Pension Reserve Computations on GPUs

Christian Harrington Nicolai Dahl Peter Sestoft David Raymond Christiansen
{cnha,ndbl,sestoft,drc}@itu.dk
IT University of Copenhagen

Abstract

New regulations from the European Union, called *Solvency II*, require that life insurance and pension providers perform more complicated calculations to demonstrate their solvency. At the same time, exploiting alternative computational paradigms such as GPGPU requires a high degree of expertise about the hardware and ties the computational infrastructure to one particular platform. In an industry where contracts literally last a lifetime, this is far from optimal. We demonstrate the feasibility of an alternative approach in which life insurance and pension products are represented in a high-level, declarative, domain-specific language from which platform-specific high-performance code can be generated. Specifically, we generate CUDA C code after applying both domain- and platform-specific optimizations. This code significantly outperforms equivalent code running on conventional CPUs.

Categories and Subject Descriptors D.3.4 [*Processors*]: Code generation; G.4 [*Mathematical software*]: Parallel and vector implementations

Keywords GPGPU; domain-specific languages; actuarial computation; code generation

1. Introduction

We all hope to live long lives, but tragedy can strike anyone. Under a life insurance scheme, our untimely demise will result in a much larger payment to our surviving dependents than a savings account would have. This is because the survivors subsidize the unfortunate by forfeiting their payments. Like other financial products, life insurance and related services such as defined-contribution pension plans are agreements between a company and a policy holder that define payment obligations for both parties. Unlike many bank products, life insurance and pension products literally last a lifetime. Additionally, they fundamentally feature the spreading of risk across a large population, with companies earning their profits through administration fees rather than speculation in this risk.

Life insurance and pension companies must live up to a number of regulations regarding *solvency*. In other words, they must be able to convince regulators that they will be able to cover the obligations imposed by the agreements that they have entered with their customers. Additionally, when designing new products or taking on new customers, insurers must have a way to determine the correct prices to charge, such that their income should be expected to be close to their costs.

The primary mathematical models used by the Danish industry are based on continuous-time Markov models, which give rise to a system of differential equations. Solving these equations yields a prediction of the capital requirements imposed by a particular product.

Unfortunately, many realistic products give rise to equations that cannot be solved analytically. Additionally, new regulatory requirements such as the EU directive *Solvency II* [18], demand a wider variety of computationally intensive analyses. Insurers must predict the obligations that result from millions of pension contracts in thousands of potential future scenarios. The old way of doing business may no longer be possible. Fortunately, modern off-the-shelf computer hardware, in the form of graphics processing units (GPUs), offers affordable high-performance parallel computing.

The Actulus project, a collaboration between the University of Copenhagen, the IT University of Copenhagen, and the actuarial software firm Edlund A/S, is developing a domain-specific functional language for representing insurance and pension products. This language is called the Actulus Modeling Language, or AML. Previous work [5, 6] has described AML as a notation for these products, but did not go into detail about the execution strategy. Here, we begin to bridge the gap from the high-level functional language to the efficient execution of an interesting computational workload on a modern parallel architecture.

Our compilation process, from a high-level language based on actuarial theory to low-level CUDA C code, provides numerous opportunities for optimization. We describe optimizations motivated by actuarial domain concepts as well as by the low-level architecture of NVIDIA GPUs.

Contribution

This paper describes 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 performing *Solvency II* calculations on an NVIDIA GPU.

This framework represents the first step towards enabling AML to target heterogeneous parallel architectures.

Overview

Section 2 gives an overview of the actuarial theory and technology that underlies our approach as well as the core terminology of the article. Section 3 describes our solution for calculating reserves

FHPC'14, September 04 2014, Gothenburg, Sweden.

http://dx.doi.org/10.1145/2636228.2636230

on GPUs, including optimization details. Section 4 provides a brief description of the AML language. Section 5 describes our computational framework, bridging the gap from AML to GPUs. Section 6 extends our GPU solution to work across a collection of virtual servers with GPUs. Section 7 presents specific performance results for our generated GPU code and Sections 8 and 9 present related and future work. Our final conclusions are presented in Section 10.

Relation to Earlier Work

This paper is much revised and expanded from a presentation [12] to the 2014 International Congress of Actuaries. We target computer scientists, go into more detail about the generated GPU code and the AML language and its relationship to the rest of the system, and include preliminary experience with GPU computing on Amazon's Elastic Compute Cloud.

2. Background

Since terminology differs from country to country and even from company to company, we define a few core terms:

Product A product is a specific coverage offered by an insurance company, such as a disability annuity (an amount paid to the insured so long as he or she is disabled and alive) or a death benefit (a lump sum paid upon the insured's death).

Policy A policy describes all the insurance products held by a particular insured customer.

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

2.1 Actuarial Concepts

Following standard Danish practice, we base our models of life insurance and pension products on continuous-time Markov models. These models consist of a finite number of states and transitions between them. Together, these describe the life of the insured, where the states represent situations such as "the insured is alive", "the insured is able-bodied", or "the insured is dead". Transitions represent stochastic occurrences, such as death or disability, that cause the insured to leave one state and enter another. These models exhibit the Markov property — in other words, the chance that a transition will occur in a particular time interval is determined only by the present state of the insured and the bounds of the interval in question, and not by the previous history of the insured life. The Markov property is a significant simplifying assumption, which rules out many realistic assumptions that could otherwise complicate the model. For example, we cannot model that a previously-disabled customer might be more likely to become disabled again than a customer who has not previously been disabled.

The likelihood of a transition from state i to state j is represented using a time-dependent *transition intensity* $\mu_{ij}(t)$. Roughly speaking, integrating the transition intensity over a time interval gives the probability of a transition occurring in that interval. Figure 2.1 demonstrates an example Markov model, with states representing active labor market participation, disability, and death.

An insurance product associates payments with a stochastic model. In particular, it may specify that while in a state j, the insured or co-insured will receive (or make) a continuous payment $b_j(t)$, in dollars per year. Upon a transition from state j (e.g. active) to state k (e.g. dead), the insured or co-insured may receive a lump sum payment $b_{jk}(t)$, in dollars.

The paradigmatic computation on products is to determine the *reserve*, which is the expected net present value of the future payments in a product, taking interest rates and stochastic transitions into account. Reserves are important due to the principle of *equivalence*, according to which the price of a product should exactly cover the necessary reserves. The state-wise reserve for a customer 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 [11] that is derived from the continuous-time state model, its transition intensities, payments, and interest rate $r(t)$. Each state gives rise to an equation, so the reserves for an n-state model are computed by solving a system of n deterministic ordinary differential equations (ODEs).

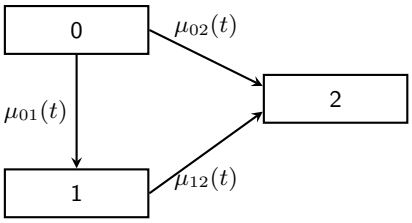


Figure 1. State model in which 0 = active, 1 = disabled, 2 = dead.

Analytical (closed-form) solutions to Thiele's differential equations have traditionally allowed insurers to determine the correct prices for their products without resorting to expensive computations. However, some instances of Thiele's equations cannot be solved analytically, in particular those that result from models whose transition graphs include cycles. Unfortunately, cyclic models arise naturally when a customer who has previously been declared disabled might recover and return to the labor market: there is a transition from the active state to disabled, and vice versa. In the present work we solve Thiele's differential equations numerically, so we can support more products and more realistic models.

Further definitions of actuarial concepts and terminology can be found in an earlier paper about AML [6] and in Gerber's text [11], but the present paper should be comprehensible without them.

2.2 Technological Background

Modern CPUs, such as those found in most desktop and laptop computers, are very powerful, but not suited for doing many calculations in parallel. GPUs originated as a technology for displaying advanced graphics in 3D games, and are found in the majority of personal computers today. GPUs were once extremely specialized, and had no other function than to produce 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, numerically solving the differential equations in parallel is an attractive possibility. Fortunately, modern graphics cards are very well-suited for this. Instead of just a few cores, they have hundreds or even thousands. There are, however, some challenges, as the individual cores on graphics cards are slower than on a CPU. There are also some restrictions on what they can compute in parallel, and thus a performance optimized solution entails carefully tailoring programs to make the most of the GPU architecture.

CUDA and NVIDIA GPUs

A leading supplier of GPGPU hardware, NVIDIA, has developed an infrastructure for GPGPU programming called the Compute Unified Device Architecture (CUDA). CUDA is both a hardware specification that NVIDIA's graphics cards implement and software tools that allow developers to write code that executes on the graphics cards. The architecture of graphics cards implementing CUDA is very intricate, and consists of many layers of memory and processing units. Here we give only a brief introduction to CUDA and GPGPU programming in general.

A GPU consists of multiple streaming multiprocessors (SM) that in turn have several cores. For example, the already somewhat aging NVIDIA Tesla C2075[1] graphics card, which is a high-end card built for high-performance computing, has one GPU with 14 streaming multiprocessors, each with 32 cores. This results in a total of 448 cores, supporting parallel execution of 448 instructions. For example, a multiprocessor can multiply the 32 numbers $a[0]$ to $a[31]$ by 7, computing $a[0] \cdot 7$, $a[1] \cdot 7$, ..., $a[31] \cdot 7$, in a single instruction cycle. On a CPU, these would be computed sequentially, which would take much longer.

CUDA uses a Single Instruction, Multiple Data (SIMD) execution model. Threads are bundled into groups of 32, called *warps*. This means that within a warp, *only* one single type of instruction can be performed at a time. Given 32 numbers, multiplying each of them with 7 would be perfectly parallelizable, 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 2 shows an example where a result is needed for each of 32 customers. Each customer is different, and the result is dependent on the customer 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.

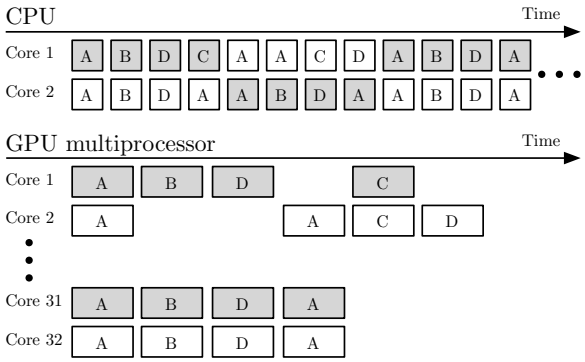


Figure 2. Calculations for 32 customers on a 2-core CPU and a 32-core GPU.

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 3.

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

3. Calculating Reserves on GPUs

To speed up calculations using a GPU, one must first identify a decomposition of the problem into a number of parallel threads, each of which follows a substantially similar control path. In the case of Thiele's differential equations, an efficient parallelization strategy should identify a decomposition in which the same equations can be solved with different parameters. Ideally, there would be an interesting and computationally demanding computational workload that performed a large number of analyses for one policy, with its unique collection of atomic products.

Technical Paragraph 64 of *Solvency II* requires insurance companies "...to ensure that ruin occurs no more often than once in every 200 cases" [18]. This means that an insurance company must be able to cover the combined prospective reserve of all obligations in 99.5% of a large collection of potential future scenarios. Our initial GPU solver, called *CUDA Life Insurance Reserve Estimator* (CLIRE, pronounced "clear"), implements this requirement by concurrently simulating thousands of interest rate paths, solving the customer's differential equations in each.

CLIRE generates simulated interest rate paths on the GPU, using a method developed by Cecilie Horn [14] at Edlund A/S. These interest rate paths are simulated in the Cox–Ingersoll–Ross model [7], based on data from the Danish Financial Supervisory Authority, *Finanstilsynet*.

Once the interest rate paths have been generated, CLIRE calculates the state-wise reserve for each customer by numerically solving Thiele's differential equation [11, p. 71], using the fourth-order Runge-Kutta method (RK4) [20]. It does this for every interest rate path, thereby producing many different possible reserves. This is a perfect opportunity for parallelization on the GPU, as we can calculate all the reserves for each customer in parallel. In principle it is necessary to perform (linear) interpolation in each interest rate path, but since the RK4 method uses a fixed step size, all the required interpolations of a given interest rate path can be precomputed and tabulated before the reserve calculations are performed.

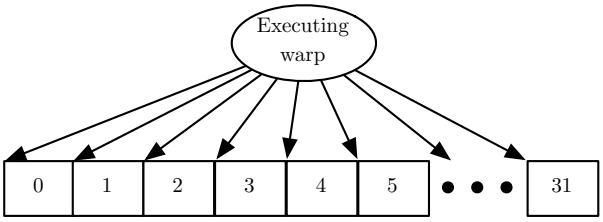
If a single customer does not provide enough computational work to saturate the multiprocessors on the GPU, it is still possible to achieve a high degree of parallelism by also parallelizing with respect to customers. This is because the lock-step execution model of the GPU only applies to each warp of (32) threads. In other words, as long as we are working with at least 32 interest rate paths, which we always are, we can also parallelize across different customers, even if they hold different products.

3.1 Optimizing performance on GPUs

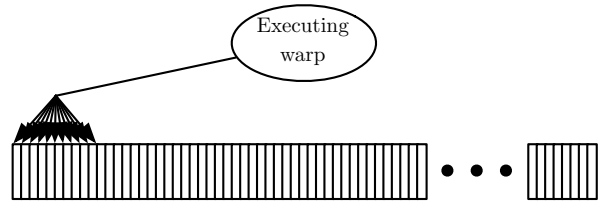
The challenge of performing calculations for millions of customers each with multiple products to be simulated with thousands of different interest rate paths, is a tremendous task even for a highly parallel architecture. As a consequence, great effort has gone into optimizing the various parts of CLIRE dealing with parallelization and memory access.

When optimizing GPU code, there are several ways to get increased performance, beyond regular optimization techniques. Two quantities in particular are important: *thread divergence* and *occupancy*. Thread divergence is the measurement of how often work has to be performed sequentially, due to differing instructions across a warp. Ideally, this should be 0%, although this can be hard to achieve in practice. In CLIRE thread divergence is minimized by parallelizing with respect to interest rate paths.

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps [17]. This is affected by many different factors. The launch configuration, register use per thread, and shared memory use per thread all play a part in determining occupancy. Counterintuitively, higher occupancy does *not* always equal higher performance [22]. This



(a) Interest rate paths: Several reads per step



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

Figure 3. Memory access without and with interleaved interest rate paths.

means the optimal occupancy must be determined on a case by case basis. In this project, the optimal launch configuration and register usage was determined experimentally. We only use shared memory as a cache, so it did not play a large role in determining occupancy.

Besides optimizing occupancy and minimizing thread divergence, there is a third area of optimization that is substantially different on GPUs: memory access. NVIDIA GPUs have a complicated memory model, with several layers exhibiting different performance characteristics.

The generated interest rate paths are both the largest data and the most frequently accessed dataset. Because of this, much effort was put into optimizing how this data is accessed. As mentioned in Section 2.2, calculations on products are bundled in warps of 32. These calculations are carried out in parallel on a single multiprocessor. Every time a multiprocessor reads a value from memory, it returns the adjacent values as well. This can be exploited to improve memory throughput by arranging for information required by each concurrently-executing instruction to be adjacent in memory.

In a single warp each of the reserve calculations are for the same customer with the same life insurance product, but for different interest rate paths. Figure 3.1 depicts two ways of accessing the very first number (that is, year zero) of an interest rate path for a warp of reserve calculations. The naïve method of Figure 3.1 (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 Figure 3.1 (b) ensures one coalesced memory read for each step in the calculation by placing the first numbers from the 32 paths next to each other. The second numbers for each of the paths are placed together in the same fashion, and so forth. This technique ensures a speed increase by utilizing knowledge about the memory access patterns of the GPU. In practice, this lead to a fourfold increase in performance.

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 insurance companies' entire portfolios. The *Solvency II* requirement that insurers verify their solvency in a multitude of future scenarios has introduced a golden opportunity for parallel processing, allowing parallelization not only at the level of customers and products but also at the level of future scenarios.

```
statemodel LifeDeath where
  states = alive | dead
  transitions = alive -> dead

product PE(expiry : TimePoint) : LifeDeath where
  obligations = at t pay $1
                when (t = expiry)
                provided (alive)

product TLA(expiry : TimePoint) : LifeDeath where
  obligations = at t pay $1 per year
                provided (alive and t < expiry)

riskmodel Mortality(p : Person) : LifeDeath where
  intensities = alive -> dead by gm(p)

basis MortalityBasis(p : Person) : LifeDeath where
  riskModel = Mortality(p)
  interestRate = (t : TimePoint) => 0.05
  maxtime = p.birthDate + 120
```

Figure 4. Example AML code

While CLIRE is extremely fast at what it does, a problem arises: Each possible insurance product must be specified in CUDA C. This creates an unfortunate linkage between programmers and actuaries. Actuaries cannot experiment easily with different insurance products, and programmers must spend time implementing potentially hundreds of different products, tuning each of them. In the next sections, we will describe how we can overcome this challenge using the domain-specific language AML and code generation.

4. AML

The Actulus Modeling Language, or AML, is a domain specific language for describing the actuarial models that were introduced in Section 2.1. AML is a functional programming language with first-class support for concepts like state models, risk models, and life insurance or pension products. AML is described in more detail in previous papers [5, 6], which include examples of larger models and programs. Here, we only provide enough details to illustrate its relationship to the work described in this paper. In particular, we demonstrate only the product definition sub-language.

Typically, life insurance and pension funds have far more products than they have underlying statistical models. Thus, AML separates these concerns, having independent notions of *risk model* and *product*. The risk model consists of the transition intensities $\mu_{ij}(t)$, while the product specifies the lump-sum payment functions $b_{ij}(t)$ and the payment intensities $b_i(t)$. Both risk models and products are specified inside a state model, which constrains the available states and transitions.

Figure 4 demonstrates some simple AML definitions. The state model `LifeDeath` is a two-state model, corresponding to that represented in Figure 5. In the AML representation, the state numbers have been replaced by the names `alive` and `dead`. In Figure 4, two products are defined in this state model: a *pure endowment*, in which the insured is paid a fixed amount upon survival to a given date, and a *temporary life annuity*, where the insured is paid a fixed amount per time period until his or her death, or until some expiry date, whichever comes first. These products are called `PE` and `TLA`, respectively. When computing reserves (and other quantities) for `PE` and `TLA` we use the risk model `Mortality` to define the intensity of mortality $\mu_{01}(t)$ in terms of the (partially

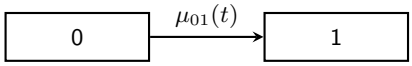


Figure 5. State model in which 0 = alive, 1 = dead.

applied) library function `gm`, which maps a representation of a person `p` a to his or her mortality intensity function. In this case, `Mortality` uses the standard Gompertz-Makeham model, whose AML implementation is called `gm`. Note that `gm` is an ordinary function, written in AML, rather than a language primitive.

To calculate the prospective reserves (and therefore the necessary price) for a product, an AML implementation must combine the product with a *calculation basis*, which consists of a risk model, a model for interest rates (the function $r(t)$ in Section 2.1), and the time in the future when we assume the insured to be dead and all obligations imposed by the product to be discharged, which serves as a boundary condition for the resulting system of equations. From the product and calculation basis, an AML implementation generates Thiele's equations and then solves them numerically.

The sample calculation basis in Figure 4 defines a constant interest rate of 5%, stops the solver when the insured is 120 years old, and uses the transition intensities from `Mortality`. In fact, `Mortality` and `MortalityBasis` represent *families* of risk models and calculation bases, parameterized over the customers.

While AML is intended to be a stand-alone language, development and implementation are not yet complete. However, an early prototype of the language is implemented as an embedded domain-specific language in Scala. The embedded implementation uses Hofer et al.'s *polymorphic embedding* [13] technique, a variant of the final tagless embedding [4] that allows embedded languages to be composed modularly and then run in a variety of different semantics. This makes it possible to easily compare the output from our solver with a reference implementation.

5. Design of a Calculation Framework

While it is our intention that AML be sufficiently general to describe all interesting insurance and pension products, a calculation engine becomes much more useful if it can interoperate with legacy systems. Descriptions that have other sources should not be placed at a direct performance disadvantage. In this section, we describe a framework in which multiple front-ends can be defined for a common intermediate representation that is inspired by actuarial theory, such that products from any front-end can take advantage of both domain-specific and platform-specific optimization techniques. This framework, which we call the Actuarial Calculation Processor (ACP), can generate and execute code for calculating the prospective reserves for products in any form that can be translated to Thiele's differential equations.

5.1 Main Components

Figure 6 depicts the workflow structure of the ACP with a few concrete classes. A workflow consists of one or more Product Processors that translates products to differential equations, a Policy Processor that applies optimizations at the level of Thiele's equations, a code Generator to transform the internal differential equation representation to executable code, and finally an Executor and a Result Processor, which run the generated code and aggregate the results. Each of these components can be varied or replaced, as technologies and needs change, without requiring a rewrite of the entire system. Figure 6 shows a few concrete components of the ACP project. The AML Processor enables us to handle life insurance products specified in AML, whereas the Test Processor specifies a range of standard products in the internal differential equation language for testing purposes. The CLIRE Generator generates CUDA C code for the CLIRE project as described in Section 3, and the Multicore Generator generates an optimized CPU version of CLIRE mainly used for performance comparisons (see Section 7).

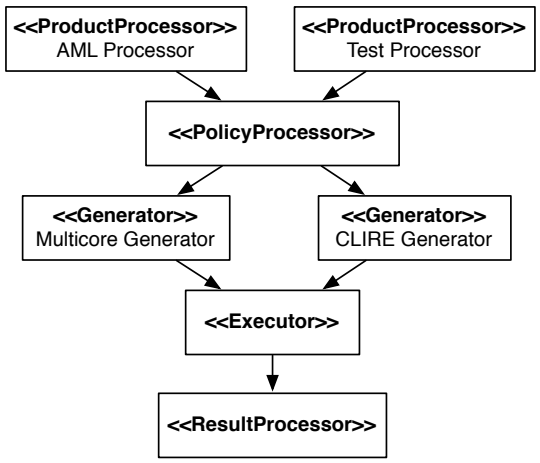


Figure 6. Abstract and concrete components of the Actuarial Calculation Processor.

5.1.1 Product Processors

A product processor transforms descriptions of products into our internal representation of Thiele's differential equations. This internal representation is a straightforward abstract syntax tree for each equation in the system, along with a global environment that maps variable names to further expressions. The ACP can be extended to support a new language or system for defining products by simply adding an additional product processor. Currently, the ACP contains two product processors: a low-level product specification language used internally by Edlund A/S for testing ODE solvers and a direct representation of the internal ODE syntax. Additionally, an early prototype of the AML product definition language has been embedded in Scala. This embedded AML can generate Edlund's low-level product specification language. Using this embedding, typical products can be specified using 10–20 lines of code. The embedded version of AML has not yet been fully integrated into the ACP.

By requiring all product processors to convert their input to our common differential equation format, we can apply certain optimizations no matter which product description syntax is used. This optimization step is performed by the framework just before the products are handed to the Policy Processor. A example of an optimization step would be the empty death state. In many life insurance products, all obligations between the insured and the insurer terminate with the death of the insured. This means that Thiele's equation corresponding to the death state is a constant equation, so one need not solve it numerically, which leads to a significant performance increase [8]. Such optimizations will benefit any component that makes use of the equations later in the workflow.

5.1.2 Policy Processors

A collection of products is insufficient for calculating a reserve. They must first be combined into specific policies held by specific customers using specific risk models, so that we can insert the appropriate particular transition intensities $\mu_{ij}(t)$ into the differential equations. A policy processor performs this combining and outputs a Portfolio object that forms the basis for calculations. In practice, this information will come from a variety of systems. For test pur-

poses, the ACP includes a Policy Processor that generates realistic but artificial customers and policies.

5.1.3 Generators

Once a collection of optimized differential equations has been generated by a policy processor, one or more *generators* are invoked to generate platform-specific calculation code. The architecture of the system imposes no particular algorithm or paradigm on the generators, so in theory a generator could use any method to solve the equations. The implementation technology is therefore independent of the product specifications and the syntax in which they are written. 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.

As a proof of concept, we developed the CLIRE generator, which generates optimized CUDA C code, as described in Section 3. A hand-written high-performance skeleton, with careful management of the memory hierarchy as described in Section 3.1, is specialized to a particular system of differential equations by adding the code that is specific to those equations.

5.1.4 Executors and Result Processors

The final steps in the ACP pipeline are executors and result processors, which execute the generated code and interpret the results. The present system simply outputs a table representing the numerical solution to the differential equations.

5.2 Architecture 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 collection of Product Processors and Policy Processor can be used with any Generator, Executor, and Result Processor pipeline. The combination of domain-specific optimizations at the level of the differential equations and platform-specific optimizations in specific code generators allows the construction of very fast solvers. Specific benchmark results will be discussed in Section 7.

6. Parallelizing Across GPUs in the Cloud

As demonstrated in Section 3, *Solvency II* calculations can be efficiently performed on GPUs by parallelizing with respect to interest rate paths. However, given a large enough portfolio, simulated with enough interest paths, a single GPU might not be enough to deliver timely results. Since the calculations for each customer/interest rate path pairing are independent, it is easy to parallelize across multiple GPUs as well. As graphics cards for computational purposes are relatively expensive, and demand can be unpredictable, this is a perfect fit for cloud computing.

To test our hypothesis that parallelization across GPUs is feasible, we developed a proof of concept framework for estimating reserves on Amazon's web services. This solution consists of a single master server exposing a web service. This master server can then spin up multiple worker instances, each with a GPU attached. These worker instances run CLIRE to perform a subset of the calculations. The master server then simply retrieves the partial results and combines them. Our initial results are promising, showing a nearly linear increase in performance with added GPUs.

This proof of concept solution currently exists separately from ACP, but the vision is to integrate these two projects. It should be possible to write a new execution stage for ACP that sends the calculations to the cloud. Another possibility is to move ACP itself to the cloud. This would completely free the client computer from performing reserve calculations.

Table 1. GPU 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. CPU CLIRE running on Intel Core i5 2400

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

These results vindicate the architectural decision to separate the high-level declarative description of the insurance products from the low-level details of ODE solvers, as well as the decision to separate GPU code generation from execution plans. It becomes much easier to adapt to the rapidly-changing landscape of today's heterogeneous parallel architectures.

7. Results

The following results are from two versions of CLIRE generated by the ACP. The first runs on NVIDIA GPUs, and the second runs on multicore CPUs. This CPU 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 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 GPU version of CLIRE, while the CPU version of CLIRE was run on an Intel Core i5 2400 at 3.4 GHz, using 4 threads on 4 cores. All computations use double-precision floating point.

As can be seen from the results, CLIRE is more than 100 times faster when run on the graphics card. In fact, CLIRE running with 100,000 test customers takes less time to compute reserves than the CPU version takes for 1,000 customers! It should also be noted that execution time scales almost linearly with the number of customers. The running times of CLIRE on 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 CPU version of CLIRE, due to the excessive running time.

8. Related work

General Purpose ODE solvers Prior work concerning the implementation of ODE solvers on CUDA includes Murray's study of several fixed-step and adaptive-step Runge-Kutta solvers [16]. Unlike these solvers, CLIRE benefits from only needing to solve one particular differential equation. For instance, in general it may seem that the variation in step size between threads in an adaptive-step solver would lead to thread divergence and hence poor performance on graphics processors.

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.

The concept of using domain-specific knowledge to optimize code written in a DSL occurs throughout the literature. In fact, it was one of the major design goals for Hofer et al.'s polymorphic embedding technique, used by the embedded AML prototype.

The SPIRAL system [21] for defining DSP transforms doesn't stop with domain- and platform-specific optimization passes. During compilation, SPIRAL configures its optimizations by executing test code with these optimizations applied and recording the results. This could be an interesting source of inspiration for future improvements to CLIRE.

Financial contracts Peyton Jones, Eber and Seward present a language for building complex financial contracts by combining simple contracts [19]. 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. [10] and Flænø Werk, Ahnfelt-Rønne and Larsen [9]. The latter is especially relevant, as they use an embedded domain-specific language to generate optimized GPGPU code. However, these systems calculate contract prices by aggregating the results of many simulations, rather than by numerically approximating the solution to a system of differential equations. Likewise, they hard-code a particular code generation technique, while ACP is designed for pluggable backends.

9. Future Work

There are a number of possibilities for future extensions to the ACP. One direction concerns the generation of product-specific C and/or JVM code for standard multicore (desktop and server) hardware. Such hardware is likely to be somewhat slower than the graphics processor hardware for the *Solvency II* workloads that we are interested in, but it is more readily available and may be more readily applicable to other actuarial calculations. Like CLIRE, such a backend could be extended to support cloud computing on a system like Amazon's Elastic Compute Cloud or Microsoft's Azure.

Another interesting area of future work is to generate more sophisticated numerical differential equation solvers, such as adaptive-step-size Dormand-Prince solvers. Initially, one might expect that adaptive solvers are unsuitable for GPUs, because variation in step size between threads might lead to thread divergence and hence poor performance. However, preliminary work by students at the IT University of Copenhagen [15] leads us to believe that this may not be a problem in practice.

One major missing feature in the current ACP is support for what are called *collective* products in the Danish context. In these products, a stochastically-dependent benefit is paid to a recipient who is unknown at the beginning of the contract. For example, a life insurance product may provide a life-long annuity to a surviving spouse, should such a spouse exist. These products require special pre-processing to avoid massive increases in algorithmic complexity.

The above directions for future work require new code generators and executors (Figure 6), but they require no fundamental changes to product descriptions. However, there is also potential in expanding the class of allowed actuarial models.

Many perfectly reasonable assumptions about risk cannot be encoded in the continuous-time Markov model described in Section 2.1. For example, one might want to encode that short-term disabilities improve the chances of recovery. Semi-Markov models allow transition intensities to additionally take the length of sojourn in the present state as an argument. However, the corresponding extension of Thiele's equations leads to partial differential equations, which do not admit the same straightforward treatment as the ordinary differential equations that the ACP can solve. A PDE backend for the ACP would enable a major increase in AML's expressivity.

10. Conclusion

We have demonstrated that it makes sense to decouple high-level descriptions of actuarial products in a domain-specific functional language from the execution technology that efficiently computes quantities of interest. Additionally, we have demonstrated an architecture for combining multiple sources of information and multiple execution technologies. In particular, we have demonstrated how to use these techniques to generate efficient code for NVIDIA GPUs. Our results indicate significant performance increases relative to comparable code running on traditional desktop and server processors. Additionally, preliminary results indicate that this system can scale up to modern cloud computing environments.

Acknowledgments

This work was supported by the Danish Advanced Technology Foundation (*Højteknologifonden*) through grant 017-2010-3. We would like to thank our company partners from Edlund A/S for their time and expertise, especially Cecilie Horn for her help with generating interest rate paths, Jeppe Woetmann Nielsen for his help with generating test customers, and Henning Niss for bridging the gap between the computer scientist's and the actuary's worldview. We would also like to thank Allan Ryan for his comments on our earlier ICA presentation.

References

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

[2] J. Andersen, E. Elsborg, F. Henglein, J. Simonsen, and C. Stefansen. Compositional specification of commercial contracts. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(6):485–516, Nov. 2006.

[3] B. Arnold, A. V. 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] J. Carette, O. Kiselyov, and C. C. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(05):509–543, 2009. .

[5] D. R. Christiansen. Software development for the working actuary. In *Proceedings of the Fourth International Symposium on End-User Development*, pages 266–271. Springer Berlin Heidelberg, 2013.

[6] D. R. Christiansen, H. Niss, K. Grue, K. S. Sigtryggsson, and P. Sestoft. An actuarial programming language for life insurance. 2014. Presented at *International Congress of Actuaries 2014*, Washington, D.C., USA.

[7] J. C. Cox, J. Ingersoll, Jonathan E., and S. A. Ross. An intertemporal general equilibrium model of asset prices. *Econometrica*, 53(2):pp. 363–384, 1985. ISSN 00129682.

[8] N. Dahl and C. Harrington. CUDA life insurance reserve estimator. Technical report, IT University of Copenhagen, May 2012.

[9] M. Flænø Werk, J. Ahnfelt-Rønne, and K. F. 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.

[10] S. Frankau, D. Spinellis, N. Nassuphis, and C. Burgard. Commercial uses: Going functional on exotic trades. *Journal of Functional Programming*, 19(01):27–45, 2009.

[11] H. U. Gerber. *Life Insurance Mathematics, Third Edition*. Springer, 1997.

[12] C. Harrington, N. Dahl, P. Sestoft, and D. R. Christiansen. High-performance reserve calculations for life insurance portfolios. 2014. Presented at *International Congress of Actuaries 2014*, Washington, D.C., USA.

[13] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of DSLs. In *Proceedings of the 7th international conference on Generative programming and component engineering*, GPCE '08, pages 137–148, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-267-2. .

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

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

[16] L. Murray. Gpu acceleration of Runge-Kutta integrators. *IEEE Transactions on Parallel and Distributed Systems*, 23(1):94–101, 2012.

[17] NVidia. Cuda c programming guide, 2014. URL `http://docs.nvidia.com/cuda/cuda-c-programming-guide`.

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

[19] S. Peyton Jones, J.-M. Eber, and J. 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.

[20] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 1988-1992.

[21] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE*, 93(2):232–275, Feb. 2005. ISSN 0018-9219. .

[22] V. Volkov. Better Performance at Lower Occupancy, Sept. 2010. URL `http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf`.