Accelerating Financial Computation

Wayne Luk

Department of Computing
Imperial College London

HPC Finance Conference and Training Event
Computational Methods and Technologies for Finance

13 May 2013
Accelerated System Architecture

- accelerators
  - multiple functions
  - in clouds

- common types
  - FPGA
  - GPU
  - mixed
Acceleration@imperial

• security: Elliptic Curve Encryption
  – 35MHz XC2V6000: \textbf{1150x} 2.6GHz Xeon processor

• bio-informatics: canonical labelling
  – xc4vlx60: up to \textbf{400x} 2.2 GHz Quad-Opteron

• combinatorial optimisation: tabu search for TSPLIB
  – 1.15GHz C2050: \textbf{112x} 2.67GHz Xeon X5650 12-cores

• medical imaging: 3D image registration
  – 412MHz XC5VLX330: \textbf{108x} 2.5GHz Quad-Xeon

• financial: Monte Carlo credit risk modelling
  – 233MHz XC4VSX55: \textbf{60-100x} 2.4GHz Quad-Xeon
Why Accelerators?

• features
  – parallelism: many heterogeneous cores
  – customisable: operation and data, e.g. precision

• benefits: improve over CPU based systems
  – speed
  – latency
  – size
  – power
  – energy
  – cost
Challenges

• maximise efficiency: best trade-offs in:
  – speed
  – size
  – power and energy
  – ...

• maximise productivity
  – high-level description
  – support users + experts
  – facilitate design re-use
  – ...
Customisation Example

1. Monte Carlo framework
   – HJM based interest rate derivatives payoff evaluations
   – 3 levels of functional specialisations

2. Specialisation
   – Domain-Specific Language: specialise for applications
   – optimise data-width on FPGA

3. Evaluation
   – 1.36 times faster than GPU
   – 3 times more energy efficient than GPU

Joint work with Qiwei Jin, Diwei Dong, Anson Tse, Gary Chow, David Thomas, and Stephen Weston
Background

• Monte Carlo Method
  – useful numerical technique
  – used for options with no closed-form solution
  – easily parallelisable
  – time-consuming to obtain accurate result

• FPGA: natural fit for Monte Carlo simulations
  – deep pipelining
  – customisable data-width
  – low power consumption
  – efficient random number generating
Concerns

• FPGA
  – complexity in mapping algorithm to hardware
  – adversarial to change if design is optimised

• real-world Monte Carlo applications
  – complex control logic
  – prone to change
  – short deadline for delivery

• financial interest rate derivatives
  – payoff evaluation: family of interest rate curves
  – bespoke products: different payoff, continuously emerging
  – Monte Carlo: can be the only feasible way of valuation
Heath-Jarrow-Morton

- Heath-Jarrow-Morton (HJM) framework
  - general mathematical model
  - models instantaneous forward interest rate curve

- mathematical description

\[
df(t, T) = \mu(t, T) dt + \sigma(t, T)^T dW(t)
\]

\[
\mu(t, T) = \sigma(t, T)^T \int_t^T \sigma(t, u) du
\]

- \(f(t, T)\): instantaneous forward rate at time \(T\) as seen from time \(t\)
- \(\sigma(t, T)\): forward volatility column vector of size \(d\) (no. of factors)
- \(W(t)\): d-dimensional standard random process
Forward Curve Dynamics

\[ f(0,T), \ 0 \leq T \leq 8 \]
Forward Curve Dynamics

$f(1,T), \ 1 \leq T \leq 8$
Forward Curve Dynamics

\[ f(1, T), 1 \leq T \leq 8 \]
Forward Curve Dynamics

\[ f(2,T), \ 2 \leq T \leq 8 \]

Random displacement
HJM Monte Carlo: Single Path

**Input:** \( f(0, T) = \text{initial forward curve}, \sigma = \text{volatility model} \)

**Output:** \( f(t, T) = \text{forward surface} \)

1: \( \text{for } t=0 \text{ to } t_{\text{max}} \text{ do} \)
2: \( \text{for } T'=0 \text{ to } T'_{\text{max}} \text{ do} \)
3: \( \text{Calculate Drift:} \)
   \( \text{obtain } \sigma(t, T) \text{ and calculate } \mu(t-\delta t, t+T') \)
4: \( \text{Update forward Surface: get } f(t, t+T') \)
5: \( \text{Price Derivative State 1:} \)
   \( \text{Use } f(t, t+T') \text{ to price the target derivative} \)
6: \( \text{end for} \)
7: \( \text{Price Derivative State 2:} \)
   \( \text{Use result from State 1 to price the target derivative} \)
8: \( \text{end for} \)
HJM Monte Carlo: Single Path

**Input:** \( f(0, T) = \) initial forward curve, \( \sigma \) is volatility model

**Output:** \( f(t, T) = \) forward surface

1: for \( t=0 \) to \( t_{\text{max}} \) do
2: for \( T'=0 \) to \( T'_{\text{max}} \) do
3: **Calculate Drift:**
   - obtain \( \sigma(t, T) \) and calculate \( \mu(t-\epsilon t, t+T') \)
4: **Update forward Surface:** get \( f(t, t+T') \)
5: **Price Derivative State 1:**
   - Use \( f(t, t+T') \) to price the target derivative
6: end for
7: **Price Derivative State 2:**
   - Use result from State 1 to price the target derivative
8: end for
1. Multi-level Customisation

• efficiency: two phases in development
  – model developing phase
  – payoff evaluator developing phase

• productivity: two types of developers
  – platform experts: expertise in target platform
  – platform users: expertise in applications

• 3 levels of modular functional specialisations
  – Heavy, Medium, Light
Specialisations

• Heavy
  – stable modules: highly optimised, platform dependent
  – require detailed knowledge of platform, done by experts

• Medium
  – semi-stable modules: optimised, platform dependent
  – limited variations: specified by users ahead of time
  – building blocks: in payoff evaluator developing phase

• Light
  – volatile modules: still under development
  – ease of use: domain specific languages
  – may involve platform dependent configuration files
Customisation: Two Phases

Model development phase

1. Experts develop heavily specialised modules
2. Experts and users define templates for mediumly specialised modules
3. Experts optimise the modules for potential target platform

payoff evaluator development phase

4. Users choose a mediumly specialised module as a base component and a target platform
5. Users using a platform independent domain specific language to generate payoff evaluators
Multi-level Customisation for HJM

Interest Rate Engine
  - Interest Rate Generator (Hand Optimised)
  - Volatility Logic (From Template)

Payoff Evaluation Logic (Programmed by User)

HJM Payoff Evaluation Kernel

Parameters
  - From CPU

Results to CPU

Parallel Kernels

Heavily specialised module
  - By expert

Mediumly specialised module
  - By expert

Lightly specialised module
  - By user

Prone to change
Customise: volatility + payoff evaluation

<table>
<thead>
<tr>
<th>Volatility Structure</th>
<th>$\sigma(t, T)$</th>
<th>$\mu(t, T)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant$^1$</td>
<td>$\alpha$</td>
<td>$\frac{1}{2} \alpha^2 [T^2 - (T - t)^2]$</td>
</tr>
<tr>
<td>Exponential$^1$</td>
<td>$\alpha e^{-\beta(T-t)}$</td>
<td>$\frac{\alpha^2}{\beta} \left( e^{-2\beta(T-t)} - e^{-\beta(T-t)} \right)$</td>
</tr>
<tr>
<td>Stochastic$^2$</td>
<td>$\tilde{\sigma}(t, T)f(t, T)$</td>
<td>$\sigma(t, T) \int_t^T \sigma(t, u)du$</td>
</tr>
</tbody>
</table>

$^1$ $\alpha$ and $\beta$ are calibrated model constants  
$^2$ $\tilde{\sigma}$ is a stochastic volatility process

From Template: max re-use

<table>
<thead>
<tr>
<th>Target Instrument</th>
<th>Payoff Evaluation Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bond</td>
<td>$B(t, T) = \exp \left( - \int_t^T f(t, u)du \right)$</td>
</tr>
<tr>
<td>Bond Option$^1$</td>
<td>$(B(t, T) - K)^+$</td>
</tr>
<tr>
<td>CMS$^2$</td>
<td>$Y(t, T) = \frac{1-B(t,T)}{\sum_{a=t}^T B(t,a)}$</td>
</tr>
<tr>
<td>Swaption</td>
<td>$(Y(t, T) - K)^+ \sum_{a=t}^T B(t, a)$</td>
</tr>
<tr>
<td>CMS S.O.$^3$</td>
<td>$(Y(t, T_1) - Y(t, T_2) - K)^+$</td>
</tr>
</tbody>
</table>

$^1$ $(x)^+ \equiv \max(0, x)$  
$^2$ Constant Maturity Swap  
$^3$ CMS Spread Option

In C-based domain-specific language
Workflow: Experts + Users

1. Volatility, Accuracy & Parallelisation Configuration
   - 2. Volatility Templates
   - 3. Interest Rate Generator Template
   - 4. User Programming File
   - 5. Glue Logic

Change Infrequently
- Interest Rate Engine Generator
- Payoff Evaluation Logic Compiler
- Payoff Evaluation Logic Description
- HJM Kernel Generator

By expert

Change frequently
- Application Specific Programming File for Target Platform Tool Chain
- 6. HJM Kernel Description

By user
2. Application Specialisation Flow

• domain specific programming environment
  – to specialise the framework to particular application

• data-width optimisation
  – to find the optimal data format
  – ensures good performance on FPGA
  – while retaining result accuracy
Domain Specific Programming

• “C” style and control-based
• provides environment parameters per iteration
• operator latency is implicit
• platform user
  – create input/output variables
  – create intermediate variables
  – defines payoff evaluation logic
Present value calculator for a Zero Coupon Bond $B(t_{I_{\text{max}}}, t+T_{J_{\text{max}}})$
Data-Width Optimisation: Errors

• results from numerical techniques
  – discretisation error
  – finite precision error

• discretisation error
  – intrinsic

• finite precision error
  – increases as data-width decreases
Data-Width Optimisation

- data-width reduction: improve FPGA performance
Data-Width Optimisation

• problem: determine optimal data-width
  – preserve result accuracy
  – consume minimal FPGA resources

• Welch’s t-test
  – assess statistical significance of finite precision error
  – compare reduced precision and full precision
Welch’s t-test: Optimised Data-Width

Number of mantissa bits:

p-value in log scale for Swaption
3. Results

- MaxWorkstation: Xilinx Virtex-6 SX475T FPGA
- 4-Core Intel i7-870 CPU, 2.93GHz
- 448-Core NVIDIA Tesla C2070 GPU, 1.15GHz

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>FPGA</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler</td>
<td>Intel</td>
<td>Max Compiler</td>
<td>nvcc</td>
</tr>
<tr>
<td>Native Language</td>
<td>C++</td>
<td>MaxJ</td>
<td>CUDA</td>
</tr>
</tbody>
</table>
Resource Use: Optimised Data-Width

% Resource Consumption

- **Bond Option**: Wf=53 LUT: 6%, Wf=17 LUT: 3%, Wf=53 BRAM: 2%
- **Swaption**: Wf=53 LUT: 8%, Wf=17 LUT: 4%, Wf=53 BRAM: 2%
- **CMS Spread Option**: Wf=53 LUT: 12%, Wf=17 LUT: 5%, Wf=17 BRAM: 2%

Wf: number of mantissa bits
Speed Up

Speed up over single core software implementation

<table>
<thead>
<tr>
<th></th>
<th>Bond Option</th>
<th>Swaption</th>
<th>CSM Spread Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-Core CPU</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>FPGA</td>
<td>44,8</td>
<td>42,4</td>
<td>39,2</td>
</tr>
<tr>
<td>GPU</td>
<td>32,8</td>
<td>30,04</td>
<td>27,1</td>
</tr>
</tbody>
</table>
Power Consumption

Power Consumption for Different Implementations, using Power Measuring Socket from Olson Electronics

<table>
<thead>
<tr>
<th>Bond Option</th>
<th>Swaption</th>
<th>CSM Spread Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>183 (4-Core CPU)</td>
<td>184 (FPGA)</td>
<td>184 (GPU)</td>
</tr>
<tr>
<td>87 (FPGA)</td>
<td>87 (FPGA)</td>
<td>85 (GPU)</td>
</tr>
<tr>
<td>240 (GPU)</td>
<td>238 (GPU)</td>
<td>240 (GPU)</td>
</tr>
</tbody>
</table>
Current Work

• extend framework to support more
  – platforms, e.g. those with multiple accelerator types
  – volatility structures, payoff evaluation functions
  – financial, risk and other applications

• improve performance + energy efficiency
  – mixed precision
  – more automation
  – run-time reconfiguration
Why Reconfigurability

• growing fabrication cost
• time-share large design
• accelerate demanding applications
• potential for low power/energy consumption
• support health monitoring
• enhance reliability + fault tolerance
• speed up design cycle: incremental development
Why Reconfigurability

- growing fabrication cost
- time-share large design
- accelerate demanding applications
- potential for low power/energy consumption
- support health monitoring
- enhance reliability + fault tolerance
- speed up design cycle: incremental development
Run-time Reconfigurability

- multiple reconfigurations
  - interleave or concurrent with data processing
- mixed precision computation
  - low precision: maximise parallelism
  - high precision: improve accuracy
- multi-stage computation: multiple precisions
  - high precision: fewer iteration, each takes longer
- eliminate idle functions
  - active functions in same configuration
Recent Results: MAX3 Accelerator

- finance: pricing Asian options
  - 44.6x speed, 40.7x energy efficiency of quadcore i7-870
  - 4.6x speed, 5.5x energy efficiency of C2070 GPU

- seismic imaging: reverse time migration
  - 103x speed, 145x energy efficiency of quadcore i7-870
  - 2.5x speed, 10.2x energy efficiency of GTX280 GPU

- biomedical: genetic sequence matching
  - 293x speed of Xeon X5650 with 20 threads
  - 134x speed of NVIDIA GTX 580 GPU
Current and Future Research

• functional and performance models
  – correctness + performance: generalise reconfigurability

• aspect-oriented design: software + hardware
  – multi-source e.g. OpenCL, design re-use, portability

• machine learning: smarter systems
  – adapt to application and device behaviour at run time
Summary

• accelerators: becoming main-stream
  – Improving speed, latency, size, power, energy, ...

• key challenges
  – best trade-offs in efficiency and productivity

• compilation, verification, performance analysis
  – models, machine learning, run-time reconfigurability