Scala Parallel Collections
VS
GPU Frameworks

Marco Antognini

Spring 2013

Bachelor Semester Project under the supervision of:
Aleksandar Prokopec & Prof. Martin Odersky
Programming Methods Laboratory LAMP

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Abstract

The objectives of this bachelor project are multiple. First, it aims to identify the currently available technologies to perform heavy computation like clusters of servers but also more compact computational devices like GPUs.

The second objective is to implement a few benchmark applications to compare the performance – that is, the processing time but also more subjective metrics like the number of code line or the implementation cost in manpower – of a GPU framework and the parallel collections of Scala that work on CPUs.

The selected GPU framework is Thrust, developed by NVIDIA on top of CUDA. Alternatively, this document also presents other technologies like Microsoft's C++AMP and uses very quickly TBB, a C++ framework for parallel computation developed by Intel.

A subsidiary point approached by this paper is the comparison of performance between the current parallel collections of Scala and the underdevelopment Workstealing framework.

This project concludes that GPU implementations are definitely much faster for short and easy algorithms but for bigger problems the use of GPU computational device is much harder, takes longer to develop and is not always faster. It also looks for future developments opportunities for Scala.

Acknowledgments

I would like to thank the LAMP for having given me the opportunity and ways to work on such fascinating and actual topic. I'm also very grateful to my friends and family members who encouraged me with interesting ideas and reflexions, and, of course, anyone else who helped me, directly or indirectly.

1. Introduction

The first programming models invented were all sequential. After years of development, a big research area of Computer Sciences is nowadays dedicated to understand and improve parallel programming models. There is the now common multithreading approach where several unrelated tasks are executed in parallel – high API systems call this *future* or *promise* – based on the low level idea of *Threads*. Alternatively there is the *Actor pattern* which is implemented by, for example, the Akka framework for Scala.

Another way of using the parallel computation power, available in Scala, is based on collections. For example, if each element of a sequence can be transformed to another value independently from any other elements then the sequence can be considered as parallel sequence – i.e. the mapping can be performed by using multiple CPU cores.

In this paper we first discuss briefly the difference and common similarities of Scala and C++. Then we make a quick tour of the currently available computation models that use common hardware material such as CPU or GPU.

Afterwards we have a closer look at the GPGPU programming model illustrated with CUDA and its derived product: the Thrust framework.

This paper then covers the benchmark results of three applications coded in C++, Scala and with Thrust: a Monte Carlo simulation, the computation of Mandelbrot sets and the implementations of a Genetic Algorithm to maximize a given function. The main differences, performance, issues and advantages are detailed and a global conclusion is made to determine which implementation is the best in a given situation by using several metrics.

Finally, we look for future development opportunities for Scala that can benefit from GPU computational device and technics.

1.1. Scala & C++

Scala and C++ are two different programming languages. Yet they share many common features and theoretical notions: they are both statically typed and type-safe, have inherited the curly brace syntax from C and are compiled languages, to mention only a few.

```
#include <iostream>
int main(int, char**) {
    std::cout << "Hello, World!"
              << std::endl;
    return 0;
}
```

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, world!")
  }
}
```

Beyond these details they both let the programmer choose how he wants to use to design his software with several paradigms: data abstraction, object oriented, imperative, functional or meta-programming. However these paradigms are approached with very distinct theoretical backgrounds. For examples, the meta-programming in C++ allows compile-time computations whereas Scala can handle type covariance and contravariance; the functional aspect introduced by C++11, the last standard of the language, is not as advanced as Scala's functional features but yet can ease the design of some softwares as we will see later in this document with the Genetic Algorithm implementation in C++11.

C++ allows very good optimizations at compile time and is known for its speed efficiency but its syntax usually requires more line of code to write the same algorithm than in Scala which can make big software hard to read and understand. On the other hand, the Just In Time compiler of the Java Virtual Machine has already proved to be capable of doing important optimizations and considerably reducing the performance gap between native assembly languages and Java Bytecode.

Scala provides software developers with high level API. C++ has a relatively high API but also gives the developers low level API and complex but very powerful features like the *const* modifier for methods, the *move semantics* or *variadic templates*.

Scala integrates the idea of *immutability* which tends to make program design easier and less bug-prone. This concept is not enforced by the C++ Standard Template Library (STL).

Another important contrast between the two languages is the memory management system. Modern C++ technics drag manual memory management utilities like the *new* or *delete* operators to the closet. Nonetheless, the developer is in charge to make sure the object bound to a reference is still «alive». He also has to make sure his software design respects the *Resource Acquisition Is Initialization* principle, abbreviated RAII, by providing *destructors* for objects deallocation, when appropriate, since there is no *Garbage Collector*. Despite looking awfully complex to Java developers this idiom has at least the two following advantages. First, no garbage collector is required and therefore no CPU time is periodically «lost» to clean up the memory. Furthermore, resources like files or sockets can be easily released within a precise timeline without the need of the *dispose pattern* on user-site code.

Scala is a very young programming language compared to 30+ year old C++ and yet provides its users with incredible computational power with its parallel collections or its actors and takes benefit from the full power of the available CPUs, even in big clusters with a high level API.

1.2. CPU & GPU

Central Processing Units are the main computational component of personal computers and servers. They carry out the assembly instruction of a computer program with basic arithmetical operations on binary data. They also manage the input and output transactions of the system. Their major parts include the Arithmetic Logic Unit (ALU) and a Control Unit (CU) that extract and execute the instructions from memory.

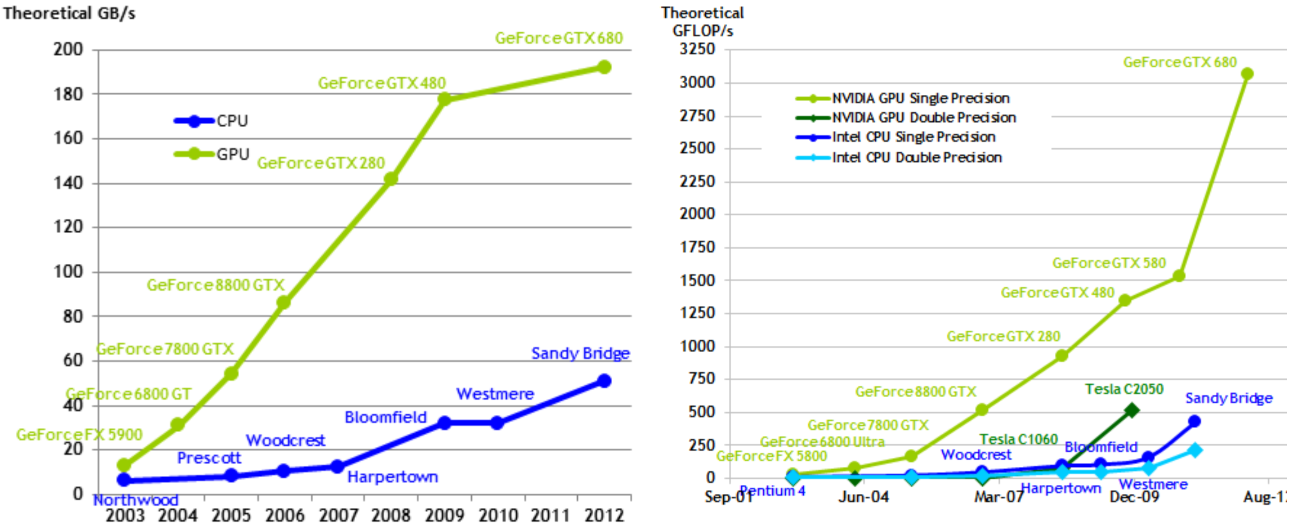
CPUs can optimize the latency of instructions by using piplining mechanisms but we cannot consider such optimizations as true parallelism. When they were initially designed CPUs were single core processors and a first kind of parallelism was introduced in the Operating Systems' kernel to run processes «at the same time» by using different kinds of scheduling policies (e.g. Round Robin, Time Slicing, FIFO, ...) and by switching the active job frequently.

This led to the introduction of threads – sometimes called lightweight processes – to let a process executes multiple tasks «concurrently». However, with a mono core CPU the assembly instructions of two given tasks can still not be run at the same time.

When the first multiprocessors computer was introduced the «concurrent» programming was sent to a whole new level. This evolution was then followed by the creation of multi-core CPUs. Nowadays a personal computers on the market usually have between 2 and 8 cores at decent prices. Some algorithms can benefit from a speedup by a factor of *N* when run on an *N*-core CPU as the paper entitled «Parallel Machine Learning Implementations in Scala», by Pierre Grydbeck, demonstrates.

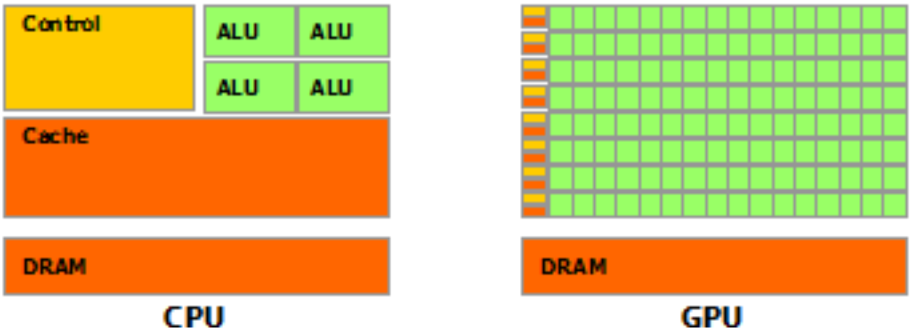
During the last 60 years CPUs were improved in many ways: the size of their circuits was constantly reduced, the number of transistors and bus width increased, caches were introduced and all kinds of material and hardware design breakthroughs resulted in an always increasing clock rate – reaching more than 3GHz today in common CPUs – and therefore reducing significantly the latency of assembly instructions.

On the other hand, Graphics Processing Units were designed from the beginning to optimize the throughput and not the latency by providing highly parallelised computation facilities. For example, there are 384 cores clocked at 850 MHz on the GeForce GT 650M graphics card which delivers a floating-point performance of 652 Gflops while an Intel Core i7-3720QM clocked at 2.60 GHz, with 4 physical cores, gives around 34 Gflops.



NVIDIA's statistics on GPU and CPU bandwidth and theoretical Gflops

GPUs were initially created to render images but their computation power turned out to be an incredible tool for heavy computations. Since most GPUs offer a functionally complete set of operations performed on arbitrary bits, general-purpose computing on graphics processing units technologies – or GPGPU for short – as OpenCL or CUDA were created to use this computational power in non-graphics softwares like folding proteins, financial simulations, astrophysics experiments or medical researches.



CPU and GPU processor structures

Another way to get higher computation power is to cluster CPUs or GPUs. For examples, EPFL’s Callisto introduced in 2008 is an IBM Intel Harpertown cluster with a total of 1024 cores and a theoretical power of 12.29 Tflops, or the newly installed Blue Gene/Q supercomputer with a 209 Tflops theoretical power at an undisclosed price, or the Electra cluster, acquired in 2011, gathering several CPUs and GPUs to obtain a theoretical power of 52 Tflops.

Some recent developments have brought interesting inventions. For example, Bill Hsu and Marc Sosnick-Pérez from the Department of Computer Science of San Francisco State University have developed a real time finite difference-based sound synthesis. In other words they are using the parallel power of the GPU to compute in real time, for example, the sound produced by a metallic plate to generate gong or cymbal-like sounds.

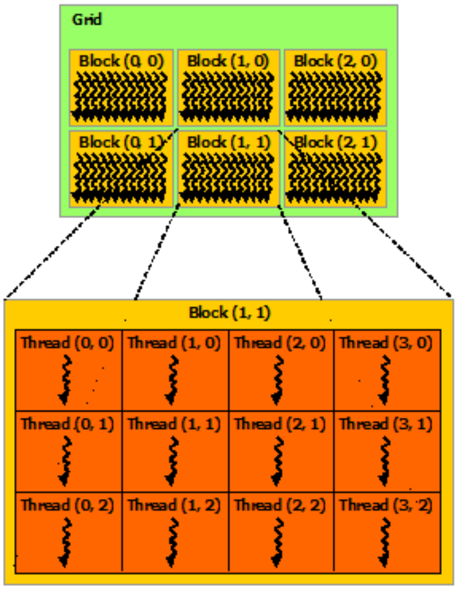
2. Introduction to GPGPU

GPU computation model differs from the usual software structure used on CPU. A brief introduction is required.

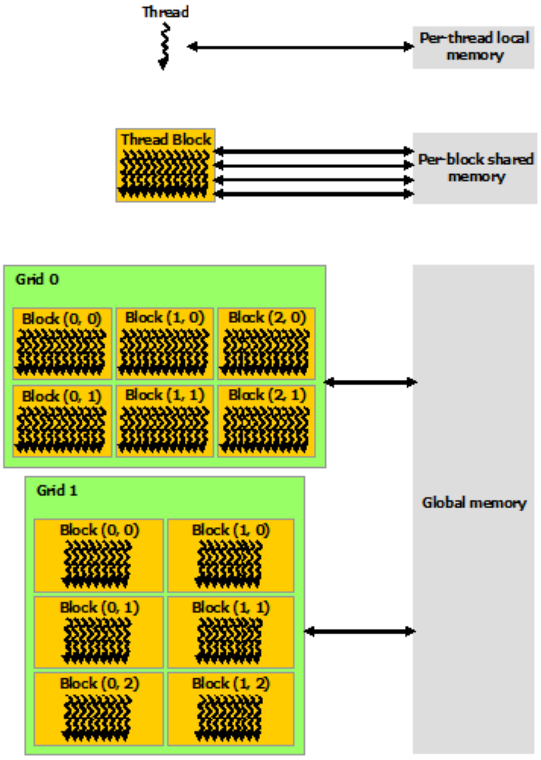
2.1. CUDA Kernel

Compute Unified Device Architecture – or CUDA – is a programming model created by *NVIDIA* for their GPUs based on the C language and a few features from C++ are also supported like *templates* or *operator overloading*. Fundamentally, the programmer define a kernel function that is run with a given input data (e.g. an array of values). The kernel is then run simultaneously on many *threads*, all executing the same instructions.

Threads are grouped into *thread blocks* and *thread blocks* together build a *grid*. The grid can have 1, 2 or 3 dimensions. Each thread and thread block has a unique identifier which let the programmer decide which part of input data a given thread should process.



2.2. Memories



There are several kind of memories available, each with different advantages and limitations. There is, of course, the classic memory of the *host*, usually DDR or alike, which is used by the CPU. There is also global memory on the *device*, of type GDDR, available to the kernel function. The global memory is divided into two regions: a read & write one, and a read-only one for constants. Then each thread block shares a part of the device memory. Each thread has its own memory space on the device for its registers. Finally, each core of the GPU has some memory caches.

Each of these memories has a different access speed. The main rule is relatively simple: the smaller the memory size the faster the access. Here is a table representing the time required for memory accesses:

registers	read/write	per-thread	~1 clock cycle
shared memory	read/write	per-block	~5 clock cycles
global memory	read/write	per-grid	~500 clock cycles
constant memory	read-only	per-grid	~5 clock cycles with caching

GPU Memories

Memory access, especially shared and global memories, can be highly optimized when the accessed data is coalesced, that is, the fetched memory entries are contiguous and can be read or written with a minimum number of transfers on the memory bus.

Since many threads are accessing the memory at the same time, the storage hardware is divided into *memory banks*: multiple rows and columns of storage units. Two read or write operations cannot concurrently access the same memory bank and the hardware will automatically serialize conflict accesses. However, this will result in a huge loss in performance.

2.3. Execution

With many threads running truly at the same time, the issue of synchronization has to be well understood. Fortunately, CUDA has some barrier mechanisms. The *__syncthreads()* function will make sure every threads within the same thread block have reached the same point of the kernel function. Of course, dead lock can stuck the GPU if this function is called inside an *if* statement or a loop and one (or more) thread within a thread block doesn't call the synchronization procedure.

The performance of a kernel function also highly depends on the *divergence* of *warps*. A warp is a set of 32 threads within a thread block that are executed in group and run the same instruction together. In other words, when reaching an *if p then s else t* statement the threads of a warp for which *p* holds will execute *s* and then only the other threads of the same warp will execute *t*. The same can happen when some but not all threads leave a loop.

This divergence phenomenon can dramatically reduce the performance of a kernel function. In the worst case, if one thread per warp takes an expensive branch, then there is lose by a factor of 32 in performance! It is therefore very important to take this in consideration when writing kernels.

2.4. CUDA Program Structure

The usual structure of a CUDA program is the following:

(1) The kernel input data is initialized on the host system;
(2) Memory for input and output data is allocated on the device;
(3) The input data is transferred from the host to the device;
(4) The kernel function is launched, parametrized by the grid dimensions and the number of threads per block;
(5) Once the kernel function has finished, the output data is copied back to the host;
(6) The device memory is deallocated;
(7) Finally, the kernel's answer is printed – or used in any other way.

CUDA kernel code are usually written in *.cu* files and compiled with *nvcc*.

2.5. Vector Addition Sample

```
#include <stdlib.h>
#include <cuda_runtime.h>

/**
 * CUDA Kernel Device code
 *
 * Computes the vector addition of A and B into C.
 * The 3 vectors have the same number of elements numElements.
 */
__global__ void vectorAdd(const float* A, const float* B, float* C, int numElements)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < numElements) {
        C[i] = A[i] + B[i];
    }
}

int main(void)
{
    // Compute the vector's size
    int numElements = 50000;
    size_t size = numElements * sizeof(float);

    // Allocate the host input vectors A & B and output vector C
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);
    float* h_C = (float*)malloc(size);

    // Initialize the host input vectors
    for (int i = 0; i < numElements; ++i) {
        h_A[i] = rand()/(float)RAND_MAX;
        h_B[i] = rand()/(float)RAND_MAX;
    }

    // Allocate the device input vector A & B and output vector C
    float *d_A = NULL, *d_B = NULL, *d_C = NULL;
    cudaMalloc((void**)&d_A, size);
    cudaMalloc((void**)&d_B, size);
    cudaMalloc((void**)&d_C, size);

    // Copy the host input vectors A and B in host memory
    // to the device input vectors in device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Launch the Vector Add CUDA Kernel
    int threadsPerBlock = 256;
    int blocksPerGrid = (numElements + threadsPerBlock - 1) / threadsPerBlock;
    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);

    // Copy the device result vector in device memory
    // to the host result vector in host memory.
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device & host global memory
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
    free(h_A); free(h_B); free(h_C);

    // Reset the device and exit
    cudaDeviceReset();
    return 0;
}
```

(Inspired by NVIDIA's samples.)

The error handling part of this example was stripped. Yet, for such simple kernel so much code is needed and the software become highly error-prone. Thankfully, Thrust provides a much higher API to solve the same problem and the code is extremely simplified as we will see later.

2.6. C++AMP

Microsoft has developed a similar technology called C++ Accelerated Massive Parallelism, or C++AMP for short. This framework is based on DirectX 11 and takes advantage of the GPU in a similar way. Unlike CUDA, C++AMP uses some features from C++11 like the lambda functions. Here is the vector addition example provided by Microsoft:

```
#include <amp.h>
#include <iostream>
using namespace concurrency;
const int size = 5;

void CppAmpMethod() {
    int aCPP[] = {1, 2, 3, 4, 5};
    int bCPP[] = {6, 7, 8, 9, 10};
    int sumCPP[size];

    // Create C++ AMP objects.
    array_view<const int, 1> a(size, aCPP);
    array_view<const int, 1> b(size, bCPP);
    array_view<int, 1> sum(size, sumCPP);
    sum.discard_data();

    parallel_for_each(
        // Define the compute domain,
        // which is the set of threads that are created.
        sum.extent,
        // Define the code to run on each thread on the accelerator.
        [=](index<1> idx) restrict(amp)
        {
            sum[idx] = a[idx] + b[idx];
        }
    );

    // Print the results. The expected output is "7, 9, 11, 13, 15".
    for (int i = 0; i < size; i++) {
        std::cout << sum[i] << "\n";
    }
}
```

With their higher API the code becomes much easier to understand and designing bigger algorithm is therefore simplified.

2.7. OpenCL

OpenCL stands for Open Computing Language. It is a free and open standard for general purpose parallel programming across CPUs, GPUs and other processors that gives software developers portable and efficient access to the power of these heterogeneous processing platforms. Its API is much lower than CUDA's API, though.

Major companies like Apple, IBM, ARM, NVIDIA, AMD, and more, work together to develop its specifications.

3. Thrust

Thrust is a C++ general purpose, high level, GPU and CPU accelerated library. It has several backends: CUDA, Intel's Threading Building Blocks or OpenMP.

3.1. API

Its API is very similar to STL so that C++ developers can start very quickly to write GPU applications. Since most algorithms' input data are arrays, Thrust defines two kinds of container that also match STL's std::vector interface: thrust::host_vector and thrust::device_vector. As their names state the former is used on the host system and the latter is used on the GPU with, for example, the CUDA backend. Many of Thrust's generic algorithms have the same interface as their C++ counterpart and heavily use the mechanism of iterators to represent ranges. The framework implements searching, copying, reductions, sorting, mapping, scanning algorithms, just to mention a few.

Unfortunately, Thrust doesn't yet support C++11 additional features like anonymous lambda functions; the programmer currently has to stick with C++03 standard. The developer team wants to support these features but are facing limitations with the CUDA backend. However, they could reimplement some features like advanced random number generators or tuples.

3.2. Vector Addition Sample

The vector addition program we described earlier can be implemented as follow with Thrust:

```
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <iostream>

int main(void)
{
    const int N = 5;
    int raw_a[N] = {1, 2, 3, 4, 5};
    int raw_b[N] = {6, 7, 8, 9, 10};

    // Create device vectors
    thrust::device_vector<int> A(raw_a, raw_a + N);
    thrust::device_vector<int> B(raw_b, raw_b + N);
    thrust::device_vector<int> C(N);

    // Perform the sum of A and B and save it into C
    thrust::transform(A.begin(), A.end(),
                      B.begin(), C.begin(),
                      thrust::plus<int>());

    // Copy the result back to the host system
    thrust::host_vector<int> host_c = C;

    // Print the result
    for (int i = 0; i < N; ++i) {
        std::cout << host_c[i] << "\n";
    }

    return 0;
}
```

The higher level of Thrust compared to CUDA hides all the manual memory allocations, the low level transfer of data between the host and device systems and the generic algorithms like thrust::transform handle the logic of setting up grid dimensions and the number of threads per block. The developer can then focus mostly on the algorithm he is implementing.

3.3. Limitations

In this particular implementation of the vector addition, the addition operator is provided by Thrust directly. In more general cases the developer will have to write the function himself. The common technic to implement a parametrized function in C++03 is to create a functor: a structure with an overloaded *operator()* for function call.

Here is another example illustrating how a Single-precision real Alpha X Plus Y operation, or SAXPY for short, can be implemented:

```
struct saxpy_functor : public thrust::binary_function<float,float,float> {
    const float a;

    saxpy_functor(float alpha) : a(alpha) {}

    __host__ __device__
    float operator()(const float& x, const float& y) const {
        return a * x + y;
    }
};
```

Functors regrettably have the negative effect to create bloated and complex code structures. Fortunately, Thrust is aiming to support in a near future more C++11 features like its functional aspect to reduce the code complexity.

Thrust's specific design has an inherent downside: it is possible to perform inner product-like function – with the thrust::inner_product function that can be parametrized with custom + and * operators to serve a more general purpose – but it is not possible to perform a Cartesian product on two vectors. Hence, it is not possible to implement matrix multiplication efficiently. Other tools like CUSP, a library for sparse linear algebra and graph computations on CUDA, are more appropriate for such applications.

It is also important to mention that Thrust is a recent development and was born only in 2009. While it is evolving quite quickly it also suffers from some sever stability issues. During the creation of the benchmarks described below, the operating system has crashed several times due to kernel panics when the GPU software was not well formed and contained some bugs (*e.g.* accessing an out of bound index of an array). The graphics card's driver are probably responsible for this along with some complex operations happening in the backend of Thrust.

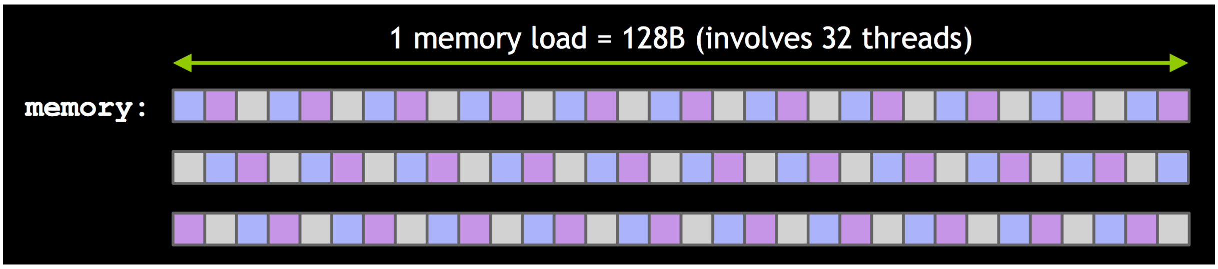
3.4. Memory Optimizations

Even with this relatively high level API, the coalescing issue is still present. An interesting optimization to this problem can be applied when dealing with *Array of Structures* (AoS). The data structure design can be viewed from another angle and designed as *Structure of Arrays* (SoA) instead. With this approach the size of the loaded memory by a kernel function can be reduced to its minimum. Here is an example.

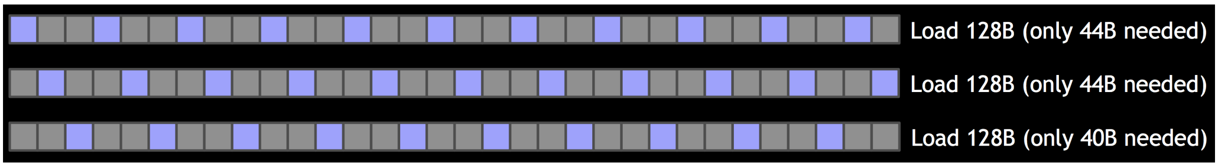
Let the following AoS be:

```
struct Float3 { float x, y, z; };
thrust::device_vector<Float3> data;
```

And assuming the size of the data is 32, then we have the following memory storage, where X's are in blue, Y's in mauve and Z's in light grey:[1]



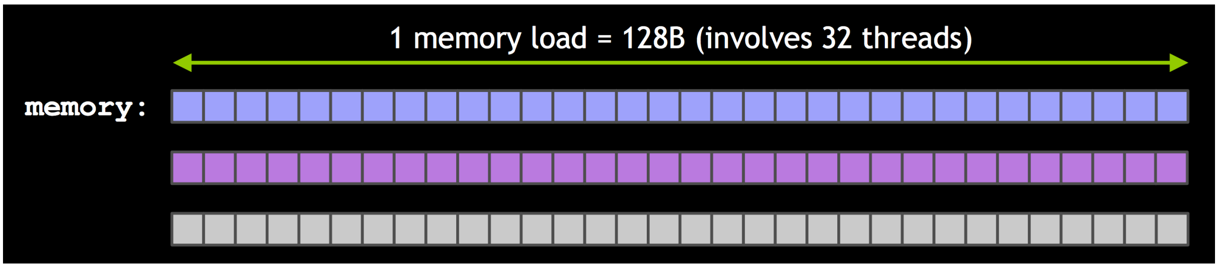
When a warp's threads read all 32 X's simultaneously, the whole 384 Bytes are loaded:


Load 128B (only 44B needed)
Load 128B (only 44B needed)
Load 128B (only 40B needed)

So when the Y's and Z's are also loaded a total of 1.125 KBytes is eventually loaded instead of the 384 Bytes really needed. However, with the following SoA:

```
struct StructOfFloats {
    thrust::device_vector<float> x;
    thrust::device_vector<float> y;
    thrust::device_vector<float> z;
};
StructOfFloats data;
```

The memory is arranged as follow:



This time, when X's are loaded by the warp's threads only the first 128 Bytes are loaded. It follows that when Y's and Z's are loaded, only the required 384 Bytes are read from memory. This can lead to a significant speedup when many Bytes are fetched from memory.

[1] The images present on this page are inspired from NVIDIA's guide

4. Benchmarking Environment

The different softwares implemented for this project were all run on the same computer and with the same benchmarking technics. They were cautiously designed to prevent as much boxing as possible, workaround the contention issue arising with multithreading or too much data flow between the host and the GPU device.

4.1. Hardware

The computer used for the benchmark is a MacBookPro with the following hardware spec:

Model	MacBookPro10,1
CPU	Intel Core i7 3720QM @ 2.6 Ghz
RAM	16 GB DDR3 @ 1600 Mhz
Discrete GPU	Intel HD Graphics 4000
Integrated GPU	NVIDIA GeForce GT 650M with 1GB of GDDR5
Storage	512 GB APPLE SSD SM512E

The discrete GPU was used for the benchmark. The Operating System automatically switch to this better graphic card when intensive GPU activity is detected. *gfxCardStatus* was used to confirm this switch. When the discrete graphic card is substituted by the integrated one a noticeable latency is observed with the very first processing time measurement. The first record was therefore removed from the dataset.

4.2. Software

On the software side, the selected Operating System was Mountain Lion, fully up-to-date and running in 64 bits mode. Oracle's Java 7 was used along with Scala 2.10. For the C++ implementations the last and official Clang compiler shipped by Apple with Xcode was used. The CUDA compiler used was NVCC, published with CUDA 5, which uses GCC to compile C and C++ code.

Scala and TBB were installed with the help of *Homebrew*, a package manager for OS X similar to *aptitude* on Debian. Thrust was downloaded directly from its source on Github. The Workstealing framework was compiled into jars from its master branch on April 18, 2013.

The following table summaries the softwares and libraries used and their version:

OS X	Mountain Lion 10.8.3
Java	Oracle’s Java 1.7.0_15 for OS X
Scala	Scala 2.10.1 via Homebrew
Clang	Apple LLVM version 4.2 (clang-425.0.28) (based on LLVM 3.2svn)
GCC	GCC-LLVM 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2336.11.00)
NVCC	Release 5.0, V0.2.1221
Thrust	1.7 via Github.com
TBB	4.1u3 via Homebrew
Workstealing	Version 06378cb75bd5ff7061678b9e2791ac5bfe225143 of April 18, 2013

4.3. JVM Parameters

In order to get the best performance possible, the benchmarks were run using the server version of the Java JVM and with the following settings:

```
-Xmx8G -XX:+UseConcMarkSweepGC
-XX:MaxPermSize=8G -XX:+UseCondCardMark -server
```

The Workstealing framework was used with its *Workstealing.DefaultConfig* settings and the following JVM parameters:

```
-Dstep=1 -DincFreq=1 -DmaxStep=4096
-Drepeats=1 -Dstrategy=FindMax$ -Ddebug=true -Dpar=8
```

Three measurement systems were used for the Scala benchmarks detailed below. The Genetic Algorithm uses *Workstealing.StatisticsBenchmark*, the Monte Carlo implementation uses *Scalameter* 0.2 and the Mandelbrot program calls *System.nanoTime()* to determine the processing time. Each time the software is run multiple times so that we can get as much optimizations as possible with the Just In Time compiler.

4.4. C++ & Thrust Compilation Parameters

Most C-like languages have the advantage to be able to do some great optimizations at compile time with the right compiler settings. For this project Clang was set to use «-O3», the higher optimization level for speed. NVCC was also set to use this kind of aggressive optimizations. SFML's clock, an open source multimedia library for C++, was used to measure execution time.

4.5. Performance criterions

Comparing technologies as different as Scala and Thrust requires particular care. This paper, of course, discusses the processing time performance of the different implementation, since it is an objective metrics. It also presents more subjective metrics like the SLOC, aka the Source Lines Of Code, or the intuitiveness of the implementations. We also discusses the complexity of the code and relatively quickly the required man power required, from a relative perspective, to implement the different versions.

5. Benchmark: Monte Carlo

The Monte Carlo method used to compute, for example, π is commonly used to benchmark a system's performance.

5.1. Algorithm

Consider a circle inside a unit square. Their areas have a ratio of π/4. We use a uniform distribution for the x-axis and a similar but independent distribution for the y-axis to distribute random points in the unit square. π is then approximated by the ratio of the number of points inside the circle and the total number of points distributed in the square.

This algorithm can be honestly considered as an easy one and can be used to understand many aspects of GPU computing.

5.2. C++ Implementations

Two sequential implementations written in C++ of this Monte Carlo method were realized for this project. Both versions use the random generator API introduced in C++11 and are parameterized by the number of points distributed inside the unit square (called *pc* below for point count).

✴ Version C++#1

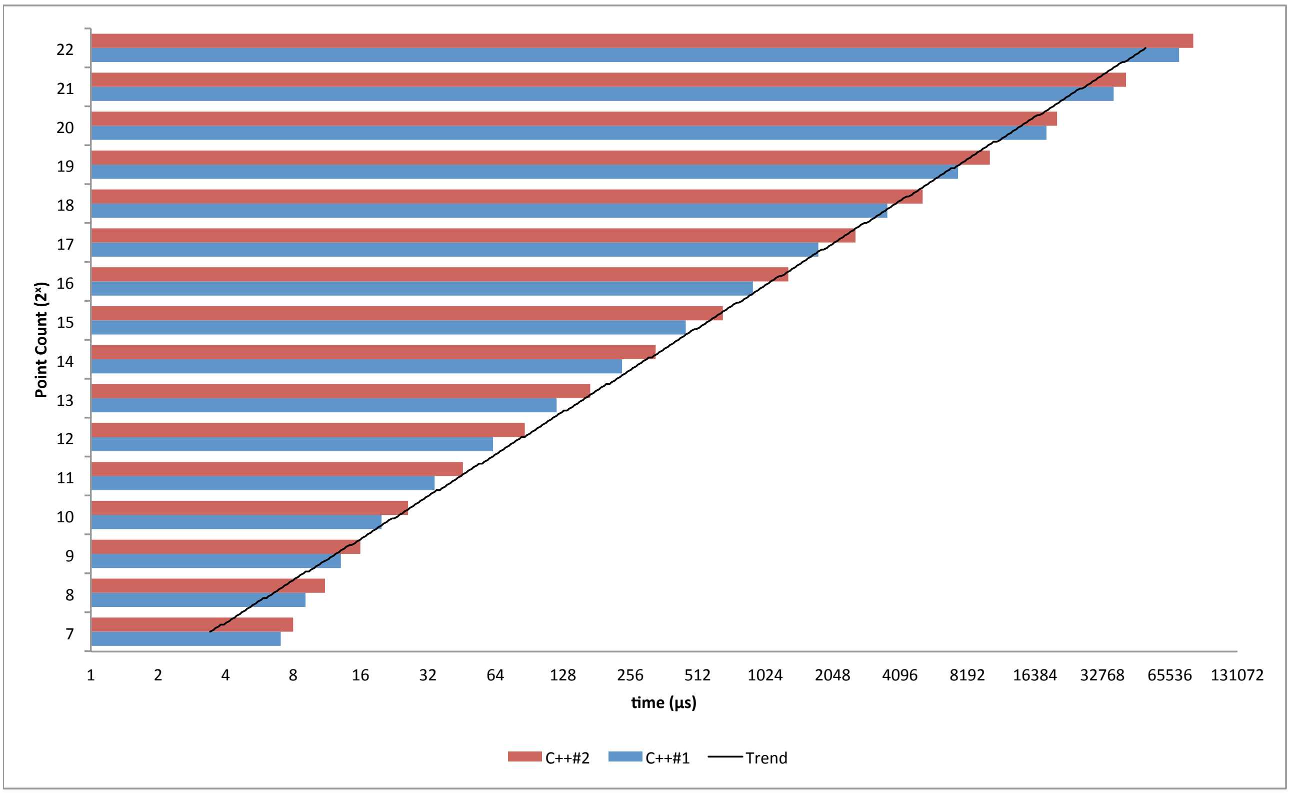
This first implementation is based on a simple for loop from 0 to *pc* and a counter. At each round two random numbers are generated from the two uniform distributions to represent the coordinates of a point. If the random point is inside the unit circle then the counter is incremented by one. When the loop is over the ratio can be computed to approximate π.

✴ Version C++#2

The second version is a little bit more complex. Instead of a for loop to generate the *pc* points all the points are generated and stored in an array with the help of std::generate generic function from STL. Then the number of point inside the unit circle is determined with another generic function: std::count_if. Finally, π is computed similarly as in the first version.

✴ Performance

The first implementation seems intuitively simpler and faster than the second one because it doesn't need to store any intermediate data. This is confirmed by the following chart:



Performance of the C++ implementation

The number of point generated ranges between 2^7 and 2^{22} and the computation time is expressed in microsecond on the x-axis in a logarithm scale of base 2. These two implementations don't use any parallelism which explains why the time is linearly proportional to the number of points. This logarithmic scale is used for most charts in this paper.

5.3. Thrust Implementations

Again two implementations of the Monte Carlo method were created using Thrust framework. When using random generator with multiple threads the developer has to be particularly cautious with the seed used. If two threads call the generator function with the same seed at the same time then both will get the same number and this occurs very often on GPU as illustrated before with the system of warps. To workaround this behavior the *discard* function becomes really handy: by using a thread identifier we can discard the right amount of random number to get a correct uniform distribution.

For this algorithm this solution doesn't introduce much complexity but we will see later that it can become really complex to tell to each thread what its unique identifier is when the implemented software is a little bit more complex.

The thread's identifier is transmitted to the kernel function manually since Thrust hides the logic of threads and thread blocks. Here is a piece of code illustrating how thread identifier can be set:

```
thrust::transform(
    thrust::counting_iterator<std::size_t>(0), // first identifier is 0
    thrust::counting_iterator<std::size_t>(MAX_ID), // last identifier is MAX_ID
    output_vector.begin(),
    transform_function);
```

Where the *transform_function* takes a parameter of type *std::size_t* that represents the thread's identifier. In this snippet, the identifiers range from 0 to *MAX_ID*.

✳ Version Thrust#1

The first implementation is very similar to C++#2; in fact the differences of the main logic between the two codes are mainly the use of *thrust::transform* and *thrust::count_if* instead of *std::generate* and *std::count_if*, respectively. The computation is also parametrized by *pc*.

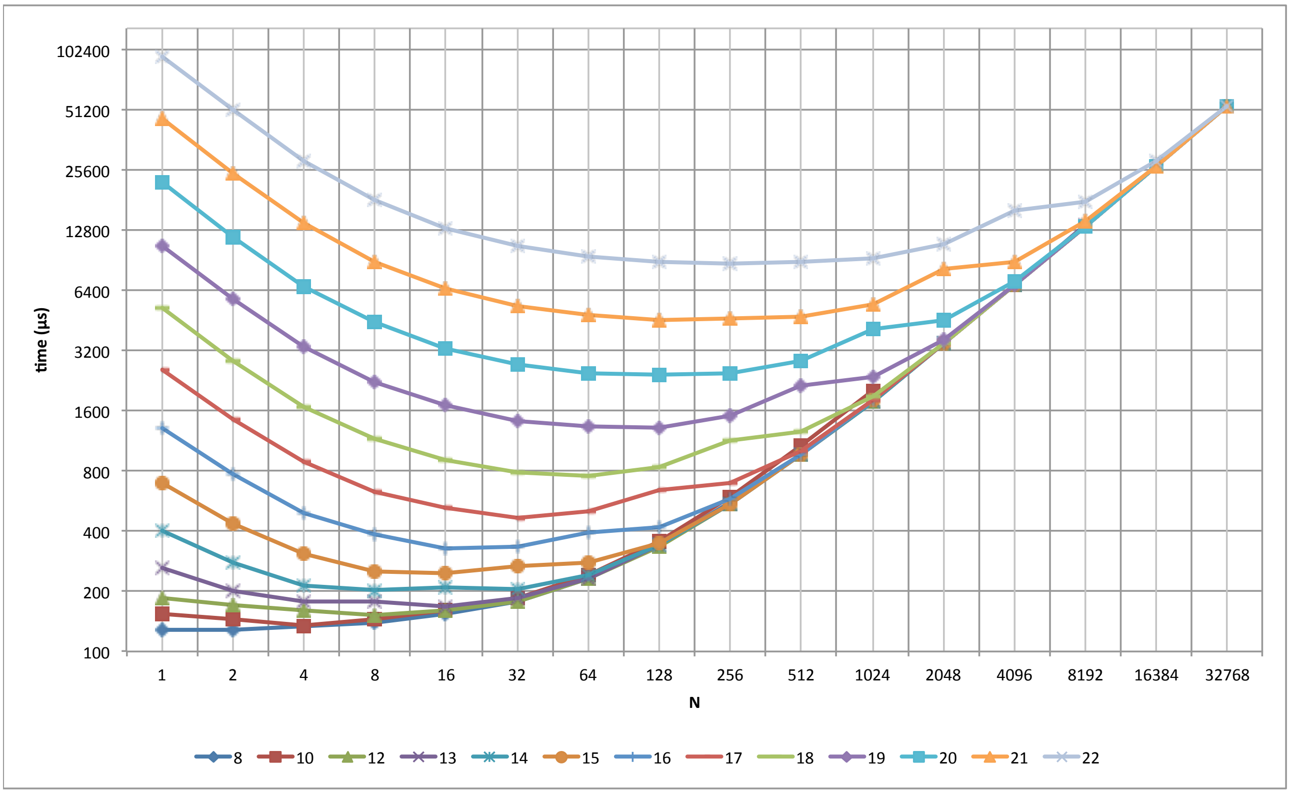
✳ Version Thrust#2

An alternative version takes an additional parameter such that each threads on the GPUs will compute *N* points of the total *pc* points to compute a first approximation of the ratio π/4. The final ratio is determined by using *thrust::transform_reduce* function: the ratios computed by each thread are added together to get a faction representing *pc / N * π*.

✳ Performance

Impact Of Workload Per Thread

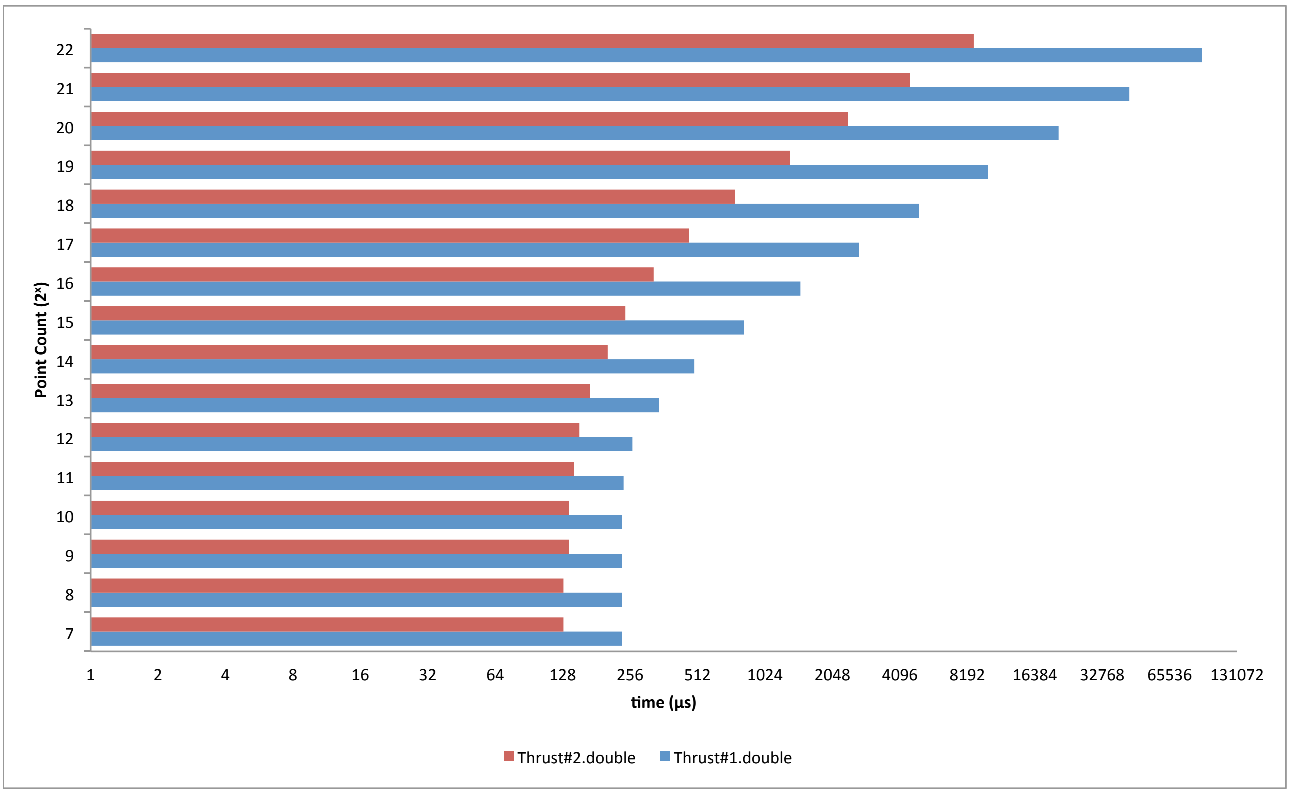
The goal of the second implementation is to reduce the number of threads needed to compute π and to increase the workload per thread. The parameter *N* affect performance as shown by this chart where each curve correspond to a *pc* value (2^{pc} generated points to be exact):



Performance of Thrust implementation: minimizing the processing time with *N*

We can see that the N minimizing the process time depends on the number of generated point: when too much, or too little, workload is put on each thread the performance decrease. For example, the best *N* for a point count of *pc* = 2^{16} is *N = 16* and for *pc* = 2^{22} the algorithm is the fastest with *N = 256*.

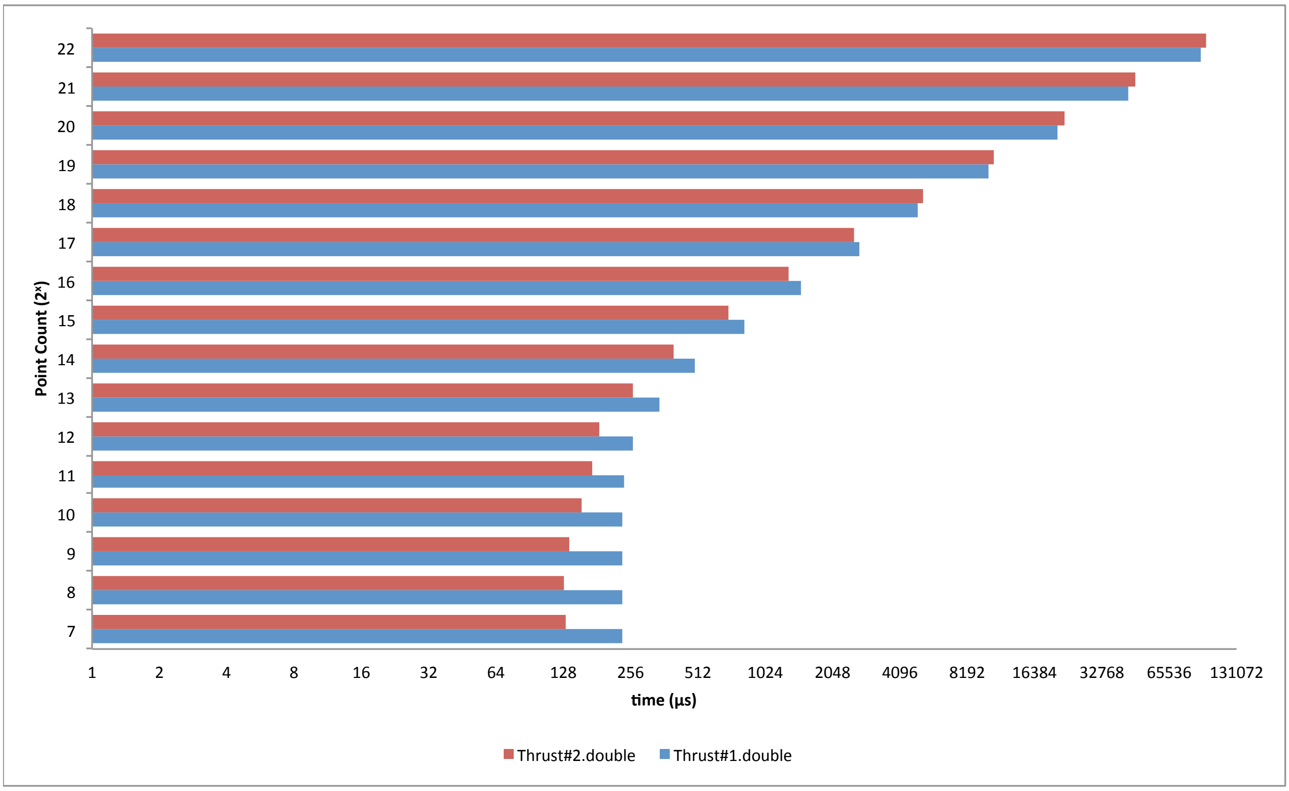
When we take the best N for each *pc* for this benchmark and compare the processing time with the first implementation: we can see that the second version is definitely much faster.



Performance of Thrust implementation with the best *N*

Preventing Intermediate Result

When we take *N = 1* for the second implementation and compare it to the first one we have the following performance relation: the second one is slightly faster, except with huge numbers of points (2^{18} and above).

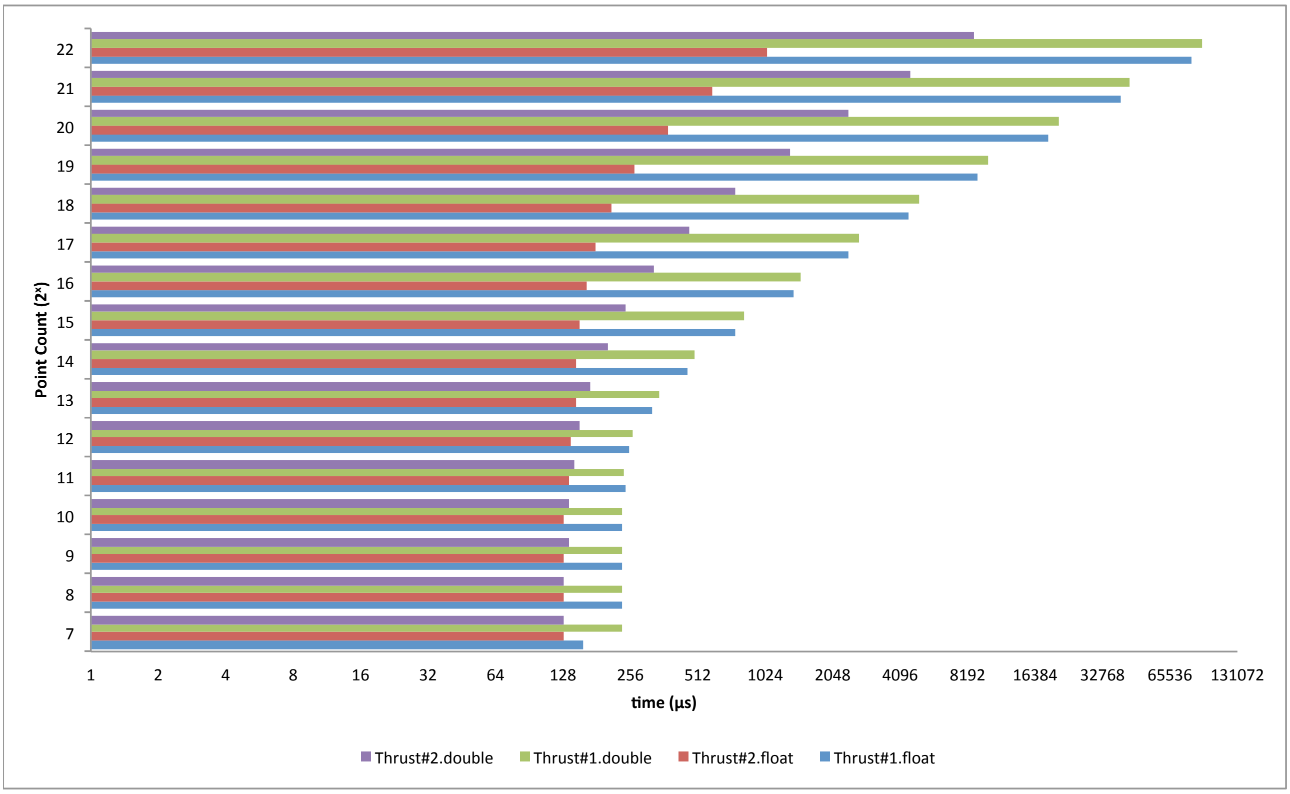


Processing time with *N = 1*

This can be explain by the important overhead of sending data between the host and device system With the second version there is no intermediate result whereas two different kernels are launched in the first one. This overhead becomes less significant with big *pc* values, though.

Single VS Double Precision Floating-Point Format

The performance of GPU applications depends highly on the data type used. Recent GPU devices now support double precision floating-point operations. However, operations on that format are much slower. The two implementations were initially benchmarked with a real type set to *double.* Here we benchmark them again but using *floats* instead of *doubles* to compare their relative performance for their best settings (with the best *N* for the second version):



Performance comparison between single and double precision formats

The process time is significantly improved when using single precision instead of double precision floating-point format.

5.4. Scala Implementation

The Scala implementation is relatively similar to Thrust#2. The *thrust::transform_reduce* function is essentially replaced by the *TraversableOnce.aggregate* method. Both versions are contention free with the number generators involved.

✳ Version Scala#1

This version has three parameters: *pc* and *N*, as above, plus a *cores* setting for the parallel collections defining the number of cores available. If *cores* is 1 (or lower) then the sequential collections is used instead of its parallel homologue.

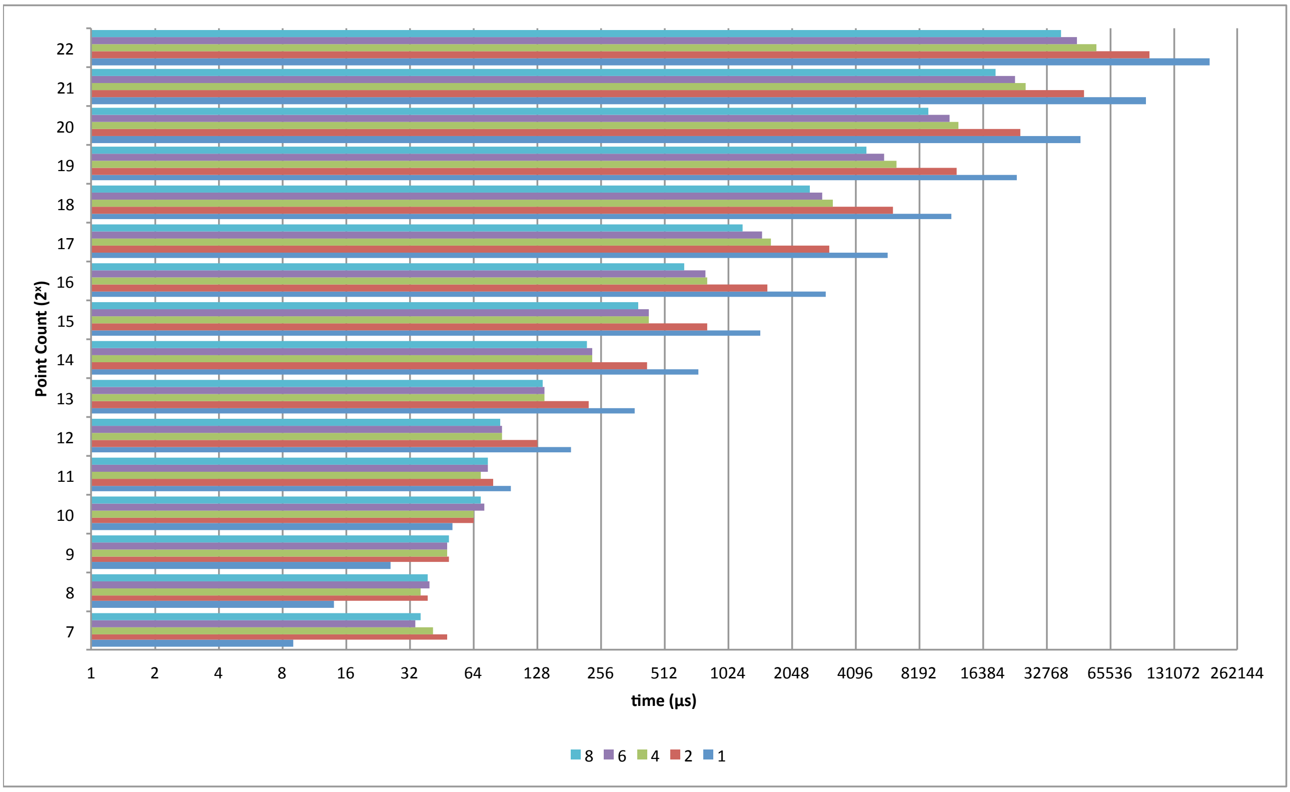
✳ Version Scala#2

The last implementation of this Monte Carlo method uses the currently in development Work-stealing parallel collections framework. The benchmark is also parametrized by *pc* and *N* and designed to use all the 8 available cores of the computer described earlier.

✳ Performance

Impact Of Available Cores

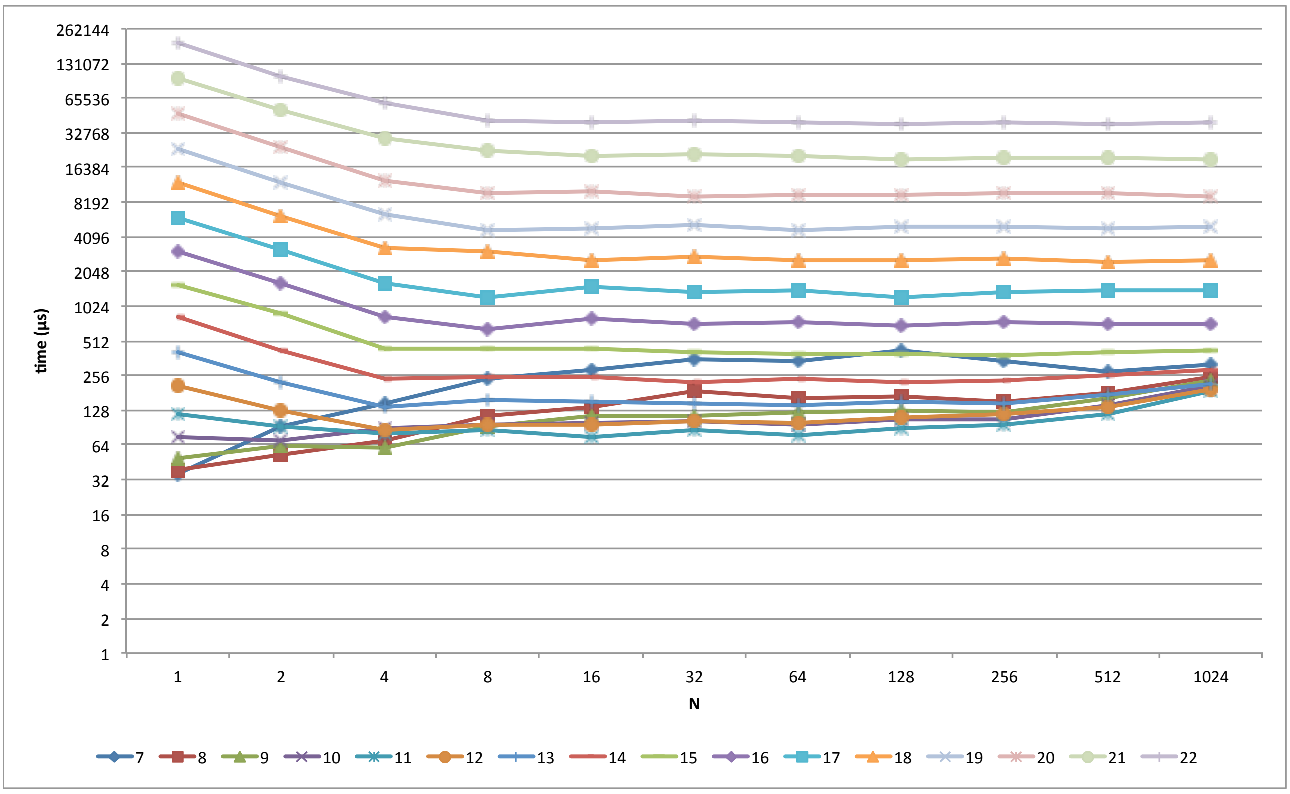
Scala#1 is parameterized by the number of used cores during the computation. The following graph shows that with relatively small *pc* one core is more efficient than *8* – i.e. the sequential implementation is faster – but with higher *pc* values the multithreading significantly enhance the processing time. The speed up factor is relatively close to *8* with *pc* = 2^{22}. The value of *N* was chosen to be the best for each *pc* and *cores* settings for this statistic.



Performance regarding the available number of cores on Scala#1

Impact Of Workload Per Thread

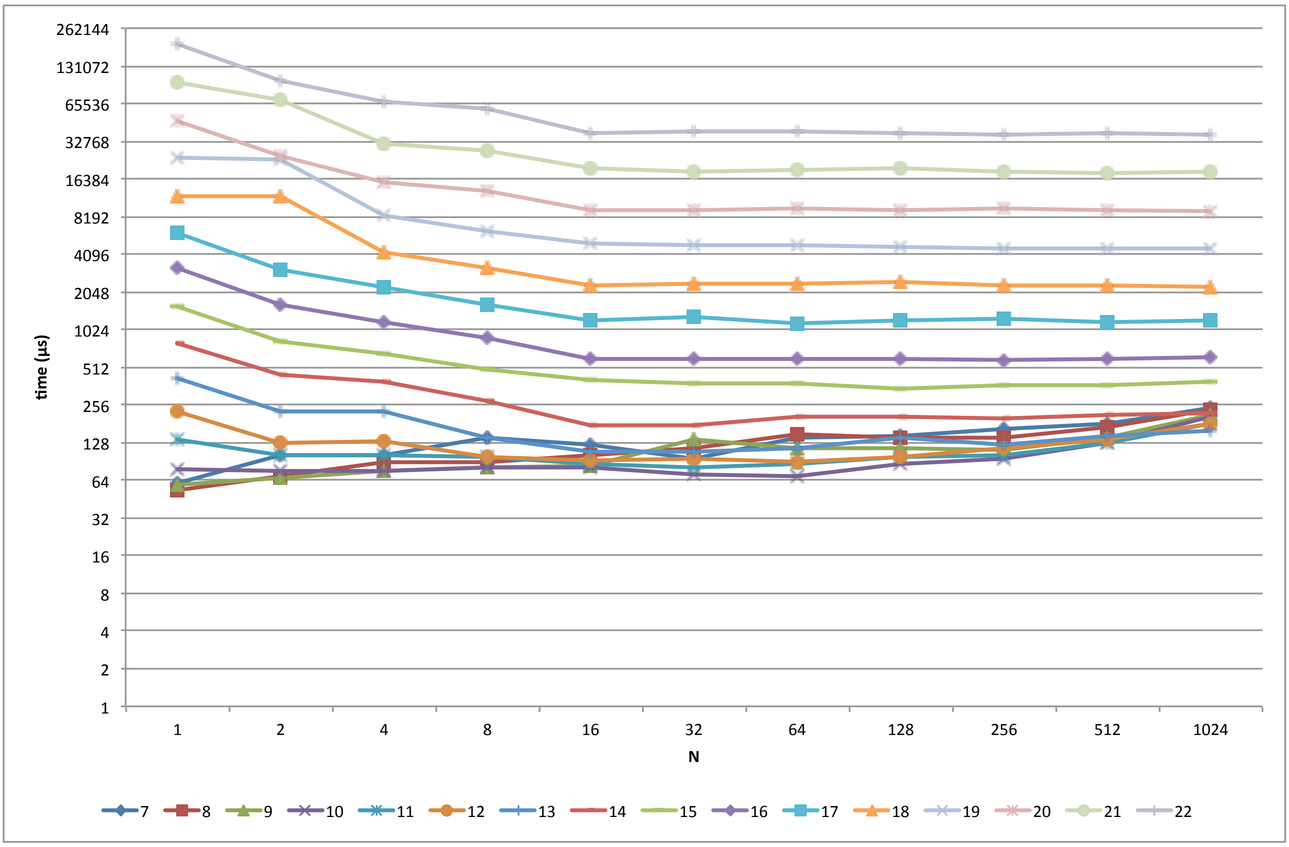
In a similar fashion as above with Thrust#2, we can plot the measurements to understand what is the effect of *N* on the processing time. Here is a graph for the first implementation with 8 cores:



Performance regarding the workload by thread with 8 cores on Scala#1

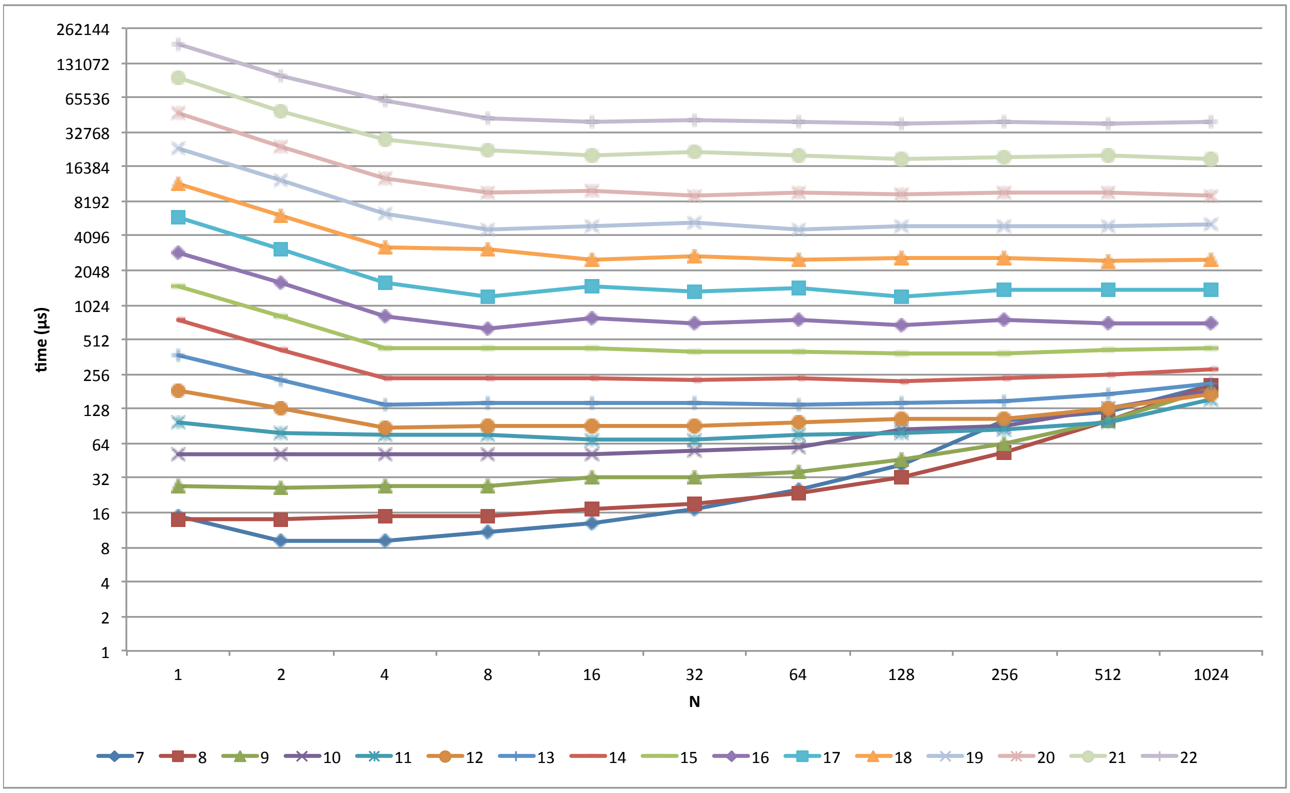
With low point count, that is less than 2^{10}, the software is faster with a value of *1* for *N* but for higher point count a value of *16* is good. Higher values of N don't change anything regarding the performance.

With the second implementation Scala#2, which uses exclusively 8 cores, the influence of *N* is about the same. For *pc > 2^{10}* a value of *32* is the best for higher performance:



Performance regarding the workload by thread on Scala#2

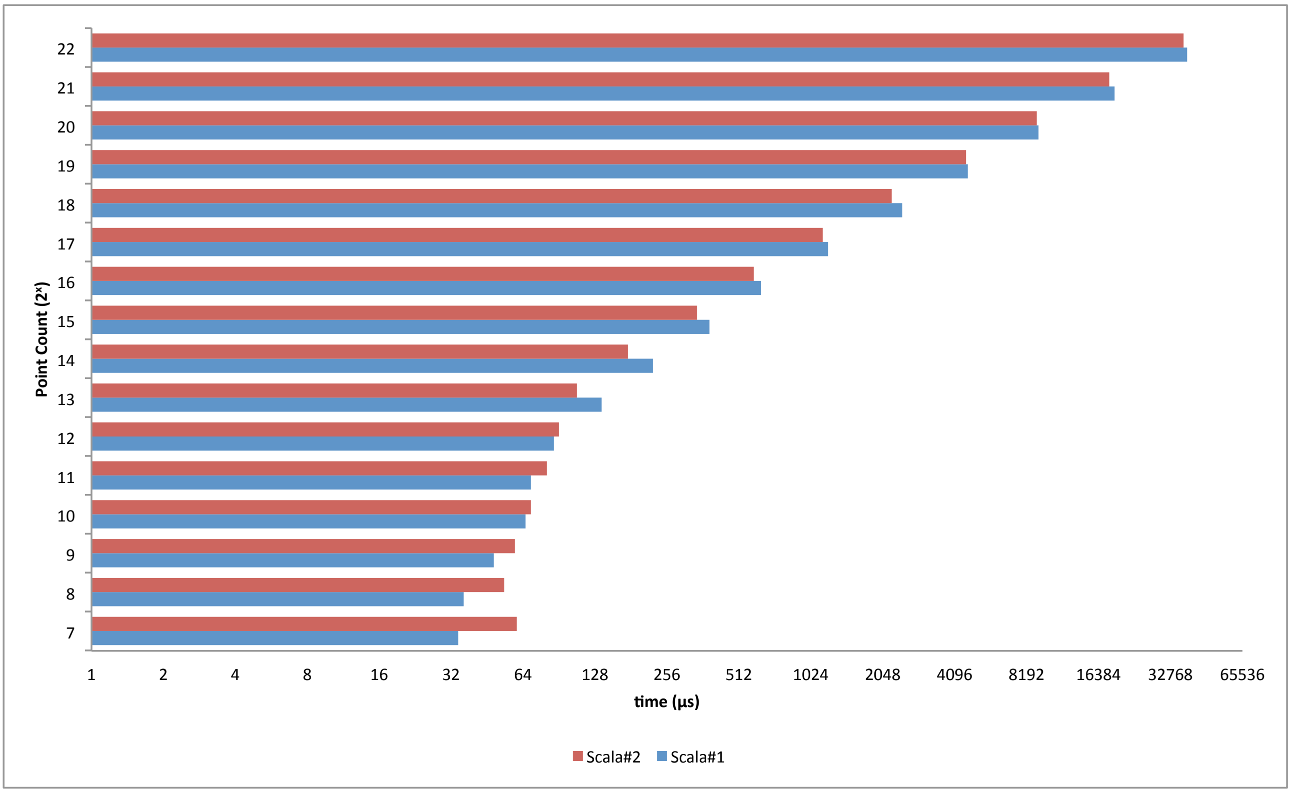
The next chart displays the impact of *N*, with the best number of cores, on the first version. In this more general case *N = 16* is best for any number of random generated points.



Performance regarding the workload by thread with the best number of cores on Scala#1

Workstealing framework VS Scala Parallel Collections

When comparing the performance of the current parallel collections of Scala – with the best values for *cores* bigger than *1* – and the work realized with the Workstealing framework we can see that for small *pc* the first implementation finishes faster than the second one. However, with bigger *pc* the Workstealing implementation is slightly faster.



Performance comparison between Scala#1 and Scala#2

For the two versions their respective best value for *N* was selected. The Workstealing framework would probably be faster with less cores with low *pc* values.

5.5. Synthesis

The Monte Carlo method for computing π is relatively easy to implement in those three technologies. The following table summaries the SLOC metrics.

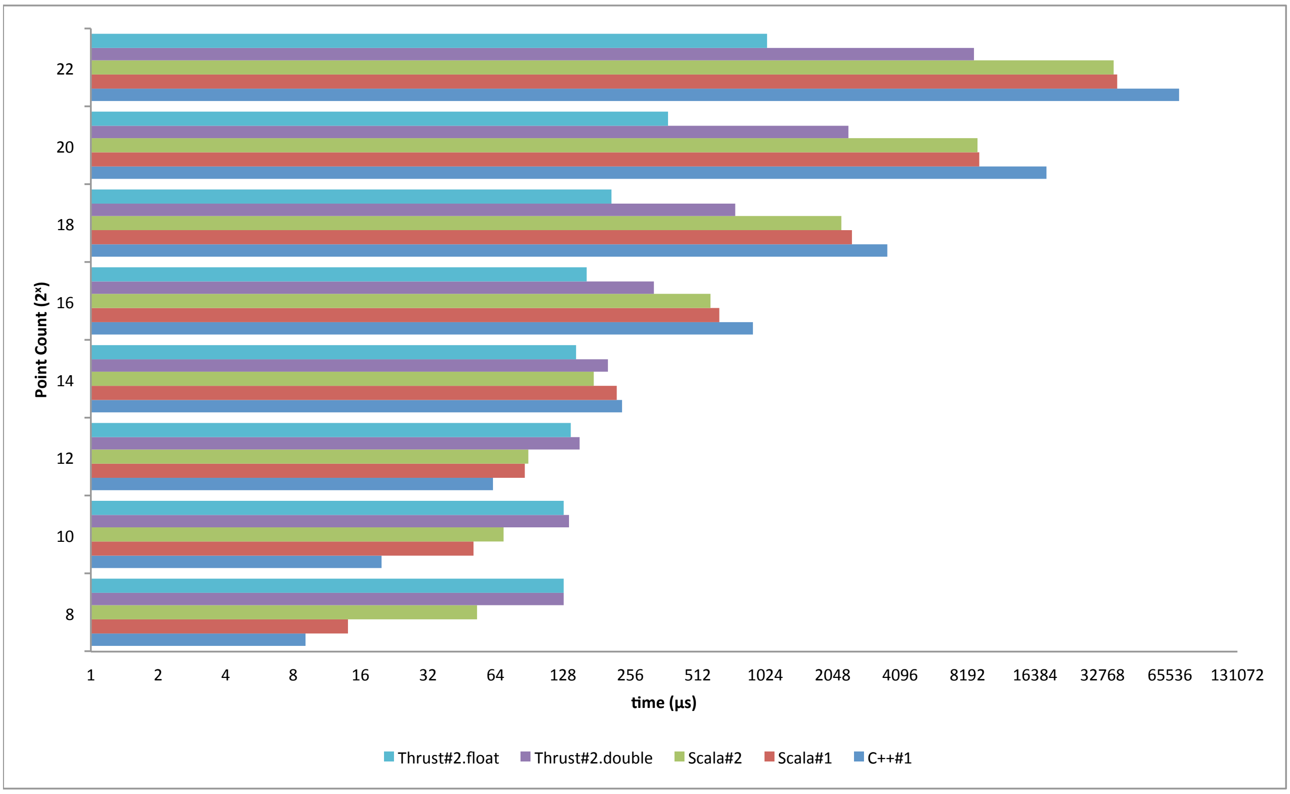
C++#1	35
C++#2	36
Thrust#1	60
Thrust#2	65
Scala#1	36
Scala#2	28

Number of line of code

As we can see the Scala and C++ implementations are nearly twice shorter than their Thrust counterpart. However, for a software that sort this information is not really conclusive.

Since the five programs are really short and similar we cannot differentiate much their complexity from an algorithm perspective. Nevertheless, when it comes to the code on itself, it is easy for the developer to forget a *__host__* *__device__* before the declaration of a function and get an cryptic and interminable error message from the compiler. It is challenging to manage correctly the randomness of generated points. Moreover developing the C++ or Scala versions was faster, easier and less error-prone.

The critical metrics when it comes to performance is the processing time. The next chart reflects the performance of the different implementations – C++#2 and Thrust#1 are not displayed since they are significantly slower – with their best settings (*N* or *cores*) for each of them:



Performance comparison between all implementations with their respective best parameters

Globally, every implementation has an initial overhead that is amortized and become less important with high *pc* values. For example, the minimum computation time required by the Thrust implementations is roughly 128µs. It can be explained by the transfer of data between the CPU and GPU systems.

The workstealing framework looks less attractive than the current version of the parallel collections on this graph. However, this is probably because the Scala#2 was only run with 8 cores, which is too much for low point count like 2^8 where a sequential collections is faster.

Finally, with $pc = 2^{22}$, the Thrust#2.float is about 70 times faster than C++#1, the slowest for this setting. Moreover it beats Scala#2 by a factor of approximately 35.

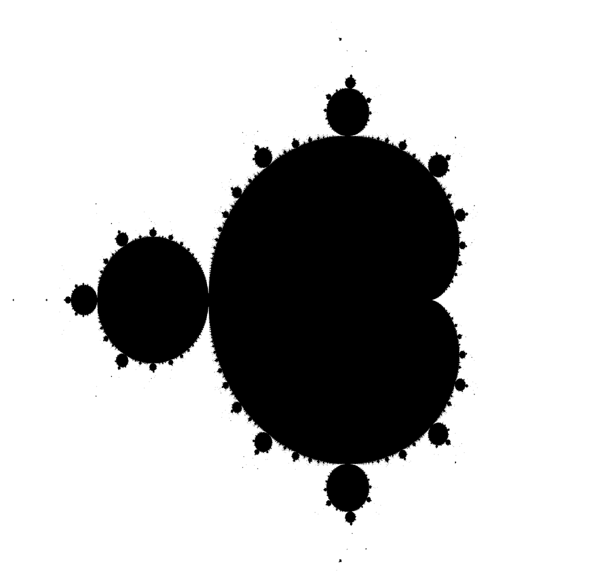
6. Benchmark: Mandelbrot Set

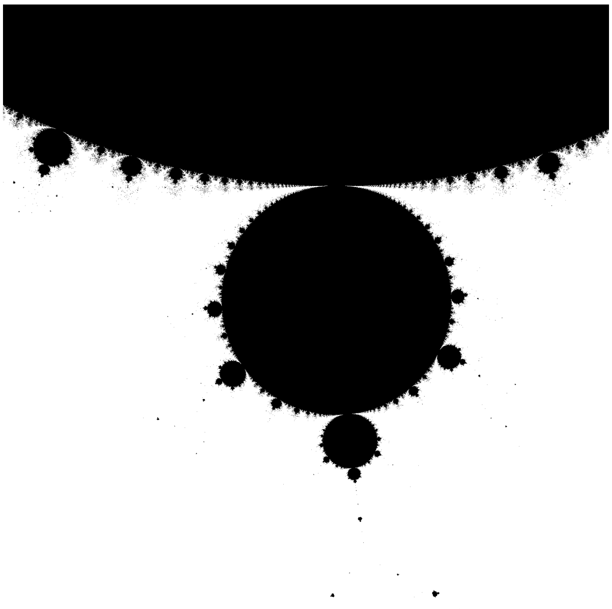
6.1. Algorithm

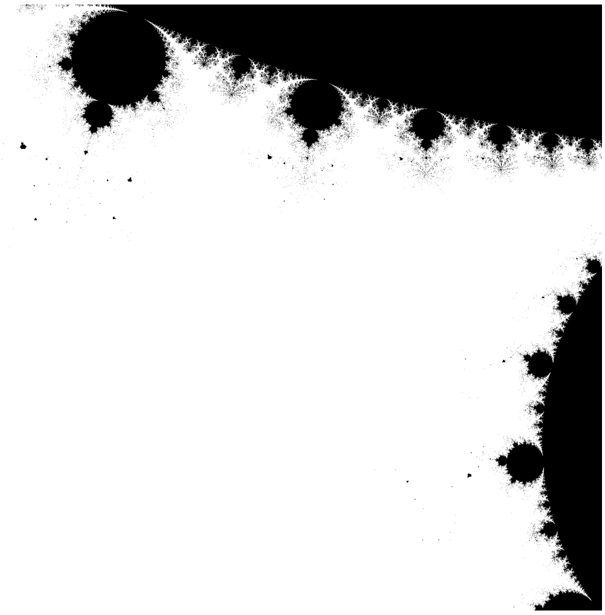
The formal definition of a Mandelbrot Set M states that, for a given complex number c, a point z in the complex space belong to M if the operation $z \longmapsto z^2 + c$ can be chained n times, for every n in $\mathbb{N}$. The common software implementations define a precision number to represent the upper bound of n.

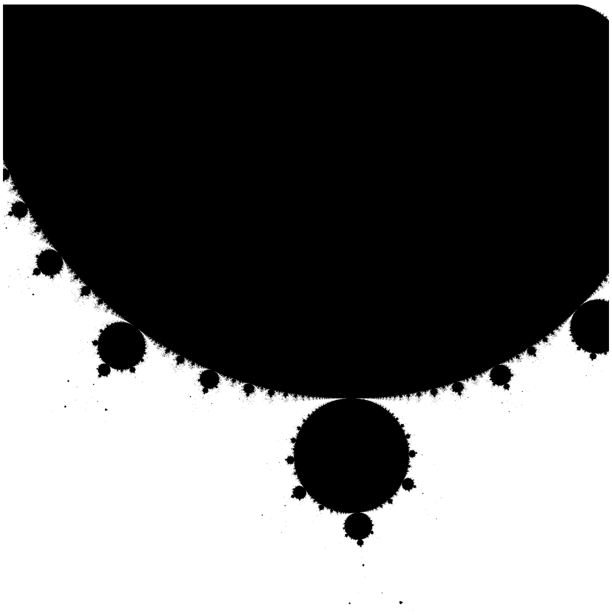
The different implementations were benchmarked on different complex sets and with variable image sizes and set precision levels.

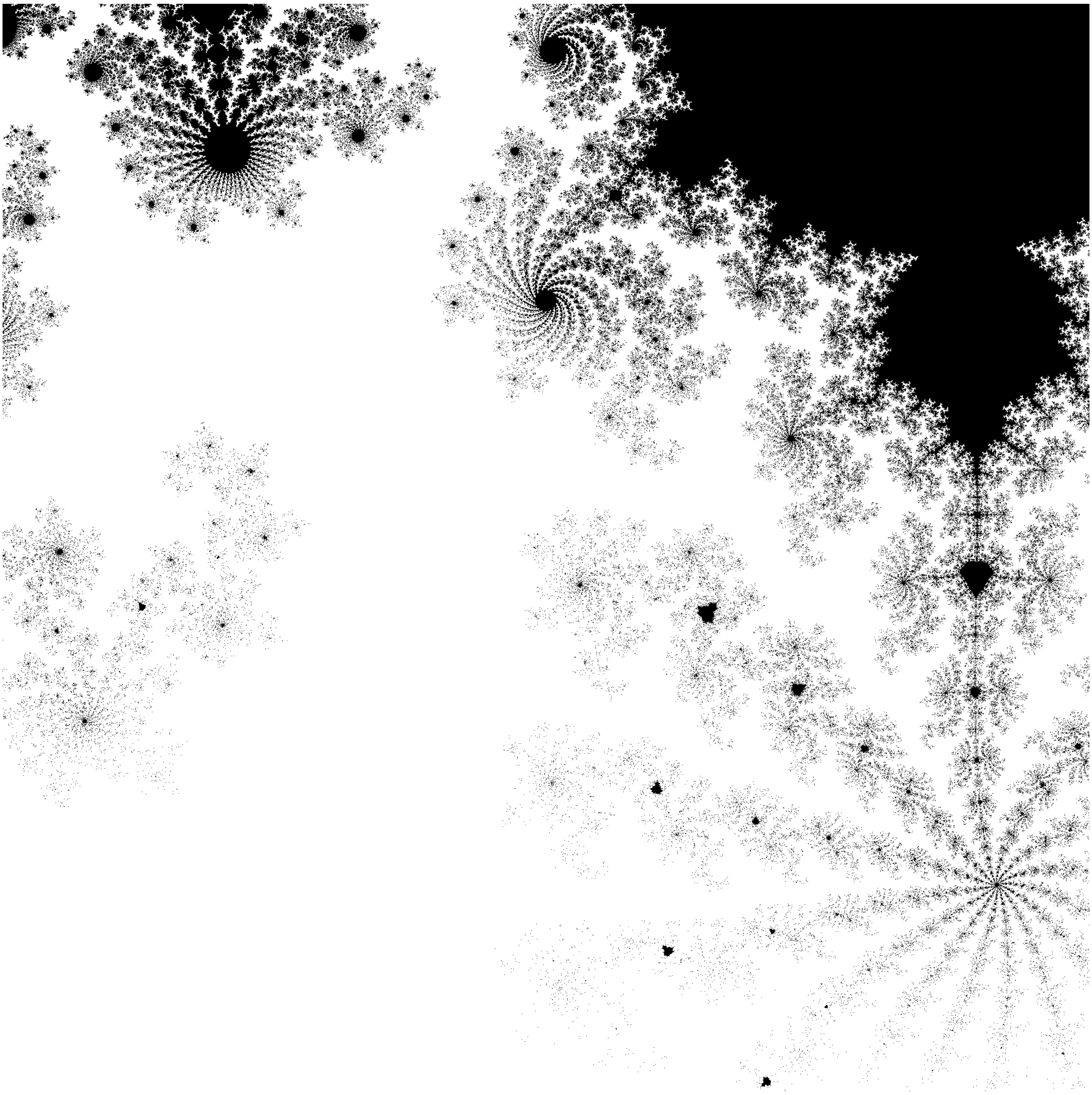
Here are the 5 sets rendered with an image size of 2000 px and a precision of 250 iterations:











Their respective complex set are, from left to right and top to bottom:

{(-1.72;1.2);(1;-1.2)}
{(-0.7;0);(0.3;-1)}
{(-0.4;-0.5);(0.1;-1)}
{(-0.4;-0.6);(-0.2;-0.8)}
{(-0.24;-0.64);(-0.26;-0.66)}

Each 2-tuple of points defines a rectangle area in the complex plane.

6.2. Implementations

The three implementations don't differ very much from an algorithm point of view: they all iterate on a container representing the image and compute for each point the Mandelbrot function with the given precision to determine whether the point is inside or outside the set.

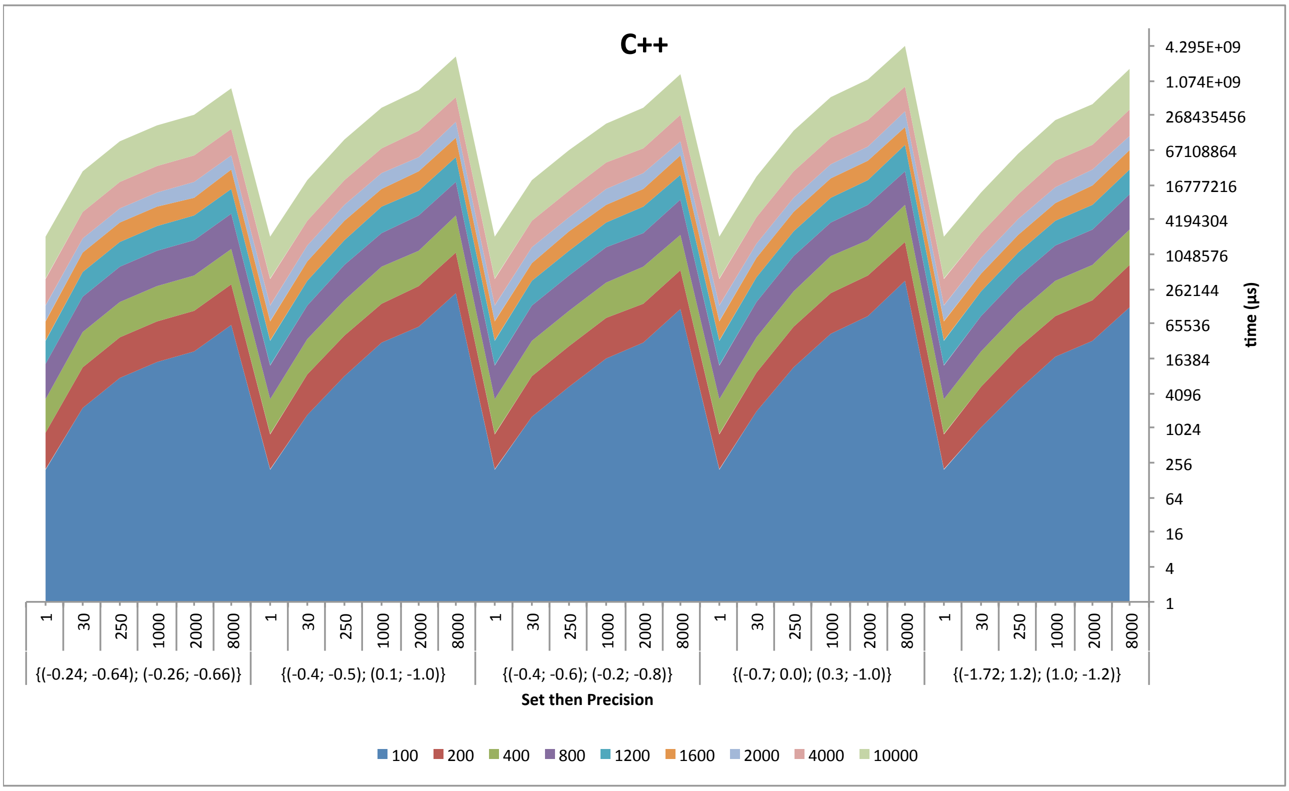
The Scala implementation is declined into two versions: the first, denoted Scala#1 below, uses the current parallel collections, the second, denoted Scala#2 afterwards, uses the Workstealing framework.

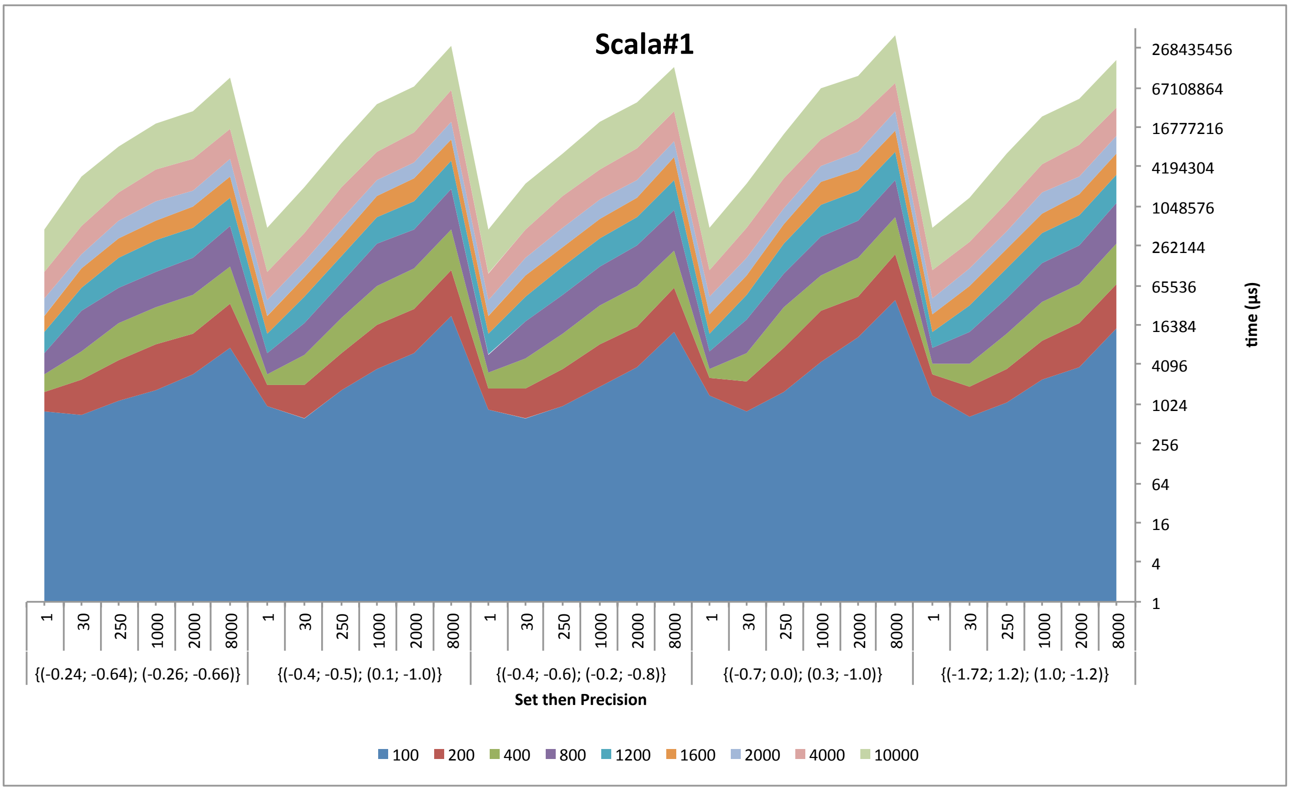
The Thrust version was set to use single precision floating-point format and the C++ one is still sequential. Time measurement is presented µs on a logarithmic axis of base 2.

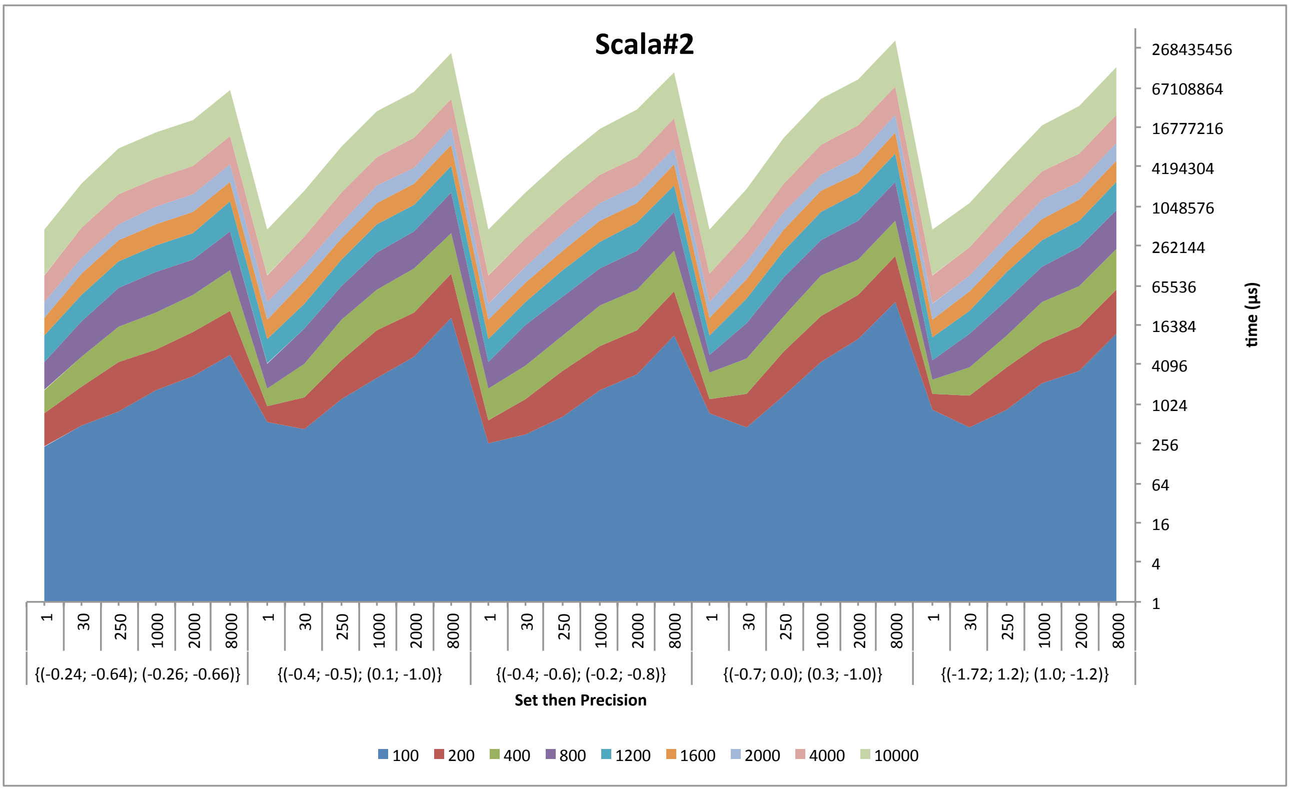
6.3. Synthesis

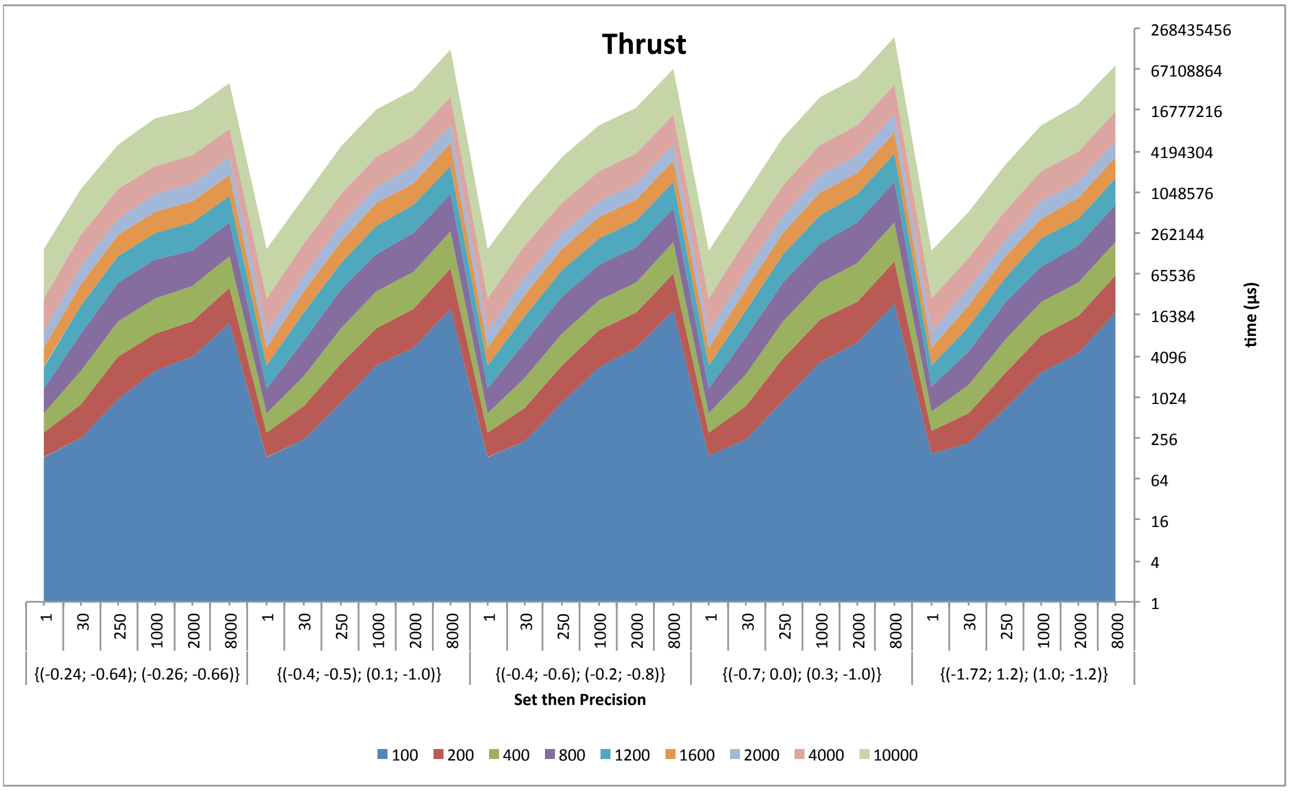
✳ Correlation With Side

The following four graphs show that, for the five set, the processing time of the four versions is linearly correlated by the size of the image: each colored area represents a different value of the image's side. The charts also demonstrate that the precision level has the same effect on each set with every implementation.



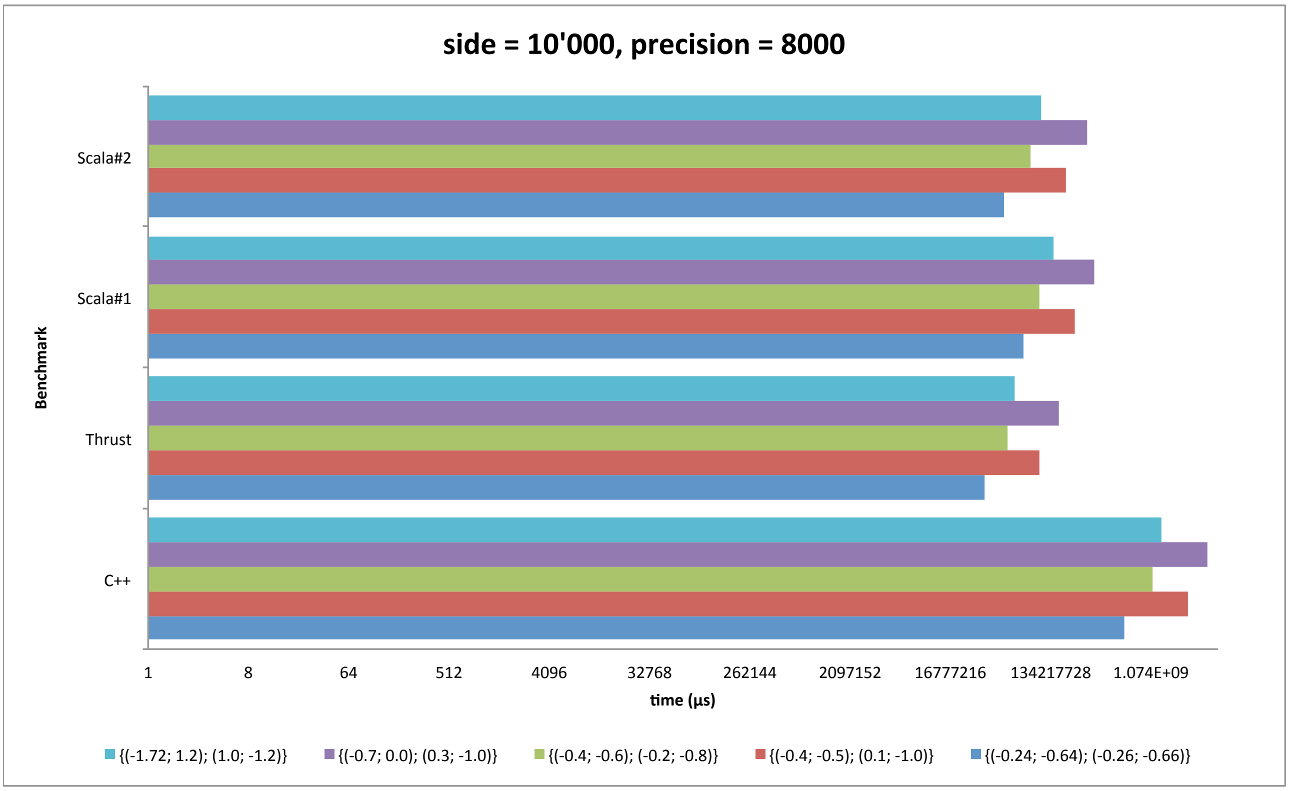

Scala#1
268435456
67108864
16777216
4194304
1048576
262144
65536
16384
4096
1024
256
64
16
4
1
time (µs)
1
30
250
1000
2000
8000
{(-0.24; -0.64); (-0.26; -0.66)}
{(-0.4; -0.5); (0.1; -1.0)}
{(-0.4; -0.6); (-0.2; -0.8)}
{(-0.7; 0.0); (0.3; -1.0)}
{(-1.72; 1.2); (1.0; -1.2)}
Set then Precision
100
200
400
800
1200
1600
2000
4000
10000


Scala#2
268435456
67108864
16777216
4194304
1048576
262144
65536
16384
4096
1024
256
64
16
4
1
time (µs)
1
30
250
1000
2000
8000
{(-0.24; -0.64); (-0.26; -0.66)}
{(-0.4; -0.5); (0.1; -1.0)}
{(-0.4; -0.6); (-0.2; -0.8)}
{(-0.7; 0.0); (0.3; -1.0)}
{(-1.72; 1.2); (1.0; -1.2)}
Set then Precision
100
200
400
800
1200
1600
2000
4000
10000



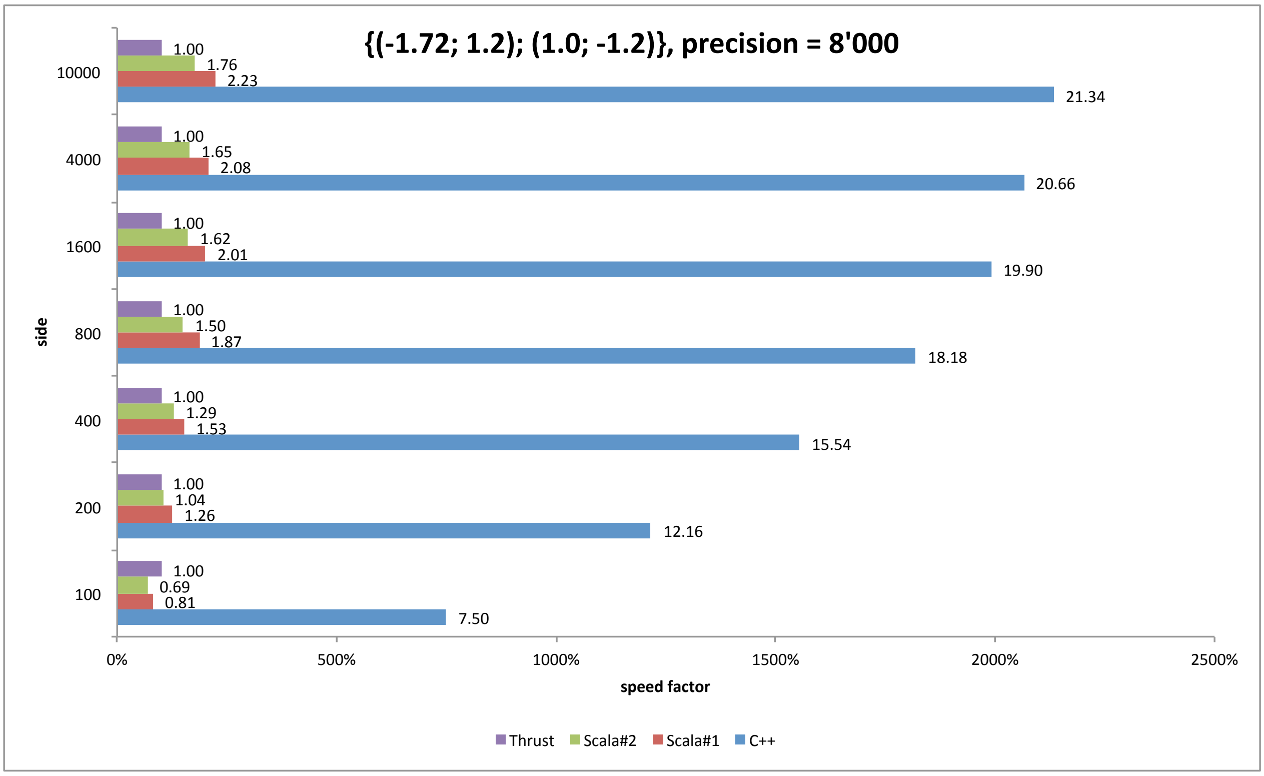
✴ Set Performance

We can also analyze the performance of each set:



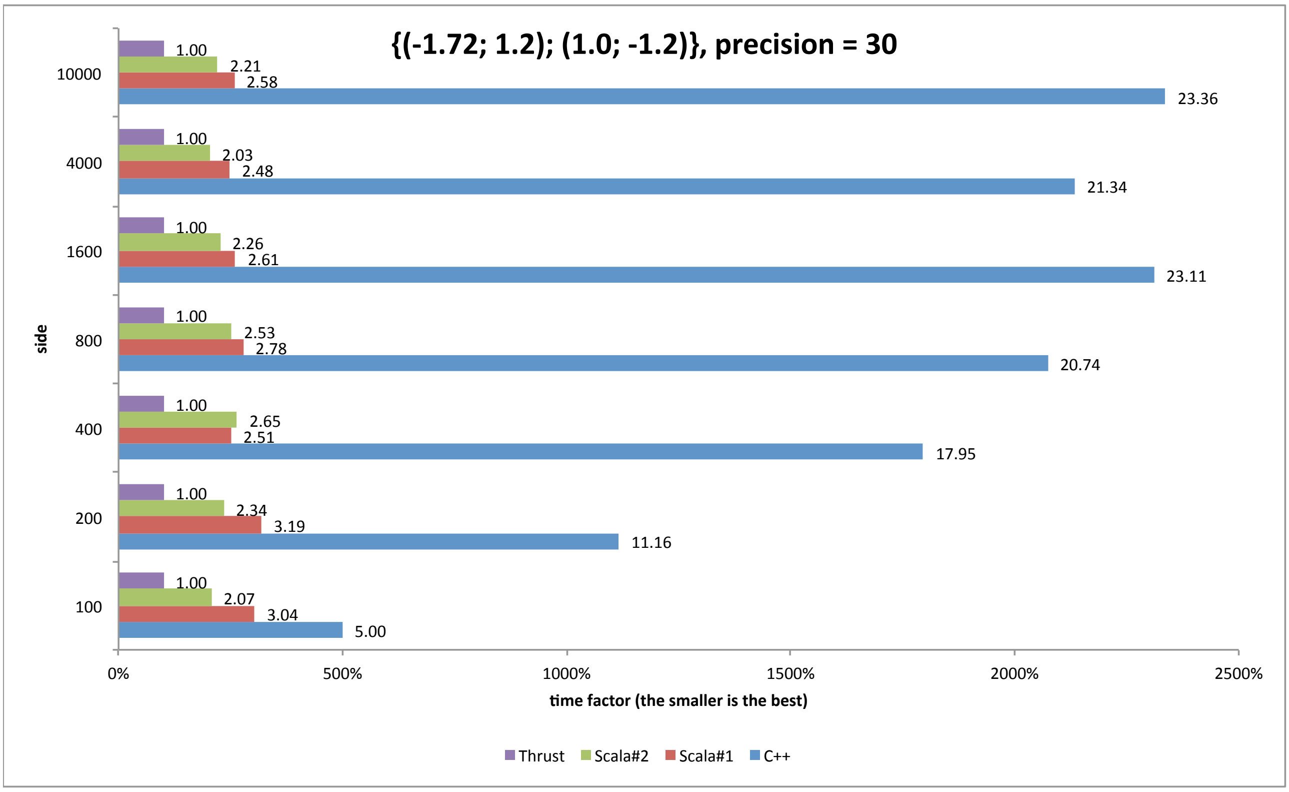
It appears clearly that the relative performance of the complex sets is the same with each implementation with an image size of 10’000 by 10’000 pixels and a precision of 8’000 iterations. Also Thrust looks faster on this logarithmic scale of the time axis.

We can confirm that the sequential implementation written in C++ is more than 21 times slower than the GPU implementation and that the latter goes 2.23 times faster than the current parallel collections of Scala with an image’s side of 10’000 and a precision of 8’000. The Workstealing framework is already a little bit faster and is beaten only by a factor of 1.76 by Thrust.



Similarly as seen before, small data set are slower on GPU.

The difference of speed factor between a precision of 8’000 and a precision or 30 for the same set, which is displayed below, can be explained by the divergence phenomenon that happens on GPU. A warp of 32 threads will process close points of the Mandelbrot set. However, some points can quickly be considered outside the set but some need much more iterations to finally realize they are outside, or, after the maximum iterations authorized by the chosen precision, considered to be inside the set. When a high precision such as 8’000 is selected the divergence inside a wrap can be very significative in term of performance when a warp computes a group of points near the border of the set. The relative performance of Thrust with a precision of 30 is slightly better compared to Scala and C++ implementations and significantly beats them with low image size such as *side = 100* while it offers worse performance with a high precision of 8’000:



✴ Additional Comments

As with the previous benchmark, the code of each implementation is relatively short: between 100 and 200 lines of code in total, benchmark code included.

Their respective complexity are however slightly different. Both C++ and Thrust version are a little bit simpler when it comes to complex number arithmetic. When objects are introduced in the Scala versions to manipulate complex number with a higher level of abstraction the performance drop significantly: many objects are created and destroyed in a short amount of time, because of the immutability pattern, so that the garbage collector must be run during the computation more often. Hence, to keep the performance high in Scala a lower level approach is needed and complex number must be computed by using *var*s.

On the other hand, because the GPU memory is smaller than the CPU one, the algorithm must be modified to prevent *out of memory* issues: in the case of the Mandelbrot set computation, we had to launch several GPU kernel to compute the total image part by part when the side or precision level were too high.

7. Benchmark: Genetic Algorithm

7.1. Algorithm

Some Artificial Intelligence algorithms try to mimic phenomenons present in the Nature. The branch of Genetic Algorithms is typically one of them.

This kind of algorithm takes its origins from the observations of the evolution of animal populations on a macroscopic level such as the phenotype of individual members but also includes the microscopic behavior of mutating DNA segments such as the crossover of two individual when they produce a new one or when an individual ages and mutate its DNA segments.

There exist many variants of such algorithm. The three implementations described below use the following steps:

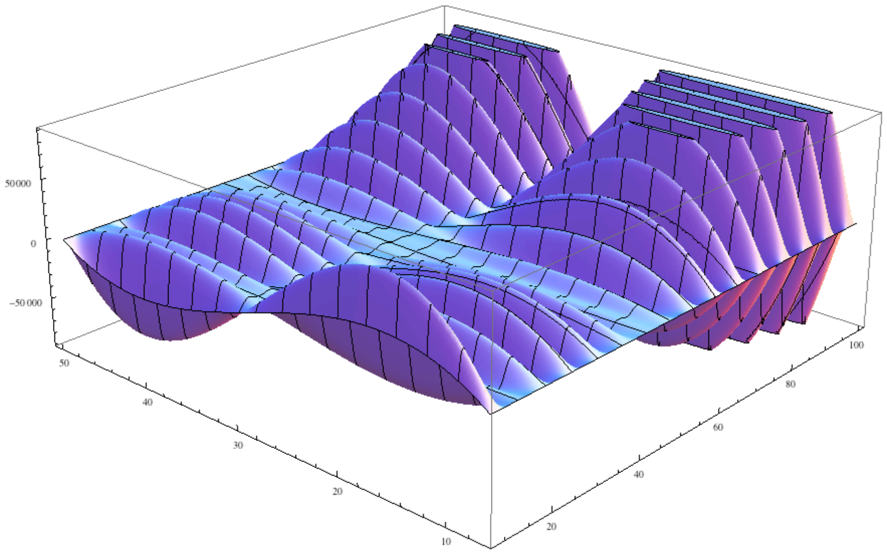
(1) An initial population is randomly generated;
(2) Each entity of the population is evaluated according to a fitness function
and the *K* worse individual are removed;
(3) From the remaining population, *M* are selected randomly and mutated;
(4) Then *C* individuals are generated with a crossover from two random individuals each;
(5) The population is increased by *N* newly generated entities
to keep a constant population size;
(6) If the new population meets some termination criterions then the application stops;
Otherwise the algorithm loops back to step (2).

This algorithm is still relatively easy to understand and implement; yet it is considerably more complex than the previous ones. The computation time is also generally longer so millisecond are used as time unit for the graphs below.

In these implementations, the Genetic Algorithm was configured to find the local maximum of the following function for a given range defined by the relation 9 ≤ x ≤ 100 and 7 ≤ y ≤ 50:

$$\frac{\sin(x-15)}{x}(y-7)(y-30)(y-50)(x-15)(x-45)$$

Here is the function plotted on this specific range:



The x-axis is on the right while the y-axis in on the left.
The z-axis is truncated when the value of the function at a given *(x; y)* position is too big.

This Genetic Algorithm is particularly precise for this problem: the three softwares detailed below find a maximum at *(98.2728; 16.5769)* while Mathematica thinks the maximum is at *(82.5735; 41.4231)*. Apparently it gets stuck on a non-optimal local maximum when using its numerical solver.

All implementations use the same population parameters for the algorithm. Therefore there is less data to analyze for this benchmark.

7.2. C++ Implementation

This first version intensively uses the new lambda system to create a very generic API based on functional programming. The generator, mutation and crossover are all customizable without changing the internal behavior of the algorithm.

The execution time of this algorithm is quite stable and the implementation is relatively straightforward.

7.3. Scala Implementation

The next implementation is extremely similar to the C++ one with its functional approach. The Workstealing framework was selected for the parallel computations instead of the default parallel collections.

As with the C++ version, this one has a stable processing time and a relatively similar code complexity.

7.4. Thrust Implementation

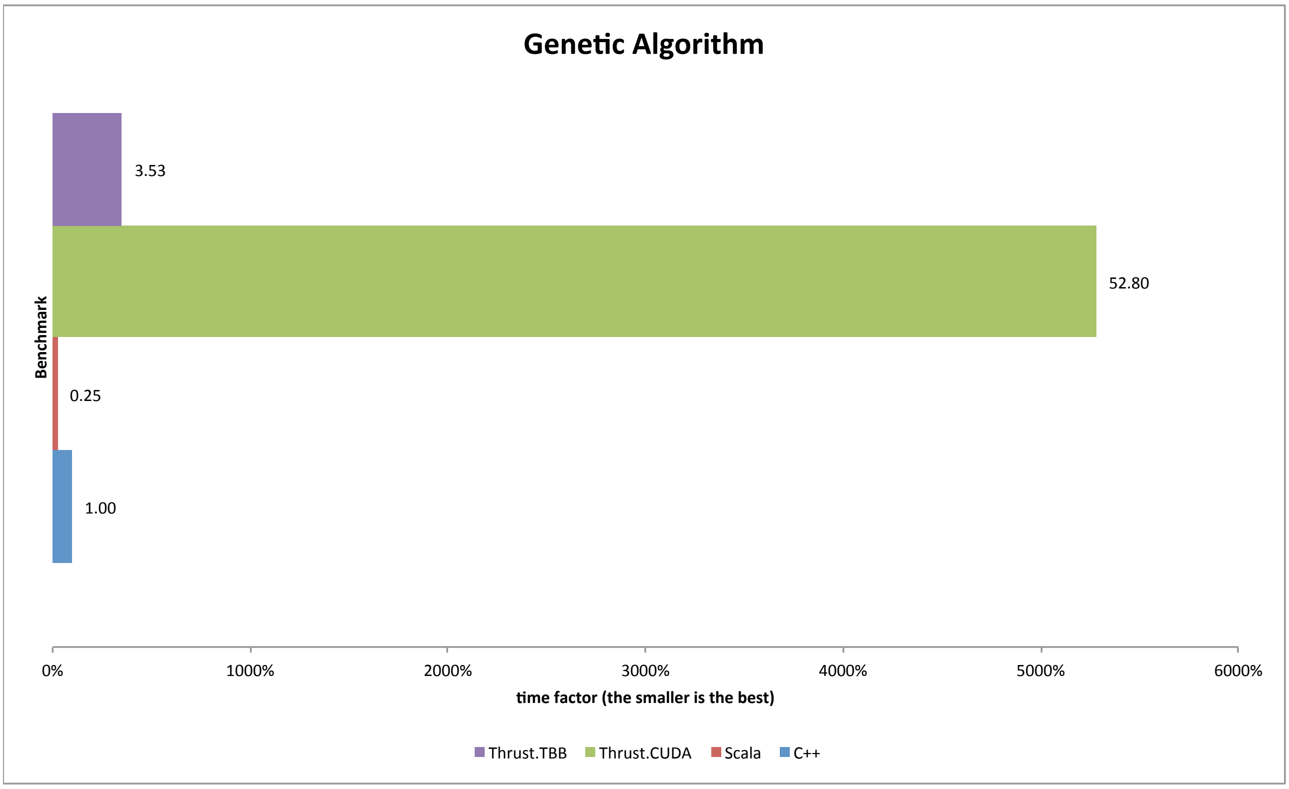
We initially tried to create a relatively close generic Genetic Algorithm for Thrust. Unfortunately, without C++11 support for lambda calculus it was extremely complicated to create a generic interface. That is why the final Thrust version implements the algorithm implements the generator, mutator, ..., directly inside the main object.

In an earlier version the entities and their respective fitness value were stored in a pair and the whole population was an *Array of Structure*. Later, the implementation was optimized to use *Structure of Array* with the method exposed above in the presentation of Thrust.

The performance of each implementation is detailed below. However, the performance of the GPU application is really bad and unstable compared to the sequential C++ or multithreaded Scala versions. This last program was also executed with TBB backend to have another point of comparison.

7.5. Synthesis

The chart below illustrates how fast the Scala implementation is in comparison to the C++, CUDA and TBB ones:



As we can see the Scala version goes 4 times faster than the C++ one. Thrust with CUDA backend is way behind with a processing time 52 times bigger than the sequential implementation. When using TBB backend the performance are much better than with the GPU is used but are still behind by a factor of 3.5 in comparison to the C++ version.

The poor behavior of Thrust can be explained by the important memory operations that occur: the population vector is copied between the host and device system during every round of the main loop. Since the population is relatively small – only one thousand entities – and the kernel functions are very short and transferring memory from the two systems is slow, the

GPU high latency execution cannot be compensated. Hence, the GPU is not at its advantage for such algorithm.

Moreover, the code is more elaborate and complex when using Thrust. The number of code line is relatively the same as the C++ version but the use of *SoA* with less scalable, reusable and straightforward code make it less pleasant and more expensive to develop. On the other hand, the Scala code is 30% shorter than the C++ one, goes directly to the point with its lightweight syntax and is extremely scalable with a low development cost.

8. Conclusion & Future development

8.1. Synthesis: Thrust VS Scala

When it comes to comparing technologies as different as GPU framework and Scala parallel collections, many factors have to be considered. Comparing directly the execution speed of two implementations of the same algorithm doesn't make much sense since one implementation might have cost months of development by a team of experts and the second only a few hours with higher level framework by an amateur during his free time.

A point that was not discussed earlier is the time required to get to know a given framework. Today, there exist a bunch of books on both Scala and CUDA programming. The documentation provided by Typesafe or NVIDIA for Thrust are quite complete and easy to navigate. Moreover they usually include code snippets to illustrate a particular method or idea. Anybody who already has some strong programming experience is at least capable of using the basic features of both Thrust and Scala within a reasonable amount of time.

However, when it comes to debugging Thrust is far behind. GPU debugger are currently not as handy as GDB or LLDB for C++ and JDB for the Java runtime. Moreover, a GPU application directly communicates with the kernel of the graphical device. It means that a malformed software can crash the kernel and the host Operating System which can cause sever data loss.

We have also experienced that random number generation with highly parallelized threads is harder to handle correctly with Thrust than with Scala, without considering the fact that the C++ API, introduced with the last standard and reimplemented by Thrust, has a lower level to allow better mathematical configuration of the random distributions.

It was demonstrated that Thrust can be extremely faster than Scala, even with its parallel computation power, when it comes to simple and short algorithms like Monte Carlo method or Mandelbrot set computation on a modern personal computer. In those scenarios the GPU power can be easily used to improve the processing time by huge factors while keeping the application code short and understandable.

With slightly more complex algorithm, like the Genetic Algorithm to compute the local maximum of a function, the GPU computation capabilities are worse than common multithreading on CPU. It can also be slower than a sequential implementation of the same problem.

This last benchmark might have been implemented from another perspective and designed completely differently to improved the performance, slightly or considerable. However, this would have required more manpower and a higher development cost and the performance-cost ratio would have been still much lower than with Scala in our opinion.

The software rule «*the tools must be carefully chosen according to the needs*» is again verified.

Although at run time it has no influence, the time spent to compile C++ or GPU softwares is much longer than with the Scala compiler. This however also increases the cost of development since it takes longer for a developer to test even the slightest modification of his software.

8.2. Synthesis: Workstealing Framework

The different benchmarks confirm the fact that the Workstealing framework offers better performance than the current parallel collections with good speedups and a better workload balance between cores and threads. Its API is still a little bit limited – e.g. the *map* function doesn't exist in trait *scala.collection.workstealing.ParIterableLike* – but we can hope that it will be completed in a near future.

8.3. The missing analyses

Although this paper presents several benchmarking applications with varying performance measurements and constraints it would be interesting to get a wider look at GPU computing and include other frameworks like C++AMP or more benchmarked algorithms, e.g. Traveling Salesman Problem, Text Segmentation, or micro-benchmarks to measure more closely on GPU devices the efficiency of sorting, counting, reducing, etc...

A precise study of the material cost currently available on the market should also be performed to compute the computational power/price ratio. This study can also include the analysis of CPUs and GPUs clusters.

8.4. Future developments

It would also be good to study the upcoming improvements of the C++ language and its standard library that are coming with the C++14 standard, to compare more precisely the core features of Scala and C++, and anticipate the development of Thrust regarding the C++11 norm, cluster support and *future*[2] support for cluster and multi GPUs computing. Moreover the Thrust team wants to improve the API interface and provide an alternative to the common C++ iterator system with the idea of ranges presented by Andrei Alexandrescu that can potentially make code cleaner.

There already exists a prototype of Scala compiler for LLVM. If this project is successful it would be great to allow developers to make part of their code executable on GPU. Another development of the Scala platform could be to use heuristics and decide at run time with JIT technologies if a particular block of code should be executed in parallel on GPU. This would of course be hard to implement but the performance improvement are really important in some situations.

[2] or *promise*

9. References

- Hardware specifications: http://support.apple.com/kb/SP653
- Homebrew website: http://mxcl.github.io/homebrew/
- Oracle's Java download page: http://www.oracle.com/technetwork/java/javase/downloads/index.html
- Scala homepage: http://www.scala-lang.org/
- Scalameter homepage: http://axel22.github.io/scalameter/
- OpenCL 1.2 specifications: http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf
- Heterogeneous Parallel Programming class on Coursera, by Prof. Wen-mei W. Hwu from University of Illinois: https://class.coursera.org/hetero-2012-001/class/index
- NVIDIA's CUDA homepage: https://developer.nvidia.com/category/zone/cuda-zone
- Balaji Vasan Srinivasan introduction to Graphical processors and CUDA, and the «New» Moores' Law: http://www.umiacs.umd.edu/~ramani/cmsc662/GPU_November_10.pdf
- Realtime GPU Audio, by Prof. Bill Hsu and Marc Sosnick-Pérez from San Francisco State University: http://queue.acm.org/detail.cfm?id=2484010
- Thrust homepage: https://developer.nvidia.com/thrust
- Microsoft's C++AMP homepage: http://msdn.microsoft.com/en-us/library/vstudio/hh265137.aspx
- CUSP homepage: http://cusplibrary.github.io/
- Monte Carlo method on Wikipedia: http://en.wikipedia.org/wiki/Monte_Carlo_method
- Mandelbrot Set on Wikipedia: http://en.wikipedia.org/wiki/Mandelbrot_set
- Genetic Algorithm on Wikipedia: http://en.wikipedia.org/wiki/Genetic_algorithm
- Introduction to Genetic Algorithm, by Nobal Niraula: http://www.slideshare.net/kancho/genetic-algorithm-by-example
- Creating a genetic algorithm for beginners, Lee Jacobson: http://www.theprojectspot.com/tutorial-post/creating-a-genetic-algorithm-for-beginners/3
- Genetic Algorithm for Variable Selection, by Jennifer Pittman: http://www.niss.org/sites/default/files/04%20Jennifer.ppt
- «Parallel Machine Learning Implementations in Scala» by Pierre Grydbeck: http://lamp.epfl.ch/files/content/sites/lamp/files/teaching/student-project-reports/semester/PierreGRYDBECK.pdf

Benchmarking technics and issues

- «Micro-Benchmarking Done Wrong, And For The Wrong Reasons» by Sasha Goldshtein: http://blogs.microsoft.co.il/blogs/sasha/archive/2012/06/22/micro-benchmarking-done-wrong-and-for-the-wrong-reasons.aspx
- «Microbenchmarking C++, C#, and Java» by Thomas Bruckschlegel: http://www.drdobbs.com/184401976
- «PARALLEL COLLECTIONS, Measuring Performance» by Aleksandar Prokopec and Heather Miller: http://docs.scala-lang.org/overviews/parallel-collections/performance.html

- «Java theory and practice: Anatomy of a flawed microbenchmark» by Brian Goetz: http://www.ibm.com/developerworks/java/library/j-jtp02225/index.html
- «Profiling performance in Scala» by Trond Olsen: http://www.steinbit.org/words/programming/profiling-performance-in-scala
- «Boxing and unboxing in Scala» by Joachim Ansorg: http://www.ansorg-it.com/en/scalanews-001.html

Various

- «Iterators Must Go» by Andrei Alexandrescu: http://www.slideshare.net/rawwell/iteratorsmustgo
- «Akka and CUDA» by Jan Machacek: http://www.cakesolutions.net/teamblogs/2013/02/13/akka-and-cuda/
- «Solving the Traveling Salesman Problem using Branch and Bound» by Robert Clark and Danny Iland: https://parallel-tsp.googlecode.com/files/teamDharmaPresentation.pdf
- «High Performance GPU Accelerated Local Optimization in TSP» by Kamil Rocki and Reiji Suda: http://olab.is.s.u-tokyo.ac.jp/~kamil.rocki/rocki_pco13.pdf
- «C++11 - the new ISO C++ standard» by Bjarne Stroustrup: http://www.stroustrup.com/C++11FAQ.html
- OpenMP homepage: http://openmp.org/wp/
- gfxCardStatus homepage: http://gfx.io/
- SFML library homepage: http://www.sfml-dev.org/
- Fork of Scala that includes llvm backend, by Geoff Reedy: https://github.com/greedy/scala

Workstealing Framework

The Workstealing parallel collections framework for Scala is currently developed by several members of the LAMP. No public information was released yet.

Finally, the source code and datasets used for this paper, plus some additional materials, are available from the following git repository: https://github.com/mantognini/scala-vs-gpu/.