GPU Optimization of Base Form 820 Collective Artificial Spouse Retirement Plan

Niklas Schalck Johansson, Nikolaj Aaes and Hildur Flemberg
{nsjo, niaa, hufl}@itu.dk

IT University of Copenhagen

Table of Contents

Abstract. This paper explores how utilizing CUDA C and general purpose graphic processing units can make the calculation of reserves for the Base Form 820 Collective Artificial Spouse Retirement Plan, faster.

In its current state this calculation take more than 20 minutes per customer if a decent approximation of the reserve is to be achieved. The parallelized solution presented in this paper makes it up to INDSÆT TAL times faster than the original solution, meaning that the previous 20 minute execution now takes INDSÆT TAL seconds.

The solution enables insurance companies to estimate the reserves for Base Form 820 using relatively cheap hardware, much faster than previously.

Keywords: CUDA, parallelism, threads, blocks, insurance, capital, software, programming languages, Runge Kutta 4th order, death benefit.

1 Introduction

This report is written at the IT University of Copenhagen (ITU) in the spring term of 2013 in connection with a CUDA project supervised by Peter Sestoft from ITU and Hans Henrik Brandenborg Sørensen from the Technical University of Denmark (DTU). The report is addressed to people interested in exploiting the benefits of using general purpose graphical processing units (GPGPUs)[1] to optimize feasible computations in general, but specifically to people interested in CUDA.

In this project, we take a problem from the insurance industry that requires a significant amount of time to solve and show how to speed it up using the GPU. The problem concerns life insurance policies for married people, where one of them receives an amount of money some time after the other person has died. The challenge faced by the insurance company is to estimate its current and future holdings as accurately as possible such that it is always able to satisfy the obligations to the policyholders. Such an estimate is largely dependent on an estimate of when each policyholder is going to die which can be anytime between signing of the policy and some, possibly large, number of years ahead in time. Furthermore, the time of death is dependent on several variables like age, gender etc which has to be taken into account when making the estimate.

The insurance company has a mathematical model that unifies all these variables and calculates the reserves needed to insure a single policyholder, should

[1]Graphical Processing Units (GPUs) are most commonly used to render graphics in computer games. GPGPUs are "general purpose" variants that share the same architecture as GPUs but are optimized for computing purposes out of line with the graphics category. Since this project is not about graphics, we will refer to GPGPUs simply as GPUs, for shortness, throughout the rest of the report.

he die at any time between the time where the computation is done and some number of years ahead. The model describes the problem to an acceptable level of detail and can be solved by a computer but it takes a lot of time to do, even with modern CPUs. Moreover, insurance companies typically issue several thousands of policies each of which needs to be plugged in to the model and solved separately. Thus, the time it takes to solve the model for a single policy has to be as small as possible in order not to blow up the complexity of the overall computation, which, at the end of the day, is the one the insurance company is interested in.

All these facts together make computing power a valuable entity. The goal of this project is to analyze the opportunity to speed up the computation for a single policyholder through parallelism with CUDA, i.e. to optimize the time it takes to compute the size of the reserves needed at any time during the life of a policyholder such that the insurance company can fulfill its obligations when the person dies.

The reader of the report is assumed to know the basic principles of concurrency and parallelism. No CUDA specific knowledge is required. The rest of the report is outlined as follows. Section 2 provides some background information on CUDA,

1. Introduction 2. Math 3. Implementation 4. CUDA 5. Realisation 6. Testing 7. Benchmarks and Comparison 8. Discussion 9. Future Work 10. Conclusion

2 Background

TODO

2.1 Problem definition

How can CUDA C be utilized to optimize the calculation of the baseform 820 collective artificial spouse retirement plan using the Runge Kutta 4th order method?

2.2 Scope

The scope of this project is to handle a single customer at a time. We do not explore the possibilities for optimizing the process of using the algorithm on several customers at a time. Neither do we explore alternative methods for approximating differential equations.

2.3 Assumptions

This project was developed under several assumptions to limit the size of it. We assumed that the spouse of an insurance holder was always of the opposite gender. It would now be possible to take same-gender couples into account but it would require som modification to the code. We assumed that the function to determine the interest rate always returns 5 %.

3 The Math

The algorithm in this project is used to determine the lump sum of money the insurance company needs to possess to be able to pay the insurance holder's spouse in the case of his or her death. The payment to the spouse will be in the form of a life interest which are disbursed from the time of death of the insurance holder unless the death occurs before the pension age. In that case the money will be disbursed after a grace period determined by the insurance company at the time the insurance was taken out.

We assume that the spouse of the insurance holder is of the opposite gender. If the insurance holder does not have a spouse at the time of his or her death the insurance is forfeited.

The algorithm used here is a 4th order Runge-Kutta solution with a fixed step-size (indsæt reference) where we use a series of constants determined before any calculation begins:

Constants

Name	**Meaning**
τ	The time of death of the insurance holder
r	The pension age
g	The grace period
x	The age of the insurance holder at calculation time ($t = 0$)
t	The time of calculation
h	The Stepsize of the Runge-Kutta solution

τ is expressed as $x + t$. Apart from this there is also a constant k which are determined by τ, r and g in the following manner:

How k is defined

If this statement holds	**then k equals**
$\tau < r$	g
$r \leq \tau < r + g$	$r + g - \tau$
$r + g \leq \tau$	0

The algorithm can be described as a combination of three models; an outer model that describes the life/death state of the insurance holder, a middle model that describes the married/unmarried state of the insurance holder and an inner model that describes the life/death state of the potential spouse.

NOTE: The description in the three model sections is loosely based on a describtion for Edlund[1] and some passages are directly translated from this document.

3.1 Outer model

The outer model is expressed as the following equation:

$\frac{d}{dt}f(t) = r(t)f(t) - \mu_t(x+t)(S^d_{x+t}(t) - f(t))$

Where $S^d_\tau$is the death benefit that for a τ year old at the time t is needed to cover the payment from the insurance company, $\mu_t(x+t)$ is the mortality rate for a $(x+t)$ year old and $r(t)]$ is the interest rate function. In this project the interest rate function returns the constant 0.05.

The differential equation is solved from $t = 120-x$ to $t = 0$ with the boundary condition $f(120-x) = 0$

3.2 Middle model

The middle model is used to calculate $S^d_{x+t}(t)$ and is expressed with the following equation:

$$S^d_\tau(t) = \begin{cases} g_\tau \int f(\eta|\tau) a^I_{[\eta]+g}(t) d\eta & \tau \leq r \\ g_\tau \int f(\eta|\tau) a^I_{[\eta]+r+g-\tau}(t) d\eta & r \leq \tau \leq r+g \\ g_\tau \int f(\eta|\tau) a^I_{[\eta]}(t) d\eta & r+g \leq \tau \end{cases}$$

Where g_τ is the propability that a τ year is married and $f(\eta|\tau)$ is the probability distribution for, that a τ year old is married to a η year provided that the τ year old is married. This equation can be rewritten to this form:

$\frac{d}{dn}f(\eta) = -g_\tau f(\eta|\tau) a^I_{[\eta]+k}(t)$

here the subscript for a is rewritten to $[\eta]+k$ where k can fall into three different categories as shown in THE-TABLE-CONSTANTS.

This differential equation is solved from $\eta = 120$ to $\eta = 1$ with the boundary condition $f(120) = 0$

3.3 Inner model

The inner model is used to calculate $a^I_{[\eta]+k}(t)$ and is expressed with the following equation:

$\frac{d}{ds}f(s) = r(t+s)f(s) - 1_{s \geq k} - \mu_{t+s}(\eta+s)(0 - f(s))$

Where t, η and k are constants and $\mu_{t+s}(\eta+s)$ is the mortality rate for a $(\eta+s)$ year old.

This differential equation is solved from $s = 120-\eta$ to $s = 0$ with the boundary condition $f(120-\eta) = 0$

3.4 Runge-Kutta 4th order

The Runge-Kutta method is a method for approximating differential equations that builds on Eulers method (INDSÆT REFERENCE) and the midpoint method

(INDSÆT REFERENCE). When given a start point and a differential equation one can choose a stepsize and approximate the graph. When given a point (x_n, y_n), a differential equation f and a stepsize h one can use the Runge-Kutta method to approximate the next point in the following way:

$k1 = hf(x_n, y_n)$
$k2 = hf(x_n + \frac{h}{2}, y_n + \frac{k1}{h})$
$k3 = hf(x_n + \frac{h}{2}, y_n + \frac{k2}{h})$
$k4 = hf(x_n + h, y_n + k3)$

$x_{n+1} = x_n + h$
$y_{n+1} = y_n + \frac{k1}{6} + \frac{k2}{3} + \frac{k3}{3} + \frac{k4}{6} + O(h^5)$ HVORFOR KAN VI IGNORERE $O(h^5)$??

3.5 Description of execution

To provide a better overview of how the math is applied we go through how an execution is performed. Before the execution begins certain value are give before hand, namely g, r and x. Additionally we need to decide on a stepsize before the execution can begin.

The first step in the execution is to use the outer model with a starting point. Since the outer model is to be solved for $t = 120 - x$ to $t = 0$ with the boundary condition $f(120 - x) = 0$ we can use the point $(120 - x, 0)$ as our starting point. The next step is to actually solve the outer model equation to calculate the next approximate point. All the components in the outer models differential equation can be computed using normal equations except $S^d_{x+t}(t)$. This component can be calculated using the middle model and when it computed the calculation of the next approximate point is completed. The outer model equation is solved four times for each approximation of the next point corresponding to calculating $k1$, $k2$, $k3$ and $k4$ in the Runge Kutta method. This can be repeated $(120 - x) \times \textit{stepsize}$ times until the approximate point becomes $(0, y)$ which is the final point.

To calculate the missing component using the middle model we need to establish the value of k for the point. Additionally we are given the value t from the outer model. Like the outer model we need to use the middle model with at starting point. The middle model is to be solved for $\eta = 120$ to $\eta = 1$ with the boundary condition $f(120) = 0$ which means that we can use $(120, 0)$ as our starting point. As with the outer model all the components of the differential equation in the middle model can be computed using normal equations except $a^I_{[\eta]+k}(t)$. This can be calculated using the inner model which is the last model in the chain. When the inner model has computed a result the calculation of the next approximate point is completed. As in the previous model the middle model equation is calculated four times corresponding to calculating $k1$, $k2$, $k3$ and $k4$ in the Runge Kutta method. This can be repeated $119 \times \textit{stepsize}$ times until the approximate point becomes $(0, y)$ which is the final point and the y-value of this point is returned to the outer model.

As the last part of the execution the inner model needs to calculate the last component used by the middle model. The inner model uses the t, k and η values provided by the middle model. This model uses a starting point in the same way as the other models. This model is to be solved for $s = 120 - \eta$ to $s = 0$ with the boundary condition $f(120 - \eta) = 0$ which means we can use the point $(120 - \eta, 0)$ as a starting point. Contrary to the other two models the inner model equation only contain components that can be computed using normal equations to find the next approximate point. For each approximation of a point the inner model equation is solved four times corresponding to calculating $k1$, $k2$, $k3$ and $k4$ in the Runge Kutta method. This can be repeated $120 - \eta \times stepsize$ times until the approximate point becomes $(0, y)$ which is the final point and the y-value of this point is returned to the middle model.

In any execution the outer model equation will be solved
$(120 - x) \times stepsize \times 4$ times,
the middle model equation will be solved
$((120 - x) \times stepsize) \times (119 \times stepsize \times 4)$ times,
and the inner model equation will be solved
$((120 - x) \times stepsize \times 4) \times (\sum_{i=120\times stepsize}^{stepsize+1} ((120 - \frac{\eta}{stepsize}) \times stepsize \times 4) + ((120 - \frac{2\eta-1}{2\times stepsize}) \times stepsize \times 4) + ((120 - \frac{2\eta-1}{2\times stepsize}) \times stepsize \times 4) + (120 - \frac{\eta-1}{stepsize}) \times stepsize \times 4))$ times.

For a normal execution with $stepsize = 4$, $g = 30$, $r = 80$ and $x = 115$ the outer model equation will be solved 80 times, the middle model equation will be solved 152.320 times and the inner model equation is solved 145.008.640 times.

4 Implementation

After we established the mathematical base we implemented a C solution. Instead of implementing the solution directly into CUDA C we chose to implement a C solution first to eliminate any bugs that were purely C specific. This also made it easier to debug since CUDA C program are generally harder to debug (INDSÆT REF?). The implementation is based on the provided C# code and to conform to that, the middle model's stepsize is always 2 and the insurance holder is always a woman and the spouse is always a man.

The first step was to implement the utility methods used for calculating trivial equations and were mostly taken directly from the original C# implementation. The methods in question are `gTau(double tau)`, `f(double eta, double tau)`,`k(double tau, double r, double g)`,`r_(double t)`, `GmFemale(double t)` and `GmMale(double t)`. These are all used to calculate components in each of the different differential equations.

The second step was to create the programs entry point, the main method. This would be responsible for initializing the g, r and x variable as well as the stepsizes for each model.

The next step was to split implement the base for the three models. Each model have a main method (`Outer(..)`, `Middle(..)` and `Inner(..)`), a method used for the Runge Kutta method (`OuterRK(..)`, `MiddleRK(..)` and `InnerRK(..)`) and a method that only contains the differential equation for the models (`OuterDiff(..)`, `MiddleDiff(..)` and `InnerDiff(..)`). While creating a Runge Kutta method for each model seems excessive because it serves a the same function in each model it was necessary later on because CUDA C does not support function pointers on compute capabilities lower than 2.0. If the implementation was only meant for the C programming language a single Runge Kutta method that took a function pointer would have been sufficient.

Each of the models main methods are responsible for executing the corresponding Runge Kutta method the correct number of times. This is done by calculating exactly how many full steps it needs to perform by using the `floor(double x)` method. When using this method we get an integer representation of how many steps we need to take in that particular model but since we are flooring the amount of steps there might be a step remainder smaller than 1, that we need to consider. To take this into account we use what we refer to as the first step. Before each model starts to perform the full number of steps it calculates the remainder of the flooring and takes a step to counter it. This is not necessary for the middle model since the middle model's stepsize is hardcoded to 2 which means that there will never be a remainder. It also means that the amount of full steps the middle model takes is constant and is therefore calculated in the programs main method.

As an additional optimization we found that the equations for calculating $k2$ and $k3$ in `MiddleRK(..)` are identical and chose to just calculate this value once and the use it both places.

5 CUDA

5.1 Architecture

Compute Unified Device Architecture (CUDA) is a parallel computing model created by Nvidia in 2006 based on a superset of the C programming language. In CUDA terminology, the computing system as a whole consists of a host (a traditional CPU) and one or more devices (GPUs) that are specially architectured to perform a massive amount of computations in parallel. Devices can execute kernel functions which is the CUDA term for a function that embeds code to be executed on the device.

Before a kernel execution is started, the host must make sure that all the data needed for the computation is available on the GPU. Conversely, the host is responsible for copying back the result data when the computation is completed. CUDA C exposes functions similar to the well known C functions *malloc* and *free* that allocate and free memory on the device similarly to how it is normally done on the hosting system. Additionally a function exists that performs the actual copying given source and target pointers. When all data is available on the device, the host can trigger the kernel function and start the computation.

Something When a device is assigned a kernel, the computing ressources of the device is partitioned into a number of streaming multiprocessors (SMP) that can each maintain a certain number of threads. The threads of an SMP are commonly known as a thread block. The total collection of blocks is known as a grid.

CUDA C extends the language with a whole new API with keywords and functions allowing the programmer to specify where a block of code should be executed, where memory should be allocated and so on. Furthermore, one is able to specify how the hardware ressources on the device should be configured to perform optimally while solving a particular problem, i.e. how many threads to use etc.

As the CUDA compiler uses the host compiler for C to compile traditional C code, any valid C program is accepted by the CUDA compiler, but the code will be executed entirely on the host. To execute kernel functions

As CUDA C is a superset of traditional C, any valid C program is accepted by the CUDA compiler, but the code is executed entirely on the host. To execute code on the GPU, one defines so called kernel functions which are ordinary C functions tagged with the "global" keyword.

6 Realization

The following two sections describes the two CUDA implementations made during this project. The first implementation, MiddlePar, was the first iteration on how we could parallelise the mathematical model described earlier. The second implementation, OuterPar, is an improved version of the first that further parallelise. Both CUDA implementations derive directly from the C implementation described earlier.

6.1 MiddlePar

In order to parallelise the C implementation and make use of the parallelism of the GPU, we had to analyse the each model of the implementation. Our approach was to locate the most obvious part that could be parallelised, and developed a CUDA off that. Since the solution could not be parallelised entirely, we acknowledged that some calculations would be done on the GPU, the parts that could be parallelised, and the rest would be run on the CPU. The most obvious possibility for a CUDA solution is found within the middle model. Compared to both the outer and the inner model, the steps within the middle model were not very dependent on the previous steps. We say very because they are not completely independent. Middle is calculated by adding the result of k1, k2, k3, k4 and then add it with the y value of the previous step. This means that for each step we can calculate the sum of all the k values before we add it all together. This is a good start since the calculation of either k, is essentially a call to the inner model. This solution would keep the amount of inner model calls per thread to be only 4. The CUDA implementation of this is explained in section :MiddlePar:.

Implementation
The implementation of this CUDA solution is apparent in the function OuterDiff, which is the function that needs a result from the middle model. OuterDiff prepares two variables that is to be copied to the GPU. The first is the current x value from outer, that is needed to calculate middle, and an array called kSum. kSum is an array that will be used to store the sum of k1, k2, k3 and k4, calculated by each thread. Having these results, calculating each step in middle can be done much faster, since it is just a matter of adding two double values together for each step.

Based on the step size it is possible to calculate exactly how many steps middle needs to take. Prior to starting the kernel, we need to make sure that there are enough threads available when parallelising middle, such that there are at least one thread for each step. The amount of threads per block would be set prior to program execution, and before starting the kernel the amount of blocks needed to ensure enough threads would be automatically calculated.

In the following we will explain the kernel implementation; see figure (CODE-FIGURE INSERT REFERENCE) for the changes made to middle.

Once the kernel has been started, each thread define their global thread id by calculating threadIdx.x + blockIdx.x*blockDim.x. This ensures that each thread within the kernel has a unique id, which we will need later on.

We then ensure that only threads, that has an id within the range of middle steps, continue on from line XX. We know that this will cause branch diversion in the last block if the total amount of threads is not equal to the amount of steps, however this is a necessary precaution as we will see soon.

The next couple of lines is used to calculate local variables, that is used to call both Inner(. . .) and MiddleDiff(. . .). k1, k2 and k4 is then calculated by calling MiddleDiff(. . .) and Inner(. . .). In line YY the weighted average of each k is calculated and stored in the kSum array. This is where it is important that the thread id does not exceed the amount of steps. The size of kSum is exactly the total amount of steps, so if a thread id is larger than this, we would run into problems. A solution could be to always make sure that the number of threads you have is equal to the amount of steps to be taken, however since both thread per block and, as well as middle steps varies it is not feasible, and cannot always be ensured. Once the sum of the k values have been stored in kSum for all threads, the kernel terminates.

Memory considerations

This CUDA implementation access certain constant variables quite a lot when calculating inner(. . .). Since we were ever only going to read these constants, it would be a good idea to place this information in constant memory. Constant memory is considerably faster than global memory(INSERT REFERENCE), with the limitation.

It was also considered to keep kSum in shared memory. Shared memory, like constant memory, is likewise considerably faster than global memory, however it is possible for more than one block to be part of the kernel execution. Shared memory allow tread within the same block to share data, but not between blocks, so shared memory was discarded for this solution, and kSum was kept in global memory.

NVIDIA Analyser

TODO

Implementation shortages

The solution described above have a few drawbacks. First of all, each time a call to middle are made, a new GPU context would be set up, variables would be copied to the GPU, and a new kernel would be started. Doing this for a low x value, and a high step size, would result create a large amount of kernels. Second, this solution need more than one block to parallelise middle if the amount of threads per block were low. Even though it would speed up the call to middle, it does not speed it up the amount of total calls made to middle. Third, we were not really using a lot of threads or blocks to actually make the calculations, compared to how many threads and blocks are available on modern GPU's.

6.2 OuterPar

To solve these issues the GPU parallelisation of middle would have to be changed to use only one block, and we further had to parallelise the total calls to middle. Further investigation showed that the differential solution of the outer integral, used only the current x value from the outer steps to calculate middle. This effectively meant that we could calculate all x values in advance (we knew the start x value, the end x value as well as the size of each step), and further calculate all calls to middle in advance using these values. These could all be calculated at once on the GPU using only one kernel.

Implementation

Memory considerations

NVIDIA Analyser

7 Testing

To make sure that the C#, C and CUDA programs produce similar results up to an acceptable (and adjustable) treshold, we run each program on a large data set and compare the outputs. Each of the 3 programs take 4 command line arguments which is (1) g - the number of years from the policyholder dies until the first rate is disbursed, (2) r - age of retirement, (3) x age at runtime and (4) *stepsize* - the number of steps to take each year when the outer and the inner models are solved. Each program outputs a single number which is – roflcopter –.

We want all input from the test data set to fall within intervals that are chosen realistically and individually for each command line argument. It makes little sense to test the 3 programs with a retirement age of 200 since few people experience to become that old. Likewise, a big stepsize will make the overall test take very long time to complete which is impractical when testing during development. To avoid these problems, we made a program that generates test data in the shape of a plain text file. Each row in the file contains a comma separated list of numbers representing the 4 command line arguments to be fed to each of the 3 programs.

Next, we need a program that reads the lines of the test data file one by one, invokes the 3 programs in sequence and writes the corresponding results to a line in a new file. Obviously, this could be done by a single program as just described, but we wish to be able to test the C#, C and CUDA programs separately during development without having to wait for the other 2 to finish, which might take longer time than the single program we want to test. For that reason, we make a test program for each of the 3 programs that reads a line from the test data file, invokes the program and appends the result to a text file that is private to the program being tested.

Finally, we have 3 text files with the results from each test run. We can then import the contents of each file into a column in a spread sheet and have the sheet compare the numbers on each row to identify matching and differing results.

The test scenario describe dabove was conducted on 3 machine with the following hardware specifications

Machine	The Malamanteau server	MacBook pro, Retina Mid 2012
OS	Windows 7 Pro 64-bit	OS X 10.8.3 (12D78)
CPU	Intel Xeon W3505 2.53 GHz	Intel Core i7 2,3 GHz
RAM	4 GB	8 GB 1600 MHz DDR3
GPU	Nvidia Tesla C2075[3]	Nvidia GeForce GT 650M [4]
CC	2.0	3.0
CUDA cores	448	384
Frequency of CUDA cores	1.15 GHz	Up to 900 MHz
Total dedicated memory	6 GB GDDR5	2 GB

For testing purposes we assume that the insurance holder is always a woman and the spouse is always a man. The individual test results kan be found in INDSÆT APPENDIX ELLER CD REF.

8 Benchmarks and Comparison

TODO

TODO: Graph with runtime and stepsize. Constant g,r and x. Take samples and shown that it works even for corner cases.
TODO: Graph with runtime and unused threads.
TODO: Graph with runtime and variable g,r,x and constant stepsize.
TODO: Graph with runtime and threads per block. take samples.

9 Reflection / Discussion

When computing several customers one after another it could be beneficial to store the result of each calculation of the outer, middle and inner model and what parameters they were given. Whenever the program wanted to calculate value it could check if the calculation was already performed at a previous time and instantly get the result instead of calculating it again. If the result was not found in the storage it could calculate it itself and store the result afterwards for future use.

This approach is only beneficial when the lookup in the storage is faster than performing the calculation itself. It is also not beneficial if the amount of calculations that are identical, is not large enough. It would be interesting to examine the calculation collision with a large amount of executions with different customers. A drawback with this approach is that the storage can become very large over time and that could potentially lead to slower lookups. Since the approach stops being beneficial when the lookups are slower than the actual calculations it would be smart to examine when this occurs and evaluate what results is stored. If it occurs too fast one might consider to only store the middle or outer results because they are slower to calculate than the inner results.

If we examine the differential equation used in the middle model $S^d_\tau(t) = g_\tau \int f(\eta|\tau) a^I_{[\eta]+k}(t) d\eta$ it is a constant times an integral. This means that there is no need to use the Runge Kutta method here we could have chosen another approach for integral approximating such as Simpson's rule[5]. Simpson's rule requires a even number of steps which our middle model will always have as long as the stepsize is hardcoded to 2. Where the Runge Kutta method uses four calculations for each step ($k1$, $k2$, $k3$ and $k4$), Simpson's rule only uses a single calculation. This could potentially make the computing of the middle model four times faster.

10 Future work

This project has a limited scope and we would like to document suggestions for future expansion of the project:

Formal verification of correctness of code
We assume that the code produces the correct results because it is modeled after existing code and because our test of INDSÆT KORREKT TAL values are identical to the results produced by the existing code. Our test give a sense of correctness but without formal verification there is always the possibility that a corner case is not covered and will produce incorrect results.

Handling multiple customers
This project is built around the assumption that it would be run for a single customer at a time. If it is to have any real world use it must be able to run multiple customers at the same time. While this is possible with the current implementation is has not been tested or optimized. It would be interesting to examine how GPGPUs could be utilized when handling multiple customers and if the time used per customer could be even lower than it is when handling a single customer.

Variable interest curve
In this project the interest rate is assumed to be a constant of 5 %. However in the real world this is not the case. The interest rate is variable over time and the program should be able to take this into account.

Switch genders at runtime
When handling multiple customers in the same execution being able to switch the genders of the insurance holder and spouse at runtime is necessary. This is not a complex extension of the program but it is still something that is not implemented in this project. The difference between using female and male is the function used to calculate their mortality intensities. Changing

whether the female or male function should be called for each customer and spouse should be trivial in the C and C# implementation but might have repercussions in the CUDA C implementation since it can generate branch divergence.

Same-gender marriages
As an addition to switching gender at runtime the program will be able to emulate the real world better if it supported same-gender marriages. The same considerations are present as the function to calculate mortality intensities is now the same for both the insurance holder and the spouse. Additionally the function used to calculate the probability that the insurance holder is married might be different for same-gender marriages.

Upper bound on threads per block
Check whether you try to set the threads per block higher than what a single SMP can actually contain and run.

11 Conclusion

TODO

12 Glossary

Insurance holder
The person who have made an agreement with the insurance company to enable his/her spouse to receive an amount of money in case of the his/her death.

Spouse
The person married to the insurance holder.

Life interest
A monthly payment to the spouse of the insurance holder

Death benefit
All the life interest payments added together. This is a construction created for the sake of calculation.

Grace period
The time period between the death of the insurance holder and the start of the payments.

Boundary condition
TODO

References

[1] EDLUND DOKUMENTET

[2] David B. Kirk, Wen-mei W. Hwu - Programming Massively Parallel Proccesors, A Hands-on Approach - Elsevier Inc. - 2010

[3] `http://www.nvidia.com/docs/IO/43395/NV-DS-Tesla-C2075.pdf`
[4] `http://www.geforce.com/hardware/notebook-gpus/geforce-gt-650m/specifications`
[5] `http://www.mathwords.com/s/simpsons_rule.htm`