO | 367,6 | 319 | 255,3 | 49,9 | 255,3 | 38,3 | 1285,4 | 115% |
| unknownLISO | 570,6 | 318,8 | 255,4 | 50 | 255,3 | 93,4 | 1543,5 | 107% |

**Table 4.18:** Algorithm's components performance running on a Tesla C2050 device with different datasets.

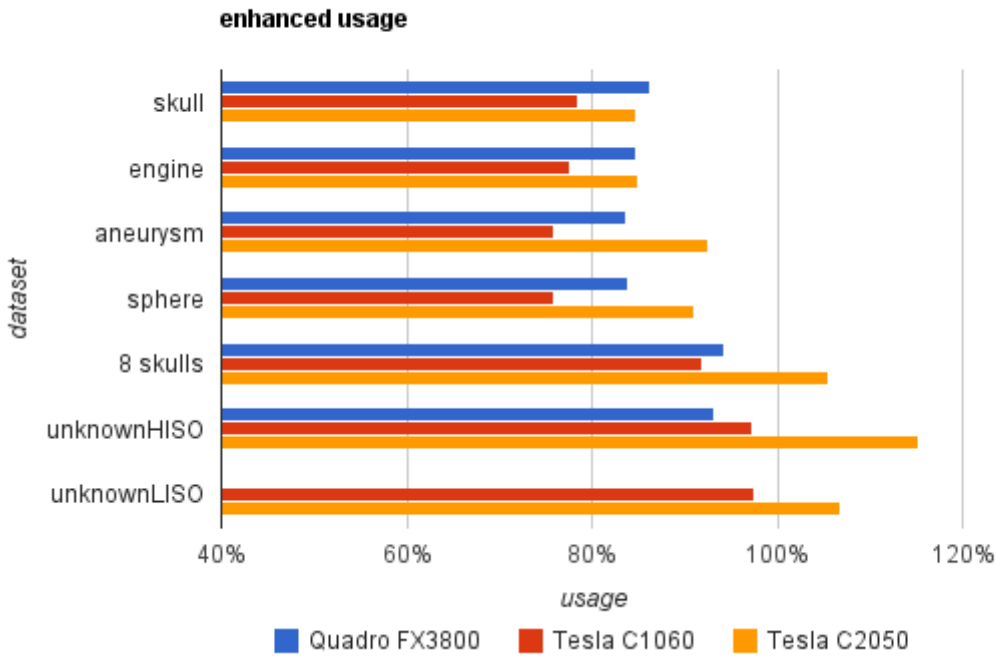



**Figure 4.14:** Device usage on different devices and with different datasets.

reaches the 50% mark.

### 4.1.2 Objects Identification

The tests performed in this subsection uses the enhancement detailed in 3.4.2.6. Costs represent the executions timings compared with the results achieved in the subsections 4.1.1.4.

| input | Quadro FX3800 | | | | Tesla C2050 | | | |
|---|---|---|---|---|---|---|---|---|
| | time | | | cost | time | | | cost |
| dataset | class. | gen. | execution | | class. | gen. | execution | |
| skull | 29,8 | 43 | 155 | 7% | 9,6 | 9,2 | 64 | 7% |
| 8 skulls | 208,1 | 176,8 | 937 | 10% | 77,1 | 36,4 | 354 | 5% |

**Table 4.19:** Algorithm performance using object identification with 2 objects.

As it's possible to see, the object identification has neglectable costs. Also, using just 2 or 65 identification tags has the same costs. One side effect of using this feature is the size grow by 1/9th, storing the extra color data.

## 4.2 Multiple Devices

These results are achieved running the algorithm just like it's described in chapter 3 (including improvements) but this time by multiple devices in parallel. Since the work is

| input | Quadro FX3800 | | | | Tesla C2050 | | | |
|---|---|---|---|---|---|---|---|---|
| | time | | | cost | time | | | cost |
| dataset | class. | gen. | execution | | class. | gen. | execution | |
| skull | 29,7 | 43,1 | 155 | 7% | 9,6 | 9,2 | 64 | 7% |
| 8 skulls | 207,4 | 177 | 938 | 10% | 77,2 | 36,4 | 354 | 5% |

**Table 4.20:** Algorithm performance using object identification with 65 objects.

already divided in smaller portions, the work units, the only thing left is to distribute them by multiple devices. In theory this modification can bring huge performance improvements (and also memory capacity) because device's resources are aggregated.

| input | Tesla C2050<br>Quadro FX3800 | | Tesla C2050<br>Tesla C2050 | |
|---|---|---|---|---|
| dataset | time | enhance | time | enhance |
| 8 skulls | 508 | -34% | 481 | 17% |
| unknownHISO | 1133 | -2% | 1607 | 17% |
| unknownLISO | - | - | 1773 | 15% |

**Table 4.21:** Algorithm's performance using multiple devices.

Unfortunately the tests reveal that using multiple devices don't bring any kind of performance improvement when using heterogeneous devices. And even when using homogeneous devices the improvements are small.

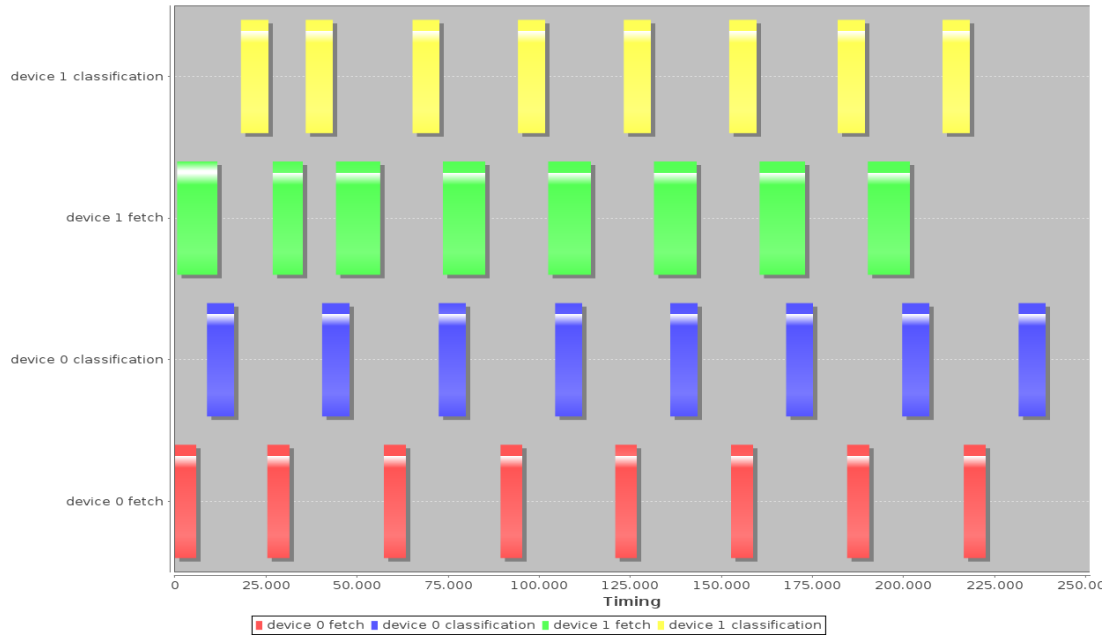



**Figure 4.15:** Simple demo test using multiple devices.

Figure 4.15 shows a simple demo test of a parallel execution with homogeneous devices, in this case **2x Tesla C1060**.

# 5

# Conclusion

The target of this work was to implement an (indirect) volume rendering solution using the Marching Cubes algorithm, which could take advantage of the OpenCL framework to speedup its execution. Such requisite was largely met, the created solution is capable of handling huge datasets and preforms much quicker than CPU versions. While much less flexible than CPU implementations can be, its potential performance gains justify the trade.

Since the Marching Cubes algorithm can be implemented in a complete parallel way, using this framework is a great way to speedup things up, tapping into the GPUs processing power (which have a theoretical peak performance much higher than the available in CPUs). Besides great processing power, GPUs also have stunning memory bandwidth a skill which is very important in this algorithm. Aside from the generation kernel, all other kernels put much more pressure on memory operations that arithmetical operations. The classification kernel is a good example of how IO-bound this algorithm is and how memory bandwidth affects its performance.

Parallel and serial versions of the Marching Cubes algorithm differ considerably. To allow an efficient parallelization, this implementation have to spent much of its execution time doing things that serial versions simply don't have. The biggest difference are in the compaction (involving the scan operation), which is completely unnecessary in a serial version. The compaction steps are used to avoid unnecessary computations, which depends much of the scan operation. Besides compaction, the execution follows a strait forward implementation of the Marching Cubes algorithm, each voxel is classified and then the relevant ones are used to generate the representation model (or isosurface).

Compared to the initial base implementation, the Marching Cubes CUDA sample

included in the NVIDIA GPGPU SDK (detailed in subsection 2.1.4), this implementation offers a much more refined solution:

- capable of handling much bigger datasets and non power of two dataset sizes;
- makes use of prefetch to hide fetch times (which account for a big portion of the execution time);
- caches binary programs to shrink initialization times;
- allows the selection of devices and the order by witch they are used;
- provides (on request) useful profiling data and/or verbose execution trace;
- provides object identification.

## 5.1 Future Work

Unfortunately there are some shortcomings in this implementation. The inability to achieve some improvements which in theory could have yielded great performance improvements. Specifically, the usage of multiple devices and the prefetch of the input data on devices using OpenCL profile 1.0. Upon some investigating and testing we can only speculate that the OpenCL driver used on ours tests have some caveats, it seems that at some point some operations are serialized in some way, even if there's more than one *command queue* or *device*.

Future work would start by deeply investigate the causes of the shortcomings in this implementation. Test and improve the rudimentary and untested load balance functionality. Investigate deeply some conclusions made from testing (running several instances of the benchmarking concurrently on the same device) that lead to believe that some devices (like the Tesla C2050 used in the tests) aren't being completely used and can be squeezed more performance out of them. Another direction that could be taken is to use data types which consume less memory space, currently the type *cl_float4* is used to store only 3 values.

## 5.2 Contributions

- An OpenCL accelerated Marching Cubes algorithm implementation capable of handling large datasets, distribute the workload across several devices (although the results achieved aren't particularly interesting) and identify different objected within a dataset;
- A layer with the intent of ease the development of OpenCL programs. This layer provides some abstraction from the low-level functions offered in OpenCL API;

- Two utilities, a viewer providing the visualization capabilities and also a benchmark to preform tests and measure performance.

# Bibliography


[DZTS08] Christopher Dyken, Gernot Ziegler, Christian Theobalt, and Hans-Peter Seidel. High-speed marching cubes using HistoPyramids. *Computer Graphics Forum*, 27(8):2028–2039, 2008.

[Elv92] T. Todd Elvins. A survey of algorithms for volume visualization. 1992.

[HSO07] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (Scan) with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*. Addison Wesley, August 2007.

[JC06] Gunnar Johansson and Hamish Carr. Accelerating marching cubes with graphics hardware. In *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, page 39, Toronto, Ontario, Canada, 2006. ACM.

[KSE04] Thomas Klein, Simon Stegmaier, and Thomas Ertl. Hardware-accelerated reconstruction of polygonal isosurface representations on unstructured grids. *IN PROCEEDINGS OF PACIFIC GRAPHICS '04*, pages 186—195, 2004.

[KW05] Peter Kipfer and Rüdiger Westermann. GPU construction and transparent rendering of Iso-Surfaces. In G. Greiner, J. Hornegger, H. Niemann, and M. Stamminger, editors, *Proceedings Vision, Modeling and Visualization 2005*, pages 241–248. IOS Press, infix, 2005.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169. ACM, 1987.

[Mac92] Paul Mackerras. A fast parallel marching-cubes implementation on the fujitsu ap1000. 1992.

[mun10] The OpenCL specification, 2010.

[Nag08] Henrik R. Nagel. GPU optimized marching cubes algorithm for handling very large, temporal datasets. Trondheim, Norway, 2008. Norwegian University of Science and Technology.

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[NY06] Timothy S. Newman and Hong Yi. A survey of the marching cubes algorithm. *Computers & Graphics*, 30(5):854–879, October 2006.

[Pas04] V. Pascucci. Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping. *IN JOINT EUROGRAPHICS - IEEE TVCG SYMPOSIUM ON VISUALIZATION*, pages 293—300, 2004.

[ZTTS06] Gernot Ziegler, Art Tevs, Christian Theobalt, and Hans-Peter Seidel. GPU point list generation through histogram pyramids. Technical Report MPI-I-2006-4-002, Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, June 2006.

# 6

# Matrix Multiplication Example

These examples were taken from `http://gpgpu-computing4.blogspot.com`. Their purpose is to show the differences between CUDA and OpenCL languages in a similar and simple program.

## 6.1 CUDA

**Listing 6.1:** CUDA kernel of matrix multiplication example.

```
1
2  #ifndef _MATRIXMUL_KERNEL_H_
3  #define _MATRIXMUL_KERNEL_H_
4
5  #include <stdio.h>
6
7  // Thread block size
8  #define BLOCK_SIZE 3
9
10 #define WA 3   // Matrix A width
11 #define HA 3   // Matrix A height
12 #define WB 3   // Matrix B width
13 #define HB WA  // Matrix B height
14 #define WC WB  // Matrix C width
15 #define HC HA  // Matrix C height
16
17 // CUDA Kernel
18 __global__ void
19 matrixMul( float* C, float* A, float* B, int wA, int wB)
20 {
```

```
21
22    // 2D Thread ID
23    int tx = threadIdx.x;
24    int ty = threadIdx.y;
25
26    // value stores the element that is
27    // computed by the thread
28    float value = 0;
29    for (int i = 0; i < wA; ++i)
30    {
31       float elementA = A[ty * wA + i];
32       float elementB = B[i * wB + tx];
33       value += elementA * elementB;
34    }
35
36    // Write the matrix to device memory each
37    // thread writes one element
38    C[ty * wA + tx] = value;
39 }
40
41 #endif // #ifndef _MATRIXMUL_KERNEL_H_
```

**Listing 6.2:** CUDA host program of matrix multiplication example.

```
1
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <math.h>
5  #include <matrixMul_kernel.cu>
6
7  // Allocates a matrix with random float entries.
8  void randomInit(float* data, int size)
9  {
10     for (int i = 0; i < size; ++i)
11         data[i] = rand() / (float)RAND_MAX;
12 }
13
14 ////////////////////////////////////////////////////////
15 // Program main
16 ////////////////////////////////////////////////////////
17
18 int
19 main(int argc, char** argv)
20 {
21
22     // set seed for rand()
23     srand(2006);
24
25     // 1. allocate host memory for matrices A and B
26     unsigned int size_A = WA * HA;
27     unsigned int mem_size_A = sizeof(float) * size_A;
```

```
28      float* h_A = (float*) malloc(mem_size_A);
29
30      unsigned int size_B = WB * HB;
31      unsigned int mem_size_B = sizeof(float) * size_B;
32      float* h_B = (float*) malloc(mem_size_B);
33
34      // 2. initialize host memory
35      randomInit(h_A, size_A);
36      randomInit(h_B, size_B);
37
38      // 3. print out A and B
39      printf("\n\nMatrix␣A\n");
40      for(int i = 0; i < size_A; i++)
41      {
42         printf("%f␣", h_A[i]);
43         if(((i + 1) % WA) == 0)
44            printf("\n");
45      }
46
47      printf("\n\nMatrix␣B\n");
48      for(int i = 0; i < size_B; i++)
49      {
50         printf("%f␣", h_B[i]);
51         if(((i + 1) % WB) == 0)
52            printf("\n");
53      }
54
55      // 8. allocate device memory
56      float* d_A;
57      float* d_B;
58      cudaMalloc((void**) &d_A, mem_size_A);
59      cudaMalloc((void**) &d_B, mem_size_B);
60
61      // 9. copy host memory to device
62      cudaMemcpy(d_A, h_A, mem_size_A,
63                 cudaMemcpyHostToDevice);
64      cudaMemcpy(d_B, h_B, mem_size_B,
65                 cudaMemcpyHostToDevice);
66
67
68      // 4. allocate host memory for the result C
69      unsigned int size_C = WC * HC;
70      unsigned int mem_size_C = sizeof(float) * size_C;
71      float* h_C = (float*) malloc(mem_size_C);
72
73      // 10. allocate device memory for the result
74      float* d_C;
75      cudaMalloc((void**) &d_C, mem_size_C);
76
77      // 5. perform the calculation
```

```
78      //    setup execution parameters
79      dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
80      dim3 grid(WC / threads.x, HC / threads.y);
81
82      //   execute the kernel
83      matrixMul<<< grid, threads >>>(d_C, d_A,
84                                     d_B, WA, WB);
85
86      // 11. copy result from device to host
87      cudaMemcpy(h_C, d_C, mem_size_C,
88                 cudaMemcpyDeviceToHost);
89
90      // 6. print out the results
91      printf("\n\nMatrix␣C␣(Results)\n");
92      for(int i = 0; i < size_C; i++)
93      {
94         printf("%f␣", h_C[i]);
95         if(((i + 1) % WC) == 0)
96            printf("\n");
97      }
98      printf("\n");
99
100     // 7. clean up memory
101     free(h_A);
102     free(h_B);
103     free(h_C);
104     cudaFree(d_A);
105     cudaFree(d_B);
106     cudaFree(d_C);
107
108 }
```

## 6.2 OpenCL

**Listing 6.3:** OpenCL kernel of matrix multiplication example.

```
1
2  // kernel.cl
3  // Multiply two matrices A * B = C
4  // Device code.
5
6
7  // OpenCL Kernel
8  __kernel void
9  matrixMul(__global float* C,
10           __global float* A,
11           __global float* B,
12           int wA, int wB)
13 {
14
15    // 2D Thread ID
16    int tx = get_local_id(0);
17    int ty = get_local_id(1);
18
19    // value stores the element
20    // that is computed by the thread
21    float value = 0;
22    for (int k = 0; k < wA; ++k)
23    {
24       float elementA = A[ty * wA + k];
25       float elementB = B[k * wB + tx];
26       value += elementA * elementB;
27    }
28
29    // Write the matrix to device memory each
30    // thread writes one element
31    C[ty * wA + tx] = value;
32 }
```

**Listing 6.4:** OpenCL host program of matrix multiplication example.

```
1
2  // Multiply two matrices A * B = C
3
4  #include <stdlib.h>
5  #include <stdio.h>
6  #include <math.h>
7  #include <oclUtils.h>
8
9  #define WA 3
10 #define HA 3
11 #define WB 3
```

```
12 #define HB 3
13 #define WC 3
14 #define HC 3
15
16 // Allocates a matrix with random float entries.
17 void randomInit(float* data, int size)
18 {
19    for (int i = 0; i < size; ++i)
20    data[i] = rand() / (float)RAND_MAX;
21 }
22
23 ////////////////////////////////////////////////////////
24 // Program main
25 ////////////////////////////////////////////////////////
26
27 int
28 main(int argc, char** argv)
29 {
30
31    // set seed for rand()
32    srand(2006);
33
34    // 1. allocate host memory for matrices A and B
35    unsigned int size_A = WA * HA;
36    unsigned int mem_size_A = sizeof(float) * size_A;
37    float* h_A = (float*) malloc(mem_size_A);
38
39    unsigned int size_B = WB * HB;
40    unsigned int mem_size_B = sizeof(float) * size_B;
41    float* h_B = (float*) malloc(mem_size_B);
42
43    // 2. initialize host memory
44    randomInit(h_A, size_A);
45    randomInit(h_B, size_B);
46
47    // 3. print out A and B
48    printf("\n\nMatrix␣A\n");
49    for(int i = 0; i < size_A; i++)
50    {
51       printf("%f␣", h_A[i]);
52       if(((i + 1) % WA) == 0)
53       printf("\n");
54    }
55
56    printf("\n\nMatrix␣B\n");
57    for(int i = 0; i < size_B; i++)
58    {
59       printf("%f␣", h_B[i]);
60       if(((i + 1) % WB) == 0)
61       printf("\n");
```

```
62    }
63
64    // 4. allocate host memory for the result C
65    unsigned int size_C = WC * HC;
66    unsigned int mem_size_C = sizeof(float) * size_C;
67    float* h_C = (float*) malloc(mem_size_C);
68
69    // 5. Initialize OpenCL
70    // OpenCL specific variables
71    cl_context clGPUContext;
72    cl_command_queue clCommandQue;
73    cl_program clProgram;
74    cl_kernel clKernel;
75
76    size_t dataBytes;
77    size_t kernelLength;
78    cl_int errcode;
79
80    // OpenCL device memory for matrices
81    cl_mem d_A;
82    cl_mem d_B;
83    cl_mem d_C;
84
85    /*****************************************/
86    /* Initialize OpenCL */
87    /*****************************************/
88    clGPUContext = clCreateContextFromType(0,
89                    CL_DEVICE_TYPE_GPU,
90                    NULL, NULL, &errcode);
91    shrCheckError(errcode, CL_SUCCESS);
92
93    // get the list of GPU devices associated
94    // with context
95    errcode = clGetContextInfo(clGPUContext,
96              CL_CONTEXT_DEVICES, 0, NULL,
97              &dataBytes);
98    cl_device_id *clDevices = (cl_device_id *)
99              malloc(dataBytes);
100   errcode |= clGetContextInfo(clGPUContext,
101             CL_CONTEXT_DEVICES, dataBytes,
102             clDevices, NULL);
103   shrCheckError(errcode, CL_SUCCESS);
104
105   //Create a command-queue
106   clCommandQue = clCreateCommandQueue(clGPUContext,
107                  clDevices[0], 0, &errcode);
108   shrCheckError(errcode, CL_SUCCESS);
109
110   // Setup device memory
111   d_C = clCreateBuffer(clGPUContext,
```

```
112         CL_MEM_READ_WRITE,
113         mem_size_A, NULL, &errcode);
114     d_A = clCreateBuffer(clGPUContext,
115         CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
116         mem_size_A, h_A, &errcode);
117     d_B = clCreateBuffer(clGPUContext,
118         CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
119         mem_size_B, h_B, &errcode);
120
121
122     // 6. Load and build OpenCL kernel
123     char *clMatrixMul = oclLoadProgSource("kernel.cl",
124                         "// My comment\n",
125                         &kernelLength);
126     shrCheckError(clMatrixMul != NULL, shrTRUE);
127
128     clProgram = clCreateProgramWithSource(clGPUContext,
129                 1, (const char **)&clMatrixMul,
130                 &kernelLength, &errcode);
131     shrCheckError(errcode, CL_SUCCESS);
132
133     errcode = clBuildProgram(clProgram, 0,
134               NULL, NULL, NULL, NULL);
135     shrCheckError(errcode, CL_SUCCESS);
136
137     clKernel = clCreateKernel(clProgram,
138                "matrixMul", &errcode);
139     shrCheckError(errcode, CL_SUCCESS);
140
141
142     // 7. Launch OpenCL kernel
143     size_t localWorkSize[2], globalWorkSize[2];
144
145     int wA = WA;
146     int wC = WC;
147     errcode = clSetKernelArg(clKernel, 0,
148               sizeof(cl_mem), (void *)&d_C);
149     errcode |= clSetKernelArg(clKernel, 1,
150               sizeof(cl_mem), (void *)&d_A);
151     errcode |= clSetKernelArg(clKernel, 2,
152               sizeof(cl_mem), (void *)&d_B);
153     errcode |= clSetKernelArg(clKernel, 3,
154               sizeof(int), (void *)&wA);
155     errcode |= clSetKernelArg(clKernel, 4,
156               sizeof(int), (void *)&wC);
157     shrCheckError(errcode, CL_SUCCESS);
158
159     localWorkSize[0] = 3;
160     localWorkSize[1] = 3;
161     globalWorkSize[0] = 3;
```

```
162     globalWorkSize[1] = 3;
163
164     errcode = clEnqueueNDRangeKernel(clCommandQue,
165                clKernel, 2, NULL, globalWorkSize,
166                localWorkSize, 0, NULL, NULL);
167     shrCheckError(errcode, CL_SUCCESS);
168
169     // 8. Retrieve result from device
170     errcode = clEnqueueReadBuffer(clCommandQue,
171                d_C, CL_TRUE, 0, mem_size_C,
172                h_C, 0, NULL, NULL);
173     shrCheckError(errcode, CL_SUCCESS);
174
175     // 9. print out the results
176     printf("\n\nMatrix␣C␣(Results)\n");
177     for(int i = 0; i < size_C; i++)
178     {
179        printf("%f␣", h_C[i]);
180        if(((i + 1) % WC) == 0)
181        printf("\n");
182     }
183     printf("\n");
184
185     // 10. clean up memory
186     free(h_A);
187     free(h_B);
188     free(h_C);
189
190     clReleaseMemObject(d_A);
191     clReleaseMemObject(d_C);
192     clReleaseMemObject(d_B);
193
194     free(clDevices);
195     free(clMatrixMul);
196     clReleaseContext(clGPUContext);
197     clReleaseKernel(clKernel);
198     clReleaseProgram(clProgram);
199     clReleaseCommandQueue(clCommandQue);
200
201 }
```

# 7

# Datasets

These are the datasets used in chapter 4.

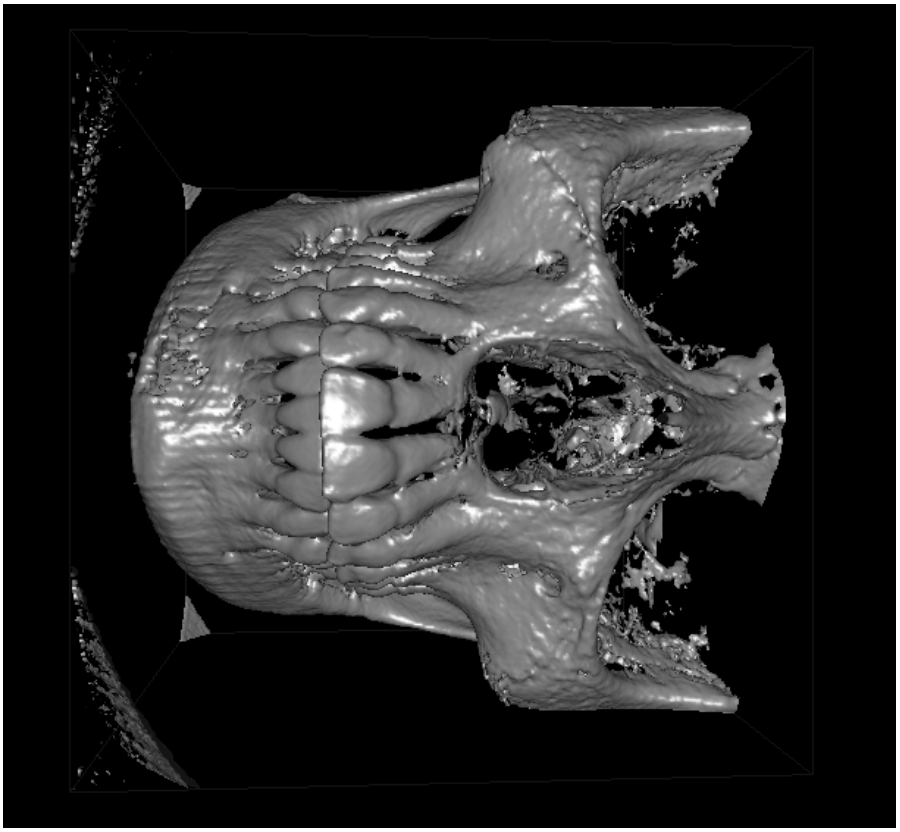


**Figure 7.1:** Skull dataset.

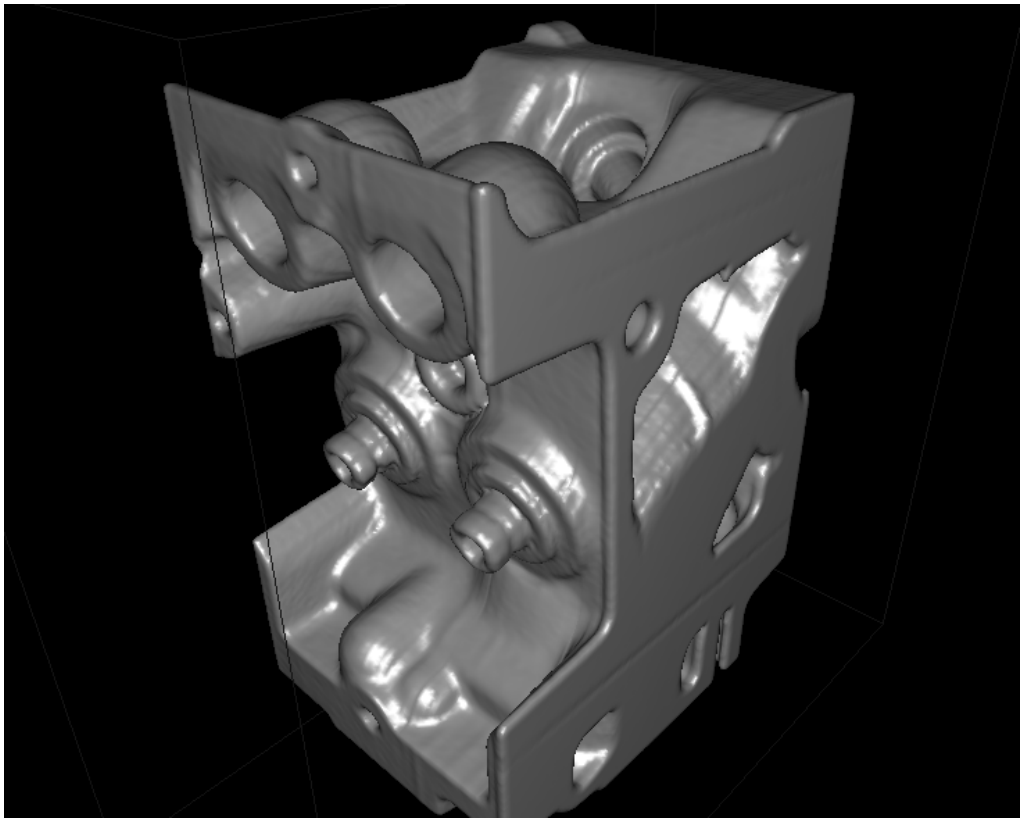


**Figure 7.2:** Engine dataset.

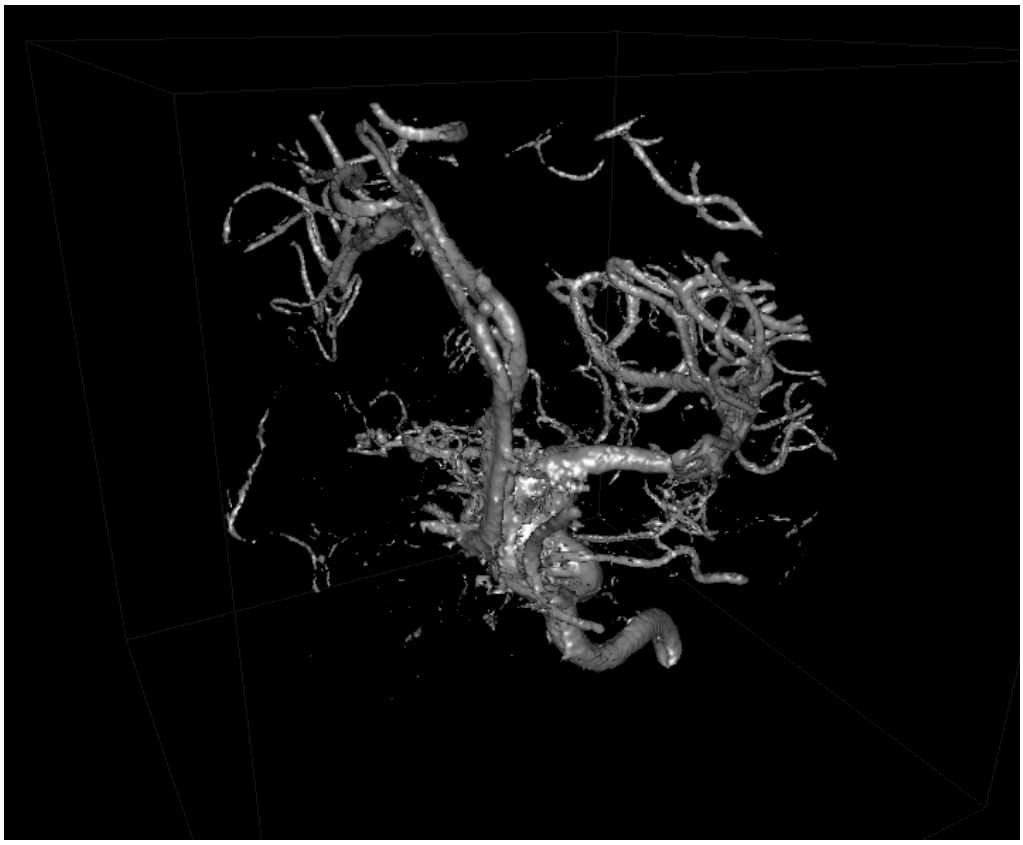


**Figure 7.3:** Aneurysm dataset.

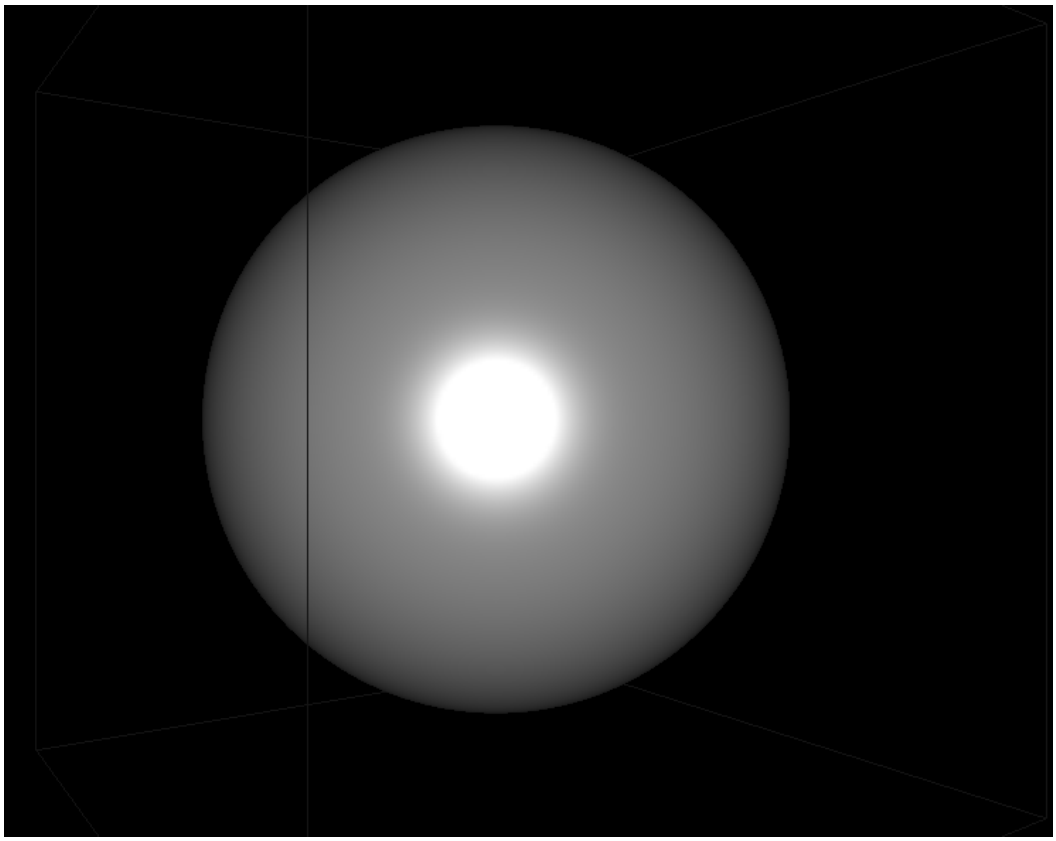


**Figure 7.4:** Sphere dataset.

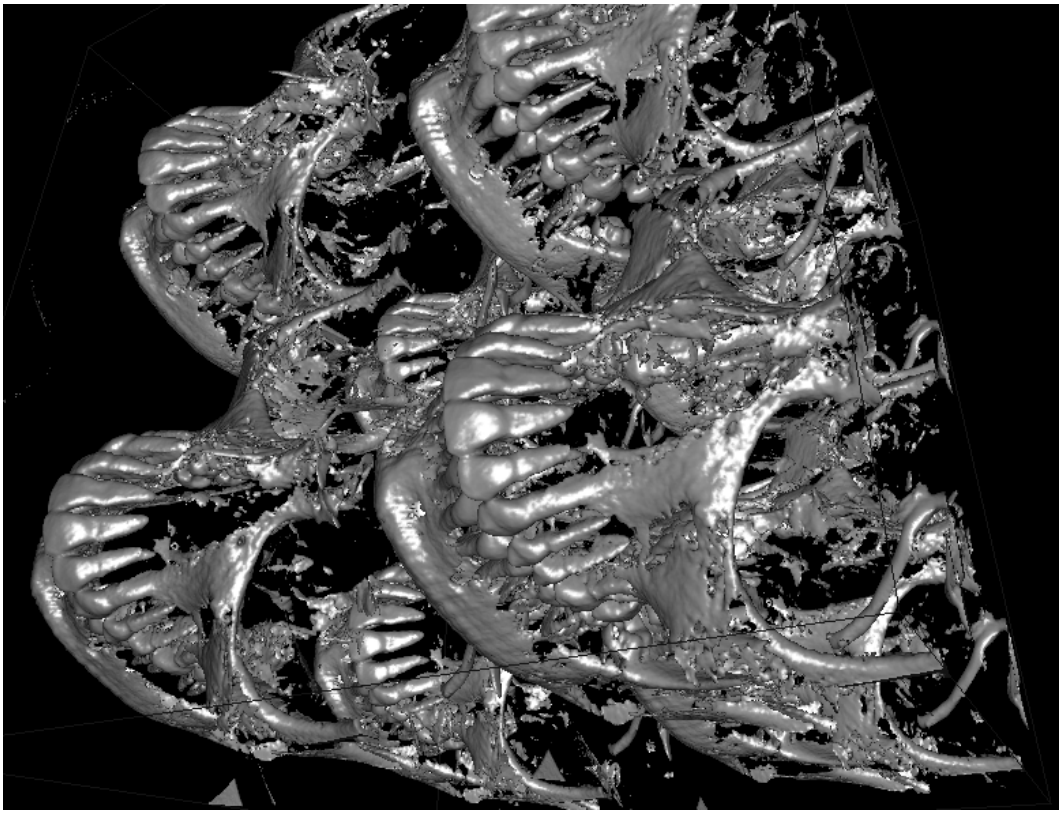


**Figure 7.5:** 8 skulls dataset.