Accelerating Financial Applications with SLX FPGA

Date	2019-09-19
Author	Syed Shahrukh Hussain

SILEXICA

Abstract

This white paper demonstrates how engineers creating FPGA-based hardware accelerators for financial market models can take advantage of SLX FPGA. SLX FPGA can be used to accelerate optimization efforts for financial market models targeting option pricing. In this paper, two implementations of computation intensive models for pricing options are discussed, namely the Black-Scholes [1] and Heston [2] pricing models. Using SLX FPGA, the implementations of these models for FPGAs are optimized while using a High-Level Synthesis (HLS) based methodology. Finally, a report is generated to compare achieved results with non-optimized versions of the hardware accelerators created using HLS. The Heston SLX FPGA optimized implementation outperforms the non-optimized implementation by 25x. The Black-Scholes SLX FPGA optimized implementation achieves 29x improvement over the non-optimized version.

1 Introduction

Efficient, low latency algorithms have emerged as an attractive option to improve the competitiveness of forecasting in financial markets. An algorithm that is originally implemented to run on commercial off-the-shelf systems, using multicore processors and GPUs, typically reaches a point where adding more computational resources is of little or no benefit. Programs running on a single computational unit consists of three parts: a setup section, a computation section, and a finalization section. The total execution time of a program is equal to the sum of the three parts [3].

$$T_{total}\ (1) = T_{setup} + T_{compute} + T_{finalization}$$

$T_{compute}$ can be parallelized by adding more computational units, but system overheads like T_{setup} and $T_{finalization}$ that are the serial part of a program prohibit all efforts to fully exploit an algorithm's parallelism potential [4]. In addition to this, on a shared architecture, communication overhead within system components also adds to total execution time.

To fully achieve optimized performance for a given algorithm, there is a need for dedicated hardware that hides the latency of sequential application parts in a pipeline and has zero communication latency overhead. Due to their inherent parallel nature, FPGAs are an excellent candidate for the optimized hardware implementation of algorithms traditionally designed for execution on multiple sequential processors (e.g., CPUs and GPUs). However, moving from a software implementation of an algorithm to an optimized hardware implementation is not easy.

High-Level Synthesis (HLS) is one such approach to address this problem by taking algorithms written in a high-level language, such as C/C++, and translating them into a register transfer level (RTL) hardware implementation. Existing HLS tools cannot translate every arbitrary C/C++ application into efficient hardware, and a designer needs to take in many considerations when preparing the C/C++ code for HLS. Often, the engineer needs to re-architect and refactor the algorithm using syntax that can be understood by the HLS compiler. Furthermore, HLS specific data types and libraries need to be used to guarantee that the HLS tool can create efficient hardware, and code annotations (i.e., pragmas or directives) are needed to direct the HLS compiler towards the desired optimization criteria (e.g., latency, throughput or area). SLX FPGA helps to address these challenges by providing a step by step optimization approach, helping the user transform the existing algorithm with minimal effort.

Financial terminologies used in this paper for both models are briefly introduced here as a springboard for rest of the discussion.

An option is a financial instrument which gives a certain person or institution the right to buy or sell an asset within a specified period. These financial instruments are traded in stock markets and exchanges. In order to accurately set the price of these options, different formulae are developed to calculate options call and put price in market. Among the popular formulae that exist are:

- Black-Scholes Model [1]: The Black-Scholes is a mathematical model of the financial derivatives market. The equation is a partial differential one, popularly used for pricing call and put options in the market. The model assumes two assets, one of which is risky and another which is riskless. Stock assets are normally considered risky, while money market, cash, or bond are riskless assets.
- Heston Model [2]: The Heston Model is an extension to the Black-Scholes Model, which considers more parameters that can affect the call and put price of an option. The

advantage of Heston over Black-Scholes is the correlation between underline asset price and volatility.

The multi-variable partial differential equations (PDE) of the above models are much harder to solve compared to ordinary differential equations (ODE). The Monte Carlo simulation is among the common methods for solving such problems numerically [5]. The Monte Carlo simulation can be applied to a wide variety of problems, both in non-engineering and engineering fields. The Monte Carlo simulation samples random inputs from a known probability distribution and then applies a deterministic computation on the inputs. This process is repeated for a large number of input samples, and the results of the computation are aggregated into confidence intervals, which describe the likelihood of observing a given result [5].

In Monte Carlo methods, multiple simulations are often required for accurate results. Sequentially running these simulations on a computer is not fast enough [6]. Given the fact that financial institutions need to react to market changes in real time, execution time for the option pricing algorithms is crucial to obtain good results for generating profit. Therefore, using massively parallel FPGAs to create efficient hardware accelerators for the aforementioned algorithms (and variations of them) is common in large investment banking institutions. In this paper, two existing open-source implementations of Black-Scholes and Heston[1] in C++ are examined to demonstrate how SLX FPGA can help engineers quickly convert obtained C/C++ code into an optimized hardware implementation, complementing Xilinx's Vivado HLS flow.

[1] https://github.com/KitAway

2 SLX FPGA Overview

One of the biggest challenges with using the HLS design flow is performance and design efficiency in terms of latency and area of the synthesized design. Improving the performance and area utilization of an HLS design requires parallelization and hardware-aware optimization. HLS design suites, such as the Xilinx Vivado™ provide several mechanisms for implementing parallelization and hardware optimizations but depend on the developer to specify them through HLS pragmas. Specifying these pragmas often involves making sophisticated tradeoffs that are not trivial for large complex applications. SLX employs extensive application analysis techniques to acquire deep insights into an application. These insights are then used to reason about parallelization and optimization decisions.

Most HLS tools on the market rely solely on compiler-based static analysis. However, large parts of applications can be data driven; it is often not possible for static analysis tools to extract exact metrics for these parts. For example, with static analysis, the micro-operations in a block of code are known, but its execution frequency is only known at runtime. Moreover, static analysis often fails to provide precise memory access patterns: for example, failing to see through pointer chains and detecting underlying objects or function calls. SLX uses dynamic analysis to capture such essential information when it is missing from the static analysis results. In fact, SLX FPGA is the first and (to date) only commercially available tool that applies the power of dynamic analysis to HLS optimization.

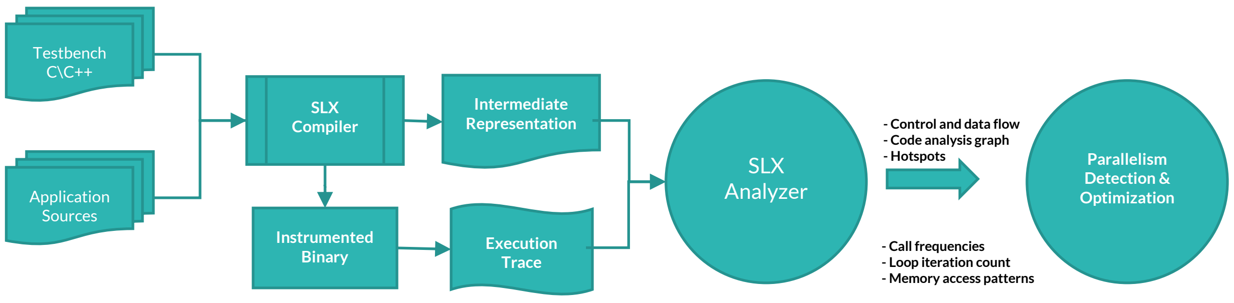


Figure 1: SLX Application analysis overview

Figure 1 provides an overview of SLX application analysis. SLX's proprietary compiler instruments the application, when required, at the level of instructions and memory accesses. A trace is generated by executing this instrumented binary. Critical information, such as memory access patterns, loop trip-counts, call, and conditional code execution frequencies are extracted from this trace. State-of-the-art analysis techniques are used to extract weighted code analysis graphs from the application, which is essential in reasoning about the different optimizations and to relate them back to the source files for code generation.

Another challenge for HLS users is synthesizability refactoring, especially for legacy applications. SLX FPGA facilitates the user with automated and guided synthesizability refactoring. The process with which SLX helps developers transform their C/C++ application code into an optimized, HLS ready code, is divided into four steps. Figure 2 gives an overview of these steps.

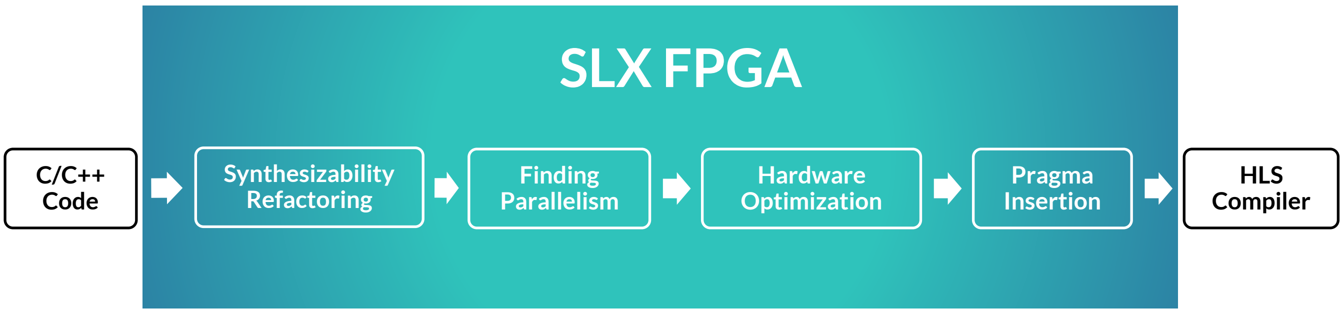


Figure 2: SLX FPGA workflow from a generic C/C++ code to a HLS ready code optimized for HW implementation.

2.1 Synthesizability Refactoring

Current HLS compilers support a restricted set of the C/C++ language. Example restrictions include the use of dynamic memory, variable sized function arguments, function overloading, function pointers and recursion. For novice users, patching these synthesizability issues often requires searching through thousands of pages of documentation. This makes synthesizing legacy software applications very cumbersome.

The SLX tool helps developers with automated and guided refactoring of non-synthesizable code. For example, it automatically replaces non-synthesizable library function calls with synthesizable ones. Detailed hints are generated to guide the developer in transforming non-synthesizable pieces of code into synthesizable code.

2.2 Finding Parallelism

The performance of an FPGA implementation relies heavily on the degree of parallelism present in the application. HLS compilers, such as Vivado HLS, support several different constructs for implementing parallelism in hardware, e.g. pipelining, unrolling and dataflow parallelism. However, these HLS compilers rely on developers to identify parallelization opportunities and specify their properties using pragmas. This is a non-trivial task that requires not only a deep understanding of the application but also awareness of the underlying FPGA architecture. In FPGAs, exploiting parallelization translates in an increased resource consumption; therefore, a system level approach is required to target resources toward the most important performance bottlenecks of the system. For medium-sized to large applications, where every loop has several possible parallelization candidates, the complexity can build up quickly.

SLX FPGA currently supports two parallelization patterns: (1) data-level parallelism and (2) pipeline-level parallelism. Data-level parallelism is selected for loops in which every computation in the for loop is either independent or can be mapped efficiently to a highly parallel hardware architecture (e.g., a multiply-add tree). Pipeline-level parallelism is selected for loops where different operations within a single loop iteration can be parallelized and where several loop iterations can execute simultaneously in individual pipeline stages [7].

2.3 Hardware Optimization

The next step in the SLX workflow is exploring and selecting hardware optimizations. The selected hardware optimizations are then passed onto the HLS compiler for implementation in the form of HLS pragmas. Currently, SLX supports five types of hardware optimizations: (1) loop unrolling, (2) loop pipelining, (3) array partitioning and interface design, (4) function inlining, and (5) loop trip count generation.

2.4 Loop unrolling

If loop unrolling is detected as a feasible parallelization candidate for a loop, it is further evaluated for speed-up yields for the application on the target FPGA platform. Moreover, the optimal (hardware-aware) unroll factors are calculated with respect to the available resources on the platform.

SLX FPGA takes a system level approach to parallelism. Utilizing the information collected in the static and dynamic analysis phases, SLX identifies the most critical bottlenecks and hotspots of the application. Cost-benefit analysis is conducted for different parallelization candidates and their various implementation variations, thus ensuring that the limited resources on an FPGA are utilized efficiently.

2.5 Loop pipelining

If loop pipelining is selected as a feasible parallelization candidate, then application speed-up from implementing this pipeline is estimated for the target FPGA. The final decision for the adaption of the parallelization candidate is made on basis of these estimates. Furthermore, optimal pipeline stages and initiation interval for the loop on a given target platform are calculated.

During the finding parallelism step, parallelization opportunities in sequential code are explored, i.e. feasible parallelization candidates are found. In the hardware optimization step, a cost benefit analysis determines if a parallelization candidate yields a significant speed-up for the application on a target platform and its fine-tuning parameters are calculated. In the final step, corresponding pragmas for these parallelization candidates are generated.

2.6 Array partitioning and interface design

Data-intensive applications are often constrained by communication and/or memory access bandwidth. To address these issues, HLS compilers support several array partitioning and interface design options but rely on developers to select the right options and specify them through pragmas. Different interface types and array partitioning implementations have different area, bandwidth and access characteristics. Therefore, making these design decisions involves complex trade-offs. Furthermore, since more parallelism requires more bandwidth, the choice of the right interface type or array partitioning is tightly coupled with the selected parallelization patterns for associated computational elements.

2.7 Function inlining

In HLS, C/C++ functions are normally synthesized as a separate hierarchy, i.e., all calling instances of these functions would use the same synthesized module, unless specified otherwise. Inlining a function forces its functionality to be synthesized in all calling functions. This has serious consequences for performance and area of the design. On one hand, non-inlined functions with multiple callers and/or those that are called from within parallelized loops often become performance bottlenecks. On the other hand, inlining functions with many instances could lead to a drastic increase in the area of the design. Therefore, making this decision manually is difficult. SLX optimization heuristics automatically inline functions where beneficial, considering a number of parameters such as calling frequency, degree of parallelization, and implementation area estimates.

2.8 Trip-counts

Trip-counts help the HLS compiler optimize loops and generate latency reports for loops. SLX uses profiling information from realistic workloads to calculate trip-counts and automatically annotates them to the target applications, saving the designer from going through this time consuming and error prone process.

2.9 Pragma Insertion

After SLX FPGA has determined the optimal pragmas that need to be inserted in the application, and their corresponding parameterization, a code generation process takes place. During this process, SLX automatically inserts the calculated annotations in the source code, in order to direct the HLS compiler towards the optimal solution. During this process, SLX presents a preview of the code that will be generated, side by side with the original source code, for the user to perform final tuning. Once the user is satisfied, SLX will generate the annotated code, create a Vivado project, and call the HLS compiler to obtain the actual synthesis results for the hardware IP block. This seamless integration enables the user to import the generated project inside Xilinx's tools and continue inside a familiar development flow.

3 Tool Evaluation Methodology

In order to evaluate SLX FPGA, we first define the standard workflows available. SLX FPGA supports primarily two workflows for optimizing an application. Table 1 shows the mapping of SLX FPGA workflows with corresponding features. Non-synthesizable applications require refactoring for HLS while for synthesizable applications we can skip SLX FPGA refactor step and start from SLX FPGA parallelism detection. The remaining steps are the same for both flows.

SLX FPGA Workflow	Refactor non-synthesizable code for HLS	Parallelism Detection	HW Optimization and HW/SW Partitioning	Pragma Insertion	Synthesize and Integrate
Non-Synthesizable	✓	✓	✓	✓	✓
Synthesizable	✗	✓	✓	✓	✓

Table 1: Application type categorization within SLX FPGA Workflow

Black Scholes and Heston Model implementations are both synthesizable, therefore SLX FPGA synthesizable workflow is chosen for optimization.

SLX FPGA is evaluated based on:

- Number of parallelism pragmas identified.
- Number arrays partitioned.
- Number of function inline pragmas added.
- Latency achieved compared to un-optimized version.
- Hardware utilized compared to un-optimized version.

Statistics computed using SLX FPGA optimized and un-optimized version of each implementation.

4 Monte Carlo Simulation for the Black-Scholes and Heston Models

Using SLX FPGA, we perform step by step analysis to reach optimized implementations for the Black-Scholes and the Heston models. In the original implementation, both models have similar function signatures with slightly different strategies for simulation. For conciseness, the Black Scholes Model (Asian Options) is considered for SLX FPGA step by step optimization, while synthesis result of both Black Scholes (Asian Options) and Heston Method are generated. The synthesis result includes clock cycle latency and utilization report for SLX FPGA optimized and un-optimized version of each model.

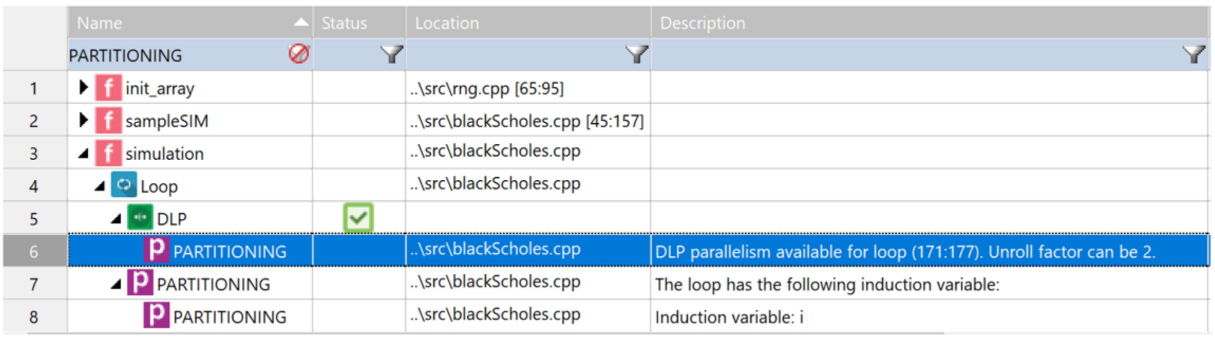
4.1 Step 1: Refactor non-synthesizable code for HLS

Prior to finding any parallelism within the application we identify synthesizable functions. Usually application functions are not synthesizable by default. SLX FPGA first identifies non synthesizable functions and then provides auto refactoring and guided refactoring to make these functions synthesizable.

As our examined Black Scholes implementation is already synthesizable, during this step SLX FPGA only catalogs information of synthesizable functions. Later this information is used in software and hardware (FPGA) partitioning.

4.2 Step 2: Parallelism Detection

SLX FPGA automatically finds data-level parallelism (DLP) and pipeline-level parallelism (PLP) within analyzed code. Further, information gathered during this step is later used to add HLS pragmas. Some potential DLPs or PLPs are ignored only due to write after read or read after write dependencies in loops. These loops can then be re-factored based on the hints provided to further increase parallelism coverage.

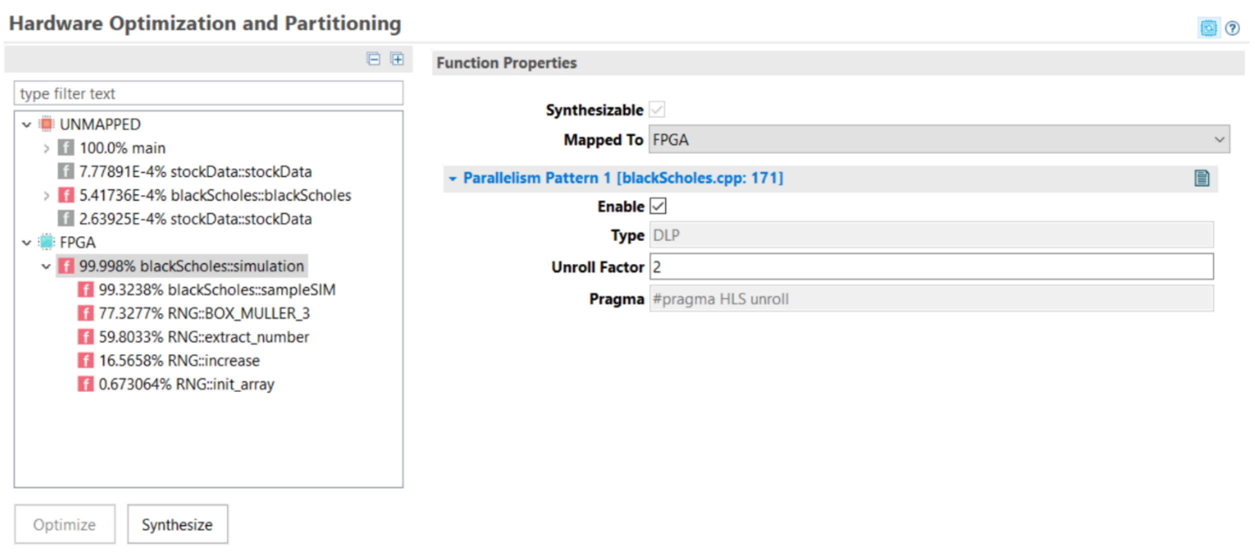


	Name	Status	Location	Description
	PARTITIONING			
1	init_array		..\src\rng.cpp [65:95]	
2	sampleSIM		..\src\blackScholes.cpp [45:157]	
3	simulation		..\src\blackScholes.cpp	
4	Loop		..\src\blackScholes.cpp	
5	DLP	✓		
6	PARTITIONING		..\src\blackScholes.cpp	DLP parallelism available for loop (171:177). Unroll factor can be 2.
7	PARTITIONING		..\src\blackScholes.cpp	The loop has the following induction variable:
8	PARTITIONING		..\src\blackScholes.cpp	Induction variable: i

Figure 3: Black Scholes (Asian option) find parallelism.

4.3 Step 3: HW Optimization and HW/SW Partitioning

Until now we have refactored functions for synthesizability and detected parallelism. During this step we partition functions into software and hardware. Hardware partitioned top-level functions along with functions in its call hierarchy are grouped under FPGA, which means that these functions are synthesizable and mapped to FPGA.



Hardware Optimization and Partitioning

type filter text

- UNMAPPED
  - 100.0% main
  - 7.77891E-4% stockData::stockData
  - 5.41736E-4% blackScholes::blackScholes
  - 2.63925E-4% stockData::stockData
- FPGA
  - 99.998% blackScholes::simulation
    - 99.3238% blackScholes::sampleSIM
    - 77.3277% RNG::BOX_MULLER_3
    - 59.8033% RNG::extract_number
    - 16.5658% RNG::increase
    - 0.673064% RNG::init_array

Function Properties

Synthesizable ✓
Mapped To: FPGA

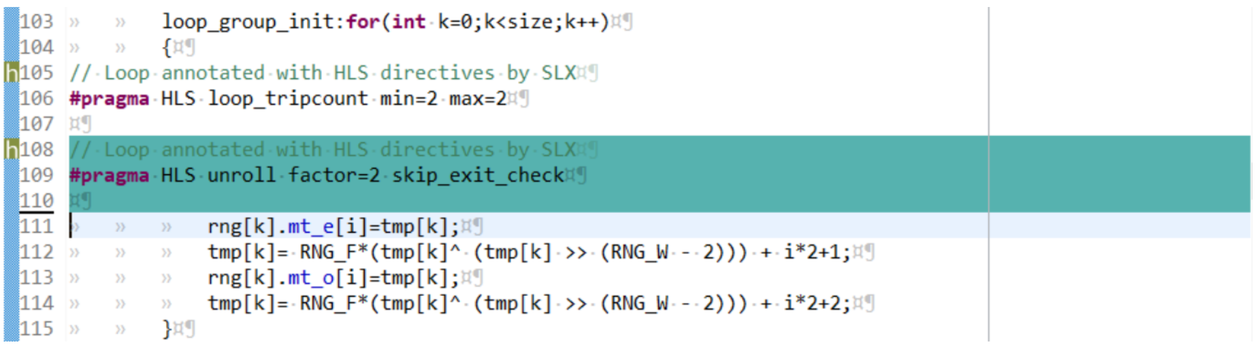
Parallelism Pattern 1 [blackScholes.cpp: 171]

Enable ✓
Type: DLP
Unroll Factor: 2
Pragma: #pragma HLS unroll

Figure 4: Black Scholes (Asian option) top level function simulation along with its call hierarchy mapped to FPGA.

4.4 Step 4: Pragma Insertion

SLX FPGA has partitioned the functions, so parallelism information already captured can be converted into HLS pragmas like array, unroll, or pipeline etc. Information gathered during previous steps help SLX FPGA understand the underlying application logic and then, based on heuristics, SLX FPGA apply best possible optimization using HLS pragmas.



```
		loop_group_init:for(int k=0;k<size;k++)
		{
// Loop annotated with HLS directives by SLX
#pragma HLS loop_tripcount min=2 max=2

// Loop annotated with HLS directives by SLX
#pragma HLS unroll factor=2 skip_exit_check

			rng[k].mt_e[i]=tmp[k];
			tmp[k]= RNG_F*(tmp[k]^ (tmp[k] >> (RNG_W - 2))) + i*2+1;
			rng[k].mt_o[i]=tmp[k];
			tmp[k]= RNG_F*(tmp[k]^ (tmp[k] >> (RNG_W - 2))) + i*2+2;
		}
```

Figure 5: HLS pragmas added to generate code in place of already identified Data Level Partition

Figure 5 shows HLS pragmas added to generated code in place of already identified data-level partitions (DLP) to optimize loops. This pragma parallelizes loop operation through data decomposition.

Table 2 contains the list of all pragmas that are added during pragma insertion. Original application sources are kept pristine by creating a copy specifically for pragma insertion. Engineers can select from the pragma generated, so they can run multiple synthesis simulation with different level of optimization to achieve best results.

Pragma	SLX FPGA
Unroll	9
Loop Pipeline	4
Interface	0
Array Partition	10
Function Inline	4
Total	27

Table 2: HLS pragmas usage report for SLX FPGA optimized Black Scholes (Asian Option).

4.5 Step 5: Synthesize and Integrate

Code is now optimized by SLX FPGA. This is the final step where SLX FPGA, using the low-level Xilinx Vivado HLS default flow, generates the IP block, ready to be added to Xilinx IP catalog. On completion of synthesis, a hardware call graph pops up showing hardware synthesized function in functional hierarchy. An HLS Simulation report is also generated on completion of synthesis; this report includes FPGA's minimum and maximum latency and resource usage.

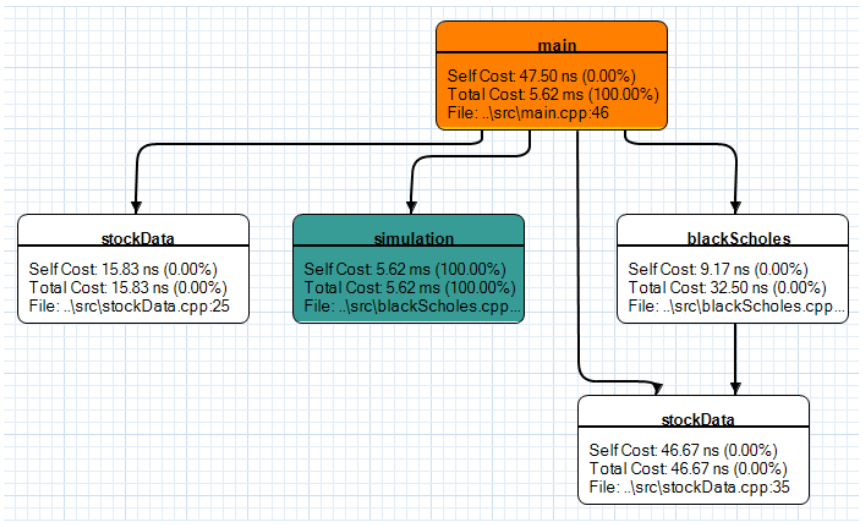

main
Self Cost 47.50 ns (0.00%)
Total Cost 5.62 ms (100.00%)
File: ..\src\main.cpp:46
stockData
Self Cost 15.83 ns (0.00%)
Total Cost 15.83 ns (0.00%)
File: ..\src\stockData.cpp:25
simulation
Self Cost 5.62 ms (100.00%)
Total Cost 5.62 ms (100.00%)
File: ..\src\blackScholes.cpp...
blackScholes
Self Cost 9.17 ns (0.00%)
Total Cost 32.50 ns (0.00%)
File: ..\src\blackScholes.cpp...
stockData
Self Cost 46.67 ns (0.00%)
Total Cost 46.67 ns (0.00%)
File: ..\src\stockData.cpp:35

Figure 6: Hardware call graph after synthesis.

5 Results

The results for each model are categorized into SLX FPGA optimized and un-optimized.

5.1 Black Scholes Model

For the Black Scholes model SLX FPGA optimized up to 29x compared to the un-optimized version. It is important to note that using SLX FPGA can achieve up to 29x without deep understanding of simulation logic.

Pragma	SLX FPGA
Unroll	9
Loop Pipeline	4
Interface	0
Array Partition	10
Function Inline	4
Total	27

Table 3: HLS pragmas usage report for Black Scholes SLX FPGA optimized.

	Latency (Clock Cycles)	Speed Up
No Pragmas	904312	0
SLX FPGA Auto Inserted HLS Pragmas	30683	29X

Table 4: Summary of latency results for Black Scholes (Asian Option).

	No HLS Pragmas	SLX FPGA Auto Inserted HLS Pragmas
BRAM_18K	8	20
DSP48E	76	84
FF	6415	45270
LUT	14563	86613
URAM	0	0

Table 5: Summary of resource usage for Black Scholes (Asian Option).

5.2 Heston Model

For the Heston model SLX FPGA optimized up to 25x compared to the un-optimized version

Pragma	SLX FPGA
Unroll	8
Loop Pipeline	3
Interface	0
Array Partition	7
Function Inline	4
Total	22

Table 6: HLS pragmas usage report for Heston SLX FPGA optimized and un-optimized.

	Latency (Clock Cycles)	Speed Up
No Pragmas	55729	0
SLX FPGA Auto Inserted HLS Pragmas	2249	25X

Table 7: Summary of latency results for Heston Model.

	No HLS Pragmas	SLX FPGA Auto Inserted HLS Pragmas
BRAM_18K	4	4
DSP48E	68	87
FF	9115	34389
LUT	16821	55074
URAM	0	0

Table 8: Summary of resource usage for Heston Model.

6 References

[1] Black, Fischer & Scholes, Myron S, 1973. "The Pricing of Options and Corporate Liabilities," Journal of Political Economy, University of Chicago Press, vol. 81(3), pages 637-654, May-June.

[2] Heston, S. L. (1993). A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options. The Review of Financial Studies, Volume 6, Issue 2, 327-343.

[3] Timothy G. Mattson (2004). Patterns of parallel programming. Addison-Wesley Professional; 1 edition (September 25, 2004)

[4] B.H.H. Juurlink and C. H. Meenderinck. 2012. Amdahl's law for predicting the future of multicores considered harmful. SIGARCH Comput. Archit. News 40, 2 (May 2012), 1-9. DOI=http://dx.doi.org/10.1145/2234336.2234338

[5] Glasserman, P. (2004). Monte Carlo methods in financial engineering. New York: Springer.

[6] Liang Ma, Fahad Bin Muslim, Luciano Lavagno(2016). High Performance and Low Power Monte Carlo Methods to Option Pricing Models via High Level Design and Synthesis. IEEE.

[7] Michael McCool, James Reinders, and Arch Robison. 2012. Structured Parallel Programming: Patterns for Efficient Computation (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

7 About the Author

Syed Shahrukh Hussain is Technical Marketing Engineer at Silexica GmbH. He previously worked for Mentor Graphics and u-blox in a variety of software development roles. Prior to that he worked as an independent consultant in embedded software and application development. He enjoys reading books and has a keen interest in computer architecture and parallel computing. He holds an MBA from University of Adelaide, Australia and a BSc (Hons) in Computing from Staffordshire University, UK.