A Domain Specific Language for Accelerated Multilevel Monte Carlo Simulations

Ben Lindsey
Imperial College London
bl2312@imperial.ac.uk

Matthew Leslie
Bank of America Merrill Lynch
matthew.leslie@baml.com

Wayne Luk
Imperial College London
wl@doc.ic.ac.uk

***Abstract*—Monte Carlo simulations are used to tackle a wide range of exciting and complex problems, such as option pricing and biophotonic modelling. Since Monte Carlo simulations are both computationally expensive and highly parallelizable, they are ideally suited for acceleration through GPUs and FPGAs. Alongside these accelerators, Multilevel Monte Carlo techniques can be harnessed to further hasten simulations. However, researchers and application developers must invest a great deal of effort to design, optimise and test such Monte Carlo simulations. Furthermore, these models often have to be rewritten from scratch to target new hardware accelerators. This paper presents Neb, a Domain Specific Language for describing and generating Multilevel Monte Carlo simulations for a variety of hardware architectures. Neb compiles equations written in LaTeX to C++, OpenCL or Maxeler's MaxJ language, allowing acceleration through GPUs or FPGAs. Neb can be used to solve stochastic equations or to generate paths for analysis with other tools. To evaluate the performance of Neb, a variety of financial models are executed on CPUs, GPUs and FPGAs, demonstrating peak acceleration of 3.7 times with FPGAs in 40nm transistor technology, and 14.4 times with GPUs in 28nm transistor technology. Furthermore, the energy efficiency of these accelerators is compared, revealing FPGAs to be 8.73 times and GPUs 2.52 times more efficient than CPUs.**

I. Introduction

Stochastic equations form the bedrock of modern quantitative finance. Such equations, found across a variety of pricing models and forecasts, are frequently intractable. Since no direct solution can be found, we must use approximations to rely upon these models. Monte Carlo simulations provide a robust and flexible implementation of such an approximation. They have been used in a wide variety of applications from computational finance [1] to biophotonic modelling [4].

A. Why build a Domain Specific Language?

Portability is a fantastic benefit of using a DSL (Domain Specific Language). Rather than writing an OpenCL model, a C++ model, a FPGA model, and so forth, this paper introduces the Neb approach, such that one model can be written that generates all these outputs. As exciting new hardware accelerators are designed and built, old models can be migrated to these platforms by updating the Neb compiler - rather than rewriting all the models by hand.

Optimising the execution time of models is a common desire but one that often leads to complex code. Even simple stochastic models quickly become challenging to read and maintain after layers of optimisations are placed upon them. However, with a DSL, optimisations can be introduced at the compiler level rather than per model. Not only does this keep models simple by clearly separating the desired behaviours from the optimisations, it also means any optimisations discovered can be quickly applied to all models through recompilation.

Since stochastic models are such critical components of some businesses, and a mistake can be so expensive, they're often heavily tested. Enforcing a common structure and reducing the amount of code that is required to write models helps minimize risk.

B. Why are GPUs and FPGAs so suitable for acceleration?

While CPUs are highly optimised to compute large sequential programs, they face strong competition when problems become parallel. The growing demand to solve graphical problems has led to the development of GPUs, hardware specialised in parallel computations. Recently, this power has attracted developers facing problems beyond the original field of graphics. OpenCL is an ongoing effort to expand the use of GPUs beyond graphical problems.

Alongside this adaption of existing parallel hardware, FPGA research has resulted in powerful reconfigurable hardware that can be further specialized to certain domains. Maxeler's dataflow engines are an extension of this research, allowing Java programs to be written that can be compiled down to energy efficient and fast designs.

C. What is Multilevel Monte Carlo?

Multilevel Monte Carlo (MLMC) is a variance reduction technique that reduces the computational cost to achieve an accuracy of $O(\varepsilon)$ from $O(\varepsilon^{-3})$ to $O(\varepsilon^{-2})$ [8]. MLMC splits the simulation into several levels, where each level has a different number of paths and time steps to generate. The algorithm continuously optimises the simulation size as a payoff function converges to a target accuracy level.

D. What are this paper's contributions?

- The Neb language for modelling and executing stochastic differential equations (Section II).
- A skeleton and kernel architecture for compiling and running Neb models on CPUs, GPUs or FPGAs (Section IV) and with Multilevel optimisations (Section V).
- A performance comparison of CPUs, GPUs and FPGAs for path generation models written in Neb (Section VI).

```
% State Variables
price

% State Initialisations
price_{0} = 100.0

% Model Parameters
Simulations = 100
Steps = 100

% Random Sources
r \sim R[-1.0, 1.0]

% State Updates
price_{t + 1} = price_{t} + r_{t}

% Output
price_{t}
```

Fig. 1. A random walk model written in Neb and the paths it generates.

II. Overview of the Neb Approach

Neb is designed to provide a simple but effective way of describing Monte Carlo simulations, which can then be used in generating efficient implementations. While domain-specific languages for Monte Carlo simulations are not new [11, 18, 17], the novel aspects of Neb include the following:

- It adopts a Literate Programming [12] style in capturing Monte Carlo models. The benefits of this approach will be explained in Section III.
- It enables the development of Multilevel Monte Carlo simulation to reduce the computational cost for a given accuracy. Note that this is different from the multilevel customisation framework [11].
- It supports automatic generation of CPU, GPU and FPGA implementations from a single high-level description; Section VI shows the experimental results comparing the performance and efficiency of such implementations.

To capture the essence of a Monte Carlo simulation, each Neb model is split into the following components (with examples in Table I):

A. State Variables

The set of variables whose path should be simulated.

B. State Initialisations

The initial values assigned to these state variables.

C. State Updates

The functions that calculate new values for the state variables at each simulation step. These functions can range from simple increments of previous values to complex updates using a mix of other state variables and random numbers.

D. Model Parameters

A set of constant values used throughout the model.

For path generation, the number of paths and the desired number of time steps should be specified here. For payoff convergence, the desired accuracy level should be specified.

E. Random Sources

The distributions and correlations of random numbers that are used in `State Updates`. Random sources can be distributed normally with a given mean and standard deviation ($r \sim N[\mu, \sigma]$), or uniformally between two numbers (for random floats: $r \sim R[\text{low}, \text{high}]$).

F. Output or Payoff

Simulations that specify an `Output` will return the generated paths, allowing further analysis by other tools. On the other hand, simulations that specify a `Payoff` will reduce the paths and return the aggregated result.

TABLE I
Neb Component Examples

Component	Examples
State Variables	randomWalk
	blackScholes
State Initialisations	$\text{randomWalk}_0 = 100$
	$\text{logspot}_0 = 100.0$
State Updates	$\text{randomWalk}_{t+1} = \text{randomWalk}_t + \text{rInt}_t$
	$\text{logspot}_{t+1} = \text{logspot}_t + \text{drift}_t + \text{vol} * r_t * \sqrt{\Delta}$
	$\text{conditional}_{t+1} = \text{conditional}_t + \text{conditional}_t > 100\,?\,5 : 1$
Random Sources	$r \sim N[0, 1]$
	$\text{rFloat} \sim R[-1.5, 1.5]$
	$\text{rInt} \sim Z[0, 2]$
	$\rho(\text{r}, \text{rFloat}) = 0.5$
Model Parameters	$\text{Simulations} = 10000$
	$\text{Strike} = 105$
	$\text{drift} = [0.5, 0.6, 0.7]$
	$\text{df} = -0.05$
Output	randomWalk_t
Payoff	e^{df} max(e^{logspot_t} - Strike, 0)

III. Neb Models

Neb models are concise descriptions of problems to be solved. Void of implementation details, each model is essentially a mathematical description of a simulation rather than a typical sequential program that instructs hardware how to calculate a solution. Instead, the implementation details and optimisations are left to the Neb compiler (Section IV).

Descriptions of Neb models are captured in a strict subset of LaTeX. This offers several advantages:

- The syntax is familiar to many researchers and developers. One of the major barriers to entry for learning a new language is having to face a strange, unfamiliar syntax. LaTeX provides a warm welcome.
- The LaTeX syntax can clearly express equations. The most complex part of a Neb model is typically the `State Updates`. Expressing these equations becomes simpler when using a syntax designed for such scenarios.
- Neb models are a strict subset of LaTeX, so they can be compiled into PDFs using standard LaTeX compilers. This allows models written for Neb to be, in a sense, self documenting. Models can be viewed and analysed in a format where square roots and fractions are visualised. Furthermore, other tools that support LaTeX development, such as syntax highlighting editors, can be utilized.

```
% State Initialisations
wave_{0} = 100.0

% Model Parameters
Simulations = 10
Steps = 16

drift  = [-0.8, -0.8, -1.0, 2.0]
\sigma = [0.2, 0.2, 0.2, 0.4]
\Delta = [0.1, 0.2, 0.3, 0.4]

% Random Sources
r \sim N[0,1]

% State Updates
w = t \bmod 4

wave_{t + 1} = wave_{t} + drift_{w}
  + \sigma_{w} * r_{t} * \sqrt{\Delta_{w}}

% Output
wave_{t}
```

Fig. 2. Array parameters and mathematical function calls in Neb.

Fig. 1 illustrates one of the simplest Neb models, a random walk. The model specifies that 100 paths should be generated, each with 100 steps. At each time step, a random number should be added to the accumulator from a uniform real distribution of numbers between -1 and 1. After the simulation completes, the paths are forwarded to a graphing tool. As we'd expect from a symmetrical walk, the paths generated spread out over time, with a mean value approximately equal to the initial value.

Fig. 2 demonstrates the use of array parameters. As a path takes a new step, it accesses the next element in any referenced arrays. Here we define a wrapped index w to reuse elements once the arrays are exhausted, forming a cycle (Fig. 4).

```
% State Initialisations
```

$$spot_0 = 100.0$$

```
% Model Parameters
```

$$Accuracy = 0.05$$

$$T = 2.0 \quad r = 0.05 \quad \sigma = 0.3 \quad K = 100.0$$

```
% Random Sources
```

$$random \sim N[0, 1]$$

```
% State Updates
```

$$h = T/\text{Steps} \qquad \Delta = random_t \sqrt{h}$$

$$spot_{t+1} = spot_t \left(1 + rh + \sigma\Delta\right)$$

```
% Payoff
```

$$e^{-rT} \max(0,\ spot_{\text{final}} - K)$$

Fig. 3. Pricing a european call option in Neb (compiled with LaTeX).

Fig. 3 highlights a more practical model, pricing a european call option using an euler discretisation. Rather than returning a set of paths, this model uses the Multilevel mode of Neb to solve a `Payoff` equation. While a fixed number of time steps is used in Fig. 1 and 2, the number of steps here changes at each level. This can be referenced to adapt the `State Updates`. Fig. 3 also demonstrates the benefits of compiling Neb to LaTeX, providing a cleaner description of the models. Features such as $\sqrt{\text{square roots}}$ and symbols (Δ, σ) are often much more comprehensible in this format.

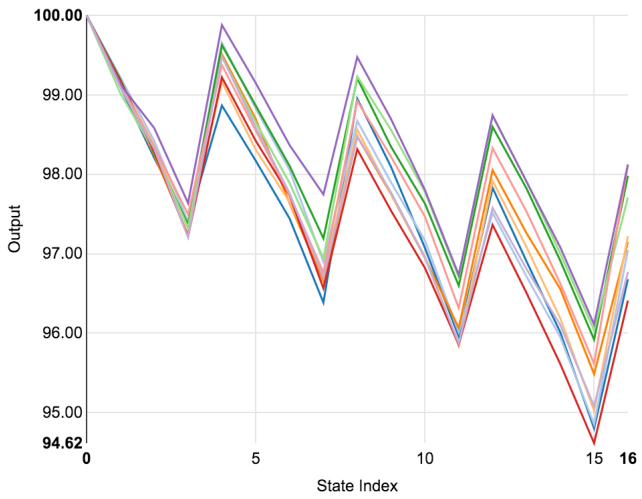

Fig. 4. Paths generated from the model in Fig. 2.

IV. Design and Implementation

Neb is an unusual DSL in the sense that it targets three very different hardware architectures. To achieve this range of outputs, the Neb compiler converts models into a high-level language for each of the targets, rather than compiling to low level implementations directly. Specifically, for CPUs Neb compiles to C++, for GPUs Neb compiles to OpenCL and for FPGAs Neb compiles to MaxJ. The Neb compiler then delegates to another compiler to build the final binary (Fig. 5).

To build these high level simulations, a skeleton implementation of Monte Carlo is defined in C++, OpenCL and MaxJ. The Neb compiler reads models and generates the missing pieces, slotting them into the skeleton to form a complete simulation.

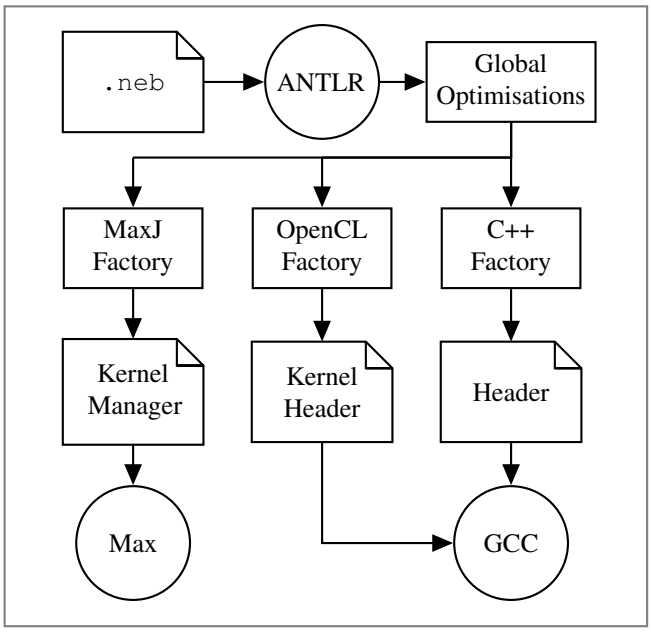

Fig. 5. Compilation flow.

A. The Monte Carlo Skeleton

The skeleton varies from architecture to architecture, but the general strategy remains the same:

1) Generate a stream of random numbers, distributed and correlated according to `Random Sources`.
2) Feed the random numbers to the target accelerator.
3) Initialise a set of state variables as directed by the `State Initialisations`.
4) Execute the simulation by repeatedly applying the `State Updates`, recording the `Output` at each time step.
5) Stream the resulting paths back to the CPU.

Generating the random numbers on the CPU allows a core part of the skeleton to be shared amongst all accelerators, reducing the implementation complexity. However, our experiments (Section VI) revealed a great deal of time was spent generating these numbers. The design of Neb will likely evolve towards supporting random number generation on the accelerators as well, increasing complexity in return for higher performance; efficient hardware implementations of random number generation for Monte Carlo simulation have been proposed [16]. Generally, the performance of Neb-generated designs will be improved as we push more stages to the accelerators.

B. The OpenCL Kernel

To perform parallel path generation with OpenCL, the simulation is split into several independent work items. The kernel is executed once for each work item, effectively simulating a single path. Fig. 6 illustrates a simple random walk kernel produced by Neb. At line 1, the kernel calculates its work id. This specifies a range in the flattened 2D array of random numbers that it should read from, and similarly a range in the 2D array of outputs it should write to.

```
1  const int id = get_global_id(1) * Steps;
2
3  // % State Initialisations
4  float price = 100.0;
5
6  for(int i = 0; i < Steps; i++) {
7      // % State Updates
8      price = price + r[id + i];
9
10     // % Output
11     output[id + i] = price;
12 }
```

Fig. 6. Key parts of the OpenCL kernel generated from Fig. 1.

C. The MaxJ Kernel

Path generation with MaxJ is achieved by forming a feedback loop in the dataflow graph for each state variable. Counter chains track the progress of the simulation, signalling whether a new path should be started or an existing one fetched from the feedback (via the ”newPath” flag in line 2). An illustration of the key elements in a MaxJ kernel is shown in Fig. 7.

```
1  // % State Initialisations
2  DFEVar price = newPath ? 100.0 : carried;
3
4  // % State Updates
5  price = price + r;
6
7  carried <== stream.offset(price, -offset);
8
9  // % Output
10 io.output("output", price, type, ready);
```

Fig. 7. Key parts of the MaxJ kernel generated from Fig. 1.

D. Type Inference

In Neb, types for `State Variables` and `Model Parameters` are implied by their initialisations, rather than explicitly set. For example, a variable set to 100 would be read as an integer, while a parameter set to [100.0, 50.5] would be infered as an array of reals. `Random Sources` are typed by their distributions. For calculations involving multiple types, the resulting type is derived from the highest priority one involved in the calculation, where real numbers have higher priority than integers.

Implementation of these abstract types is performed by the accelerator factories. For example, the real type maps to floats in OpenCL, while arrays are converted to ROM tables in MaxJ.

E. Optimisation Phases

DSLs typically allow greater optimisations to be made over general purpose languages simply because they restrict their domain of use [15]. Having fewer use cases allows stronger assumptions to be made when searching for potential optimisations.

To improve the performance of models executed with Neb, two phases of optimisation occur. Firstly, *Global Optimisations* identify platform independent improvements. For example, a `State Update` function may contain some constant calculations that can be lifted out of the simulation loop and performed only once.

Secondly, *Local Optimisations* take place. These are platform dependent improvements introduced by the three factories (Fig. 5). There are certain improvements that can only be made on specific accelerators, for example utilising multiple pipelines or mixing fixed and floating point calculations on FPGAs [6].

F. Adding an Accelerator to the Neb Compiler

As discussed in the introduction, a great benefit to DSLs is that when a new architecture is released it can be hooked into the Neb compiler, allowing all existing models to benefit. The general flow to adding a compilation target to Neb is:

1) Find an appropriate high level language for the desired hardware accelerator.
2) Write a Monte Carlo simulation using the language.
3) Extract the model specific parts of the simulations (state updates, parameters etc) into a separate header file.
4) Add a new factory to the Neb compiler that generates this header file given some state objects.

V. Multilevel Optimisation

To implement Multilevel Monte Carlo, the skeleton and kernel strategy discussed in Section IV is extended with additional control steps (Fig. 8). The plan now resembles:

1) Decide on the number of paths and time steps to simulate using the Multilevel Monte Carlo algorithm [8]. Initially, a large number of paths with few steps will be generated. As the simulation rounds progress, the number of steps generated for each path will increase.

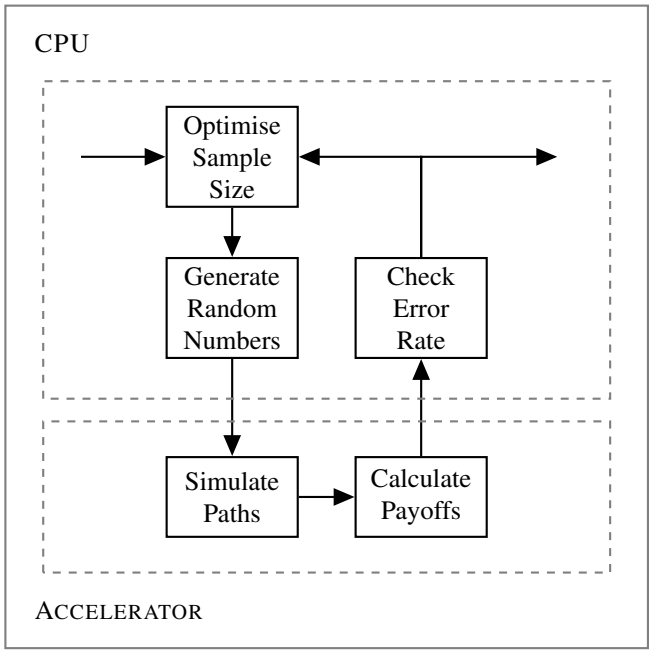

Fig. 8. Multilevel execution of models.

2) Generate a stream of random numbers, distributed and correlated according to `Random Sources`.
3) Feed the random numbers to the target accelerator.
4) Initialise a set of *fine* and *coarse* state variables as directed by the `State Initialisations`.
5) Execute the simulation by applying the `State Updates`. *Fine* state variables are updated twice at each path step, whereas *coarse* variables are updated once.
6) Reduce the *fine* and *coarse* paths using the `Payoff`.
7) Return the *fine* payoff and the difference between the *fine* and *coarse* payoffs to the CPU.
8) Calculate the error rate between the payoffs received so far. If they've converged to the desired level, successfully exit the program. Otherwise, loop back to step 1.

The Multilevel mode reuses much of the kernel logic used in normal path generation. However, since the size of the simulation is no longer known at compile time, some assumptions are broken. For example, in the MaxJ kernel (Fig. 7 line 7), the offset is no longer known. The implementation must now dynamically calculate the offset at runtime.

VI. Experimental Results

To investigate the performance benefits of acceleration in Neb, we executed a variety of models across the target architectures (Fig. 9). Simulating these different models reveals how acceleration varies with the complexity of the state updates. The random walk, a simple addition, is one of the cheapest models conceivable. Slightly more complex is the Black Scholes model, introducing multiplication and square roots. Finally, the Heston model [1] adds multiple state variables and random sources, array parameters, and further arithmetic. Each implementation uses single-precision floating-point numbers.

TABLE II
ACCELERATION OF PATH GENERATION

Model	Paths	Paths generated per second			Acceleration	
		CPU	GPU	FPGA	GPU	FPGA
Random Walk	2^{16}	2.57×10^5	5.32×10^5	8.53×10^4	$\times 2.07$	$\times 0.33$
	2^{12}	2.4×10^5	5.04×10^5	4.94×10^4	$\times 2.1$	$\times 0.21$
	2^8	5.67×10^4	1.17×10^5	7.80×10^3	$\times 2.06$	$\times 0.04$
	2^4	4.95×10^4	4.32×10^4	5.59×10^2	$\times 0.87$	$\times 0.01$
Black Scholes	2^{16}	1.2×10^5	5.24×10^5	8.63×10^4	$\times 4.36$	$\times 0.72$
	2^{12}	1.2×10^5	4.93×10^5	5.21×10^4	$\times 4.12$	$\times 0.44$
	2^8	7.97×10^4	2.06×10^5	7.82×10^3	$\times 2.58$	$\times 0.1$
	2^4	3.74×10^4	3.37×10^4	5.53×10^2	$\times 0.9$	$\times 0.01$
Heston	2^{16}	2.28×10^4	3.29×10^5	8.52×10^4	$\times 14.43$	$\times 3.74$
	2^{12}	2.23×10^4	3.11×10^5	5.45×10^4	$\times 13.98$	$\times 2.44$
	2^8	1.37×10^4	1.36×10^5	7.95×10^3	$\times 9.96$	$\times 0.58$
	2^4	7.09×10^3	2.37×10^4	5.62×10^2	$\times 3.35$	$\times 0.08$

We also varied the desired number of paths to investigate how the performance of the accelerators changes with the amount of parallelism available. With a fixed number of time steps (2^{10}), we executed the models using an Intel Xeon E5-1620 CPU, the Nvidia GeForce GTX TITAN Black GPU, and the Maxeler Vectis Dataflow Engine.

% Random Walk State Update

$$price_{t+1} = price_t + r_t$$

% Black Scholes State Update

$$logspot_{t+1} = logspot_t + drift + r_t vol\sqrt{\Delta}$$

% Heston State Updates

$$logspot_{t+1} = logspot_t + \frac{1}{2}v_t^+\Delta_t + r_t\sqrt{v_t^+\Delta_t}$$

$$v_{t+1} = v_t + \kappa_t(\theta_t - v_t^+)\Delta_t + \epsilon_t r2_t\sqrt{v_t^+\Delta_t}$$

Fig. 9. The Neb state updates used to test acceleration.

Table II shows the benefits of hardware acceleration becoming more apparent as the complexity of the models expands. For the simplest model, a maximum acceleration of 2.1 was achieved using GPUs, whereas for the more complex model a far greater increase of 14.43 times was achievable. Similarly, the hardware accelerators were able to offer greater improvements for highly parallel problems. Strangling the number of paths down to just 16 caused many of the models to execute slower on the accelerators than on the CPU.

The accelerators used in this experiment are not directly comparable, since they are based on different technologies. The FPGA system uses 40nm transistor technology, while the GPU is built using 28nm transistor technology. Another metric we can consider is energy efficiency, that is paths generated per second per watt. To measure this we connected the hardware to a power meter and recorded the mean consumption. Table III highlights the normalised efficiency of path generation with the Heston model using 2^{16} paths with 2^{10} steps, where Efficiency $= \frac{\text{Paths per second}}{\text{Load}-\text{Idle}}$. FPGAs proved the most energy efficient, providing 8.73 times the number of paths per watt.

TABLE III
EFFICIENCY OF PATH GENERATION

Accelerator	Idle (W)	Load (W)	Efficiency	Normalised
CPU	75	110	651	1.0
GPU	75	275	1645	2.52
FPGA	80	95	5680	8.73

We also investigated how the number of time steps impact the acceleration. We fixed the number of simulations at 2^{16}, then varied the number of time steps from 2^4 to 2^{24} and compared CPU and GPU results. Fig. 10 shows that the acceleration gain was independent from the number of steps.

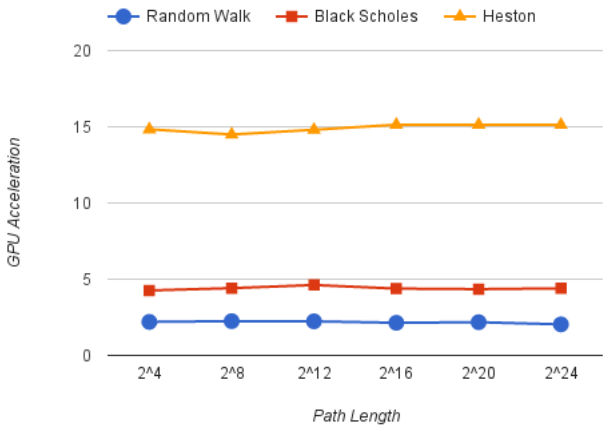

Fig. 10. GPU acceleration as the number of time steps increases.

% State Initialisations

$$v_0 = -50.0 \quad u_0 = 10.0$$

% Model Parameters

$$Simulations = 1 \quad Steps = 2000$$

$$a = 0.02 \quad b = 0.3 \quad c = 54.0 \quad d = 1.0 \quad I = 3.0$$

% Random Sources

$$r \sim R[0.18, 0.2]$$

% State Updates

$$v_{t+1} = v_t >= 30\,?\,c : v_t + r_t(0.04v_t^2 + 5v_t + 140 - u_t + I)$$

$$u_{t+1} = v_t >= 30\,?\,u_t + d : u_t + r_t a(bv_t - u_t)$$

% Output

$$v_t$$

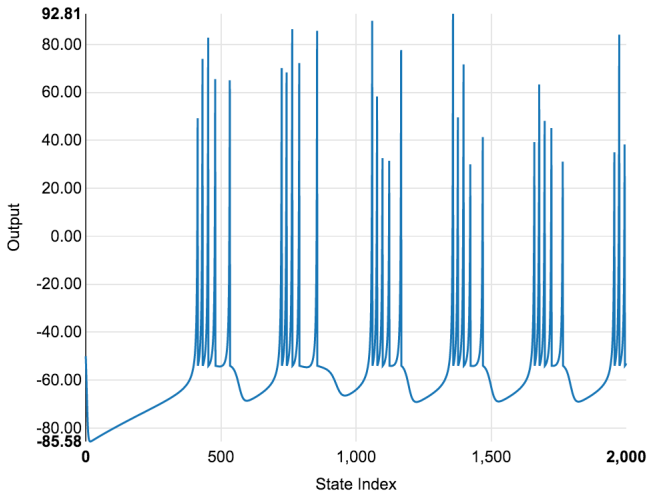

Fig. 11. Izhikevich bursting neuron model [10] written in Neb.

VII. Related Work

DSLs and their applications to Monte Carlo simulations have been the subject of exciting research. Neb builds upon this past work in several areas. DSLs designed prior to Neb typically have at most one target architecture, such as FPGAs for Contessa [18]. Since Neb models are written at such a high level, essentially composed of equations written in LaTeX, there's no constraint to a particular platform. The skeleton and kernel architecture presented in this paper allows Neb to be adapted to new accelerators easily. Existing DSLs typically require users to write imperative code in a C-style language [11] while Neb's Literate style allows more concise and mathematical models.

Several frameworks have been proposed for DSLs and for Monte Carlo simulations. Inspiration for expressing simulations by their mathematical components was found in the Monte Carlo framework proposed by Thomas et al [17]. DSL frameworks, for example Delite [15], help simplify the implementation of new languages. However, due to the Literate style of Neb and the range of target platforms, it proved simpler to implement the language using ANTLR rather than wrangling an existing framework into an unusual use case.

Multilevel Monte Carlo has shown promising results with FPGA accelerated designs [7]. Combining multilevel optimisations with mixed precision improvements has also shown strong results [2]. However, implementing these upgrades remains challenging. By abstracting away the implementation details from users, Neb simplifies the process of model development while maintaining the optimisations these features can bring. Research into the usability of high level abstractions over hardware accelerators has shown developers can create optimised programs despite not fully understanding the hardware they're using [9]. This is particularly promising news for Neb since each model can run across a variety of accelerators.

VIII. Further Considerations

The primary focus of this paper has been solving financial stochastic equations, a role Neb is particularly suited to. However, Neb can be extended to cover a range of other interesting areas. For example, in simulating the spiking of neurons, a task tackled by tools such as NeuroFlow [5]. Fig. 11 demonstrates how the Izhikevich bursting neuron model can be expressed in Neb. Rather than being tied to a particular scientific field, Neb can be used to generate paths or calculate payoffs for a variety of equations. The domain of Neb extends to any problem expressible through `State Updates`.

Looking forward, there are a range of interesting techniques that Neb could be extended to support. We have demonstrated Monte Carlo path generation and Multilevel optimisations, but the language could be adapted to use other strategies. For example, quadrature methods have proven useful for pricing options [19], while finite-difference methods have shown use in heat diffusion and fluid dynamics [13]. A common desire in computational finance is to calculate Greeks or sensitivities to instruments. Adjoint algorithmic differentiation can be used along side Monte Carlo simulations to calculate such sensitivities efficiently [3]. Expanding Neb to support algorithmic differentiation would be an exciting direction to take. Moreover, many of these additional capabilities can be captured as template libraries [14] facilitating their parametrisation and adaptation.

Additionally, the existing strategies could be improved through further optimisations. For example, currently Neb uses standard floating point numbers in its FPGA designs. Research into mixing floating point number representations with multiple levels of precision has shown promising performance gains [6]. We would integrate similar ideas into Neb, using reduced precision where possible and then falling back to higher precision when appropriate. Another powerful optimi-

sation would be to introduce some of the variance reduction techniques designed to support Monte Carlo simulations, such as importance sampling. A great advantage to using a DSL for modelling is that these optimisations can be layered onto each other in the compiler, without forcing models to become complex and incomprehensible.

IX. Conclusion

This paper has introduced Neb, a domain-specific language for describing and executing Monte Carlo simulations. Neb models are concise specifications of problems written in LaTeX. The Neb compiler takes these models and builds CPU, GPU, or FPGA designs to generate stochastic paths. The resulting paths can be reduced using a `Payoff` function or streamed to other tools. We have demonstrated (Fig. 9, 11) how differential equations written in LaTeX for publication can be lifted directly into Neb and solved via accelerated simulations.

To implement these features we have defined a skeleton and kernel system (Section IV) for integrating multiple target architectures into Neb. We have shown how this design can be extended with Multilevel optimisations (Section V) to improve convergence.

Through experimentation over a range of models, we have found that hardware accelerators can be used to improve the performance of Neb by a factor of 14.4 (Table II) with efficiency gains up to 8.73 times (Table III). Future work will look into further optimising the models generated by Neb and expanding the range of strategies it uses.

Acknowledgement

This research has received funding from European Union Horizon 2020 Programme with project reference 671653. The support by the UK EPSRC (grant references EP/I012036/1 and EP/L00058X/1), the Maxeler University Program, Altera and Xilinx is gratefully acknowledged.

References

[1] L.B.G. Andersen. "Simple and Efficient simulation of the Heston stochastic volatility model". In: *Journal of Computational Finance* 11.3 (2008), pp. 1–42.

[2] C. Brugger et al. "Mixed precision multilevel Monte Carlo on hybrid computing systems". In: *IEEE Conference on Computational Intelligence for Financial Engineering & Economics (CIFEr)*. 2014, pp. 215–222.

[3] L. Capriotti. "Fast Greeks by algorithmic differentiation". In: *Journal of Computational Finance* 14.3 (2011), pp. 3–35.

[4] J. Cassidy, L. Lilge, and V. Betz. "Fast, power-efficient biophotonic simulations for cancer treatment using FPGAs". In: *IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2014, pp. 133–140.

[5] K. Cheung, S.R. Schultz, and W. Luk. "NeuroFlow: A General Purpose Spiking Neural Network Simulation Platform using Customizable Processors". In: *Frontiers in Neuroscience* 9 (2015).

[6] G.C.T. Chow et al. "A mixed precision Monte Carlo methodology for reconfigurable accelerator systems". In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*. 2012, pp. 57–66.

[7] C. De Schryver, P. Torruella, and N. Wehn. "A multilevel Monte Carlo FPGA accelerator for option pricing in the Heston model". In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. 2013, pp. 248–253.

[8] M.B. Giles. "Multilevel Monte Carlo path simulation". In: *Operations Research* 56.3 (2008), pp. 607–617.

[9] G. Inggs et al. "Is high level synthesis ready for business? A computational finance case study". In: *IEEE International Conference on Field-Programmable Technology (FPT),* 2014, pp. 12–19.

[10] E.M. Izhikevich et al. "Simple Model of Spiking Neurons". In: *IEEE Transactions on Neural Networks* 14.6 (2003), pp. 1569–1572.

[11] Q. Jin et al. "Multi-level Customisation Framework for Curve Based Monte Carlo Financial Simulations". In: *Reconfigurable Computing: Architectures, Tools and Applications*. Springer, 2012, pp. 187–201.

[12] D.E. Knuth. "Literate programming". In: *The Computer Journal* 27.2 (1984), pp. 97–111.

[13] X. Niu et al. "A Self-Aware Tuning and Self-Aware Evaluation Method for Finite-Difference Applications in Reconfigurable Systems". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 7.2 (2014).

[14] M. Shafiq et al. "A template system for the efficient compilation of domain abstractions onto reconfigurable computers". In: *Journal of Systems Architecture* 59.2 (2013), pp. 91–102.

[15] A.K. Sujeeth et al. "Delite: A compiler architecture for performance-oriented embedded domain-specific languages". In: *ACM Transactions on Embedded Computing Systems (TECS)* 13.4s (2014).

[16] D.B. Thomas. "The Table-Hadamard GRNG: An Area-Efficient FPGA Gaussian Random Number Generator". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 8.4 (2015).

[17] D.B. Thomas, J.A. Bower, and W. Luk. "Automatic generation and optimisation of reconfigurable financial Monte-Carlo simulations". In: *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2007, pp. 168–173.

[18] D.B. Thomas and W. Luk. "A Domain Specific Language for Reconfigurable Path-based Monte Carlo Simulations". In: *IEEE International Conference on Field-Programmable Technology (FPT)*. 2007, pp. 97–104.

[19] A.H.T. Tse, D.B. Thomas, and W. Luk. "Design exploration of quadrature methods in option pricing". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 20.5 (2012), pp. 818–826.