Automated Application Acceleration using Software to Hardware Transformation

Qiwei Jin, David B. Thomas and Wayne Luk

Department of Computing, Imperial College London
United Kingdom
{qj04,dt10,wl}@doc.ic.ac.uk

***Abstract*—This paper describes an approach that allows applications to be developed in a software language, while taking advantage of hardware by facilities that automatically transform such software programs for hardware accelerators. A demonstration of this approach has been built for the C# language. Three case studies in numerical integration show that the automatically generated hardware accelerators can achieve similar speed-ups to manually optimised versions. In particular, the automatically generated accelerator running on an xc4vlx160 FPGA at 83MHz with single precision arithmetic can be more than 18 times faster and up to 143 times more power efficient than a Pentium 4 processor at 3.6GHz, while the double precision accelerator running at 64MHz is 7 times faster and 77 times more power efficient.**

I. Introduction

Field Programmable Gate Arrays (FPGAs) have applications in many areas such as digital signal processing, ASIC prototyping and computer vision. In recent years FPGAs have increasingly been used as a co-processor for high performance reconfigurable computing, where they are used to accelerate complicated numerical procedures such as finance applications. Although research has shown that FPGAs are suitable for accelerating many computationally intensive applications [1], there is limited actual deployment for high performance computing applications.

The main reason is that their time-consuming design process and error-prone debugging procedure do not meet the time-critical development requirements for many applications. Domain specific languages have been developed to allow easier utilisation of FPGA accelerators [2], however it is inconvenient for users to learn multiple domain specific languages in applications involving multiple standards, and new standards are constantly emerging. The problem can be solved if the domain specific languages can be unified to a common programming language such as C++ and C#, with domain specific transformations that automatically convert such software programs for hardware accelerators transparently without user's awareness.

Our research aims to simplify the use of FPGA co-processing for finance applications, by automatically transforming computationally intensive parts of software applications for hardware accelerators. Certain classes of computation such as numerical integration and monte-carlo simulation can be compiled into Directed Acyclic Graphs (DAGs) of functions, then high performance FPGA architectures can be produced based on the DAGs. All the user queries are initially executed as software cores on CPU, while certain software kernels will be translated into optimised hardware core in FPGA automatically if it is identified as a computational bottleneck.

The key contributions are:

- The architecture of a program transformation approach to generate pipelined hardware accelerators.
- A demonstration of the proposed approach by automatic transformation of C# programs in LINQ libraries to hardware implementations.
- A case study involving three numerical integration methods targeting financial applications.

II. Overview of the Approach

An important goal of our framework is to hide the FPGAs from application developers. The developers do not need to know whether the FPGA accelerator exists in the system; from their point of view the application simply runs significantly faster than the pure software implementation.

The framework can be used in different phases of a development process: accelerator development by hardware developers, application development by software developers, and application use by application users. Hardware developers develop active libraries for use by software developers; they define library API, and produce parameterisable hardware templates. Software developers do not program hardware; they develop and debug the application using standard programming language and software tools. Application users do not program at all; they use the application everyday. If the hardware developer can produce an accelerated library that is used by m software developers who develop one application each, and each application is used by n users, then we enable nm application users to benefit from the work of one hardware developer.

The components in the framework are shown in Figure 1. In particular:

- A user API which is defined by the hardware developer and used by the software developer.
- Domain specific libraries, along with the FPGA source generator used to generate different hardware and software implementations for different application domains automatically, to support the functionalities provided by user API.
- A performance profiler that determines if hardware acceleration is necessary.
- A bitfile library that allows hardware configurations to be stored for future use and shared between applications.

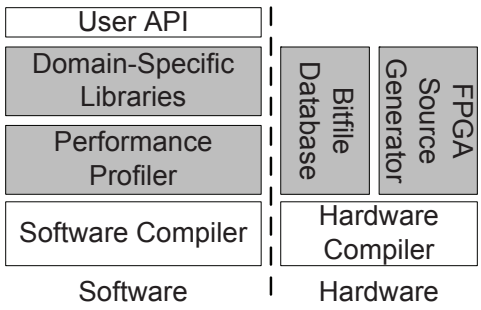

Fig. 1. Components of the framework.

```
class Integration{
  Integrate( IntegrationProvider prov,
   LambdaExpression expr,
   double low, double high,
   double var1....double varN );
};

double res1 =
Integration.Integrate( provider,
  (x, mean, std) =>
   Math.Exp(-Math.Pow((x - mean) / std, 2) / 2)/
   Math.Sqrt(2 * Math.PI),
  low, high, v_mean, v_std);
```

Fig. 2. The design and an example use of a possible integration API in C#.

- A software compiler which generates the software side communication logic to hardware.
- A hardware compiler that compiles the hardware design generated by the FPGA source generator.

The user API provides a high level abstraction for the user to utilise the FPGA accelerator; a detailed example to use the API is given later in this section. The domain-specific libraries are associated with the API to support different functions and applications. The user API allows computationally intensive software cores to be compiled into hardware accelerated versions automatically and transparently to the user.

The API is designed to support many types of finance modeling procedures, such as Quadrature method, Monte-Carlo method, Finite difference method, FFT method and Tree based Method. Tree based, FFT and finite difference methods can be supported by preparing parameterisable templates ahead of time. Quadrature and Monte-Carlo methods are parameterised with a user specified call-back function. The input call-back functions can vary significantly depending on the problem set, so preparing optimised FPGA implementations for each individual problem at compile time is not feasible. Our framework copes with this problem by including an FPGA source-code generator, to support automated generation of FPGA descriptions, parameterised by input functions defined by the user.

The performance profiler is used to check if it is necessary to accelerate the user query. The decision is made by checking the following aspects in order: the existence of a hardware accelerated version for the current query in the system; the estimated time for the query to run in software; the estimated time to run in hardware; the frequency of the query's occurrence.

Since user queries could recur, a bitfile database is used to store the FPGA bitfiles generated to aviod re-compiling. The size of one bitfile for a Virtex 4 xcvlx160 is 5MBytes, which means 1TBytes of storage space can store over 200K different bitfiles. In addition the bitfiles can be compressed to save storage space. A caching scheme could be used if fast configuration is required. If a query is worth accelerating and no existing bitfile can be found in the database, the execution of the query will be software based until the hardware accelerator is ready.

An example of the integration API in C# is illustrated in the top part of Figure 2. The integration function call takes in $N+4$ inputs. IntegrationProvider is a generalised abstract type for classes in the integration library to support different types of integrations. IntegrationProvider can be either in a software form or in a hardware accelerated form. LambdaExpression is a functional language in C#; in this case it is used to describe the function to be integrated. The two variables low and $high$ are used to define the range of the integration. The variables $var1$ to $varN$ are there to define the parameters for the function defined in the LambdaExpression. These variables are used as inputs to the LambdaExpression.

An example user experience of the API is shown in the bottom part Figure 2, which is the integration of the normal Cumulative Distribution Function (CDF) over the range low to $high$. The final result is assigned to the variable $res1$. This integration API requires six input variables. The $provider$ allocates computational resources for the integration. In this case, the provider decides the numerical method the integration will be using (i.e. Trapezoidal rule or Simpsons rule [3]) and the platform on which the integration will be running (i.e. software or hardware). The second variable, which is a lambda expression, describes the function to be integrated. In this example it is the three-input normal CDF where x is the integration variable, and $mean$ and std are the two input parameters. The last two parameters feed the values of $mean$ and std in the target function.

So far our work has focused on numerical integration procedures in order to facilitate applications based on quadrature method. In the next section we discuss the implementation of the framework in more detail.

III. Program Transformation

The implementation of the hardware acceleration framework is based on the C# programming language and Handel-C's HyperStreams library [1]. In particular, the user API, the domain-specific libraries and the performance profiler are all implemented in C#. The software compiler utilises the C# compiler library and the bitfile database uses the standard database class provided by C# to store the location of the FPGA configurations generated. The FPGA source generator uses the LINQ library in C# to generate the software and hardware components and the communication logic used between them. The communication logic in software is based on the Handel-C's DSM library which is originally a C-based library. A wrapper is used to allow execution of C code under the C# environment.

Integration Type	Equation
1. Integration with no analytical solution	$\int_a^b ((x \bmod 1)^3 - \frac{3}{2}(x \bmod 1)^2 + \frac{1}{2}(x \bmod 1))/x \; dx$
2. Normal Cumulative Distribution Function (CDF)	$\int_a^b exp(-(\frac{x-mean}{4\times std})^2)/\sqrt{2\pi} \; dx$
3. Quadrature method for European options	$A(y) \int_0^{N+\Delta x} B(y,x)V(x,t+\Delta t) \; dx$ *

TABLE I
EQUATIONS FOR CASE STUDIES ON INTEGRATION. (*REFER TO [3] FOR MORE DETAILS).

	Case Study 1			Case Study 2			Case Study 3		
	Virtex 4 xc4vlx160		Intel Core2 Duo	Virtex 4 xc4vlx160		Intel Core2 Duo	Virtex 4 xc4vlx160		Pentium 4
Number Format	single	double	double	single	double	double	single	double	double
Slices	8033 (11%)	14820 (21%)	-	19239 (28%)	53442 (79%)	-	24180 (35%)	62359 (92%)	-
FFs	7140 (5%)	11134 (8%)	-	15406 (11%)	44150 (32%)	-	20290 (15%)	48170 (35%)	-
LUTs	10545 (7%)	20160 (14%)	-	24609 (18%)	69057 (51%)	-	30924 (22%)	83745 (61%)	-
BRAMs	6 (2%)	7 (2%)	-	6 (2%)	9 (3%)	-	134 (46%)	265 (92%)	-
DSPs	15 (15%)	60 (62%)	-	30 (31%)	96 (100%)	-	39 (40%)	96 (100%)	-
Clock Rate	122MHz	89MHz	2.66GHz	94MHz	75MHz	2.66GHz	83MHz	64MHz	3.6GHz
Processing Speed (M values/sec)	122	89	6.25	94	75	5.42	83	64	9.1
Replication (cores/chip)	8	4	2	3	1	2	2	1	1
Acceleration (1 core)	19.5×	14.2×	1×	17.3×	13.8×	1×	9.1×	7×	1×
Acceleration (replicated cores)	156×	56.8×	2×	52×	13.8×	2×	18.5×	7×	1×
Max Power(Watt)	4.86	7.21	65	4.9	10.28	65	5.83	8.33	115
Energy Efficiency (M values/Joule)	25.1	12.3	0.1	19.2	7.3	0.1	14.3	7.68	0.1

TABLE II
DEVICE UTILISATION, SPEED UP AND ENERGY EFFICIENCY RESULTS FOR THREE CASE STUDIES.

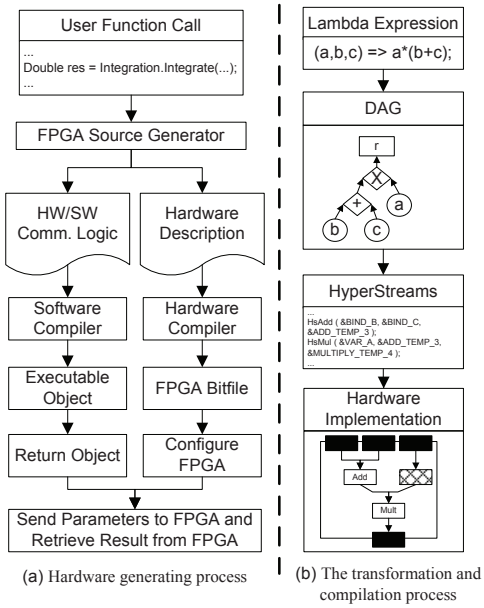

Fig. 3. Design flow of the software to hardware transformation.

The main task is to translate high level descriptions in Figure 2 into hardware implementations. To achieve this, we use "expression trees", which allow a standard C# expression to be captured as a lambda expression, and then to either execute it directly, or to examine and compile it for software or hardware execution. These expression trees map directly to DAGs, therefore they can be used to initiate the compilation of either a software implementation or a hardware accelerator. Figure 3(a) shows the compilation process to generate the hardware accelerator automatically.

The user function calls (queries) are fed into the FPGA source generator to generate both the hardware design and the communication logic between the hardware and the software. The hardware description will contain three major parts: (a) the pipelined arithmetic core, (b) the control logic to feed data into the arithmetic core, and (c) the communication logic to communicate to the software side. While (b) and (c) are fairly standard to be generated statically for different pricing procedures, (a) varies a lot and needs to be generated depending on the input function. A simplified procedure to generate the pipelined arithmetic core is shown in Figure 3(b).

IV. CASE STUDIES

The case studies of our hardware acceleration framework are based on three integration problems with no analytical solution, listed in Table I. The corresponding experimental result is shown in Table II. All the FPGA implementations are generated automatically using our framework based on the user query and are targeted to a Virtex 4 xc4vlx160 on a Celoxica RCHTX board. The power consumption of the FPGA implementations is estimated by the Xilinx XPower Estimator. In the first two case studies the reference PC is a 2.66GHz Core2 Duo processor with 4GB of RAM. The software implementation is implemented in the C# programming language and utilise one of the two CPU cores. In the third case study the reference PC is an Intel Pentium 4 CPU at 3.6GHz with 1GB of RAM, and the software implementation involves C code compiled with maximum speed optimisation options. This is to normalise our result to that reported in [3].

In case study 1, the result indicates that the single precision floating point implementation only utilises one tenth of the FPGA device. Performance can be improved by replicating the evaluation cores in the FPGA device. We can fit up to eight cores in the xc4vlx160 device to allow eight sampling points to be evaluated in parallel. For the double precision implementation we are able to replicate up to four cores in a single device. The size of a double precision implementation on the Virtex 4 FPGA is nearly two times larger than the size of the single precision version.

The acceleration results are estimated based on the assumption that the time taken for I/O between hardware and software is negligible. The assumption is valid as the data transfer rate between the RCHTX device and the PC is up to 3.2GBytes/second [4], and in this case we only transfer four floating point numbers between software and hardware.

It can be seen that one 32-bit single precision floating point FPGA core is 19.5 times faster than the software reference, while one 64-bit double precision core offers 14.2 times speedup. If we fully utilise the xc4vlx160 device by replicating cores, eight single-precision cores are estimated to be 156 times faster than the software reference. The replicated version of double precision implementation on the FPGA is expected to be 56.8 times faster than the software.

The energy efficiency result shows that a single-precision single core design on FPGA at 122MHz is 251 times more energy efficient than the Intel Core2 Duo. If the energy efficiency of double-precision arithmetic is considered, the FPGA implementation outperforms the PC by 123 times.

For case study 2, the result shows that there is enough resource on the xc4vlx160 device to replicate three single precision cores, while the double precision implementation is not replicatable due to resource limitation. The single core 32-bit single precision FPGA implementation offers a 17.3 times acceleration over the software reference, while the double precision version provides 13.8 times acceleration. If cores are replicated, the single precision version can provide an speed-up of up to 52 times. The FPGAs still outperform the Intel CPU by 192 and 73 times in energy efficiency for double and single implementations respectively.

In case study 3 we consider European option valuation using quadrature method. We generate FPGA accelerator implementations automatically using our framework and compare the results to the manually optimised implementations proposed by Tse et al [3].

Next we compare device utilisation. The automatically generated implementations are generally about 20% larger than the optimised version. The main reasons are the following: (a) the automatically generated version has extra logic to calculate the current value of the integration variable x by the iteration index and the lower bound of the integration, which requires one extra adder and multiplier pair; (b) an additional multiplier is used to cope with the different factors associated to each sampling node while applying the Trapezoid and Simpson's rules; (c) the automatically generated version utilises on-chip RAMs to store intermediate data, which leads to larger control logic in FPGA. On the other hand, the parameters transferred between hardware and software is $N+3$ floating point numbers. The amount of data transferred is halved compared to the manually optimised version as the computations of the integration variable x are now based in hardware. The system running at 83MHz requires an average bandwidth of 332MBytes/second, the bandwidth provided by the RCHTX board which is 3.2Gbytes/second is sufficient for this purpose. In addition the communication overhead can be further reduced by overlapping the data transferring and the calculation, if more than one option is being processed.

The highest clock frequencies we can get for the automatically generated implementations are about 20% slower compared to the optimised versions. This is because more complex pipelines, deeper control logic and the utilisation of on-chip RAMs (in our case xilinx block RAMs) to store intermediate values have negative effects over the timing constraints during placement and routing. However the processing speed of the single core implementations are similar (within a factor of 1.2). It can also be seen that although the automatically generated implementations consume more power than the manually optimised versions due to higher device utilisation, the differences are within a factor of 1.6.

V. Conclusion

This paper describes the design and implementation of a framework for automatic hardware acceleration of finance applications. Based on high level user function calls in software, the framework is able to generate dedicated hardware accelerators automatically. The results show that the automatically generated hardware accelerators can achieve approximately the same speed-ups compared to manually optimised versions reported in [3] to solve European option pricing problem by quadrature method. In particular, our automatically generated hardware accelerators in an xc4vlx160 FPGA can generally run more than 18 times faster and can be up to 143 times more power efficient than a Pentium 4 processor in single precision arithmetic, and 7 times faster and 77 times more power efficient in double precision arithmetic.

Future work includes developing a fully automated tool chain for an extended set of applications, and exploring how pre-compilation, efficient bitfile caching and run-time reconfiguration can be used to reduce the compilation overhead in our approach and allow software to hardware transformation at run time.

Acknowledgement. The support of J.P. Morgan Securities Limited is gratefully acknowledged.

References

[1] G. Morris and M. Aubury, "Design space exploration of the European option benchmark using Hyperstreams," in *Proc. Int. Conf. on Field Programmable Logic and Applications*, 2007, pp. 5–10.

[2] D. Thomas, J. Bower, and W. Luk, "Automatic generation and optimisation of reconfigurable financial Monte-Carlo simulations," in *Proc. Int. Conf. on Application-Specific Systems, Architectures and Processors*. IEEE, 2007.

[3] A. H. Tse, D. B. Thomas, and W. Luk, "Accelerating quadrature methods for option valuation," in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, 2009.

[4] Celoxica, "RCHTX-XV4 datasheet."