High-Performance Reserve Calculations for Life Insurance Portfolios

Christian Harrington, Nicolai Dahl, Peter Sestoft, and David Raymond Christiansen

IT University of Copenhagen, Denmark[*]

May 31, 2013

[*] Work supported by the Danish Advanced Technology Foundation (Højteknologifonden) (017-2010-3)

Abstract

We describe a software design for efficient reserve computations for life insurance and pension products, described using a flexible and precise notation that is both machine-readable and human-readable; in effect, a domain-specific language for actuaries. Mathematically, this notation is based on continuous-time Markov models.

First, from a collection of such product descriptions and from assumptions about transition intensities (e.g. survival, disability) and yield curves, the software can generate instances of Thiele's differential equation that characterize the reserve or other quantities of interest.

Secondly, these differential equations can be solved efficiently in a variety of ways. We use numerical solvers and can therefore calculate reserves for advanced pension products, for which the Thiele equations do not have closed-form solutions. Our computationally most advanced solver generates product-specific CUDA C code for graphics processors. Experience shows that this can speed up the solution by a factor of around 100 over standard desktop computer hardware, while retaining the same accuracy.

With this approach, actuaries can develop and describe advanced and non-standard pension and life insurance products, yet still efficiently compute reserves and other quantities of interest. This is possible for large portfolios of contracts and large numbers of stress scenarios. Core to this is the use of modern programming language technology, including domain-specific languages and code generation for a variety of hardware platforms.

1 Introduction

The goal of this work is to efficiently and accurately compute the liabilities of a life insurance and pensions company. More precisely, for a given portfolio of pension products we compute the *reserve*, that is, the expected net present value of the company's future payments.

We assume that pension products are described using AML, a domain-specific language for life insurance and pension products, explained in a companion paper [5]. From an AML product description, we can automatically generate an instance of Thiele's differential equations [10] that characterizes the reserve. We then compute the actual reserve by numerically solving these equations.

This approach is very general due to the fact that AML, which is based on continuous-time Markov models, can describe a wide variety of advanced pension products. In particular, AML allows models with an arbitrary number of states and with cycles in their transition graph.

1.1 Motivation

The motivation for this work is two-fold:

- It is desirable to describe and model life insurance and pension products more accurately, as in AML [5]. For instance, it is desirable to allow recovery (from the disability state to the active state) in disability insurance. However, this introduces cycles to the state model, which in general means that there is no closed-form formula for computing the reserve. Hence the need for the computationally intensive task of numerically solving Thiele's differential equation.

- The financial crisis of 2008 has led to a desire to improve the regulation of insurance companies in the European Union. The forthcoming *Solvency II* [15] directive will require more sophisticated modeling and calculations on life insurance and pension portfolios.

  One way to address this is to estimate reserves for a range of possible future scenarios (interest rate scenarios, mortality shocks, and so on). *Solvency II* requires that this is done at the level of whole portfolios rather than with isolated products, which greatly increases the requirement for high computational performance.

1.2 The Challenge

Hence the challenge that we face, and overcome, is the need to compute reserves numerically, which is quite computationally demanding, even for products computed in isolation. Moreover, this may need to be done for millions of pension contracts, and these computations must be repeated for thousands of interest rate or stress scenarios.

Fortunately, modern personal computer hardware, in the form of general-purpose graphics processing units (GPGPU), offers cheap high-performance parallel computing. Our contribution uses state-of-the-art programming language technology to harness this parallelism in a general manner, automatically generating parallel numerical solvers from a portfolio of products described in AML. This paper describes that this approach can achieve the desired performance.

1.3 The Contribution

We describe the design and performance results from a prototype calculation framework, called Actuarial Calculation Processor (ACP), which provides a highly flexible way to manage, optimize and execute reserve calculations for entire portfolios of insurance products.

Our approach makes the specification of life insurance products, risk models, and customers independent of the technology on which calculation will be performed. We give an example of the capabilities of the framework by showing support for performing reserve calculations on an Nvidia Graphics Processing Unit (GPU) given products specified in the Actulus Modeling Language (AML) [5].

1.4 Paper Outline

Section 2 gives an overview of the technological background as well as the core terminology of the article. Section 3 presents our computational framework and is followed by the presentation of a concrete high-performance differential equation solver, called the CUDA Life Insurance Reserve Estimator (CLIRE), in Section 4. An example of optimizing the computational performance of CLIRE is presented in Section 5. Finally, Section 6 presents the results from an instance of the framework and Sections 7 and 8 present related work and conclusion.

2 Background

As terminology can differ from region to region, and company to company, a few core terms that are used throughout this article are listed below:

Product A specific coverage offered by an insurance company, such as a disability annuity or a death benefit

Policy A customer will typically have several products, which together comprise a policy

Portfolio An insurance company's portfolio consists of all its customers with their respective policies, as well as definitions of the products that the company offers.

2.1 Actuarial Concepts

Our modeling of life insurance and pension products is based on continuous-time Markov models; these are more general than discrete-time state models. A state in the model (such as *active*, *disabled*, *dead*) describes the insured or co-insured. A transition from one state to another describes a stochastic occurrence, typically with a specified age-dependent intensity, such as a death intensity.

Within a state, the insured or co-insured may receive (or make) a continuous payment, for instance specified in dollars per year. On a state transition, the insured or co-insured may received a lump sum payment.

The reserve is the expected net present value of the future payments in a product. The state-wise reserve for a customer, $V_j(t)$, is the reserve at time t given that the insured is in state j at time t. The state-wise reserve may be computed by solving an instance of Thiele's differential equations arising from the continuous-time state model, its transition intensities, payments, and interest rates.

The companion paper [5] defines further actuarial concepts, but the present paper should be comprehensible also without them.

2.2 Technological Background

Modern Central Processing Units (CPUs), such as those found in most desktop and laptop computers, are very powerful. But they are not suited for doing many calculations in parallel. Most desktop CPUs have 2–4 cores, which are the units that execute program instructions. Each core can execute one "thread" of instructions at a time. This means that the CPU can at most be performing a handful of calculations at the same time[1]. This is well suited for many tasks that commonly take place on consumer computers.

2.2.1 Modern-day GPUs

Graphics Processing Units (GPUs) originated as a technology for displaying advanced graphics in 3D games and are today found in the majority of personal computers. GPUs used to be extremely specialized, and had no other function than producing graphics. In the last few years, however, it has become possible to accomplish other tasks than graphics processing on a graphics card. This is known as general-purpose computing on graphics processing units (GPGPU).

Due to the size of the problem at hand, it would be huge gain if it was possible to numerically solve differential equations in parallel. Fortunately, modern graphics cards are very well suited for this. Instead of just a few cores, they can have hundreds, or even thousands. There are, however, some challenges, as the individual cores on graphics cards are slower, and much simpler. There

[1] The actual number can be slightly higher, using technologies like hyper-threading and superscalar architectures, but not enough to make a difference in this case.

are also some restrictions on what they can compute in parallel and thus a performance optimized solution entails carefully tailoring a program design that makes the most of the GPU architecture while still fulfilling the requirements of the problem at hand.

2.2.2 CUDA and Nvidia GPUs

A leading supplier of GPGPU hardware, NVidia, has manufactured a solution for GPGPU programming, called the Compute Unified Device Architecture (CUDA). CUDA is both a hardware specification that NVidia's graphics cards implement, as well as software that allows developers to write code that executes on their graphics cards. The architecture of graphics cards implementing CUDA is very intricate, and consists of many layers of memory and processing units. Explaining all this is far beyond the scope of this article, so what follows is a very brief introduction to CUDA, and GPGPU programming in general.

A GPU consists of multiple streaming multiprocessors that in turn have several cores. For example, the NVidia Tesla C2075[2] graphics card, which is a high-end card built for high-performance computing, has one GPU with 14 multiprocessors, each with 32 cores. This results in a total of 448 cores, supporting 448 parallel instructions. The multiprocessors are what is known as Single Instruction, Multiple Data (SIMD) processors. This means they can perform a single kind of instruction on different data inputs, in parallel. For example, if you need to multiply the numbers 1–32 by 2 (1×2, 2×2, 3×2, ...), it is possible to perform all these operations in parallel. On a CPU, these would be computed sequentially, resulting in longer running time.

The SIMD architecture of CUDA also comes with a drawback: it can *only* perform one single type of instruction at a time. Given 32 numbers, multiplying each of them with 2 would be perfectly parallelizable on a GPU and could be executed significantly faster than on a CPU. However, if we wanted to add 3 to the first number, multiply the second number by 5, divide the third number by 11, and so on, these instructions would be performed sequentially on the GPU just as if they had been executed on a CPU. The problem is that since each core in the GPU is less powerful than a core in a CPU, the calculation for all 32 numbers would actually be slower.

Figure 1 shows an example where a result is needed for each of 32 customers. Each customer is different, and the result is dependent on this data, so one cannot simply calculate the result once, and use it for every customer. To calculate the result, the instructions A B D A must be executed, except for customer 1 and 2. They are special cases, and require different instruction sequences, A B D C and A A C D, respectively. Because some of the instructions for customer 1 and 2 are different, they cannot be calculated in parallel with the rest. Even though each instruction takes longer to run on the GPU, the parallelism results in a shorter run time. If many of the customers had required different instructions, the run time on the GPU would have been longer.

[2] Specifications available from `http://www.nvidia.com`

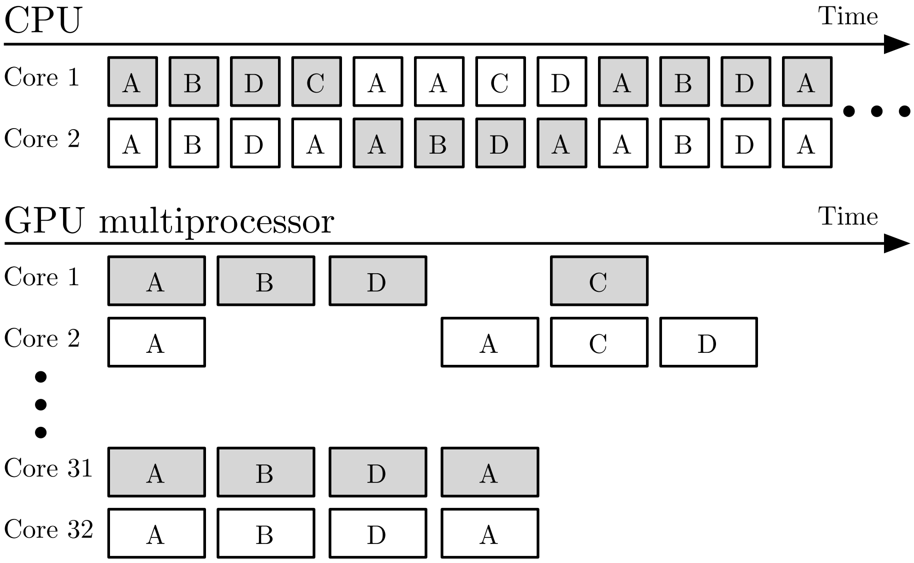


Figure 1: A result being calculated for 32 customers on a CPU and a GPU.

This means that the speed of calculations on GPUs is influenced not only by the instructions themselves, but also by the number of identical instructions. Since a portfolio consists of thousands of customers, each with many possible products, this poses a challenge, as different instructions are needed to calculate the reserves for the different products. This challenge will be addressed in Section 4.

3 Design of a Computational Framework

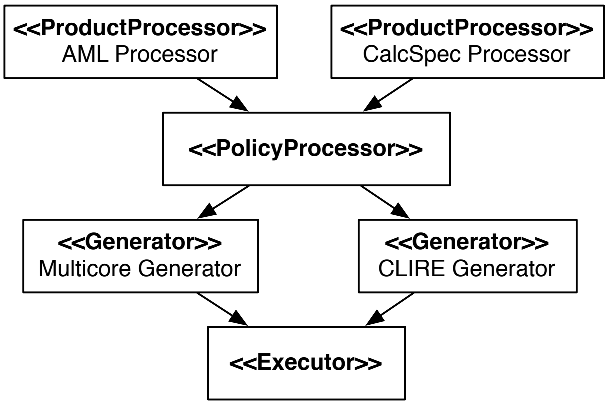
On one hand it is desirable to have a very fast and low-level implementation of these numerical solvers e.g. in CUDA-C, but on the other hand one must also be able raise the level of abstraction to where the users that actually design products are able to take advantage of the powerful implementation. The design of the Actuarial Calculation Processor (ACP) has the following key goals:

- To make product, risk model, and customer descriptions independent of execution technology
- To provide a central optimization and simplification mechanism for state-based ODEs.

The ACP, written in the programming language Scala [7], is as a framework for carrying out high-performance reserve calculations on entire portfolios of life insurance companies. It is a well-developed project that combines the advantages of low-level, hardware-specific, fast implementations with the benefits of manageability and modularity that come from abstraction.

3.1 Main Components

As depicted in Figure 2, the key components of the ACP are the ProductProcessor, PolicyProcessor, Generator, and Executor.



<<ProductProcessor>> AML Processor
<<ProductProcessor>> CalcSpec Processor
<<PolicyProcessor>>
<<Generator>> Multicore Generator
<<Generator>> CLIRE Generator
<<Executor>>

Figure 2: Abstract and concrete components of the Actuarial Calculation Processor.

A product processor transforms descriptions of products into an internal representation of Thiele's differential equations. The ACP can be extended to support a new language defining products by simply adding an additional product processor. Presently, the ACP contains two product processors. One of these supports an early prototype of AML's product definition language [5], embedded in Scala. Using this embedding, products can be specified using 10–20 lines of code. Furthermore, the ACP supports a low-level product specification language, called CalcSpec, used by Edlund A/S internally.

A policy processor bundles all inputs to a coherent Portfolio. It loads in customer specifications, risk models, and the product specifications provided by the product processor to produce a representation of the entire Portfolio that forms the basis for calculations. Note here that the risk models are also represented in the internal ODE format.

It is then up to the specific Generator implementation to handle this Portfolio and generate whatever calculation kernel it desires. This means we are theoretically able to generate any solvers; efficiently making the implementation technology completely independent of the product specifications and the syntax they are written in. Today, the ACP only handles those products that can be described using ordinary differential equations, but the framework can potentially be extended in the future to work with partial differential equations.

Many different technologies could be used to parallelize solvency calculations. For example, the Akka[3] framework for distributed and concurrent computation could be useful in a cluster environment. The ACP project, however, only currently contains the CLIRE Generator, which generates CUDA code for execution on GPGPUs. Lastly, an executor takes the generator's output and executes it on the system, preferably outputting the result of the generated solver.

3.2 ACP Evaluation

The decoupled and flexible architecture of the ACP framework makes it possible to specify the products, risk models, and customers completely independently of the implementation technology used by the concrete Generator. Any combination of ProductProcessor, PolicyProcessor, Generator, and Executor can be used.

The common, internal ODE format, used by all ProductProcessors, forms the basis for a central optimization mechanism. Known optimization strategies can be performed independent of the product description syntax. For example, for many common pension products, all obligations are terminated at the death of the insured. Thus, Thiele's equation for this state is trivial and can be eliminated, leading to a significant performance increase [4]. Any Generator will benefit from any optimization steps carried out on the central ODE representation, as the optimization always happens before the concrete Generator is invoked. Specific benchmark results will be discussed in Section 6.

4 CLIRE

CLIRE is the first solver written for ACP and it was written specifically to help address Technical Paragraph 64 of the *Solvency II* EU directive, which requires insurance companies "...to ensure that ruin occurs no more often than once in every 200 cases" [15]. This means that an insurance company must be able to cover the combined prospective reserve of all obligations in 99.5% of all future scenarios. One factor in these scenarios is the future interest rate, and it is specifically this point that CLIRE attempts to address.

CLIRE generates simulated interest rate paths on the GPU, based on the work of Cecilie Horn [11]. These interest rate paths are simulated in the Cox–Ingersoll–Ross model [6], based on data from the Danish Financial Supervisory Authority, *Finanstilsyn*. Thousands of these are generated, and each represents a different possible future eventuality. Once the interest rate paths have been generated, CLIRE calculates the state-wise reserve for each customer by numerically solving Thiele's differential equation [10, p. 71], using the fourth-order Runge-Kutta method (RK4) [16]. It does this for every yield curve, thereby producing many different possible reserves. This is a perfect opportunity for

[3] `http://www.akka.io`

parallelization on the GPU, as we can calculate all the reserves for each customer in parallel.

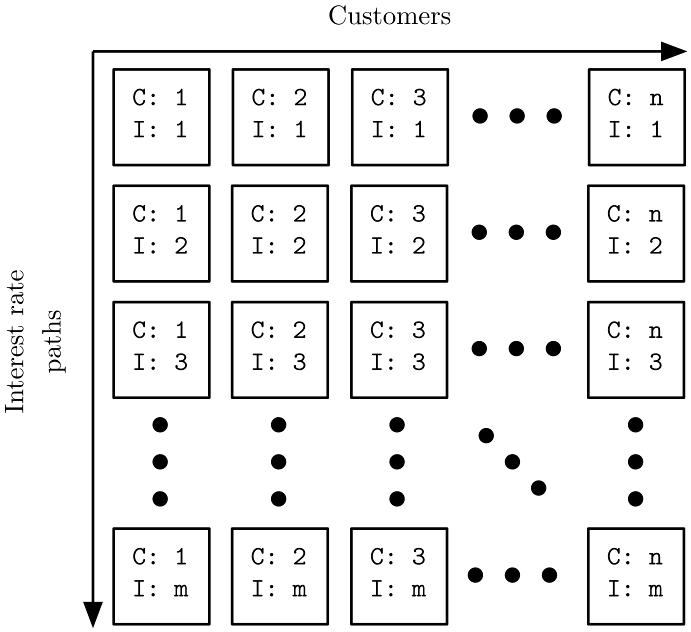


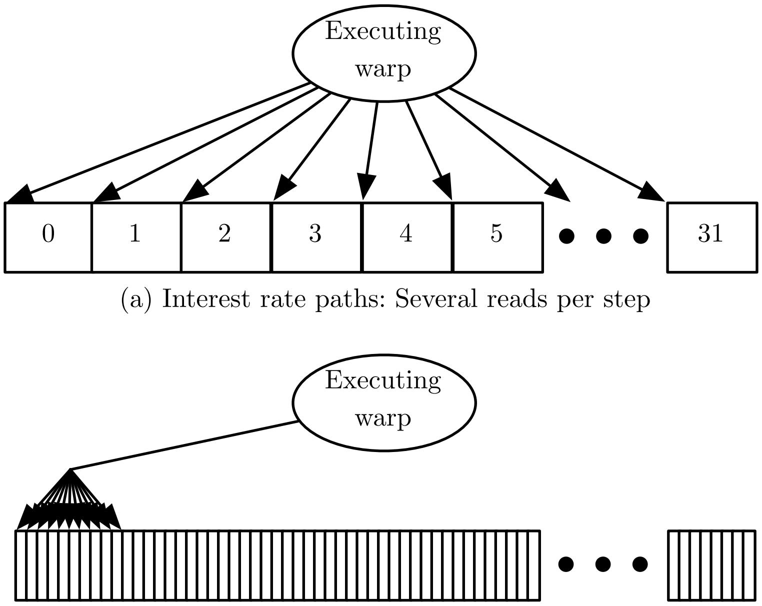
Figure 3: The problem is quadratic. With 1,000,000 customers, n, and 2,000 interest rate paths, m, 2,000,000,000 calculations are needed.

Developing code for CUDA is notoriously difficult even for professional programmers. By leveraging the AML language and the ACP, we can make high-performance code within the reach of actuaries.

5 Optimizing performance on GPGPUs

The challenge of performing calculations for millions of customers each with multiple products to be simulated with thousands of different interest rate paths, is a tremendous task even for a highly parallel architecture. As a consequence, great effort has gone into optimizing the various parts of CLIRE dealing with parallelization and memory access. In this section we present a technique for arranging the most heavily accessed memory, the generated interest rate paths, in a fast and uniform way that ultimately enables CLIRE to perform calculations on an insurance portfolio in a highly parallel fashion, using CUDA. This particular performance optimization has resulted in a 4 times faster execution than that of a non-optimized equivalent, thereby making it one of the most significant speed improvements of the CLIRE project.

Due to the architectural design of NVidia GPUs, calculations on products are bundled in "warps" of 32. These calculations are carried out in parallel on a multiprocessor and can greatly benefit from each other under the right conditions as explained in Section 2.2. An example of this is the way in which we store and access interest rate paths to perform memory operations faster.



(a) Interest rate paths: Several reads per step

(b) Interleaved interest rate paths: One read per step

Figure 4: Memory access without and with interleaved interest rate paths.

In a warp where each of the 32 reserve calculations are for the same customer with the same life insurance product but for different interest rate paths, we arrange the data in a special way. Figure 4 depicts two ways of accessing the very first number of an interest rate path for a warp of reserve calculations. The naïve method of 4(a) lays out the interest rate paths sequentially, which requires 32 different reads for each step in the calculation of a reserve. On the other hand, the interleaved method of 4(b) ensures one coalesced memory read for each step in the calculation by placing the first numbers from the 32 paths right next to each other. The second numbers of each of the 32 paths are placed next in the same fashion, and so forth. This technique ensures a speed increase by utilizing knowledge about the memory access patterns of the GPU.

Performance optimizations like these that are specifically designed for the problem at hand, are crucial in reducing the running time. Even a speed increase of only a few percentage points can save hours of computing when it comes to solvency calculations on entire portfolios of life insurance companies.

Table 1: CLIRE running on Tesla C2075

Customers	Time (seconds)	Standard deviation (seconds)
1,000	7.1	0.00035
10,000	66.4	0.00358
100,000	641.7	0.01589

Table 2: Modified CPU version of CLIRE running on Intel Core i5 2400k

Customers	Time (seconds)	Standard deviation (seconds)
1,000	897.7	0.05779
10,000	8,546.6	0.49972
100,000	N/A	N/A

6 Results

The following results are from two versions of CLIRE. The first was generated by ACP, and the second is a modified version of CLIRE that runs on a multicore CPU, that is, not on a graphics card. This modified version of CLIRE was produced to serve as a comparison to the high performance achievable through GPGPU computing.

For both versions of CLIRE, the test was run 10 times with different numbers of customers randomly generated customers. All tests where done with 2,000 generated interest rate paths, and with 100 steps per year for the RK4 algorithm. The NVidia Tesla C2075 was used for the unmodified version of CLIRE, while the modified version of CLIRE was run on an Intel Core i5 2400k at 3.5Ghz.

As can be seen from the results, CLIRE is more than 100 times faster when run on the graphics card. It should also be noted that execution time scales almost linearly with the number of customers. The running times of CLIRE on the the graphics card are extremely consistent, as can be seen from the standard deviations. This is due to the fact that the computations are executed on a system where no other processes compete for the resources of the Tesla graphics card. The last test, with 100,000 customers, was not run on the modified CPU version of CLIRE, due to the excessive running time.

7 Related work

General Purpose ODE solver Prior work concerning the implementation of ODE solvers on CUDA, includes Murray's study of several fixed-step and adaptive-step Runge-Kutta solvers [14]. The present work is more specialized. For instance, in general it may seem that the variation in stepsize between threads in an adaptive-step solver would lead to thread divergence and hence poor performance on graphics processors. However, there are initial indications [13] that when solving Thiele's differential equations, these thread divergence can be mitigated simply by grouping “similar” pension contracts and solving those in the same warp.

Other work includes Ahnert and Mulansky [1] who describe a C++ library, called Odeint, for solving the initial value problem of ODEs. For solving larger problems Odeint uses CUDA, as well as other parallelization technologies. Our CLIRE solver is highly specialized, implementing custom optimization strategies for maximum performance. Odeint does not to the same degree prioritize performance over generality.

Domain-specific languages As far as we know, the first application of code generation and domain-specific languages to the financial sector was Risla, described by Arnold, van Deursen and Res [3]. Like ACP, Risla used a high-level and readable description of a financial product to generate complicated, low-level code for administering the product.

Financial contracts Peyton Jones, Eber and Seward present a language for building complex financial contracts by combining simple contracts [12]. This language is an embedded domain-specific language in the programming language Haskell. This work has been expanded upon throughout the years by Andersen et al [2], Frankau et al [9] and Flænø Werk, Ahnfelt-Rønne and Larsen [8]. The latter is especially relevant, as they use an embedded domain-specific language to generate optimized GPGPU code.

8 Conclusion

Several directions of future work could be pursued. One direction concerns the generation of product-specific C and Akka code for standard multicore (desktop and server) hardware. Such hardware is likely to be somewhat slower than the graphics processor hardware, but more readily available. This direction could be extended to support cloud computing, using Amazon EC2 or Microsoft’s Azure, which basically consists of “farms” of standard multicore machines.

Another direction of future work is to generate more sophisticated numerical differential equation solvers, such as adaptive-stepsize Runge-Kutta-Fehlberg. It might seem preferable to execute these on multicore hardware, because variation in step size between threads might lead to thread divergence and hence poor performance on graphics processors. However, there are initial indications that the thread divergence problem need not be critical [13].

Furthermore, these directions may be combined in a number of ways, such as running generated CLIRE solvers on multiple GPUs in the cloud.

These possible development directions require new code generators and executors to be implemented (Figure 2), but they require no changes in the actuarial product descriptions as expressed in AML.

Thus our Actuarial Calculation Processor prototype and the experimental results show that the description of life insurance and pension products can be decoupled from the computational platforms used to numerically estimate reserves and other quantities of interest. Moreover, even though the description language AML is very general, such estimates can be computed efficiently using appropriate software and hardware technology.

References

[1] Karsten Ahnert and Mario Mulansky. Odeint — Solving ordinary differential equations in C++. *Computing Reasearch Repository*, abs/1110.3397, 2011.

[2] Jesper Andersen, Ebbe Elsborg, Fritz Henglein, Jakob Simonsen, and Christian Stefansen. Compositional specification of commercial contracts. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(6):485–516, November 2006.

[3] B.R.T Arnold, A. Van Deursen, and M. Res. An algebraic specification of a language for describing financial products. In *ICSE-17 Workshop on Formal Methods Application in Software Engineering*, pages 6–13. IEEE, 1995.

[4] Nicolai Dahl Blicher-Petersen and Christian Harrington. CUDA life insurance reserve estimator. Technical report, IT University of Copenhagen, May 2012.

[5] David Raymond Christiansen, Henning Niss, Klaus Grue, Kristján S. Sigtryggsson, and Peter Sestoft. An actuarial programming language for life insurance. 2014. Submitted to *ICA 2014, 2014*.

[6] John C. Cox, Jr. Ingersoll, Jonathan E., and Stephen A. Ross. An intertemporal general equilibrium model of asset prices. *Econometrica*, 53(2):pp. 363–384, 1985.

[7] Martin Odersky et al. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[8] Michael Flænø Werk, Joakim Ahnfelt-Rønne, and Ken Friis Larsen. An embedded DSL for stochastic processes. In *Proceedings of the 1st ACM SIGPLAN workshop on Functional high-performance computing*, FHPC '12, pages 93–102, Copenhagen, Denmark, 2012. ACM.

[9] Simon Frankau, Diomidis Spinellis, Nick Nassuphis, and Christoph Burgard. Commercial uses: Going functional on exotic trades. *Journal of Functional Programming*, 19(01):27–45, 2009.

[10] Hans U. Gerber. *Life Insurance Mathematics, Third Edition.* Springer, 1997.

[11] Cecilie Horn. Simulating interest paths, 2012. Internal Edlund A/S technical note.

[12] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: An adventure in financial engineering (functional pearl). In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 280–292, New York, NY, USA, 2000. ACM.

[13] Jens Dahl Møllerhøj. Runge-Kutta-Fehlberg i CUDA C. Technical report, IT University of Copenhagen, May 2013. (In Danish).

[14] Lawrence Murray. Gpu acceleration of runge-kutta integrators. *IEEE Transactions on Parallel and Distributed Systems*, 23(1):94–101, 2012.

[15] EU Parliament. Future rules, Solvency II. `http://ec.europa.eu/internal_market/insurance/solvency/future/index_en.htm`, may 2012.

[16] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C.* Cambridge University Press, 1988-1992.