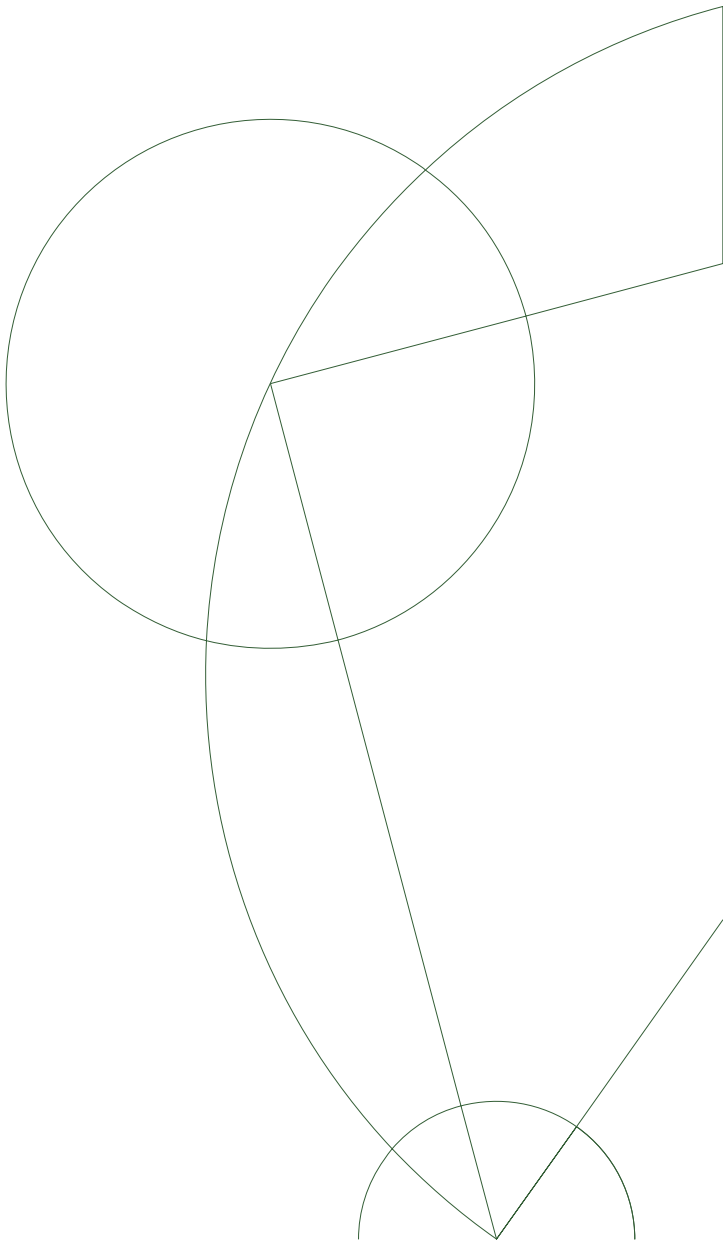
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Master's Thesis

Philip Carlsen – plcplc@gmail.com
Martin Dybdal – dybber@dybber.dk

Option pricing using data-parallel languages



Supervisor: Ken Friis Larsen

March 2013

Abstract

We explore the state of current functional data-parallel programming languages targeting GPUs, by means of evaluating a set of option pricing algorithms, namely the binomial option pricing algorithm [15] and the least squares Monte Carlo approach introduced by Longstaff and Schwartz [30], as well as related algorithms employed in Monte Carlo integration.

Among the surveyed languages we find both Nikola [32] and Accelerate [12] constrained by their expressiveness due to restrictions of the target platform.

We improve on the limitations we have found and the improvements take the form of additional primitive constructs for iterative array construction and a new suggested *cursor* for marking the division between sequential and parallel code, much inspired by constructs found in Repa [27].

We believe that there is much promise in continuing work on our suggestion, but we were unable to finish implementing and testing the extensions due to project time constraints. We give an analysis of what practical work remains.

Contents

Preface

This dissertation is submitted in fulfillment of the graduate education in computer science (*Datalogi*) at the University of Copenhagen, for Philip L. Carlsen and Martin Dybdal. In addition, the dissertation also serves as partial fulfillment of Martin Dybdal's 4+4 Ph.D. education at the University of Copenhagen.

The thesis has been supervised by Ken Friis Larsen and we would like to thank him for valuable supervision hours. We would also like to thank Geoffrey Mainland and Trevor McDonell for valuable email correspondences and quick responses in cases of problems with their libraries. Finally we would like to thank Cosmin Oancea, Christian Andretta, Jost Berthold, Martin Elsman, Andrzej Filinski for their input and help during brief discussions and email correspondences.

Chapter 1

Introduction

1.1 Background

Ever since the first electronic computers were built, there has been a wish for tackling larger and more complex problems, which in turn has created an increasing demand for performance improvements and increased programmer productivity. Performance improvements originate from improvements in either hardware or the employed algorithms. Programmer productivity, on the other hand, correlates with the features of the used programming language and programming environment (IDEs, debuggers, etc.). A language providing high programmer efficiency should make it easy to implement and reason about algorithms, make mistakes easily avoidable and when a mistake happens, make it easy to uncover its cause.

For a handful of decades, we have been able to increase performance through hardware improvements of sequential processors and we have thus been able to stick with more or less the same model of computation. Recently, hardware developers have faced physical barriers, making further performance improvements of sequential processors impractical, and they have had to go new ways to obtain the desired speed-up [47]. These new architectures call for new algorithms and models of computation, as well as new languages and programming tools to keep the complexity of software development at a tolerable level.

We will focus on software development for *graphics processing units* (GPUs), which in recent years have been found cost-effective alternatives to ordinary CPUs, for many other domains than computer graphics. Today they are employed in fields as diverse as bioinformatics, computational chemistry, medical image analysis, relational databases, computational finance and simulation in both engineering and the sciences [23, 24, 44]. Solutions to many problems in these domains can be expressed naturally as *data-parallel* algorithms, where each data item can be processed independently (often from a linear algebra specification of the problem). This is thus where GPUs popularity arise; executing such data-parallel tasks is the core capability of GPUs.

1.2 Data-parallel programming languages

Even though we think of graphics processors as *modern hardware*, the software development tools in widespread use for programming them are far from modern. CUDA [37] and OpenCL [35], the two main languages for programming GPUs are low-level languages with manual memory management, limited abstractions, and tedious hand optimization are often required. It is, for instance, important to streamline memory accesses to get decent performance. The low-level nature of these languages makes automated code analysis difficult. The NVIDIA CUDA compiler thus cannot make an optimization such as stream fusion (deforestation), that removes intermediate data structures and thus removes the need for very costly memory transactions. The programmer thus has to fuse programs by hand, sacrificing clarity as abstractions between individual parts has to be broken down and fused together. Also, the programmer has to be well aware of the exact hardware his program is to be executed on, thus making a tight coupling between program and hardware.

High-level languages for GPU programming have been developed and employed, but none of them have found as widespread use as programming directly in CUDA or OpenCL. These newer languages include Theano[4], Accelerate[12], Nikola[32], Obsidian[48], Intel Array Building Blocks [36], Bohrium [49] and Copperhead [11]. Their common aim is to abstract away the underlying hardware platform and let data-parallel problems be easily specified.

These languages share the characteristica of lifting operations to work collectively on vectors rather than individually on primitive values (such as integers and floating point number).

We want to join in and contribute to these languages, and we believe that such an endeavour should start with evaluating current state of the art, such that we can make sure our contributions will be relevant and beneficial. Thus, the first part of this report is a survey of some of the existing parallel functional programming languages. We have taken the strategy of implementing some real world example applications from the financial domain, to study the limitations of current approaches.

1.3 Contributions

We have conducted a small survey comparing a few Haskell libraries and languages with respect to their performance, expressivity and the apparent health of their supporting projects infrastructure.

Taking an outset in two common financial algorithms for option pricing and a quasi-random number generator, we have found that the lack of nesting in a language such as Accelerate and Nikola is too limiting to implement concise solutions and that the specification of some algorithms would lead to incomposable contructs.

As this lack of nesting is due to excessive and concious limitation, we suggest taking a step back and give the programmer more control, though giving away some of the guarantees made by a strict type system of performance and

absolute correctness. Of our surveyed languages, Nikola fits this view the best, and is selected as the subject for further study.

This we go about by exploring the implementation of a few but motivated extra array operators, and last we present a proposal to deal with the problem posed by mapping nested array operators[1] to flat parallel hardware. This proposal uses a cursor to mark the limit between sequential and parallel code, and we find at promising alternative to vectorisation. There is a large body of future work in deducing the consequences of such an addition.

All the source code associated with the practical execution of the survey, as well as this report is hosted at `https://github.com/HIPERFIT/vectorprogramming/`. The extensions developed for Nikola are to be found at `https://github.com/HIPERFIT/nikola/`.

1.4 Report outline

The remainder of the report is structured as follows. This introductory chapter is supplemented with two chapters introducing terminology: one on hardware platforms such as GPUs and one on the financial example problems we are going to use throughout the report.

The rest of the report after these introductory chapters is divided into two more or less independent parts. The first part contains a survey of currently used data-parallel languages. The second part describes our own efforts into extending the GPU language Nikola with functionality we found lacking in the experiments conducted throughout the survey.

[1] We avoid the term nested data-parallelism, as that is often connotated with the vectorisation transformation

Chapter 2

General-purpose programming of GPUs

Graphics processing units (GPUs) where originally produced for computer graphics applications, such as computer games or CGI. In the first decade of the 21st century they were found useful outside this field, as the problems in computer graphics had many similarities to problems in both science and engineering, as well as several other fields.

GPUs show a high degree of data parallelism, originating from their *single instruction, multiple data*-architecture (SIMD), where a set of computation units executes the same program on each their own data item. Many tasks in scientific and engineering, such as simulation or genome pattern matching, are data-parallel in nature. For instance, when simulating physical systems, there are often many millions of interacting agents (e.g. reacting atoms in chemistry) that can be updated independently in each step of the simulation. In other cases we might want to run the same simulation thousands of times and average the outcomes.

Originally, the only programming interfaces for GPUs were based on concepts from computer graphics, such as textures and shader programs. Applications outside the domain of computer graphics had to be encoded to fit the models of programming interfaces such as OpenGL or DirectX and developers needed deep knowledge about the GPU-architecture [39]. Since scientists started to use GPUs for research projects, several programming frameworks have been developed. The two most prominent and widely used today are OpenCL and CUDA.

In this chapter we will look at the architecture of a modern NVIDIA GPU and the programming model of NVIDIA's CUDA framework. The chapter is an edited version of a text on GPUs previously produced by one of the authors (see [16]).

2.1 GPU hardware

The execution of a GPU program has to be conducted by an accompanying program executing on the CPU. The traditional CPU is referred to as the *host*

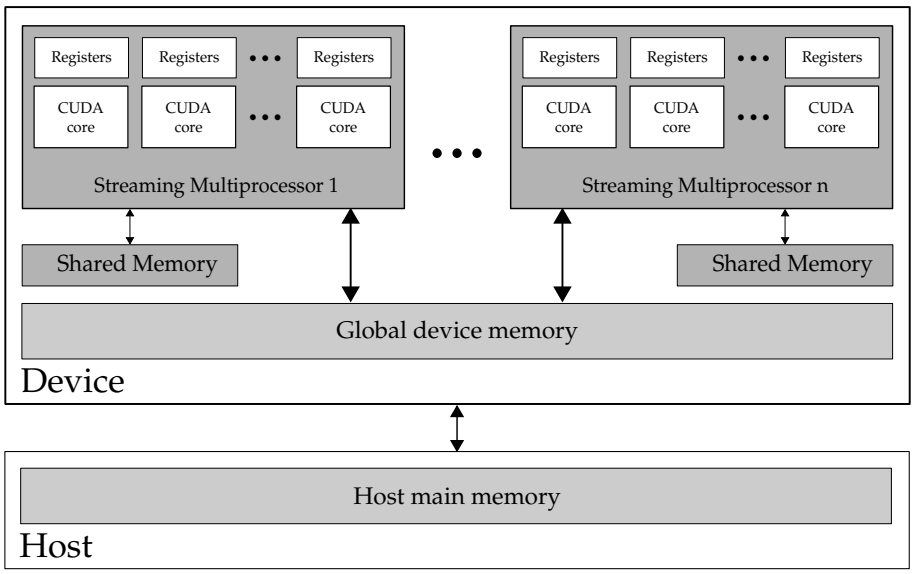


Figure 1: CUDA GPU terminology: Host, devices, streaming multiprocessors and CUDA cores. The arrows show the possible path of data movement. The host can command movement of data to and from host memory and device memory. Kernel code is responsible for moving data between global memory, shared memory and registers.

and the GPU is called the *device*. Other concepts such as pointers or arrays are often prefixed with either of these to specify where they reside.

A CUDA device is divided into a number of *streaming multiprocessors* (SMs), each of which are in turn divided into individual CUDA cores. It is these CUDA cores that performs the actual computation on a GPU. How these concepts are related is shown in Figure 1, together with the three corresponding layers of GPU-memory and communication pathways. Each CUDA core has an amount of register space available, which are memory local to the current thread executing on that core. To communicate and synchronize between other threads, each streaming multiprocessor provides an amount of shared memory, which are a bit slower than registers. As a last layer, the GPU provides a large amount of global memory, shared between all SMs, with the drawback of very long access times.

The architecture of NVIDIAs latest line of GPGPU devices is named Kepler. The current Kepler based GPUs consists of up to 2880 CUDA cores grouped into streaming multiprocessors of 192 cores each [40]. At the HIPERFIT Research Center we have access to a machine with two GeForce GTX 690 GPUs, each of which contains two Kepler GK104 GPUs. The Kepler GK104 GPU is equipped with 1536 CUDA cores arranged into eight multiprocessors and shared between all SMs is an L2 Cache (512 KB) and 2 GiB global memory [38].

Figure 2 shows the structure of a Kepler GK104 streaming multiprocessor. All 192 cores share the same register file and 64 KB of additional memory which can be used both as L1 cache and shared memory between for the CUDA cores

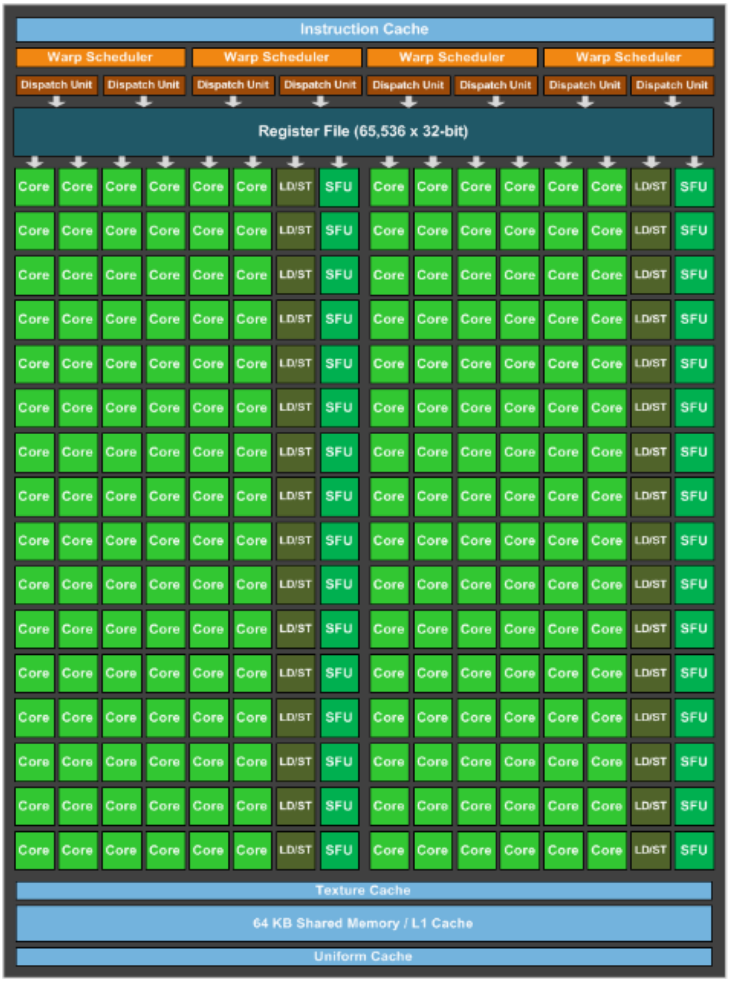


Figure 2: A diagram of a single streaming multiprocessor from a Kepler GK104 GPU. The illustration is borrowed from NVIDIAs GK104 whitepaper [38], and a few irrelevant parts are cropped away..

of the SM. The amount of memory used as cache and as shared memory is configurable.

2.2 GPU programming

Since modern GPUs contains hundreds or thousands of cores, GPU programs must be written such that their work can be executed independently and in parallel on as many cores as possible. A single thread of execution on a CUDA core is the smallest unit of work on a GPU and programming a GPU is done by writing *kernel programs* or simply *kernels*, which specifies the work done by a single thread.

These threads are arranged into equal sized groups called *blocks* and blocks are arranged into a grid. It is the task of kernel program itself to find the subset

of the data that it has to work on. That is, all executions of the kernel receives exactly the same arguments, but the kernel can query where it is located in its block and where that block is located in the grid, to determine which part of the input arrays to work on.

When executing a grid of CUDA blocks on a Kepler architecture GPU, each block is assigned to a single SM (each SM can process several blocks). Each block is partitioned into *warps* which are groups of 32 threads, which are scheduled on the 192 CUDA cores of the streaming multiprocessor.

All threads in a *warp* always executes exactly the same instruction, this is called *SIMD*, single instruction, multiple data. Moreover, when warps are executed by a SM it can execute two warps from the same block simultaneously, as each SM have access to two schedulers and the register file can be used for both of the two warps independently. This is called *SIMT* (single instruction, multiple thread) as two threads are executed on the same processor simultaneously. Each streaming multiprocessor can be assigned a certain number of warps, which it switches between executing. This is useful for hiding latency endured by memory access or data dependencies between instructions.

Synchronization

It is not uncommon that several threads has to interact, when they are cooperating to solve a problem. In these cases synchronization primitives are necessary to avoid race-conditions. In CUDA there are two levels of inter-thread synchronization. A kernel can synchronize with the rest of the threads in its block by calling the `__syncthreads` CUDA function. To synchronize across blocks, the programmer has to split program into two independent kernels and call them sequentially with a call to `cudaDeviceSynchronize` in between.

Optimization considerations

Determining how the grid of threads are partitioned into blocks are of importance when optimizing for fast execution. There should at least be as many blocks as there are streaming multiprocessor, to avoid having a stalled processor. If blocks have to wait for synchronization with other blocks it might also be beneficial to have more than one block per SM.

The size of blocks should also be considered, as having enough available warps can hide latency from memory transactions. Accessing global memory can stall a streaming multiprocessor for 200-400 cycles, where as local memory access only partakes a couple of cycles [41]. This is not the largest bottleneck though, as GPUs are presently connected to the host through PCI Express ports which have a throughput limit of a little under 8 GB/s (with PCI Express 3.0). Accessing global memory on the device can be done at 192.3 GB/s, for the GeForce GTX 690 (and the double of that if you count both of its GPUs).

We just argued for increasing block size, to make sure there are enough warps, but it is not always good for efficiency to have large blocks, as each streaming multiprocessor has a limited amount of registers and local memory. All threads currently executed on a SM shares the same register-file, and blocks are only assigned to a SM if there are enough available registers for all of its

threads. Thus, to get optimal occupancy, blocks must be sized such that the size of the register-file is divisible by the total amount of needed registers. An occupancy calculator created by NVIDIA is available as a spreadsheet on the NVIDIA website[1].

Memory access patterns

Accessing global device memory is always done in segments of 32, 64 or 128 bytes, and these accesses must be aligned such that segments are placed in physical device memory with a segment, starting on addresses that are a multiple of the segment size [41]. When a warp gets executed, the memory transaction of individual threads are coalesced such that the needed memory can be fetched by a single memory transaction. To minimize the number of memory transactions, it is thus important to write kernels, such that nearby threads which are scheduled in the same warp, will access memory in the same memory region. Previous architectures had strict rules for how data accesses should be distributed. If simultaneous operations on memory from a warp was out of sequence on such a device, all the operations was performed as separate memory transactions. The same would happen if accesses were not aligned correctly. From Compute Capability 2.X and higher, the global L2 Cache is used, such that out of order accesses can be coalesced into single transactions and misalignments can be handled as long as they do not cross 128 byte boundaries. If a segment of 32 bytes are accessed such that this access overlap two physical 32-byte segments, the 64-byte segment containing both of them are loaded from global memory instead.

These considerations are explained in detail in "The CUDA C Programming Guide" by NVIDIA [41, Section 5.3.2].

[1] `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`

Chapter 3

Cases

In this chapter we will introduce the cases which we will use as benchmarks in our survey and as offset for our extensions of the Nikola language. All cases are drawn from the financial domain and the affiliated domain of scientific computing.

3.1 Option pricing

Finance as a mathematical discipline is concerned with the pricing of assets. Financial assets can be material objects such as goods, or contracts on other assets. The *stock* issued by a company is a common example of a financial contract. However, as this definition of a financial asset is recursive, contracts on other contracts are possible. An *option* to buy or sell some other asset for some price at some point in time is a common example, and the one that we will invest some of our effort treating. These types of contracts are commonly termed *financial derivatives*.

Option is the name used collectively for a range of different financial contracts that all share some features, namely that an option on an underlying asset A gives the holder the right to exchange A for a certain strike price K in a certain time interval up until a prespecified expiration time T.

The different types of options vary the processes that determine the size of strike price and the time span where the option may be exercised. We make a distinction between options that grants a right to buy and those that grants a right to sell. A *call option* on asset A grants the right to buy A, whereas a *put option* grants the right to sell.

For a concrete example, consider this:

Example 1 (European Call Option). *European options*[1] *are characterised by having a pre-determined constant strike price K, and may be exercised only on the expiration time T.*

- *Assume that we at time t_0 acquire an European call option for the stock A which is quoted as $S(t_0) = \$90$, and that the option has a strike price of $K = \$100$ and expiration time $T = t_1$.*

[1] The continent of Europe is completely unrelated to the naming of this class of contracts.

- *When upon the arrival of t_1 we observe that A is now exchanged at value $S(t_1) = \$105$, we choose to exercise our option right and thus recieve one A for \$100.*
- *Immediately, we sell this stock on the market for \$105 and have thus earned $S(t_1) - K = \$5$. Our option was "in the money".*

Note that in reality, the underlying asset rarely changes hands. Instead, only the net difference in strike price and market price is exchanged.[2]

The challenge of option contracts resides in how to determine their price. The above option would have been appropriately priced at $\$5$ discounted to t_0 prices. The complexity of the pricing challenge varies along with the differences in parameters. The value of European options for instance is closely approximated by the Black-Scholes formula [6], which is a closed form analytical solution. In contrast to European options lie American options[3], characterised by granting the right to exercise at any point in time up until expiration, rather than only at expiration.

This difference in granted rights makes American options require a different pricing method than European options. But as of today, only numerical approximation schemes are known for this type of contract. While this may be detrimental for those that depend on accurate pricing of their financial assets, it is all the more interesting for researchers of computing to treat.

There are three classes of algorithms for American options, finite difference, lattice and Monte Carlo based methods [25]. The ensuing sections will introduce examples of both lattice and Monte Carlo based methods. We will not discuss finite difference methods.

3.2 The binomial option pricing model

A relatively simple discrete time model for computing the price of an American style option is the *standard binomial model* [15]. The model is widely used in the financial industry [18], and is thus a relevant case for our investigation.

The basic assumption is that the price of the underlying follows a binomial process over equally spaced time steps. This makes it possible to write out the possible future states of the underlying. Moving a single time step forward, the binomial process produces two possible future states of the underlying. The value of the underlying can go either up or down with probabilities q and $1 - q$ respectively. We denote the rate of up and down movement as u and d respectively. The change over one period Δt is thus given as:

$$S(t + \Delta t) = \begin{cases} S(t)u & \text{with probability } q \\ S(t)d & \text{with probability } 1 - q \end{cases}$$

We skip the discussion of how to compute the constants q, u and d and refer to the paper by Cox, Ross and Rubinstein that introduced the model [15]. After

[2] A consequence of this is that you are not required to actually own any quantity of the underlying asset for which you issue an option for.

[3] Which are similarly irrelevantly named.

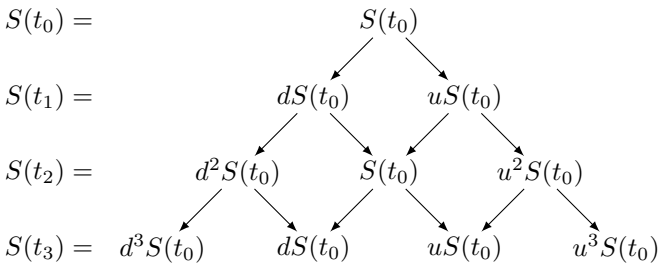


Figure 3: Lattice generated by the binomial process of a single underlying over three periods ($T = t_3$). The root node represents the current price of the underlying and the leafs represents possible values at expiration time.

several time steps the binomial process will unfold into a lattice of possible futures states as shown in Figure 3. In this example we have assumed that u and d has been selected such that $u \cdot d = 1$, but that is not a necessity.

Each layer in the tree corresponds to a time step t, and the nodes of the layer are the different possible prices $S(t)$ of the underlying. To distinguish between the nodes on a single layer we attach a unique index i to each node of the tree and define left-child(i) and right-child(i) to navigate the indices of the tree. We use the notation $S_i(t)$ to refer to the particular node i at layer t.

The final row represents the possible values of the underlying at expiration time, T. We can use the same procedure as for European options (discussed in Example 1 above) to find the value of our option at each of these possible cases. Thus, for each possible $S_i(T)$, that is, for each leaf node, we compute the option value as either $V_i(T) = max(S_i(T) - K, 0)$ for call options or $V_i(T) = max(K - S_i(T), 0)$ for put options, where K is the strike price.

The algorithm now proceeds by estimating the price V_i of the option in the remaining lattice points, by iterating backwards (towards the root). The price estimate at each lattice point for time t depends only on its two children at time $t + 1$, which prices was calculated in the previous iteration. This is done by recursively discounting:

$$V_i = e^{-r\Delta t} \cdot (q \cdot V_{\text{left-child}(i)} + (1 - q) \cdot V_{\text{right-child}(i)})$$

where the probability q is used to attain the expected value from either of the two alternative futures, and $e^{r\Delta t}$ is the guaranteed scale of increment over a single time period Δt for a riskless investment with interest rate r.

Implementation Techniques

As seen above, the value of $V_i(t)$ depends only on values associated with time $t + 1$. Therefore, for each t, the evaluation of $V_i(t)$'s may proceed in parallel.

Algorithm 1 illustrates this in pseudocode that uses only two arrays in tandem to hold the intermediately produced values. It should also be observed that it is not necessary to build the complete lattice, as we are only interested in the values at expiration date.

function BINOM(Δt, T, u, d, q, $S(0)$, K, r)
  $S(T) \leftarrow$ **parmap** $(\lambda i.\ d^{T-i}u^i S(0))\ \{0..T\}$ ▷ Value at expiration
  $B \leftarrow$ **parmap** $(\lambda S_i(T).\ \max 0\ (K - S_i(T)))\ S(T)$ ▷ Perform exercise decision
  $t \leftarrow T$
  while $t > 0$ **do**
    $A \leftarrow$ **parmap** $(\lambda i.\ e^{-r\Delta t} \cdot (qB[i] + (1-q)B[i+1]))\ \{0..t\}$
    $t \leftarrow t - 1$
    $B \leftarrow$ **parmap** $(\lambda i.\ e^{-r\Delta t} \cdot (qA[i] + (1-q)A[i+1]))\ \{0..t\}$
    $t \leftarrow t - 1$
  end while
  return $B[0]$
end function

Algorithm 1 Binomial algorithm

As defined by Blelloch [8], the above code may be reasonably assumed to have time complexities $Work(T^2/2)$ and $Depth(T)$, and memory complexity $O(2T)$.

While the above code is both short and simple in structure, the main motive for pursuing parallelism is in gaining performance rather than expressivity. To explore further algorithm samples, we examine the binomial pricing algorithm included in the CUDA SDK [43], depicted in Algorithm 2. Here the PARTITION(i, n, k, o) procedure partitions the index range $[i, n]$ into k sized chunks with o indices overlapping. Here it is written as a separate procedure for clarity, but in the actual code the partitioning logic is fused into the index computations of the `for`-loop.

This algorithm explicitly divides the pricing problem into packages that fit exactly into the amount of shared memory availabe for a Streaming Multiprocessor. This includes a configurable amount of duplicated computation for each iteration, because it is faster to recompute some part of the solution if doing so brings down the number of accesses to global devce memory. Thus, the optimal value of Δ_{mem} is hardware specific. Algorithm 2 is only capable of employing mem threads at a time. Therefore, to scale it must be applied to an entire portforlio of options. In the CUDA SDK source code, this is done by doing a kernel call with a CUDA block for each option to be priced. That way, each option pricing instance gets its own streaming multiprocessor, and no penalty has to be paid for divergence when the options in the portfolio vary in size.

3.3 Longstaff & Schwartz pricing

Another approach to American option pricing is manifest in the Least Squares Monte Carlo algorithm (LSM), introduced in 2001 by Longstaff and Schwartz [30]. Here Monte Carlo simulation is used to generate a corpus of possible price developments of the underlying asset, and the option price is then calculated by successively applying linear regression at each time step from expiration to initiation time.

function GPUBINOM(mem, Δ_{mem}, Δt, T, u, d, q, $S(0)$, K, r)
  $S(T) \leftarrow$ **parmap** $(\lambda i.\ d^{T-i}u^i S(0))\ \{0..T\}$ ▷ Value at expiration
  $B \leftarrow$ **parmap** $(\lambda S_i(T).\ \max 0\ (K - S_i(T)))\ S(T)$ ▷ in global memory
  $t \leftarrow T$
  while $t > 0$ **do**
    for each (i, j) of PARTITION(0,t, mem, Δ_{mem}) **do**
      $C \leftarrow B[i : j]$ ▷ Copy to shared memory
      $C \leftarrow \Delta_{mem}$ iterations of the body of BINOM
      $B[i : j - \Delta_{mem}] \leftarrow C[0 : j - i - \Delta_{mem}]$ ▷ Update global memory
    end for
    $t \leftarrow t - \Delta_{mem}$ ▷ All of B now corresponds to $t - \Delta_{mem}$
  end while
end function
function PARTITION(i, n, k, o)
  if $i + k > n$ **then**
    return $[(i, n)]$
  else
    return $(i, i + k) :$ PARTITION($i + k - o + 1$, n, k, o)
  end if
end function

Algorithm 2 Binomial portfolio pricer

function LSM(T, M, N, s_0, K, r, σ)
  $\Delta t \leftarrow \frac{T}{M}$
  $S \leftarrow$ GENERATEPRICEPATHS(N, M, s_0, σ, Δt, r)
    ▷ $N \times (M + 1)$ matrix of N paths
  $V_{M+1} \leftarrow max(K - S_{M+1}, 0))$ ▷ Cash flow at expiry
  for $t \leftarrow M - 1$ **to** 1 **do**
    $E_t \leftarrow max(K - S_t, 0)$ ▷ Exercise value
    $f_t \leftarrow$ LEASTSQUARES($\overline{S_t}$, $e^{-r\Delta t} \cdot \overline{V_t}$)
    $\hat{C}_t \leftarrow f_i(S_t)$ ▷ Expected value for continuation
    $V_t \leftarrow$**if** $E_t > C_t > 0$ **then** E_t **else** $e^{-r\Delta t} \cdot V_t$
  end for
end function

Algorithm 3 Least Squares Monte Carlo algorithm for American put options. $\overline{S_t}$ and $\overline{V_t}$ selects those paths which are in the money, that is where $E_t > 0$.

We will use the names "Longstaff and Schwartz" and "Least Squares Monte Carlo" (LSM) interchangeably in the rest of the report

Algorithm 3 presents the LSM algorithm. The approach is to first generate a large number of possible price developments of the underlying, up till expiration. The algorithm proceeds by moving backwards in time from expiration time, and at each step estimating the value of continuation by least squares regression. These estimates are then used to calculate the current option price, before continuing to the next iteration.

We have left out how to generate price paths and perform the actual least squares regression, these will be treated in more detail in the subsequent

function GENERATEPRICEPATHS(N, M, s_0, σ, Δt, r)
  $S_0 \leftarrow$ Initialize vector with N repetitions of s_0
  for $i \leftarrow 0$ **to** $M - 1$ **do**
    $V \leftarrow$ Generate N normally distributed random numbers
    $D \leftarrow$ **parmap** $(\lambda v.\ e^{\Delta t(r-\sigma^2/2)+\sigma v\sqrt{\Delta t}})\ V$
    $S_{i+1} \leftarrow$ **parallelZipWith** $(+)\ S_i\ D$
  end for
  return S
end function

Algorithm 4 Brownian motion path generation

sections.

Path generation

A standard approach to constructing the corpus of price paths is by sampling paths of geometric Brownian motion.

This can be done by iterative application of the following equation:

$$S_{t+\Delta t} = S_t e^{\Delta t(r-\sigma^2/2)+\sigma v\sqrt{\Delta t}}$$

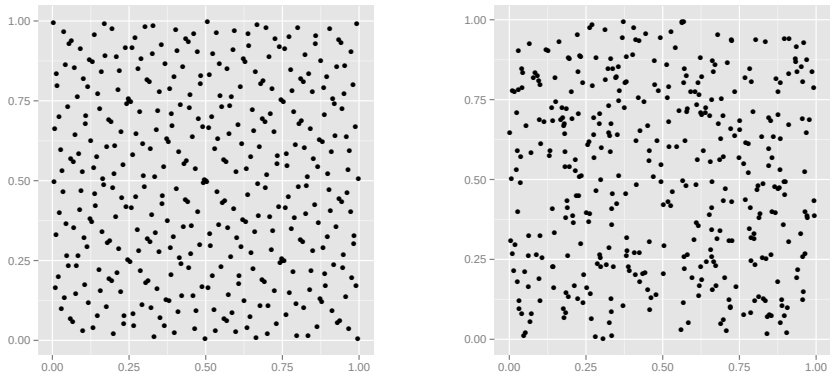
where v is a real number drawn from $\mathcal{N}(0, 1)$, Δt is the size of the time period, and σ is the volatility. Volatility is the standard deviation of the option value over time and corresponds in function to that of u and d in the binomial model. The right-hand side consists of two parts where the first exponent corresponds to the deterministic evolution of the price by the riskless interest rate, and the second part corresponds to the variation introduced from the Brownian motion. A derivation of the equation can be found in the book *Monte-Carlo Methods in Financial Engineering* by Paul Glasserman [19, Section 3.2].

As long as we can generate independent random numbers in parallel this generation can also be parallelized, as each path can be generated independently.

Random number generation

A necessity for all applications of Monte Carlo methods is a source of good random numbers. We cannot computationally generate "true" random numbers, as they will always follow our chosen algorithm, but through the use of *pseudo-random number generators* (PRNGs) we can generate sequences of numbers with approximately the same properties as a completely random sequence. Such algorithms are widely used in both Monte Carlo simulation and cryptography, but the choice of algorithm depends on the application area as different PRNGs displays different statistical properties.

An alternative scheme is that employed by a *quasi-random number generator* (QRNG). Algorithms in this category aims for sequences with low discrepancy, which is a measure of spacing in the sequence. If we are to generate a sequence over some interval $[a, b]$, discrepancy would be high if there were any subintervals $[c, d]$ with proportionally few samples compared to other



(a) (b)

Figure 4: (a) A 2D sequence of low-discrepancy numbers generated using the Sobol generator **(b)** A uniform 2D sequence of pseudorandom numbers generated with Mersenne twister PRNG

intervals. Low discrepancy thus guarantees that proportionally equal amounts of samples are made in all subintervals. We thus avoid gaps in the sample space with few or no samples or conversely, regions with high sample density. To illustrate, Figure 4 shows a graph of a low-discrepancy sequence compared with sequence of uniformly generated numbers from a PRNG.

QRNGs does not assure any other properties of random sequences and does indeed follow rather visible patterns (see Figure 4(a)). It is not the aim to get close to statistically random numbers, instead QRNG provides numbers suitable for numerical integration, optimization and simulation. QRNGs have several times been shown useful in financial Monte Carlo simulations [13, 14, 52] and a survey paper shows QRNGs beating PRNGs when it comes to option pricing [2]. The survey is 15 years old, so newer PRNGs might perform differently.

We will look closer at one particular low-discrepancy generator, namely the Sobol sequence generator, by the Russian mathematician I. M. Sobol [45]. We chose the Sobol sequence because it was the one used by an unrelated sample application from one of HIPERFIT's industri partners. Also, the above mentioned survey concludes that "Sobol' points appear to give the best results among the QMC sequences tested".

Sobol generator

We will not discuss why the numbers generated by the Sobol generator maintain low discrepancy, but only focus on the algorithmic aspect.

Given a sequence of so called *direction numbers*, $v_1, v_2, \ldots$, the computation of Sobol-sequences is pretty straight forward:

$$x_i = b_1 v_1 \oplus b_2 v_2 \oplus \ldots \oplus b_l v_l$$

where $b_l \ldots b_3 b_2 b_1$ is the binary representation of i in l-bit representation. This simple algorithm is presented in Figure 5, where TOBITVECTOR converts an

function SOBOLINDUCTIVE(v,i)
  return fold $\oplus$ 0 (**zipWith** ($\times$) v (TOBITVECTOR i))
end function

Algorithm 5 Generate element i of the Sobol sequence using vector v as direction numbers.

function SOBOLRECURSIVE(v,A,i,n)
  $x_i \leftarrow$ SOBOLINDUCTIVE(v, GRAYCODE i)
  $end \leftarrow i + n$
  while $i < end$ **do**
    $c \leftarrow \mathrm{lsb}(i)$
    $x_{i+1} \leftarrow x_i \oplus v_c$
    $i \leftarrow i + 1$
  end while
end function

Algorithm 6 Sequentially generate elements i through $i + n - 1$ of the Sobol sequence and store results at the appropriate indices of the x vector.

integer i to its binary representation as an array of integers. In our own implementations we have used 32-bit integers. As computing each number is independent of the other numbers, we can parallelise by mapping SOBOLINDUCTIVE over the indices.

The SOBOLINDUCTIVE algorithm as well as the remaining algorithms produces integer values in the range $[0, 2^l]$ and we thus has to normalize this to $(0, 1]$ to get a uniform distribution. This normalization is not included in the algorithms of this section. In addition, we have to convert this uniform distribution to a normal distribution. This will be discussed in Section 3.3.

For multidimensional sequences, such as the 2D sequence in Figure 4(a), a different set direction vector v is used for each dimension. A list of good direction vectors is available online at [46].

If one seeks for a sequential algorithm for Sobol sequence generation, we can improve on the above by using the Gray code binary representation of the index instead of the ordinary, whereby a recursive definition of Sobol sequences can be made [10]. It has been shown that

$$x_{i+1} = x_i \oplus v_c \tag{3.1}$$

where $c = lsb(i)$ is the index of the least set bit in i. A sequential algorithm using this strategy is presented in Figure 6.

This approach does not output the same sequence though, but it has been shown that Gray code based sequences also displays the property of low discrepancy, as it is merely a scrambling (reordering) of prefixes of the original Sobol sequence [10]. Conversion to Gray code representation can be done through single shift and an exclusive-or operation, which makes for a cheap sequential algorithm.

The two approaches can be combined to achieve an alternative parallel implementation. The sequence that needs to be generated is divided in chunks onto each processing element, the inductive algorithm is used to initialize each

function SOBOLPARRECURSE(v,n)
  $x \leftarrow$ allocate vector of length n
  $m \leftarrow$ number of processor cores (e.g. CUDA cores)
  $r \leftarrow \lceil n/m \rceil$
  parfor $p \leftarrow 0$ to $m - 1$ **do**
    $start \leftarrow p \cdot r$
    SOBOLRECURSIVE(v,x,$start$,$\min(r, n - start)$)
  end parfor
end function

Algorithm 7 Parallel Sobol sequence generator.

of the chunks and the recursive algorithm is then used to fill out the space. The algorithm is shown in Figure 7

The problem with this algorithm is that its memory behavior is not suitable for GPUs. Programs that does not coalesce memory accesses, such that all cores act on the same block simultaneously, incurs a big performance penalty. With this algorithm, each thread will write very dispersed, as each thread will fill out each its own region of the array A.

Luckily, an alternative strategy has been shown by Thomas Bradley et al. [23, Chapter 16]. They observed that applying (3.1) several times, would lead to repeated exclusive or operations with the same direction number. As the *exclusive or* operation is associative, commutative and is its own inverse, some of these operations will cancel out. Especially, when skipping a power of two ahead, all but two exclusive or operations will cancel out. The least set bit will change in a systematic way, and analysis shows that (see the above mentioned book chapter):

$$x_{i+2^p} = x_i \oplus v_p \oplus v_{q(i)} \tag{3.2}$$

where $q(i) = lsb(i \veebar (2^p - 1))$. We use $\veebar$ for bitwise or.

This fact can be used to develop a parallel algorithm which is memory efficient on GPU architectures. Instead of each letting each thread fill a separate block of memory using (3.1), we can instead let all threads cooperate on filling one block at a time, by using the inductive algorithm (Algorithm 5) to fill the first block of memory and use (3.2) to fill each successive block. This thus requires a skipsize of a power of two.

On a GPU this can done by letting a group of 2^b blocks of size 2^p threads cooperate on filling a block of size 2^{p+b} before continuing work on the next block.

To generate multidimensional sequences, their suggestion is to launch several instances of the same algorithm in parallel, which would lead to a an unzipped (tuple of arrays) memory representation.

In the same book chapter they present skip-ahead strategies for two PRNGs, the Mersenne Twister (the MT19937 variant) and L'Ecuyer's Multiple Recursive Generator (MRG32k3a).

function SOBOLSKIPPING(v, n, p, b)
  parfor $i \leftarrow 0$ **to** $2^{p+b} - 1$ **do**
    $x_i \leftarrow$ SOBOLINDUCTIVE(v, GRAYCODE i)
    while $i < n$ **do**
      $q \leftarrow lsb(i \Vee (2^{p+b} - 1))$
      $x_{i+2^{p+b}} \leftarrow x_i \oplus v_{p+b} \oplus v_q$
      $i \leftarrow i + 2^{p+b}$
    end while
  end parfor
end function

Algorithm 8 Parallel Sobol sequence generator. v is the direction vector, n is the length of the sequence, 2^p is the block size and 2^b is the number of blocks.

Sampling the standard normal distribution

The Sobol sequence generator as well as most other PRNGs and QRNGs draws samples uniformly from the interval $(0, 1]$ (0 is not included, as we often want to avoid negative samples). In our application we need samples from the standard normal distribution, $\mathcal{N}(0, 1)$, and thus a transformation is necessary.

The standard approach to convert from a uniform distribution to any other distribution is by inversion of the cumulative distribution function (CDF). It not possible to compute the inverted CDF exactly for the normal distribution, but approximations exists. The algorithm that seems the most popular when we have to do with QRNGs is called the "Beasley-Springer-Moro" algorithm or "Moro's inversion". This algorithm uses a result by Beasley and Springer from [3] to approximates the inverse CDF of the standard normal distribution by

$$norminv_1(u) = \frac{\sum_{n=0}^{3} a_n (u - 1/2)^{2n+1}}{1 + \sum_{n=0}^{3} b_n (u - 1/2)^{2n}}$$

on the interval $u \in [0.5, 0.92]$ for a specific set of constants a_n and b_n, see [19, page 67-68]. Boris Moro found [34] that he could approximate the remaining interval $u \in (0.92, 1]$ by

$$norminv_2(u) = \sum_{n=0}^{8} c_n log^n(-log(1 - u))$$

where c_n are additional constants, also given in [19]. With this, the inverse CDF of the standard normal distribution can now be written as:

$$cdf^{-1}(u) = \begin{cases} norminv_1(u) & \text{if } 0.5 \leq u \leq 0.92 \\ norminv_2(u) & \text{if } 0.92 < u < 1 \\ -cdf^{-1}(1 - u) & \text{if } 0 < u < 0.5 \end{cases}$$

Least Squares solvers

The only remaining element of the Longstaff & Schwartz algorithm that we need to discuss is the least squares regression. What we need is a way of smoothing out the variation in the paths, by representing their change from

one point in time to the next by a function. We have a set of stock prices S_t and a set of option values V_t and wants to find the polynomial f of a degree d that is as close to these coordinates as possible. That is, we want the coefficients $\mathbf{x}$ for the polynomial that minimizes the error:

$$min_{\mathbf{x}} \sum_{i=0}^{N} (V_{t,i} - f(S_{t,i}, \mathbf{x}))^2$$

to generalize this, we will use a and b in the rest of this section, such that:

$$min_{\mathbf{x}} \sum_{i=0}^{N} (b_i - f(a_i, \mathbf{x}))^2$$

In our case, f will take the shape of a polynomial:

$$f(\mathbf{a}, \mathbf{x}) = x_1 + x_2 a_i + x_3 a_i^2 \dots x_d a_i^{d-1}$$

and this gives rise to a linear system of equations. Each equation originating from a coordinate (a_i, b_i). We write the system as:

$$\mathbf{Ax} = \begin{pmatrix} 1 & a_1 & a_1^2 & \dots & a_1^{d-1} \\ 1 & a_2 & a_2^2 & \dots & a_2^{d-1} \\ & & \vdots & & \\ 1 & a_n & a_n^2 & \dots & a_n^{d-1} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} = \mathbf{b} \tag{3.3}$$

The $\mathbf{A}$ matrix is called a Vandermonde matrix. To find the minimizing x vector, we want to minimize the residual vector $\mathbf{r} = \mathbf{b} - \mathbf{Ax}$. Michael T. Heath shows how this can be done by solving the linear system $\mathbf{A}^T\mathbf{Ax} = \mathbf{A}^T\mathbf{b}$ in *Scientific Computing: An Introductory Survey* [22, Section 3.2.1].

There are several ways to solve linear systems, for instance LU-decomposition, QR-decomposition or Cholesky-decomposition. We have chosen to implement Cholesky-decomposition.

With Cholesky decomposition we find a lower triangular matrix $\mathbf{L}$ such that $\mathbf{A} = \mathbf{LL}^T$. And now forward and then backward substitution can be used to find the minimizing $\mathbf{x}$.

Part I

Survey

Chapter 4

Survey scope and methodology

In this and the following chapters we present a small-scale survey of a few current vector languages. The survey was originally conducted to orient ourselves in the current landscape of data-parallel functional languages and their implementation, but the results might be of interest for others. We do not know of other comparisons with a similar scope.

We will also identify some key issues used in some of the current languages, which will be valuable knowledge for future research in the area.

In the rest of this chapter we define the scope of our survey and go through the methodology we have followed.

Subject languages

As this is a half-year project, we had to limit our survey considerably in size, and thus only included a few programming languages. We decided to evaluate and compare R, CUDA, Accelerate [12], Repa [27], Nikola [32] and the `Data.Vector` library[1].

Nikola and Accelerate was included in the survey as they are functional approaches to GPU computation. We had initially also wished to include Obsidian[48], Copperhead[11] and GPU-NESL[5] in this category, but time constraints and installation problems made us leave them out.

The remaining languages included in the survey are not meant as being subjects for the study, but to serve as reference models for the comparison.

We have included the R programming language and NVIDIA's CUDA platform because of their use in the financial industry and research communities. Financial engineers and mathematicians use R for quick prototyping and expressing algorithms succinctly and CUDA when high-performance computing is required. Repa and `Data.Vector` are two Haskell libraries, that provides array operations for CPU-execution. They are included, because of their popularity as functional approaches to data-parallel programming languages. R, Repa and `Data.Vector` are thus mostly included to evaluate how easy the Nikola and Accelerate languages are to `use`. Whereas CUDA is included as a measuring stick and goal for their performance in execution speed.

[1] `http://hackage.haskell.org/package/vector`

Since we initially started, we have discovered several other languages which would have been interesting to look closer at. These include Bohrium[49], Theano[4], CnC-CUDA[21] and R+GPU[50] (library of common R functions running on CUDA).

4.1 Scope of Experiments

There are three language facets that we want to examine: Their expressivity, project health and execution performance. In the next three subsection we detail and motivate these aspects.

Expressiveness

By the expressiveness of a language, we intend to ... the ease expression of ones thoughts, mathematical models and natural implementation of algorithms.

Specifically, we want to test whether the selected languages allows us to write the financial algorithms presented in Section 3, and that the underlying algorithms when implemented, remains in a natural and readable form. Quoting a famous line from the preface to *Structure and interpretation of computer programs* [1]:

> "Programs must be written for people to read, and only incidentally for machines to execute"

We thus want to check that the techniques necessary for implementing our sample programs does not insert unnecessary clutter and that performance can be obtained by writing in an idiomatic style for the language.

Project health

When evaluating project health of programming languages, we want to take the side of the language user, who that has to select a stable language for a program, and we thus want to look at aspects such as which projects seems guaranteed continued development, is not subject to large changes in programming interface and are properly documented.

The health of a software project is an aspect worth investigating, if you intend to use the product of that project in your own work.

We have written a short chapter on the project health of `Data.Vector`, Repa, Accelerate and Nikola.

Performance

The final thing we want to evaluate is how the languages compare in raw execution performance, by comparison with high-performance implementations (CUDA) of the same benchmarks.

4.2 Methodology

To make a fair comparison, we have to be objective in our evaluation of the languages. Obtaining objectivity in such a comparison is not an easy task, as it is hard to quantitatively measure aspects such as the quality of language documentation (longer is not always better, and it might be outdated). Also, as mentioned by Lutz Prechelt in his 2001 paper "Empirical Comparison of Seven Programming Languages":

> "Any programming language comparison based on actual sample programs is valid only to the degree to which the capabilities of the respective programmers using these languages are similar."

We would thus have to either acquire the same level of experience in all the languages ourselves or find experts in each of the languages to do the implementation. We have not had the resources to conduct a survey following the standards set in the paper by Lutz Prechelt. For each language under comparison he acquired 10-20 implementations of the same algorithm from different programmers (mostly graduate students). We could have set up a similar experiment by presenting a programming challenge to online Haskell community (e.g. the "Haskell-Cafe"-mailing list), and surely we could perhaps get a decent benchmark in terms speed and memory usage for different implementations by different developers, but we would not get answers to qualitative questions about the development process. Another aspect is that these languages are all in their early stages, and most people we would find on those channels might be amateurs.

To perform the above mentioned experiments we set out to implement both the binomial pricing algorithm and the Longstaff and Schwartz algorithm.

We also considered the Black-Scholes option pricer a possibility, but we knew that Black-Scholes was possible to implement in both Nikola and Accelerate (and thus the remaining languages in our survey), as they were documented in their respective research papers [32, 12]. It was thus a less interesting example for the Devil's advocate, and as we had enough work in implementing the binomial pricer and LSM, we ended up leaving it out.

As we discovered that Longstaff and Schwartz was hard (perhaps impossible) to implement in Accelerate and Nikola, we changed our scope and put a larger focus on the Sobol algorithm than the actual LSM algorithm.

It is from these implementations that we have documented our findings of expressivity and performance, which will be presented in Chapter 5 and 7.

When it comes to project health, we have investigated by tracking information about number of contributors, number of reverse dependencies and so on, from information available on Hackage and commit logs. These findings are presented in Chapter 6.

We discuss the results of the entire survey in Chapter and present our conclusions.

Chapter 5

Evaluation of expressiveness

In this chapter we discuss and compare the expressiveness of each of the languages we are considering. The property of programming language expressiveness is traditionally associated only with how beautiful and simple encodings a set of algorithms lend themselves to. But since we are dealing with a domain of algorithms where high computer performance is prerequisite for acceptability, the need for control over the management of resources invariably enters the picture – at least with current programming technology. In this survey therefore, resource management features of a programming language may contribute to the lanugages expressivity.

We evaluate expressiveness of a lanugage by discussing its general features, and by discussing noteworthy issues we have encountered while implementing our benchmarks in the evaluated languages.

Although we already know that our eventual choice of language to extend stands only between Nikola and Accelerate, here we also include CUDA, Repa, Data.Vector and R to give a better perspective.

5.1 CUDA/C

CUDA/C is close to a one-to-one mapping of hardware capabilities to language primitives, thus it is possible, though not necessarily easy, to implement anything that the hardware is capable of running.

Every single resource must be explicitly managed – only in CUDA version 5.0 is there even a function call stack, supporting only a maximum of 20 levels of nesting. While this manual resource management enables the coding of very fast algorithms, programs are also more tightly coupled with the parameters of the hardware, and to assumptions about input size. So even though a CUDA programmer has access to highly optimised linear algebra libraries such as CUBLAS [42], optimisations such as loop fusion must be coded by hand, which rules out the use of such libraries. This optimisation often has profound effects on performance [33].

Implementing Algorithm 2, and adapting it to use just a single option proved to be a substantial and error prone effort for us. This we attribute to both that programming in CUDA is relatively foreign to us and the lack

of conventional debugging tools, such as a single-step debugger or a machine simulator. The concrete reason for our trouble was almost always index miscalculations in for-loops rather than misunderstandings of the general programming model.

5.2 R

The R programming language is designed as a tool for exploring data sets using statistics and plotting, and for coding prototypes of data analysis programs. It is safe to say that the focus of R lies mainly on expressivity in the domain of statistics and data analysis.

R is a very dynamic language, complete with anonymous functions, general recursion and an immense library of high level operations. Standard R programs execute on the CPU in their entirety. There is however an effort to utilise GPUs in R to some extent through the `gputools` package[26].

R has become popular in scientific fields that are not primarily centered on computer programming, such as biology and statistics That in itself is good testament to the practical usefulnes of the language.

As a result of the extensive library support for this domain in R, the case benchmarks we use in this language were able to offload much work to external linear algebra routines.

5.3 Data.Vector

The `Data.Vector` library augment Haskell with a high performance implementation of one-dimensional vectors. Vectors come in various flavours that reside in different module namespaces. There are mutable vectors, which are only usable from inside the IO or ST monads and must be allocated explicitly, and there are immutable vectors, which are usable in pure functional code. There is also a version of vectors working on unboxed values.

This vector library interfaces with the GHC compiler to provide the array fusion optimisation. Taken together with the use of unboxed vectors one is able to remove a lot of overhead associated with array and thunk construction, even from ideomatic, functional Haskell code.

5.4 Repa

Repa is a library that attempts to bring high performance data parallelism to Haskell by means of threaded multicore CPU parallelism. As a result, Repa features does not impose more limits on the expressivity of Haskell in general.

The main contribution of Repa towards this end is that arrays in Repa are parametrised by their internal representation and shape in the type: `Array r sh a`. This enables using different algorithms for processing different array representations, and restricting certain operations for certain representations. For example we have delayed arrays, which are simply represented by an index domain and a function that maps indices to values. These arrays support indexing, but each index operation will pay the cost of computing the function.

Repa provides two main operations for manifesting arrays to memory: `computeS` for sequential and `computeP` for parallel array manifestation. While this division of execution modes is simple, it opens the potential for expressing nested parallelism. In Repa, nested calls to `computeP` is considered an error. A warning is issued on the terminal, and the documentation says to expect reduced performance.

A consequence of this is that a library of Repa functions that require the use of array manifestation must provide separate parallel and sequential versions of each function, as a client program cannot use only the parallel version if it intends to parallise on a another level of the program. A remedy for this would be to adopt the policy that nested occurences of `computeP` just revert silently to `computeS`. In Chapter 11 we present another similar solution to the problem of nested parallelism.

5.5 Accelerate

Accelerate provides a variety of array operations: Both scans, segmented scans, folds, permutations, maps and zips, implemented as parallel algorithmic skeletons. Arrays may be multidimensional, denoted by a type variable in the same style as Repa.

Reductions (folds) are defined on arrays of arbitrary rank by performing the reduction on the innermost dimension, yielding an array with rank one less, or by folding all dimensions into a single scalar value. Maps are all elementwise regardless of the shape of the input array, and scans are only defined on one-dimensional arrays.

Accelerate is characterised by a clean division between the frontend and the various backends that exist. The Accelerate language is thus completely backend agnostic, and backends simply export a function such as `run :: Acc a -> a`.

Accelerate employs a meticulus partitioning of functions on arrays and functions on scalar values. Array functions are all embedded in the `Acc a` type, while scalars are embedded in the `Exp a` type. So, everything that is capable of reducing an array or producing an array is carefully placed inside `Acc`, and the functions usable for elementwise computation must reside in `Exp`. Thus, the lack of nested parallelism is directly encoded in the type system.

While this restriction probably makes it easy to ensure efficient execution of the individual contructs, it impairs the composability of the language constructs significantly, as the programmer needs to manually transform algorithms with a naturally nested definition into something that will fit into the view of Accelerate.

To illustrate this problem, consider the following small example. While the example may seem a bit abstract, it is directly derived from a problem we actually encountered while exploring the Sobol sequence generation case, depicted in Algorithm 5. The actual code is included in Appendix 12.2.

Suppose we have a function `f`, defined for normal Haskell lists for notational clarity, using the functions `k` and `h`:

```
f :: [Int] -> Int -> [Int]
```

```
f xs i = zipWith k xs (h i)
  where
    k :: Int -> Int
    h :: Int -> [Int]
```

Now, if we want to apply `f` to a range of different `i`'s, what we would reasonably do in Haskell would be just:

```
g :: [Int] -> [Int] -> [[Int]]
g xs ns = map (f xs) ns
```

And for illustration, depictions of `f` and `g` as mathematical expressions. Each application of `f` yields a row of the matrix:

$$xs = \begin{bmatrix} xs_0 \\ \vdots \\ xs_n \end{bmatrix} \qquad ns = \begin{bmatrix} ns_0 \\ \vdots \\ ns_m \end{bmatrix}$$

$$f\ xs\ i = \begin{bmatrix} k\ xs_0\ hi_0 \\ \vdots \\ k\ xs_n\ hi_n \end{bmatrix} \quad g\ xs\ ns = \begin{bmatrix} f\ [xs_0 \ldots xs_n]\ ns_0 \\ \vdots \\ f\ [xs_0 \ldots xs_n]\ ns_m \end{bmatrix}$$

Now we want to port `f` and `g` to Accelerate. For `f` there seems to be an obvious translation:

```
f :: Acc (Vector Int) -> Exp Int -> Acc (Vector Int)
f xs i = zipWith k xs (h i)
  where
    k :: Exp Int -> Exp Int
    h :: Exp Int -> Acc (Vector Int)
```

But due to the lack of nested vectors, for `g` we must resort to a two-dimensional array. And due to the restriction that mapped functions must reside in `Exp`, we cannot simply do with just: `map f (xs) ns`.

Instead we must distribute the map into the definition of `f` and replicate `xs` to fit. But then we are left with an identical problem relating to function `h`!. Now, assuming we succeed and denominate the vectorised `h` function as `h' :: Acc (Vector Int) -> Acc (Array DIM2 Int)`, here is what we arrived at for `g'`:

```
g' :: Acc (Vector Int) -> Acc (Vector Int) -> Acc (Array DIM2 Int)
g' xs ns = let ns' = h' ns
               Z :. m :. n = unlift $ shape ns'
               xs' = replicate (Z :. m :. All) xs
           in zipWith k xs' ns'
```

The transformation is illustrated below.

$$\text{ns}' = \begin{bmatrix} h(ns_0) \\ \vdots \\ h(ns_m) \end{bmatrix} \qquad \text{xs}' = \begin{bmatrix} xs_0 & xs_1 \ldots & xs_n \\ \vdots & \ddots & \\ xs_0 & xs_1 \ldots & xs_n \end{bmatrix}$$

$$g'\ xs\ ns = \begin{bmatrix} k\ xs_0\ ns'_{0,0} & k\ xs_1\ ns'_{0,1} & \ldots & k\ xs_n\ ns'_{0,n} \\ \vdots & \ddots & & \\ k\ xs_0\ ns'_{m,0} & k\ xs_1\ ns'_{m,1} & \ldots & k\ xs_n\ ns'_{m,n} \end{bmatrix}$$

Using the above notation it is relatively easy to verify that `g` for lists and `g'` for arrays should be equivalent. The transformation of the code however is quite drastic. This transformation is similar to the vectorisation transformation of NESL[7]. But since Accelerate provides no way to automate it, the programmer needs to give up composability of Accelerate terms.

This sharp division between scalar and array operations is easily the biggest hindrance to the expressiveness of Accelerate. However, as this is a pervasive part of the architecture of Accelerate, removing the distinction of `Acc` and `Exp` would result in an entirely different language, and every single backend would have to be rewritten.

The embedding of Accelerate leaves some things to be desired. It is for instance not easily possible to pattern match on tuples and shapes, as these need to be properly lifted and unlifted to be used (see above in function `g'`).

Also, lifting using `lift :: Lift c e =>e -> c (Plain e)` uses the `Plain` associated type, the definition of which is not shown in the auto-generated documentation of instances. Although the documentation generation system is arguably to blame for this, it does nonetheless make it more difficult to easily use value lifting confidently.

There is no way to define recursive functions in Accelerate. Trying to do so will result in compilation not terminating.

5.6 Nikola

Nikola does not provide the variety of array operations that Accelerate does: Only a few mapping operations are provided, and an iteration construct. Nikola has a lot of overall structure in common with Repa. As with Repa, an array's implementations and shape are represented in the array type, and the programmer has access to mutable arrays in addition to the common pure array operations.

To exploit the ability to differentiate according to array representations, operations such as **`map`** and **`zipWith`** are implemented using typeclasses. While this architecture allows for specialisation of operations to different array representations, actually using the operations results in quite elaborate types. These are often too complicated for type inference to resolve and require a human programmer to supply a type signature.

Compared to Accelerate, Nikola is embedded more naturally inside Haskell, leveraging the `RebindableSyntax` GHC-extension. Also, in our programming experience we were less required to interact with the value lifting machinery

than we were in Accelerate. Value lifting in Nikola is subject to the same documentation inadequacies as we encountered with Accelerate.

While Nikola does not present any means for expressing nested parallelism, it does not explicitly ban it either in the style of Accelerate's `Acc/Exp` division. Thus, extending the expressive power of Nikola in this aspect appears at first to be a less elaborate endeavour than in Accelerate.

There is no way to define recursive functions in Nikola. Trying to do so will result in compilation not terminating.

Nikola functions are compiled to CUDA code and then linked dynamically into the running Haskell program. Then they are wrapped Haskell functions of the corresponding type. Exactly which types are possible is derived from a menu of instances of the typeclass `Compilable a b`.

An instance of this typeclass denotes that a Nikola value of type `a` may be compiled and wrapped into a regular Haskell value of type `b`. There are instances for all the various Nikola values, and it is possible to wrap Nikola arrays in many different vector implementations. Therefore it is not in general the case that `a` determines `b`, and often one is best served by just supplying a type signature.

However, this typeclass is not yet implemented for a lot of the possible choices of array representations and combinations. During the implementation of our case studies, we observed that trying to compile a function with an instance missing often resulted in very lengthy and elaborate type errors that were hard for us to decode. Luckily our ability to decode compilation-related type errors improved over time, but they are nevertheless still unpleasant.

Nikola also integrates with the array representation facility of Repa, by implementing a representation for arrays that reside in the CUDA device's memory. By using this array representation it is possible to have both Nikola programs and other foreign functions manipulate the same physical memory withot requiring any memory transfers of data to take place between host and device.

Chapter 6

Evaluation of project health

As mentioned in previously (see Chapter 4) we define the health of a project as how likely it is that development will continue, whether programming interfaces are subject to large change and properly documented.

We take the stance of a programming language user, which has to select a data-parallel language with which he will base derivative work. The goals of this chapter is not important for the rest of our project, as bad project health does not make a programming language an unsuitable subject for research.

In the following section we will look closer at the portability, documentation quality, installation process and stability of Repa, `Data.Vector`, Accelerate and Nikola.

6.1 Portability

Shared by all four languages/libraries is that they were developed for Glasgow Haskell Compiler, which currently is the defactor stsandard for Haskell development. We have only tested the libraries on this compiler and we suspect none of them will run else where because they rely on numerous GHC extensions (see Table 1).

Nikola and Accelerate only runs on NVIDIA GPUs as they rely on CUDA as intermediate language in the compilation process. An incomplete Accelerate backend for the more widely implemented OpenCL standard, has been in developed by one of the authors of this master's thesis (see [16]), and has not been maintained for quite a while.

We have collected a number of additional statistics about dependencies in Table 1, which we will let the reader interpret as he wishes.

6.2 Documentation

Repa, Accelerate and `Data.Vector` all have excellent documentation available on Hackage and both Repa and Accelerate's internals are documented in research papers [12, 27, 28, 29]. We have not been able to find information much documentation about `Data.Vector` internals, which would have been helpful in analysing its performance behavior.

Language	Dependencies	Reverse dependencies	GHC extensions	GHC version
Accelerate	5 (+18)	2	19 (+9)	7.6.1
Nikola	21 0	N/A	26	7.4.2
Repa	6	11	20	7.6.1
Data.Vector	4	150+	15	7.6.1

Table 1: Status on language dependencies. Reverse dependencies were counted on Hackage in October 2012. Nikola does not yet have a Hackage-release yet, why we could not count any reverse dependencies. When counting dependencies for Accelerate, there is one number for the `accelerate` package, the frontend, and an additional number for the CUDA backend.

Nikola does not provides almost no documentation, the most helpful resource is the research paper from 2010 [32], which is a bit outdated, as Nikola has undergone a major revision last year. Geoffrey responded quickly on any questions we sent him, which compensated a bit for the lack of documentation.

6.3 Installation process

Installation of Repa and Vector on GHC 7.6.1 was straightforward, as Repa depends on Vector. The circumstances quickly changed however when attempting to install Accelerate and Nikola. Both depended on some of the same packages, but with conflicting bounds on versions, triggering the infamous Cabal dependency hell. After having first solved the dependency hell by adjusting the version bounds of package dependencies and fixed minor superficial incompatibilities with the bumped versions we were able to install both Accelerate and Nikola. However, due to some deeper incompatibility somewhere we were unable to get both working at the same time, and thus had to work out something else.

In the end we managed to install every language package in separate package databases using the `hsenv`[1] tool. Using this method we also managed to install Feldspar, but using it was unstable, and we had to discontinue trying to get it working in order to make progress in the survey.

We spent a great amount of time just getting everything setup due to these infrastructural mishaps. See Section 8.6 for more.

6.4 Stability

There are two senses of the word stability that we want to evaluate, we both want to evaluate whether the projects are *likely to change* and whether they are *likely to break in use* (i.e. runtime errors).

Repa, Vector and Accelerate are all maintained at the University of New South Wales, and work on all projects seems active, with regular commits. We thus believe that work will continue. All three seems to have stabilized on a frontend (unlikely to change), though backend work on especially the Accelerate CUDA backend is still work in progress, and is thus unstable (likely

[1] Our fork is accessible at `https://github.com/dybber/hsenv`

Language	Project age	Latest release	License	Contributors
Accelerate	3-4 years	June 2012	BSD3	3
Nikola	1-2 years	Not released	Harvard (Berkeley style)	1
Repa	1-2 years	October 2012	BSD3	4
Data.Vector	4 years	October 2012	BSD3	9

Table 2: Project status. The number of contributors was acquired by inspection of commit logs. The data is from October 2012.

to break) and we have encountered several bugs and installation issues with the CUDA backend.

Nikola is maintained by a single author and is not yet released on Hackage. It thus receives little exposure and we've been unable to evaluate the number of users (which is thus potentially zero). There currently is not any active development activity. Also, there is almost no documentation, and we had to dig into the code to discover the capabilities and limitations. From a users perspective it thus not a wise selection for an implementation project, but that does not make it unsuitable for a research project.

Chapter 7

Evaluation of performance

We will now look at how the different languages compare in performance, by implementations of the cases described in Chapter 3.

7.1 Benchmark setup

In our benchmarks we only measure the time used by each of the different implementations. Other parameters, such as memory consumption on host and device, have been left out. For GPU implementations of the algorithms we can usually infer the device memory consumption directly from the program, but an analysis of host memory usage would have been interesting, but we had to limit ourselves and thus deemed it outside our scope.

The benchmarks are made on programs written in an as idiomatic style as we possible could, and if we have one available, we use a version written by a language expert. We are not interested in comparing fully optimized programs, as we want to measure how the languages compare in performance when the languages are used as the language author inteded it, and in a fashion appropriate for human reading. One exception is though CUDA, where we want to compare our "idiomatic" versions to a fully optimized CUDA implementation, employing all the tricks of the trade.

The output of all our benchmarks has been verified manually, through comparison of results with our reference R implementations of the algorithms.

Each of the Haskell benchmarks we have run were compiled with the same set of parameters. We use the llvm code generation backend and level-3 optimizations (via `-fllvm -optlo-O3`). Each of the CUDA benchmarks have been compiled with no extra flags. Our C benchmarks have been compiled using level-3 (`-O3`) optimizations.

Hardware

All benchmarks have been performed on a computer provided by the HIPERFIT research center running Ubuntu 12.04 LTS and CUDA 5.0.

The hardware specifications is as follows: 2 $\times$ AMD Opteron^{TM} 6274 processors (16 cores each, clocked at 2,2 Ghz), 132 GB (approx. 126 GiB) of main

Multiprocessors	2×8 SMs
Cores per SM	192 cores
CUDA cores	2×1536 cores
Clock rate	915 Mhz
Memory	2×2 GB
RAM technology	GDDR5
Memory bandwidth	2×192.3 GB/s

Table 3: Geforce GTX 690 specification

Language	**Time**
C	$67.03\mu s$
R	$89.13\mu s$
CUDA C	$66.23\mu s$
Haskell	$107.45\mu s$

Table 4: Benchmarking overhead

memory and a Quad SLI consisting of two GeForce GTX 690 (each a dual-head GPU). See full specifications in Table 3.

Ideally, we would have evaluated all benchmarks on two different systems, to make sure that our results are not specific to one setup. We did however only have access to this one machine with CUDA version 5.0 or newer, which was a required by the newest versions of both Nikola and Accelerate.

Software and compilation

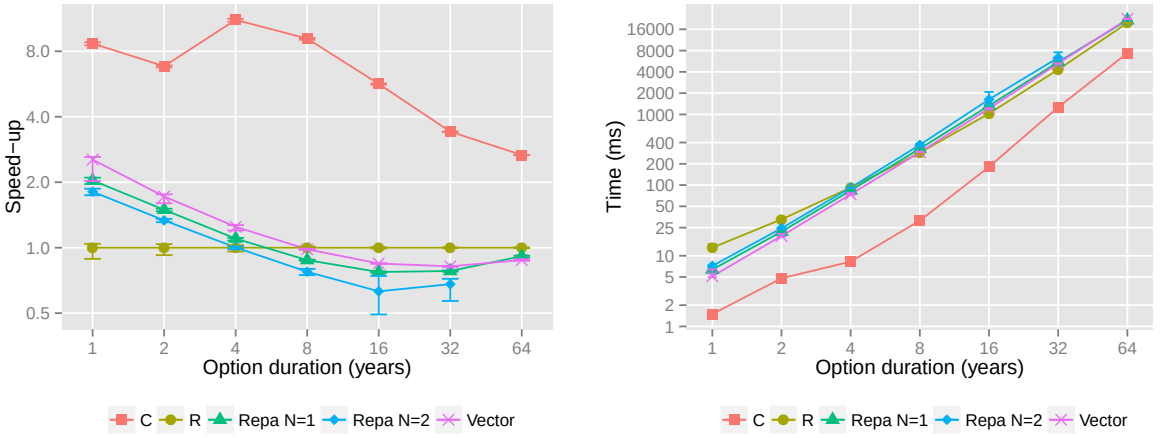
For each case, we have a set of different implementations and we want to measure the time used by each implementation on varying problem sizes. This was accomplished by developing a small benchmarking tool[1] that executed each experiment in succession, providing the input through a small protocol over Unix standard I/O, and letting the program respond with a result between each experiment. This response was used to stop the time measurement.

Several of the libraries and languages we have used requires individual compilation steps before the actual computation can take place. We have decided not to include all initialization in the running times, and only record the time used on the actual computation and eventual memory allocation and memory transfers. This is done to make the comparison fair, and because we think it is more important to optimize the actual computation. Optimizing the compilation or interpretation performance is also important, but we find that to be less of priority now and it thus out of our current scope.

The benchmarking tool was developed using the Haskell `criterion`[2] package. Each experiment was been executed 100 times and the mean and standard deviations was logged. To eliminate the possibility of errors in our approach, we have measured the overhead involved in issuing a request through standard I/O, by implementing an "empty" benchmark in each language, and measured the time used on a just responding. The results are shown in Table 4 (standard deviations are not listed, but are in all cases negligible). The significance of of course depends on the application, for most of our purposes it is too small to make a difference, we will note when it should be considered a source of error.

[1] The tool is documented in the README available at `https://github.com/HIPERFIT/vectorprogramming/blob/master/README.md`

[2] `http://hackage.haskell.org/package/criterion`



(a) Speed-up relative to R implementation.

(b) Absolute time usage.

Figure 5: Binomial option pricing (CPU).

7.2 Benchmarks

We will now present the results of our benchmarks. We have two plots for each benchmark: One showing the actual time used on each computation, and one showing relative performances. All scales are logarithmic.

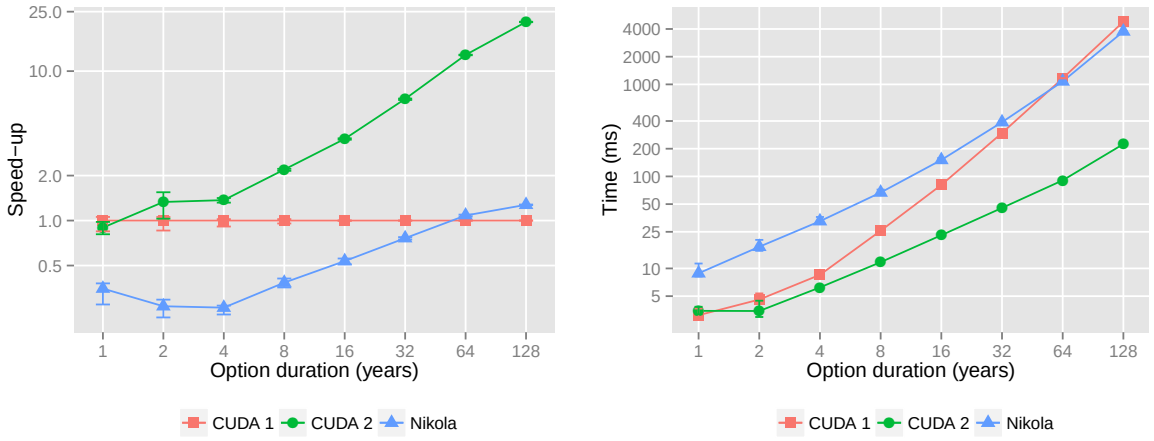
Benchmark 1: Binomial option pricing on the CPU

In our first benchmark we compare the performance of R and C languages to the Repa and the `Data.Vector` Haskell libraries, on the binomial option pricing case. All algorithms have been implemented following Algorithm 1, in Section 3.2. The R and C implementations are equivalently and are written by Rolf Poulsen, Professor of Financial Mathematics at the Department of Mathematical Sciences, University of Copenhagen[3]. The `Data.Vector` version and Repa implementations differ from these by not reusing memory, and are thus not completely fair to Algorithm 1, but using mutable arrays would not be idiomatic in these library. The Repa version has been executed with both one and two operating system threads (by setting the `+RTS -N` option).

The results are shown in Figure 5. We see that using an additional processor in Repa gives us a speed down. We have also tested with additional processors, which performs even worse. The amount of synchronization necessary between each iteration probably the problem. It would be interesting to see how Repa would perform if we were to execute it as a binomial pricer, but we have not had the time to do that experiment.

We also see that the overhead incurred by reallocating arrays in each iteration by the `Data.Vector` implementation is not that huge, we could suspect

[3]Obtained from `http://www.math.ku.dk/~rolf/FAMOES/`



(a) Speed-up relative to CUDA implementation.

(b) Absolute time usage.

Figure 6: Binomial option pricing (GPU).

that the garbage collectors liveness analysis detects that we can reuse the just deallocated memory region.

Benchmark 2: Binomial option pricing on the GPU

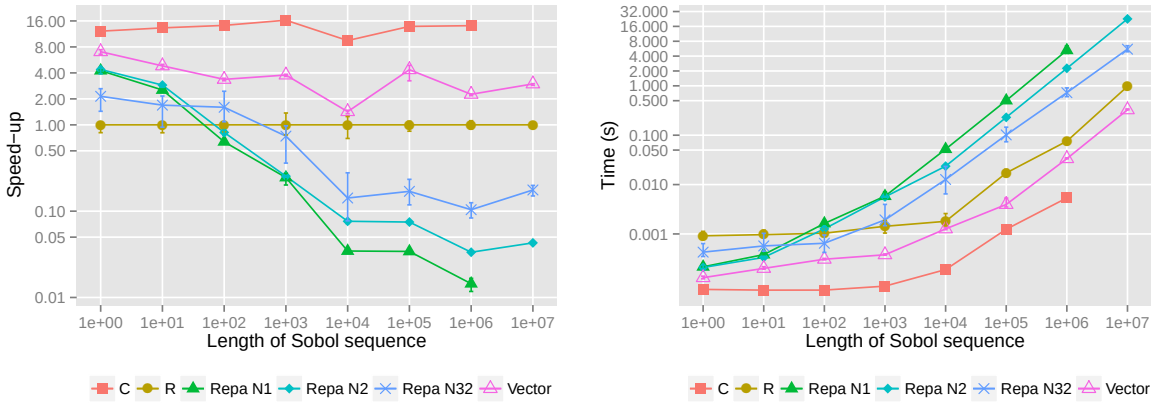
In our second benchmark we compare the performance of Nikola, written by the Nikola author Geoffrey Mainland, to two different CUDA implementations of binomial option pricing. We also wished to benchmark an Accelerate version, but we could not get a version to run consistently and often it would break down with a CUDA error ("unspecified launch failure", which signals a memory access error equivalent to a segmentation fault). The results are presented in Figure 6.

The first CUDA version is adapted from the portfolio pricer found in NVIDIA's CUDA SDK, Algorithm 2 in Section 3.2. This only runs on a single block and we see that it is not competitive for long option durations. At options running over 64 years (16384 time-steps), the Nikola version outperforms it.

The second CUDA implementation is faster in all cases, except the one-year option. We have written this version from scratch, and uses all available cores to price the option. Synchronization happens by returning to the CPU in each iteration, as we have to synchronize across all blocks. The reason why it outperforms Nikola is perhaps that the Nikola version does not reuse memory, and thus have to reallocate and deallocate in each iteration, in all other aspects this implementation is identical to the Nikola implementation.

Benchmark 3: Sobol sequence generation on the CPU

We now look at the performance of the Sobol sequence generators we have implemented, and in this section we will concentrate on the CPU versions



(a) Speed-up relative to R implementation.

(b) Absolute time usage.

Figure 7: Sobol sequence generation (CPU).

implemented in C, R, Repa and `Data.Vector`.

The R implementation is from the R-package `randtoolbox` which interfaces with a Fortran implementation that uses the recursive Algorithm 6, Section 3.3. This is also the algorithm we have implemented in `Data.Vector`, Repa and C.

The results are presented in Figure 7. We see that the `Data.Vector`-library in this case is competitive with R. Nonetheless, C beats all the others by a large margin.

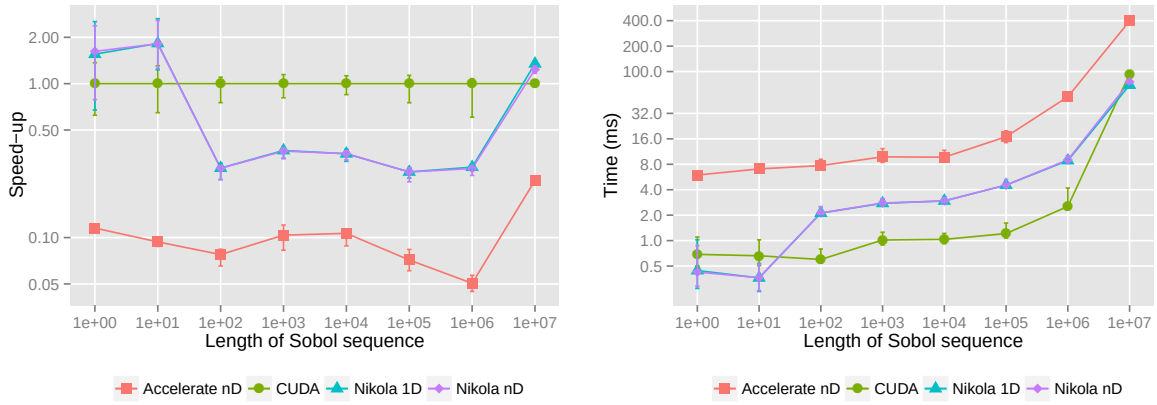
Even though we believe to have followed an idiomatic style for the Repa implementation, it does a bad job in the benchmark. This is perhaps due to the fact that the problem is not that numerically intensive, and thus we observe the overhead incurred by a more complicated runtime without realizing the full potential from parallelization. The Repa implementation running on 32 processors is only 7.2 times faster than the single-threaded version for 10^6 elements.

Benchmark 4: Sobol sequence generation on the GPU

For GPUs, we have Sobol sequence generation implementations in Accelerate, CUDA in addition to two Nikola variants. The difference between the two Nikola implementations is that the one tagged nD allows for computing multi-dimensional sobol sequence by providing several direction vectors (of the same length), and the $1D$ version is specialized to computing a single-dimensional sobol sequence. For Accelerate and CUDA we only have a multidimensional implementation.

The results are presented in Figure 8. First notice that both Nikola implementations performs equivalently, which says something to the amount of fusion Nikola is capabable of.

Even though the Accelerate and Nikola nD versions are written in identically style, Accelerate is slower in all cases. We are not sure what is the reason



(a) Speed-up relative to CUDA implementation.

(b) Absolute time usage.

Figure 8: Sobol sequence generation (GPU), including device to host memory transfer.

for that, but suspect that Nikola has done a better job at fusion. It also seems that we have had an error in this experiment, so it might not be valid to make any such conclusions.

The error can be observed for sequences of length 10^7, where Nikola suddenly perform better than CUDA. We tried profiling and found that the experiment was dominated by the memory transfer from device to host, and that it was in scheduling this memory transfer that Nikola did a better job.

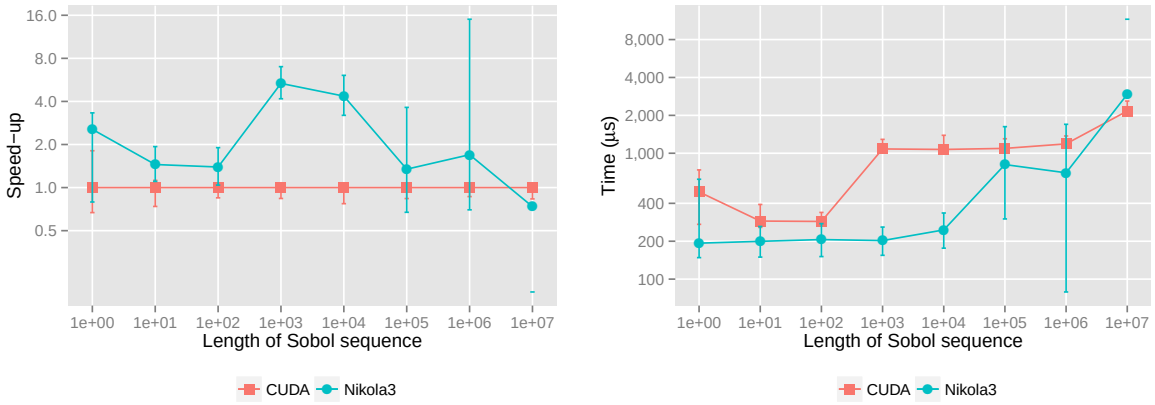
We attempted to get a better measure of what could be wrong by writing programs that only performed the actual computation on the device – where we are also most likely use the output of our QRNG – and measure the time used on the computation part, excluding time used on memory transfers. The result of this experiment is shown in Figure 9. Here we observe an opposing result, where Nikola is faster computation wise in all but the last case.

Benchmark 5: Longstaff and Schwartz option pricing on the CPU

Our last benchmark experiment is a comparison of CPU implementations of the Longstaff and Schwartz algorithm. We did not find time to implement Accelerate and Nikola implementations, and they would also require a working implementation of the `unfold`-operation described in Chapter 10. Also, some components, such as Cholesky factorization is not obvious how to implement in general in these languages, though we could always implement an unrolled version operating on a specific matrix size.

The results of the benchmark are displayed on Figure 10. Again we observe that Repa is slower than `Data.Vector` and R.

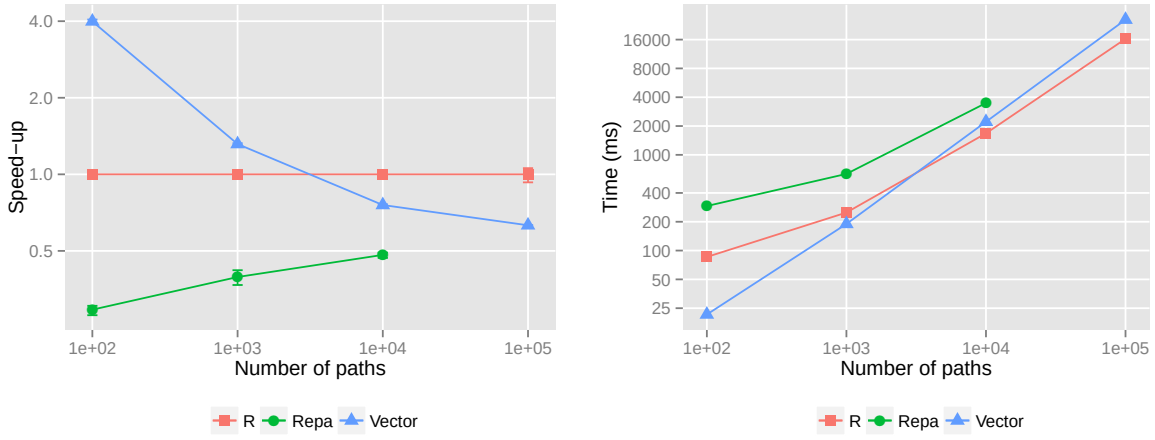
We also note that `Data.Vector` has an advantage over R for small problem sizes. This is probably because `Data.Vector` is written in a compiled language, whereas R is interpreted. For larger problem sizes, R takes the lead, which



(a) Speed-up relative to CUDA implementation.

(b) Absolute time usage.

Figure 9: Sobol sequence generation (GPU), excluding device to host memory transfer.



(a) Speed-up relative to R implementation.

(b) Absolute time usage.

Figure 10: Longstaff & Schwartz option pricing (CPU).

is probably due to much of the computation in R is delegated to external optimized linear algebra routines.

7.3 Sources of error

There are several sources of errors for our benchmarks, so we have only been able to make few conclusions based on them.

- The test machine we have used is shared among a large research, and although we have tried to avoid it, some of the benchmarks have been performed while other compute intensive tasks has been running simultaneously.
- When pricing an option, we have let most of the option configuration, except option duration, been compiled into our programs, and thus compiler optimizations might have reduced the complexity of the resulting code, and thus our benchmarks might not map directly to a real world scenario.

  The same holds for our Sobol-sequences generators, where the direction vectors have been similarly hardcoded.
- As mentioned all our Haskell programs have been executed with the llvm backend, with `-O3` given to the underlying llvm assembler. Certain GHC optimizitations, such as the rewrite rules that forms the basis for fusion in `Data.Vector`, has thus not been made and we should probably have used the GHC flag `-O3` in addition.

Chapter 8

Survey conclusion

In this chapter we summarise the conclusions made in the above summary, and make suggestions for further studies. In Part II of this dissertation we will look closer on a couple of the mentioned suggestions, the remaining can be seen as future work. Additional future work, not related to the outcomes of the survey, is presented in Chapter 12.

8.1 Language selection

We have compared the libraries Accelerate, Nikola, Repa and `Data.Vector` and our conclusion is that Nikola is the most suitable for further extensions when the scope is GPU programming. The Repa and `Data.Vector` architectures are not really compatible with code-generation, as would be necessary if they had to be modified to run on GPUs. Studying them was not entirely without results, as for instance the `computeS` of Repa can be viewed as the originator for the idea we will present in Chapter 11.

When comparing Accelerate and Nikola, we had the most pleasant installation experience with Nikola. The restrictive type system of Accelerate was also a factor that made us opt for Nikola. Especially the division between the expression and vector layer of Accelerate made us worried, as we would have to make major modifications to break this barrier down. Another plus in the favour of Nikola, is that it allows interaction with other CUDA libraries, e.g. CUBLAS, without having to get the data roundtrip to the CPU and back again (see Section 5.6).

Another choice is which languages would be suitable for application development. Currently `Data.Vector` with more than 150 reverse dependencies seems the most stable and practically useful. Repa seems stable, though hard to get to scale, though our problems might not have been the best fit for Repa.

The four languages that would be most likely to change in the future is Nikola, as only skeleton of the language exist; it still lacks many common parallel array operations such as prefix-sum, parallel reductions. It can be seen more as a testbed, than a practical language. Accelerate on the other hand has alot of such operations built in as skeletons, and if we did not have had the troubles with getting it to run on our machine, we would probably have had

been more keen at recommending it.

8.2 Unfold

As demonstrated in sections 3.3,3.3, several of our cases requires a way of building up an array from repeated application of a non-associative operator. Neither Accelerate nor Nikola provides built-in functionality that can be combined to do perform such operation. They can only construct new GPU arrays where there are no dependencies between elements (through `generate` and `fromFunction` respectively) or from existing CPU-arrays.

Chapter 10 will present our work on an implementation of such an idea in Nikola.

8.3 Nesting

We have in sections 5.5 and 5.6 shown how in certain cases, both Accelerate and Nikola breaks down in cumbersome notation, where algorithms could be written clearly if nested layers of array operations were allowed. We can in certain cases get out of this by relying on fusion of a surrounding `generate` function that indexes into the nested call. This will not work in the general case though, as it breaks down if we for instance raise the dimensionality of the involved arrays by one.

We argue that it is worth incorporating nesting in one way or another, as many algorithms are more naturally expressed in a nested style.

We will go into a possible way of mapping this to GPU devices in Chapter 11. The method is an alternative to full vectorisation and allows you to direct how parallelism is added to the program. Parallelism can be incorporated at several different layers of programs, and we thus provide a way of selecting between them.

8.4 Level of synchronization

In the implementation of the binomial option pricer we found it necessary to synchronize on the CPU to guarentee syncronization across all blocks. If we were to map this option pricer over a portfolio of different options, we would instead want something different. We would want each option to reside in their own block, and use CUDAs `__syncthreads` function to syncronize all threads in the block.

Thus, in algorithms that requires such synchronization, an analysis could be employed to determine when synchronization could happen in scope of single blocks and only when necessary, synchronize all blocks.

8.5 Memory reusage

In the binomial pricer we have seen that the performance of Nikola degraded as it didn't reuse the same CUDA memory areas that was already allocated, but had to reallocate space. In our case, where we are folding over an operation

that uses an array as accumulator, we would be able to determine whether we could reuse the array from earlier iterations. We already know the size of arrays after each iteration, since we need to allocate space before starting the operation, so we would be able to infer whether the accumulator would increase in size or not.

8.6 Haskell infrastructure

In setting up our test environment we found that the package management infrastructure of Haskell, the Cabal system, very brittle when installing different packages with common dependencies. This problem is infamously known by the name *cabal dependency hell*, and a lot of our alotted project time was spent to work around this problem.

Had the Haskell community adopted a package policy similar to that of the Comprehensive R Archive Network[1] of more agressive demands to maintainability, we would probably have been spared a lot of our trouble.

[1] `http://cran.r-project.org/`

Part II

Language Extension

Chapter 9

Nikola

As concluded in our survey we want to experiment with extending Nikola in pursuit of additional expressiveness. In order to give a more coherent picture we therefore first give a short introduction of Nikola programming and the parts of the implementation that we had to treat in order to make our extensions. Afterwards we present our takes on iterative array construction, and we present a new method for directing data parallelism.

Nikola is a deeply embedded, domain specific language in Haskell. It aims to provide a means to better exploit hardware capabilities for parallel numerical computation, as exemplified by OpenMP and (in particular) CUDA.

Surprisingly, we have found documentation on Nikola architecture and programming scarce, with the Nikola article [32] detailing mainly the embedding effort. In this chapter we try to make up for this by describing the architecture of Nikola in sufficient detail to provide a background for presenting our extensions.

First we present the frontend of Nikola, and then we describe the backend. Last we devote a section to a small overview reference of Nikola modules.

9.1 Frontend Architecture

In this section we describe the part of Nikola that serves as the language programming interface, as well as the middle layer which is still backend agnostic.

The Nikola language is made up of three data types: `Exp t a`, `Array r sh a` and the program monad `P a`. These parts are all backend agnostic. It also provides a type class directed `compile` function for compiling Nikola functions to CUDA code which is subsequently wrapped as regular haskell functions. `Exp t a` expressions are first translated to another first-order untyped abstract syntax before CUDA code generation.

Nikola scalar expressions are represented by the `Exp t a` type, with the phantom type variable `t` representing the target architecture for the expression, eg. `CUDA`. Thus, it is possible to have Nikola terms specialised for different backends. This is unusual, as programming languages are typically assumed

universal in the sense that every well-typed expression is executable on every supported machine architecture. Nikola is the first programming language we have witnessed to explicitly encode term portability in the type system. However, the ability to specialise expressions to targets doesn't extend into the lower layers of Nikola, so as such the `t` type variable only clutters up type signatures currently.

Arrays in Nikola are modeled by the type `Array r sh a`, an associated type of the typeclass `IsArray r a`. Arrays are parametrised on their representation and shape by the type variables `r` and `sh`, following in the tradition of Repa. The only common operation for all arrays is that of extracting their shape through the function `extent`. Array shapes are similar to those of Repa and Accelerate, denoted as instances of typeclass `Shape sh`.

Three principal array representations are provided by Nikola: The global array, the delayed array, and the push array, denoted respectably by the types `G`, `PSH` and `D`. This is a source of both control and complexity, as each representation gives rise to different features and restrictions. The global array represents a certain manifest range of memory cells, from which it is only referentially safe to read. Push arrays and delayed arrays however, represent array computations rather than actual areas of memory - only at the Nikola-Haskell border are they manifest into memory. An important consequence of this is that terms composed of delayed and push arrays undergo fusion by construction.

What distinguishs push arrays and delayed arrays is their perspective on the arrays they represent. A delayed array is simply a function from indices to values, while a push array may be viewed as a stream of index/value pairs, that may appear in potentially any ordering.

The type of the low-level abstract syntax is `S.Exp`, qualified to avoid confusion with `Exp t a`. This is elaborated in the nikola reference in section 9.3. `S.Exp` defines primitive constructs for annonymous functions, delayed arrays, accessing and manipulation of mutable arrays, and a `for`-loop construct. Many of these constructs map directly to corresponding C constructs.

The monad `P a` serves some of the same roles as the `IO` monad does in plain Haskell. One must for example only manipulate mutable arrays from within the `P` monad. It is a type alias for the slightly more elaborate monad type `type P a = R S.Exp a`. The monad `R r a` is the reification monad, used to convert the various frontend datatypes such as `Exp t a` and `Array r sh a` into the low-level abstract syntax `S.Exp`.

To enable this, monad `R r a` is a continuation monad. But instead of direct access to the underlying continuation, Nikola uses delimited continuations, described in [53] and introduced first in [17]. The interface to using delimited continuations consists of two special operations:

```
shift :: ((a -> R r r) -> R r r) -> R r a
reset :: R r a -> R r r
```

Which when specialised to the `P` monad becomes:

```
shift :: ((a -> P S.Exp) -> P S.Exp) -> P a
reset :: P a -> P S.Exp
```

`shift` and `reset` work in tandem. A full account of the use and details of delimited continuations is out of scope for this thesis, but consider this short typical usage pattern:

```
reset $ do
  ...
  y <- shift $ \k -> do
    ...
    x <- k ""
    ...
  ...
```

In this `shift` expression, `k` is a continuation representing all that is going to happen up until the enclosing `reset`. Upon invoking `k ""`, control shifts outside of `shift`, and `y` is bound to the empty string `""`. Upon reaching the end of the monadic action inside `reset`, control is shifted back into `shift`, and `x` is bound to the result of the enclosing action. The eventual result of the action inside `shift` then becomes the result of the enclosing `reset`. Informally, `shift` and `reset` turn the code inside-out.

9.2 Nikola Backend Architecture

Compiling a Nikola function relies on both reification to `S.Exp`, a mechanism to determine the type of the resulting Haskell function, and the translation from `S.Exp` abstract syntax to C abstract syntax. Reification is handled by the typeclass `Reifiable a b`, and the Haskell-interfacing parts of compilation by `Compilable a b`. The details of how Nikola manages to interface Haskell with the compiled Nikola code is documented in [32].

All it takes for a construct to be part of the Nikola language frontend is proper instances of these two type classes. So as such, Nikola is very extensible, provided that the programmer is capable of lifting the burden of expressing his additions in terms of the low-level abstract syntax.

Once a term has undergone reification, it is translated into C abstract syntax. This is a somewhat straightforward translation, as many of the components correspond directly to C.

9.3 Nikola module overiew reference

In this section we provide a small overview of central modules and namespaces in Nikola for reference. These modules all reside in the `Data.Array.Nikola` namespace.

Exposed frontend modules

`Exp` Exports the type `Exp t a` and typeclass `IfThenElse` for use with rebindable syntax. The type `Exp t a` will sometimes appear as qualified name `E.Exp` in case of ambiguity.

Array Defines `IsArray r e` typeclass with associated type `Array r sh e`. It also defines the `Source r e` typeclass for arrays that support indexing.

Shape Defines types `Z`, `sh :. a` and the typeclass `Shape sh`, with instances for `Z` and `sh :. a`.

Repr.* Contains modules defininig the array representations for Push, Delayed and Global arrays through `IsArray` instances.

Low-level frontend modules

Language.Syntax Exports the `Exp` datatype, which is a first-order abstract syntax for primitive Nikola expressions. The abstract syntax contains constructs for manipulating memory arrays, looping and annonymous functions. It also exports the datatypes `Type` and `ScalarType` used as church style type annotations for `Exp` values. In the case of ambiguity we quailfy `Exp` as `S.Exp`.

Language.Check Contains a type checker and simple type inference for `Exp` values. Also defines the typeclass `MonadCheck m`, to allow type checking to be performed in any monad that defines how to access to the typing environment

Language.Monad Defines the continuation passing `R r a` monad, and the type alias `type P a = R Exp a`. Also defines the combinators `shift` and `reset` for delimited continuations [53].

Language.Reify Exports the `Reifiable a b` typeclass. It provides the operation `reify :: a -> R b b`, which is the central piece in converting frontend data types `Exp t a` and `Array r sh e` to primitive `Exp`.

Language.Optimize.* Provides various optimizations on Nikola `S.Exp` values exclusively. Most notable is the detection of let-sharing, see [32]. Marking parallel for-loops at the top level to be translated to CUDA kernels is also treated as an optimisation in Nikola.

Backend modules

Backend.CUDA.{Haskell,TH} Define the `compile` function for runtime and compiletime usage respectively. This is the exposed part of the backend.

Backend.C.Codegen Generates C code from `S.Exp` values, represented as C abstract syntax.

Apart from these modules Nikola also provides various implementations of vectors in the `Data.Vector.CUDA` namespace. These vectors are stored in device memory and integrate with `compile` to allow control of the timing of host-device data transfers.

Furthermore these vector implementations are integrated with Repa as additional array representation types in modules in the `Data.Array.Repa.Repr.CUDA` namespace.

Array representation capabilities.

The operations that arrays may be used with is modeled with the use of type classes.

- The `Source r e` type class specifies an `index` function that allow a consumer to extract values of type `e` from an array with representation `r` and contents `e`.

  Instances:

  ```
  Source D e
  IsElem e => Source G e
  ```

- The `Target r e` type class enables mutable access of `r`-arrays in the `P` monad.

  Instances:

  ```
  IsElem e => Target G e
  ```

- The `Load r sh e` type class enables manifestation of `r`-arrays into `Target r' e` arrays in the `P` monad.

  Instances:

  ```
  Shape sh => Load sh D
  Shape sh => Load sh PSH
  ```

Chapter 10

Iterative array construction

In this chapter we present our takes on adding support for expressing iteratively defined arrays. Unfortunately we have not been able to get any of these extensions completely ready for use and possible inclusion into Nikola. We will however give an analysis of what is lacking for a more complete implementation.

Our need to iteratively define arrays is motivated directly from our case work on the variants of the recursive Sobol-sequence generator, depicted in Algorithms 6, 8 and 4. While in general the needs for a specific example does not always warrant language extensions, the case of incremental construction of output data is arguably a common one, and thus well motivated.

But in isolation, the ability to construct a single array iteratively is insufficient. It must also be possible to produce multiple iteratively constructed arrays in parallel. This is due to both the desire for performance and the desire for expressivity. To meet this end we will analyse two possible ways to go about this: a nested parallel version and a flat parallel version.

10.1 Nested data parallel version

Since it is not possible to use recursion in Nikola, we must implement a primitive iterative operator to incrementally construct arrays.

In Haskell, the function `unfoldr` of the `Data.List`-module has the type `(b -> Maybe (a,b))-> b -> [a]`.

`unfoldr f a` recursively populates the element of a list with `f`, as long as it evaluates to a `Just`-value. It is a well known way to construct a list, so we pursue that as an extension.

In Nikola the length of an array is the only intrinsic property shared across all array representations. Therefore we must know the length of the result array a priori to evaluating a Nikola `unfoldr`.

For a prototype that only uses the last generated element as state, we settle for the type signature:

```
unfoldr :: (b -> Maybe (a,b)) -> b -> [a]
```

It recursively populates the element of a list with `f`, as long as it evaluates to a `Just`-value. It is a well known way to construct a list, which further motivates it as an addition to Nikola.

In Nikola the length of an array is the only intrinsic property shared across all array representations. Therefore we must know the length of the result array a priori to evaluating a Nikola `unfoldr`. For a prototype that only uses the last generated element as state, we settle for the type signature:

```
unfold :: (Shape sh, IsElem a, Typeable a)
       => sh -> (sh -> a -> a) -> a -> Array PSH sh a
```

In order to be useful however, `unfoldr` by itself is not enough. We need to be able to iteratively produce entire rows or columns of a matrix as required in Algorithm 3. To remedy this, we set out to explore nested maps. Again, lacking nested arrays we must use array shapes to express the structure of our computation. Like in the case of `unfoldr` we thus must express the resulting shape beforehand.

The type we eventually arrived at for our maps capable of nesting is depicted here:

```
mapNest :: (Shape sh, Shape sh', IsElem a,
            IsElem (Exp t Ix), IsElem b, Source r a)
        => sh'
        -> (Exp t Ix -> Array D sh a -> Array PSH sh' b)
        -> Array r   (sh :. Exp t Ix) a
        -> Array PSH (sh' :. Exp t Ix) b
```

While we require the shape `sh'` to be specified, there is no guarantee in the application `mapNest f xs` that `f x` will not produce an array exactly the size denoted by `sh'`. If the resulting array is smaller than denoted by `sh'`, an eventual manifestation of the map to memory will leave some cells uninitialized. Similarly, if the resulting array is not bounded by `sh'`, manifesting it will result in writing out of bounds.

For now we have chosen to accept the existence of these fallacies, partly because we are concerned with implementing a prototype, and partly because we do not see any acceptable resolution of the problem beyond placing the actual array extent rather than just the shape into the type of the array, as done in [51], but such a change to Nikola is outside the scope of our project.

Another issue with this definition is the `Source r a` restriction on `xs`. If the mapped function does not need to index into the array, it is an unnecessary restriction to impose. Similarly, if the mapped function performs only element-wise operations, preserving the property of being indexable would be valuable to retain for possible further processing of the result array.

Risking to further complicate the types, the situation is somewhat remediable by defining `mapNest` as a function in a typeclass parametrised over the involved array representations, as is already the case for typeclass `Map`.

The most important issue about this definition however, is that it permits expressing nested array operations, which is not directly executable on the flat parallel hardware. Treating this issue is the subject of Chapter 11.

The inadequacies of our implementation

All our efforts on iterative array construction draw on Nikola's implementation of 'push' arrays. As mentioned earlier, delayed arrays and push arrays are each others opposites. Delayed arrays are `Source` arrays, and their manifestation is at the mercy of the consumer of the array. Therefore it is required that the elements of a delayed array are completely independent from each other. Push arrays on the other hand are defined as a stream of index-value pairs, and thus need not obey the same restriction on element dependence.

Push arrays in Nikola are defined by means of the loop constructor `ForE` of `S.Exp`. This loop constructor however, is very low-level in the sense that it includes as arguments a list of variables to be looped over and a list of loop bounds corresponding to each of the variables, and corresponds directly to a `for`-loop in C.

However to our regret, using this approach for defining `unfold` and `mapNest` proved unfruitful, as the part of the C code generation that pertains to for-loops in Nikola does not handle cross-iteration data dependencies properly. It assumes that variables that are overwritten inside the body of the loop constitute new, fresh variables. Due to our time constraints we were not able to address this, and have to leave our operations be for now. If however we disregard this problem, we are not aware of any more obstacles for implementing our extensions properly.

10.2 Flat data parallel version

Our first attempt at defining an unfolding operation was inspired by the collective operators of Accelerate.

```
unfold :: (IsElem a, IsElem (Exp t Ix), Shape sh, Source r a)
       => Exp t Ix
       -> (Exp t Ix -> a -> a)
       -> Array r sh a
       -> Array PSH (sh :. Exp t Ix) a
```

By avoiding the need for a nested array operations, it is guaranteed that the resulting unfolded arrays are regular. Furthermore, since the operation is a mapped version of `unfold`, the outer structure of the computation is completely static, and it is possible to assign a parallel execution semantics, given that the unfolded function is executed sequentially.

As such it fits quite well with the other array operations in Accelerate. If it were to be implemented in Accelerate or Nikola, it would enable us to express the optimised version of Sobol sequences, Algorithm 8, which is currently inexpressible.

An implementation using the `unfold`-operator can be seen in Algorithm 9. As xs is a 1D-array, the returned array, but in memory the values are in the order of the Sobol sequence. Thus, we just has to tell Nikola or Accelerate to interpret it differently, which can be done using the built-in `reshape`-function found in both languages.

function SOBOLSKIPPING-UNFOLD(v, n, p, b)
  $xs \leftarrow$ **parmap** (λi. -> SOBOLINDUCTIVE(v, GRAYCODE i)) $[0..2^{p+b} - 1]$
  $ys \leftarrow$ **unfold** n ($\lambda i.\lambda x_i$. SKIPAHEAD(p, b, i, x_i)) xs
  return ys
end function

Algorithm 9 Parallel Sobol sequence generator. v is the direction vector, $n \cdot 2^{p+b}$ is the length of the sequence, 2^p is the block size and 2^b is the number of blocks.

10.3 Discussion

Which of the two possible implementations to select depends on ones point of view. From one point of view, all programs that type checks should be correct and come with guarantees of good performance. As we have seen this style does in some cases break program composability, but the other aspects are certainly also worth pursuing.

In this case, we find that the one invariant a developer has to manually enforce when applying our `mapNest` is worth the extra cost in man hours, when compared to reading and writing programs in the style we have seen in Section 5.5.

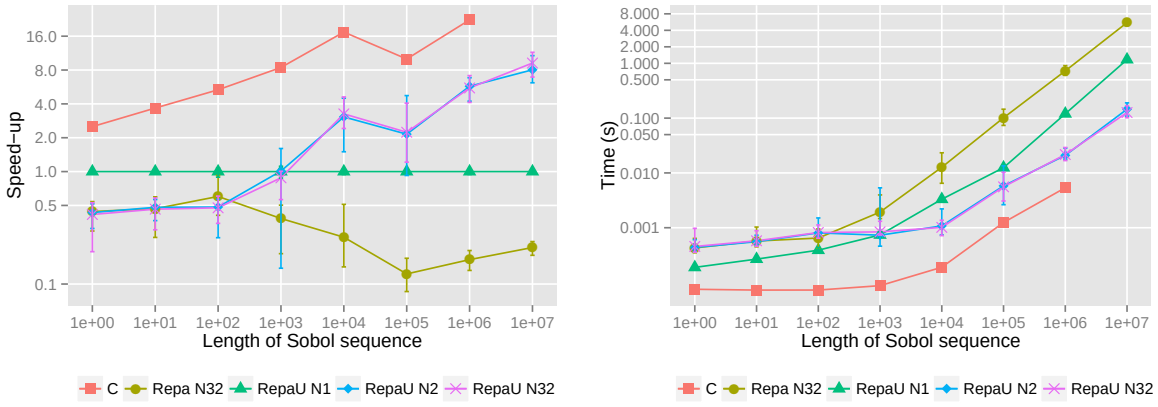
10.4 Repa implementation

As seen in our survey, we also had problems getting our parallel CPU version in Repa to deliver a speed up. Actually, it only degraded in performance when we split the problem over several processors.

We have implemented a function for Repa similar `unfold` described above, which allows us to implement Algorithm 7 (we do not need the SOBOLSKIPPING algorithm, as we do not need to address the problem of memory coalescing). The code interface is presented below and the results of a benchmark run is presented in Figure 11. We see that we now perform better for both one and two processors, but the difference between 2 and 32 processors is not noticeable. This might be an error in our benchmark setup that we have not discovered, but at least it all in all it performs significantly better than our previous Repa-version.

Another aspect worth recognizing, is that even though we scale better, we still does not catch up with the sequential version written in C.

```
unfoldFillChunkedP
 :: Int                 -- ^ Number of elements.
 -> (Int -> b -> IO ()) -- ^ Update function to write into result buffer.
 -> (Int -> a -> a)     -- ^ Recursive function to get the value at a given index.
 -> (Int -> a)          -- ^ Inductive function to get the value at a given index.
 -> (a -> b)            -- ^ Generate output (useful for fusion)
 -> IO ()
```



(a) Speed-up relative to CUDA implementation.

(b) Absolute time usage.

Figure 11: Sobol sequence generation on the CPU. We use RepaU to denote benchmarks using our new Repa implementation using the `unfoldChunkedP`-construct.

Chapter 11

Directing parallelism

As presented in the survey, Chapter 5, we have found that flat data-parallelism in the style of Accelerate and Nikola (without our mapNest extension) is a non-composable and unnatural program structure that hides the intended meaning of a program in clutter.

To look for alternatives, we have taken a look into nested data-parallelism, and how it can be implemented in a GPU language. We have contemplated an implementation of the vectorisation transformation (see [9]), but ended up opting for a solution that is more lightweight and hopefully easier to understand the execution semantics (and thus performance properties).

Currently our effort concerning the implementation of our proposal into Nikola has been mostly hypothetical but we find it worthy of further study. While we have made some effort at implementing it in Nikola, this is far from complete.

In the next section we will introduce our take on nested data-parallelism, and in the remaining sections of the chapter we will discuss the implications and alternatives to such an extension.

11.1 The `sequential`-operator

Thus motivated, we will now take a look at a typical nested data-parallel program, inspired by our binomial option pricing case, and the employ our suggested extension, to direct how the parallelization is performed.

Algorithm 1 from Section 3.2 can be written in functional style as the function:

```
binomial optionParameters = foldl (map f ...) bs [t..0]
```

Here we have left out quite a lot of detail, to keep focus on what is important for our example, the structure of the program.

We have seen one way this can be parallelized by issuing the **`map`** in parallel and synchronize between each iteration of the fold, but we will now look at the problem of portfolio pricing, where several options are priced simultaneously. Do to time constraint we have not studied this problem closely in the survey, but it serves as a good illustration in the current context. In this case, there are

two possible parallization strategies. We can either take one option at a time, and parallize in the way just described:

```
portfolio₁ = map binomial
```

Alternatively, we could issue the pricing of each option as a separate thread, where the pricing of the individual option is a sequential algorithm:

```
portfolio₂ = parmap binomial
```

where the **map** of the the `binomial` algorithm is executed sequentially.

For now we will look at the theoretical performance of these two implementations. Defining t_i to be the duration time of option number i in the portfolio consisting of n options, here is the expected depth complexities for each variant:

$$\texttt{portfolio}_1 = \text{Depth}\left(\sum_{i=1}^{n} \texttt{t}_i\right)$$
$$\texttt{portfolio}_2 = \text{Depth}\left(\max_{i=1}^{n} \frac{\texttt{t}_i(\texttt{t}_i+1)}{2}\right)$$

The first case, `portfolio`$_1$, will plausibly be an efficient strategy for few long-running options of various length, as there are few inputs that each display good parallel resource usage. On the other hand `portfolio`$_2$ will be more efficient for problems with many options of shorter and similar length.

Our proposal

Now, what we propose is to add a primitive construct to Nikola that serves as a separator between what should be executed sequentially and what should be executed in parallel, the idea being that it should be simple to move between different parallelization strategies without changing the structure of the code. We have chosen the name `sequential` for the construct that represents this division.

In the above example, we could use `sequential` to write `portfolio`$_2$, without having both a **parmap** and a **map**:

```
portfolio₂ = map (sequential binomial)
```

which should be understood as "convert `binomial` to a sequential function, whereby the outer **map** can be executed in parallel". Such a conversion is of course possible, as all parallel programs can be rewritten as a sequential program.

One should then think about a map as a "potentially parallel" operation, and only after investigating the point of use can we determine whether it is to be evaluated in parallel or sequentially. In this case the inner **map** ends up being evaluated sequentially rather than parallel, the **foldl** operation does not have a parallel version, and thus only have sequential semantics.

Informally, the suggested parallel execution semantics of programs written using `sequential` is:

The innermost potentially parallel operation enclosing the expression with the `sequential`*-marker gets to be executed in parallel.*

As mentioned, the **`foldl`** in the example can not be executed in parallel as there is a strict dependence between each iteration, but there are still more than one possible execution strategy for the reduction. If we look at the case of $\texttt{portfolio}_1$ where there is no use of `sequential`, we have the semantics that the innermost **`map`** will be executed in parallel. Thus, the **`foldl`** will have to be a host function, that serves as synchronization point between each parallel **`map`**.

That is, our proposal has the influence that we merge the host and GPU code into a single language. Depending on the placement of `sequential`-markers in the code, we get different theoretical complexities.

We can view `sequential` as a meta-language construct, which in a preprocessing step converts our language into a lower level language where array operations are materialized as concrete implementations which are either: sequential device code, parallel device code or sequential host code. We are not suggesting that this is the optimal strategy for an actual implementation of a language with `sequential`.

Using Nikola or Accelerate, one is to write GPU code in the embedded DSL and use standard Haskell constructs in all other cases. With our proposal we unify host and device code of a problem into a single language.

Evaluation strategy

There are several ways this can be mapped to a concrete machine with a graphics card. We suggest that the runtime uses the syntax tree as a an acyclic dependency graph, and for all kernels where all dependencies are met, executes them concurrently and asynchronuously.

Synchronization with possible dependencies is then necessary before any operation waiting can to be executed.

Executing the kernels asynchronously leaves CPU time for other tasks, and we might be able to extend this to a heterogeneous language. See Section 11.3 below.

11.2 Consequences and complications

Even though the semantics explained above might seem simple, which is also one of our goals, there is still a lot of complications that we have to be aware of. As an example, how does `sequential` interact with conditionals, as in the following example:

```
map ( \b -> if b
            then sequential (map f ys)
            else map f ys)
```

We have to decide the parallel execution semantics of the outer **`map`** before we can launch it. In such cases we have to make special cases for our otherwise simple rule above. One option could be to make `sequential` spill over to the **`else`**-branch, and the outer **`map`** would be a parallel map. Another option is that

the outer map is executed as a host function and depending on the value of `b` we execute the inner **`map`** sequentially on the host version or in parallel on the device.

A third alternative is to add restrictions such that the above code is not possible to write. That is, such that each branch of the sequential must located on the same execution level. This is perhaps possible by adding the execution strategy to the type of expressions, we will discuss this further below (see Section 11.3)

Another problematic, but similar, example is functions of several arguments:

```
zipWith f (sequential as) bs
```

Here **`zipWith`** is forced to proceed sequentially, as the elements of `as` must be evaluated sequentially. However, any potential parallelism in `f` and `bs` would still be exploitable, and the two even fusable.

The type of `sequential`

In the Nikola context, it is worth looking at how `sequential` can be incorporated into the existing type system. The semantics of the operation is purely about execution and it thus has a type similar to identity function, with the restriction of working on Nikola terms. One possibility is to give it the type

```
sequential :: Array r sh a -> Array r sh a
```

but in this case we would not be able to write our $\texttt{portfolio}_2$ as we did, as sequential would not accept a function argument. Our code would instead be:

```
portfolio₂ = map (sequential . binomial)
```

The only thing that all Nikola terms have in common is instances of typeclass `Reifiable a S.Exp`. So the most inclusive type would be `sequential :: Reifiable a S.Exp -> a -> a`. However, we do not take for granted that a function of that signature with our intended purpose is definable.

We have speculated about various ways to express a code marker such as `sequential`. Our initial idea was to add yet another type variable to Nikola expressions, such that the "sequentiality" of a term would be encoded in the type. The types of **`map`**, **`zipWith`** and so on should then implement the semantics of switching mode in accordance with their arguments.

While this idea appears to hold some merit, in the concrete case of Nikola we made the judgement that the type system was already saturated with modelled constraints, and thus we settled for an untyped version.

Possible runtime errors

As mentioned in the previous chapter, the introduction of nested maps in the form suggested by our `mapNest` Nikola-construct, also introduces the possibility of runtime errors, as the suggested contruct does not prevent irregular arrays, see Section 10.1.

Further problems such as this one might also arise for other operations, when defining nestable versions.

In our mindset we have adopted the notion that seems to prevail especially in high performance computing, that it is sometimes better to be complete with regards to expressivity and potentially unsafe, rather than offering full static safety but limited expression.

11.3 Extensions and future work

We of course have a lot of future work, as the suggestions in this chapter are highly hypothetical, in that we have not found time to finish an actual implementation. There might (and we are almost sure that there will) be more corner cases that we have not thought about, than those mentioned in Section 11.2 above. In addition to the obvious task of making a real implementation, we now look at a further set of extensions that might be interesting to look closer at.

Heterogeneous computing

The idea of a cursor such as `sequential` might extend to other divisions between different types of parallelism in heterogeneous computing. We might have a cursor to mark that something should not be attempted to be parallelized on a GPU, but should be executed on the CPU. Such a cursor might allow one to break out of our language and back into the full Haskell language by exposing the `IO`-monad or more restricted monad, that would allow more general computation than what we can provide on a GPU.

Another cursor might be used to indicate were we divide between block level parallelism (where threads can synchronize with its peers directly on the GPU) or full parallelism where the threads does not have the possibility of synchronization on the the device. Such a cursor might make it possible to write a version of the binomial portfolio pricer similar to GPUBINOM in Algorithm 2.

A final idea would be cursors to allow tasks to be partitioned between several devices, such as several graphics cards or a cluster.

Again, we have done a lot of speculation on these matters, but as we do not have a concrete implementation, we can not conclude further on their practicallity.

Experimentation

Another interesting aspect of such a light-weight construct as sequential, that only instructs how computation is mapped to the hardware, is that we can experiment using different parallelization strategies, just be moving it around in the code. This might even allow for automatic experimentation of different parallelization strategies on the current hardware, where `sequential` automatically, such that it will not even have to be exposed to the programmer.

One approach is to employ *machine learning* on smaller training runs to select between host or GPU execution as in the Qilin system [31]. Alternatively, an approach based on static program analysis could be employed. Here it has also been suggested to use machine learning to select whether a task is fit for

GPU or CPU execution, though using program features rather than training runs [20].

Repa

Repa also employs a notion of array manifestation in separate parallel and sequential markers.

We find that `sequential` might be able to replace both `computeS` and `computeP` to give Repa a simpler execution strategy with less space for runtime errors. As mentioned previously, nested calls to `computeP` will result in runtime errors. Contrary to Nikola, we have speculated that it might be possible to implement a `sequential` construct as a type parametrised monad, but this remains a speculation.

Chapter 12

Conclusion

12.1 Future Work

Nested data-parallelism on NVIDIA Kepler GPUs

On newer GPUs supporting CUDA 5.0, such as the one we have had access to, allows for 20 levels of nested calls, such that a kernel can spawn other parallel kernels directly on the GPU, without returning to the CPU. NVIDIA uses the term *dynamic parallelism*[1]

We only discovered this late in the project, and we have thus not contemplated much on the idea of employing them for our `mapNest`-function. It should definitely be consider to use new hardware capabilities for the problem, and it thus a thing that would be worth looking closer at in the future.

Prototype implementation of `sequential`

Our idea of 'sequential' should have been introduced in previous sections, but we should suggest implementing it more fully than we have had time to and investigate which complications will arise, which further extensions that are possible. All in all, we believe there is a great potential in a `sequential`-construct, although we can not do much more than speculate on the implementability and practicality.

For further ideas for future work on `sequential`, see Chapter 11.

Broader scope of survey

Many languages was left out in our survey, and only few parallization patterns have been tested. It would thus be worth extending our survey to cover other languages such as Theano, Feldspar, Obsidian, Bohrium and Copperhead.

When it comes to algorithms implemented, we would like to implement portfolio option pricers for both the binomial model and the LSM algorithm, as they might giver better idea about the performance of the languages for embarassingly parallel tasks.

[1] A short example is provided in `http://developer.download.nvidia.com/assets/cuda/docs/TechBrief_Dynamic_Parallelism_in_CUDA_v2.pdf`

On a final note about the future of our survey, we have during the project found that this community of data-parallel languages, really need a way of comparing their work, to make it more competitive. We suggest that the community takes inspiration from the Computer Languages Benchmark Game[2], and creates a similar comparison between data-parallel languages where results are updated regularly. There could be both a CPU and a GPU category.

For further ideas for future work on `sequential`, see Chapter 11.

12.2 Conclusion

In this master's thesis we have explored the state of contemporary data-parallel languages, with a focus on functional programming languages. We have compared the embedded languages Nikola and Accelerate with the Haskell libraries `Data.Vector` and Repa, and with the programming languages R and CUDA. This we did by selecting and describing a set of algorithms from the financial domain, and comparing the merits of the implementation in each of these languages.

In our comparison we uncovered great differences among the languages in both expressive power and performance. Some of these findings confirm our expectations from inspection of the superficial circumstances regarding each language, such as the conjecture that Haskell programs in general have an advantage to R in being compiled rather than interpreted, though R can make up for this with rich, optimised foreign function bindings, as witnessed by Figure 10(a).

More interestingly, we have described some of the protruding differences between Accelerate and Nikola, both in terms of expressive power and performance.

Accelerate adopts the view that only programs with a guaranteed effective implementation should be expressible. This view is manifest in the decision that Accelerate only exposes flat data-parallel constructs, and delegates to the compilation process the task of fusing the different components of an Accelerate program into a coherent whole. We found however that this limitation of expressivity was not only hindering us from implementing our cases at all, but also resulted in inferior performance relative to Nikola. While we do not say that it will be impossible to improve the quality of the optimisations of Accelerate to make it competitive, we do consider it detrimental to the current state of the language, being both difficult to use and slow as well.

Nikola we found to have taken an opposite approach, namely to take an offset in the capabilities of hardware, and thus exposing language constructs that more closely correspond to those capabilities. While this results in a language that tries to give less static guarantees about performance and safety, we also found the result to be a more extensible language, with a much smaller conceptual gap between the language and the hardware. While this arguably reduces the theoretical optimization options, our concrete finding for now was that Nikola programs had more expressiveness available to them, and resulted in better performance than their corresponding Accelerate implementations.

[2] `http://benchmarksgame.alioth.debian.org/`

We thus reasoned that choosing Nikola would yield a path with better chances for incremental improvements in both performance and expressivity, perhaps eventually towards the theoretical ideal of static guarantees employed by Accelerate.

Having selected Nikola for further study, we first sat out to improve the expressivity of the language, by implementing an operation for iterative array construction, and by bypassing the restriction of nested expression by permitting nested maps. Because we do not give up the restriction of only allowing regular arrays, some caveats apply to the use of nested array operations, as we have no way to ascertain statically that the mapped function does not yield arrays of different shape depending on the input.

Unfortunately, due to the selection of Nikola happening relatively late in the progression of the project, we have not been able to get our extensions working properly. We have however presented speculations of a possible solution to the compilation problem of nested vector operations through the `sequential` construct, inspired by how Repa allows the programmer to dictate the parallel or sequential execution mode by means of the `computeS` and `computeP` functions.

We reccommend the notion of dividing execution mode by the means of markers such as `sequential` as a relevant topic for future research, due to its apparent simplicity and versatility.

Bibliography

[1] ABELSON, HAROLD and GERALD JAY SUSSMAN. "Structure and interpretation of computer programs". In: (1996).

[2] ACWORTH, PETER, MARK BROADIE, and PAUL GLASSERMAN. "A comparison of some Monte Carlo and quasi-Monte Carlo techniques for option pricing". In: *Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing* 127 (1998), pp. 1–18.

[3] BEASLEY, J. D. and S. G. SPRINGER. "Algorithm AS 111: The percentage points of the normal distribution". In: *Applied Statistics* (1977), pp. 118–121.

[4] BERGSTRA, JAMES et al. "Theano: a CPU and GPU Math Expression Compiler". In: *Proceedings of the Python for Scientific Computing Conference (SciPy)*. Oral Presentation. Austin, TX, June 2010. URL: `http://www.iro.umontreal.ca/~lisa/pointeurs/theano_scipy2010.pdf`.

[5] BERGSTROM, LARS and JOHN REPPY. "Nested data-parallelism on the GPU". In: *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*. ACM. 2012, pp. 247–258.

[6] BLACK, FISCHER and MYRON SCHOLES. "The pricing of options and corporate liabilities". In: *The journal of political economy* (1973), pp. 637–654.

[7] BLELLOCH, GUY E. *NESL: A Nested Data-Parallel Language*. Tech. rep. CMU-CS-93-129. School of Computer Science, Carnegie Mellon University, Apr. 1993.

[8] BLELLOCH, GUY E. "Programming parallel algorithms". In: *Communications of the ACM* 39.3 (1996), pp. 85–97.

[9] BLELLOCH, GUY E. *Vector models for data-parallel computing*. Vol. 75. MIT press Cambridge, MA, 1990.

[10] BRATLEY, PAUL and BENNETT L. FOX. "Algorithm 659: Implementing Sobol's quasirandom sequence generator". In: *ACM Transactions on Mathematical Software (TOMS)* 14.1 (1988), pp. 88–100.

[11] CATANZARO, BRYAN, M. GARLAND, and K. KEUTZER. "Copperhead: Compiling an embedded data parallel language". In: *SIGPLAN Notices* (2011). URL: `http://www.catanzaro.name/papers/PPoPP-2011.pdf`.

[12] CHAKRAVARTY, MANUEL M. T. et al. "Accelerating Haskell array codes with multicore GPUs". In: *Proceedings of the sixth workshop on Declarative aspects of multicore programming*. ACM. 2011, pp. 3–14.

[13] CHAUDHARY, SUNEAL K. "American options and the LSM algorithm: Quasi-random sequences and Brownian bridges". In: *Journal of Computational Finance* 8.4 (2005).

[14] COUFFIGNALS, ERIC. "Quasi-Monte Carlo simulations for Longstaff Schwartz pricing of american options". MA thesis. Mathematical Institute, 2010.

[15] COX, J.C., S.A. ROSS, and M. RUBINSTEIN. "Option pricing: A simplified approach". In: *Journal of financial Economics* 7.3 (1979), pp. 229–263.

[16] DYBDAL, MARTIN. "An OpenCL back-end for Accelerate". In: (2011). URL: `http://dybber.dk/static/acc-opencl2011.pdf`.

[17] FILINSKI, ANDRZEJ. *Controlling Effects.* Tech. rep. DTIC Document, 1996.

[18] GANESAN, NARAYAN, ROGER D. CHAMBERLAIN, and JEREMY BUHLER. "Acceleration of Binomial Options Pricing via Parallelizing along time-axis on a GPU". In: *Performance Computing* (2009).

[19] GLASSERMAN, PAUL. *Monte Carlo methods in financial engineering*. Vol. 53. Springer, 2003. ISBN: 978-0387004518.

[20] GREWE, DOMINIK and MICHAEL O'BOYLE. "A static task partitioning approach for heterogeneous systems using opencl". In: *Compiler Construction*. Springer. 2011, pp. 286–305.

[21] GROSSMAN, MAX et al. "CnC-CUDA: declarative programming for GPUs". In: *Languages and Compilers for Parallel Computing* (2011), pp. 230–245.

[22] HEATH, MICHAEL T. *Scientific Computing: Computing An Introductory Survey*. Second edition (international). McGraw-Hill, 2002. ISBN: 978-0071244893.

[23] HWU, WEN-MEI W. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 2011.

[24] HWU, WEN-MEI W. *GPU Computing Gems Jade Edition*. Morgan Kaufmann Publishers Inc., 2011.

[25] IBANEZ, A. and F. ZAPATERO. "Monte Carlo valuation of American options through computation of the optimal exercise frontier". In: *Journal of Financial and Quantitative Analysis* 39.2 (2004).

[26] JOSH BUCKNER, <BUCKNERJ@UMICH.EDU> Mark Seligman, JUSTIN WILSON. *The homepage of package gputools in CRAN package archive*. 2013. URL: http://cran.r-project.org/web/packages/gputools/index.html.

[27] KELLER, GABRIELE et al. "Regular, shape-polymorphic, parallel arrays in Haskell". In: *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*. ICFP '10. Baltimore, Maryland, USA: ACM, 2010, pp. 261–272. ISBN: 978-1-60558-794-3. DOI: http://dx.doi.org/10.1145/1863543.1863582.

[28] LIPPMEIER, B. et al. "Guiding parallel array fusion with indexed types". In: *Proceedings of the 2012 symposium on Haskell symposium*. ACM. 2012, pp. 25–36.

[29] LIPPMEIER, BEN and GABRIELE KELLER. "Efficient parallel stencil convolution in Haskell". In: *ACM SIGPLAN Notices*. Vol. 46. 12. ACM. 2011, pp. 59–70.

[30] LONGSTAFF, FRANCIS A. and EDUARDO S. SCHWARTZ. "Valuing American options by simulation: A simple least-squares approach". In: *Review of Financial studies* 14.1 (2001), pp. 113–147.

[31] LUK, CHI-KEUNG, SUNPYO HONG, and HYESOON KIM. "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping". In: *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*. IEEE. 2009, pp. 45–55.

[32] MAINLAND, GEOFFREY and GREG MORRISETT. "Nikola: embedding compiled GPU functions in Haskell". In: *Proceedings of the third ACM Haskell symposium on Haskell*. ACM. 2010, pp. 67–78.

[33] MAINLAND, GEOFFREY et al. "Haskell Beats C Using Generalized Stream Fusion". In submission. 2012.

[34] MORO, BORIS. "The full monte". In: *Risk* 8.2 (1995), pp. 57–58.

[35] MUNSHI, AAFTAB. "The OpenCL 1.2 Specification". In: *Khronos OpenCL Working Group* (2011).

[36] NEWBURN, CHRIS J et al. "Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language". In: *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*. IEEE. 2011, pp. 224–235.

[37] NVIDIA. *NVIDIA CUDA C Programming Guide, Version 4.2*. 2012.

[38] NVIDIA. *Whitepaper: NVIDIA GeForce GTX 680*. 2012. URL: http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf.

[39] NVIDIA. *Whitepaper: NVIDIA's Next Generation CUDA^{TM} Compute Architecture: Fermi^{TM}*. 2009. URL: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[40] NVIDIA. *Whitepaper: NVIDIA's Next Generation CUDA^{TM} Compute Architecture: Kepler GK110^{TM}*. 2012. URL: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[41] NVIDIA CORPORATION. *NVIDIA CUDA C Programming Guide*. 2012.

[42] NVIDIA CORPORATION. *The CUBLAS website*. 2013. URL: https://developer.nvidia.com/cublas.

[43] NVIDIA CORPORATION and VICTOR PODLOZHNYUK. *NVIDIA CUDA SDK – Binomial Option Pricing Model*. 2007.

[44] OWENS, JOHN D. et al. "A Survey of General-Purpose Computation on Graphics Hardware". In: *Computer graphics forum*. Vol. 26. 1. Wiley Online Library. 2007, pp. 80–113.

[45] SOBOL, I. M. "On the distribution of points in a cube and the approximate evaluation of integrals". In: *USSR Computational Mathematics and Mathematical Physics* 7 (4 1967). DOI: http://dx.doi.org/10.1016/0041-5553(67)90144-9.

[46] *Sobol sequence generator: primitive polynomials and sets of initial direction vectors.* Last checked: March 2013. URL: http://web.maths.unsw.edu.au/~fkuo/sobol/.

[47] SUTTER, HERB. "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software". In: *Dr. Dobb's Journal* 30.3 (2005).

[48] SVENSSON, JOEL, MARY SHEERAN, and KOEN CLAESSEN. "Obsidian: A Domain Specific Embedded Language for Parallel Programming of Graphics Processors". In: *Implementation and Application of Functional Languages*. Ed. by SCHOLZ, SVEN-BODO and OLAF CHITIL. Vol. 5836. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2011, pp. 156–173. ISBN: 978-3-642-24451-3.

[49] *The Bohrium homepage*. Last checked: March 2013. URL: http://bh107.org.

[50] *The R+GPU homepage*. Last checked: March 2013. URL: http://brainarray.mbni.med.umich.edu/brainarray/rgpgpu/.

[51] THIEMANN, PETER and MANUEL M.T. CHAKRAVARTY. "Agda Meets Accelerate". In: ().

[52] UNKNOWN AUTHOR. "Valuation of American Basket Options using Quasi-Monte Carlo Methods". It seems that the author wishes anonymity, the thesis was found at http://www.maths.ox.ac.uk/prospective-students/graduate/courses/finance/part-time/dissertations/examples-dissertations. MA thesis. Christ Church College, University of Oxford, 2009. URL: http://www.maths.ox.ac.uk/system/files/private/active/0/MScThesis.pdf.

[53] WADLER, PHILIP. "Monads and composable continuations". In: *Lisp and Symbolic Computation* 7.1 (1994), pp. 39–55.

Sobol sequences in Accelerate

```
sobol_divisor = Prelude.fromIntegral (2^30)

grayCode :: Exp Int -> Exp Word32
grayCode n = fromIntegral (n `xor` (n `shiftR` 1))

-- Generate a length n Sobol sequence using the given
-- direction vector by parallelising the inductive algorithm
sobolN :: Array DIM2 Word32 -> Int -> Acc (Array DIM2 Double)
sobolN dirVs n =
  let
    Z :. i :. j = arrayShape dirVs
    cubeSize = constant $ Z :. n :. i :. j
    sobolIndices = generate cubeSize (fst3 . unindex3)
    directionNumberIndices = generate cubeSize (thd3 . unindex3)

    ps = map fromBool $ zipWith (testBit . grayCode)
                                sobolIndices
                                directionNumberIndices

    directionNumbersRep = replicate (constant $ Z :. n :. All :. All)
                                    (use dirVs)

    xs :: Acc (Array DIM2 Word32)
    xs = fold1 xor $ zipWith (*) directionNumbersRep ps
  in map ((/sobol_divisor) . fromIntegral) xs


-- Various helper functions
fromBool :: (Elt a, IsNum a) => Exp Bool -> Exp a
fromBool b = b ? (1, 0)

unindex3 :: (Elt i, IsIntegral i) => Exp DIM3 -> Exp (i, i, i)
unindex3 ix =
  let
   Z :. i :. j :. k = unlift ix :: Z :. Exp Int :. Exp Int :. Exp Int
  in lift (fromIntegral i, fromIntegral j, fromIntegral k)

fst3 :: forall a b c. (Elt a, Elt b, Elt c) => Exp (a, b, c) -> Exp a
fst3 e = let (x1, x2 :: Exp b, x3 :: Exp c) = unlift e in x1

thd3 :: forall a b c. (Elt a, Elt b, Elt c) => Exp (a, b, c) -> Exp c
thd3 e = let (x1 :: Exp a, x2 :: Exp b, x3) = unlift e in x3
```