HIPERFIT – Contract Languages, Functional Programming, and Parallel Computing

Presentation at CFA meeting, Denmark

April 16, 2013

Martin Elsman
Associate Professor, PhD
HIPERFIT Research Center Manager
Department of Computer Science
University of Copenhagen

The HIPERFIT Research Center

Funded by the Danish Council for Strategic Research (DSF) in cooperation with financial industry partners:

Department of Computer Science

Department of Mathematical Sciences

Niels Bohr Institutet

Danske Bank

LexiFi

Nordea

Nykredit

SimCorp

HIPERFIT: High Performance Computing for Financial IT.

Six years lifespan: 1.1.2011 – 31.12.2016.

Funding volume: 5.8M EUR.

78% funding from DSF, 22% from partners and university.

6 PhD + 3 post-doctoral positions (CS and Mathematics).

Additional funding for collaboration with small/middle-sized businesses.

Motivation – examples

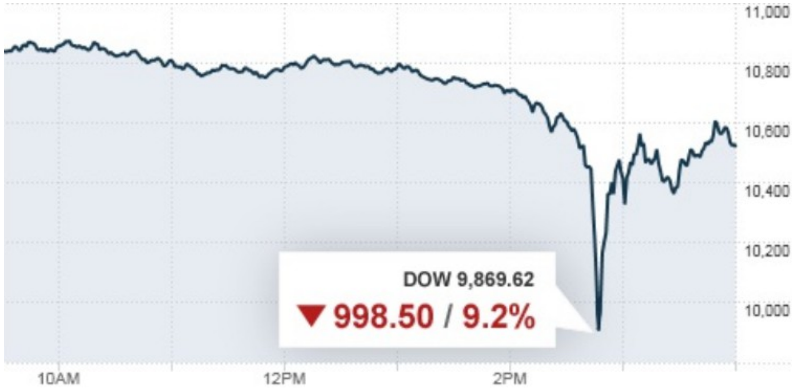
Ex 1: The Credit Crunch…

Worldwide recession in 2008:
US house market collapse.

Rating ignored interdependencies and accumulated failure.

Ex 2: The Flash Crash…

Dow Jones Index on May 6, 2010:



Almost 10% drop within minutes.

Drop almost recovered minutes later.

Systemic effect of algorithmic trading at high volume and frequency.

Performance, Transparency, Expressiveness

1. Accuracy of models
   - *reliable results, auditing*

2. Performance of computations
   - *quick reactions, handling large data*

3. Ease of development and maintenance
   - *agile, rapid and reliable development*

Claim:

Integrate solutions to achieve all three.

Principle: "**Less** is **More**"

Performance:
Compute **more** **faster**!

Apply domain-specific methods for parallel hardware.

Capture domain-specific parallelism in DSLs.

Productivity:
Express **more** with **fewer** lines!

Write high-level specifications, not low-level code.

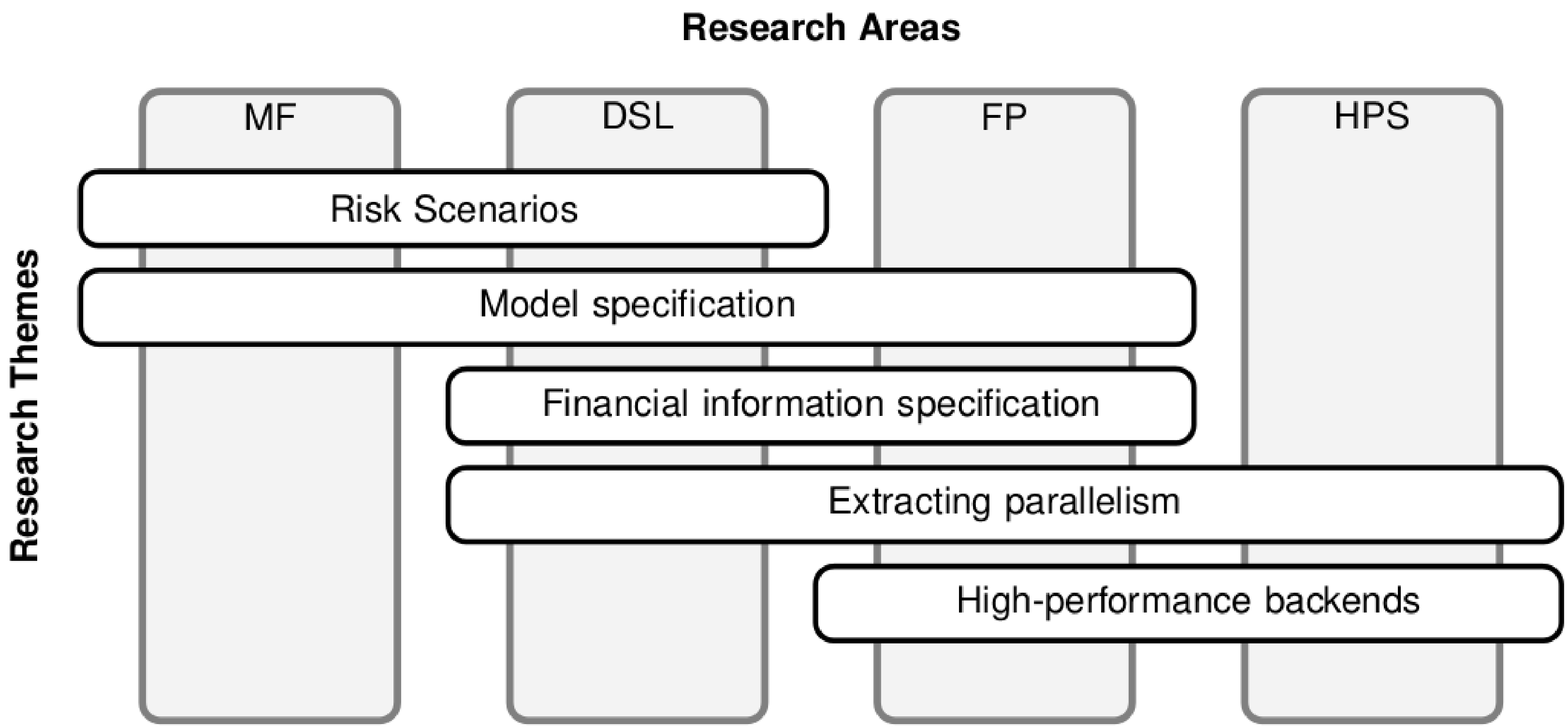
Transparency:
Understand **more** from **shorter** code!

Understand the computation as a mathematical formula with clear semantics.

Trick:
Skip the indirection of imperative software architecture.

Do not built upon sequentialized inherently parallel operations!!

Research Themes and Areas in HIPERFIT



Mathematical Finance	Functional Programming
Domain Specific Languages	High-Performance Systems

Selected HIPERFIT Projects

Financial Contract Specification (DIKU, IMF, Nordea)
Use declarative combinators for specifying and analyzing financial contracts.

Automatic Parallelization of Loop Structures (DIKU)
Outperform commercial compilers on a large number of benchmarks by parallelizing and optimizing imperative loop structures.

Streaming Semantics for Nested Data Parallelism (DIKU)
Reduce space complexity of "embarrassingly parallel" functional computations by streaming.

Automatic Parallelization of Financial Applications (DIKU, LexiFi)
Analyze real-world financial kernels, such as exotic option pricing, and parallelize them to run on GPGPUs.

Optimal Decisions in Household Finance (IMF, Nykredit, FinE)
Investigate and develop quantitative methods to solve individual household's financial decision problems.

Key-Ratios by Automatic Differentiation (DIKU)
Use automatic differentiation to derive sensibilities to market changes for financial contracts.

CVA (IMF, DIKU, Nordea)
Parallelize calculation of exposure to counterparty credit risk.

Bohrium (NBI)
Collect and optimize bytecode instructions at runtime and thereby efficiently execute vectorized applications independent of programming language and platform.

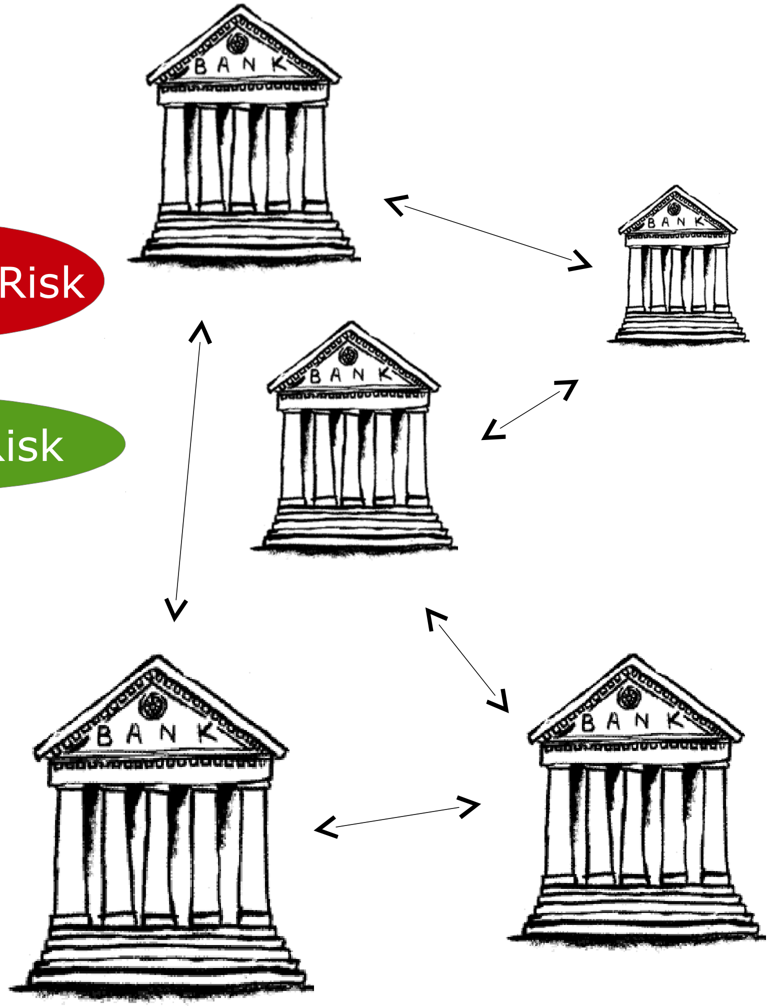
APL Compilation (DIKU, Insight Systems, SimCorp)
Develop techniques for compiling arrays, specifically a subset of APL, to run efficiently on GPGPUs and multicore-processors.

Project: Financial Contract Specification

Banks (and other financial institutions) use financial contracts for both

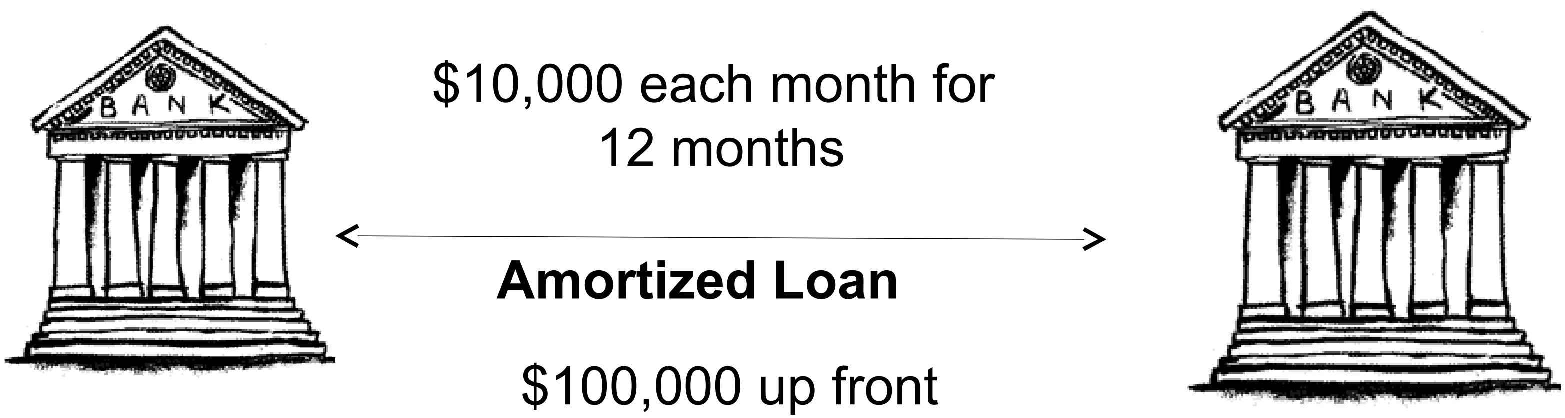
- Speculation — Increase Risk
- Insurance (hedging) — Decrease Risk

Many contracts are ”**Over The Counter**” (OTC) contracts, which are negotiated agreements between a bank and another bank (its counter party).



The Term Sheet – the financial contract

- A financial contract is typically agreed upon on a so-called ”Term Sheet”.
- The *term sheet* specifies the financial flows (amounts, dates, etc.) and under which *conditions* a flow should happen.
- Flows can go in both directions.



- A ***derivative*** is a contract that depends on an underlying entity (e.g., a stock)

Many Types of Contracts are Traded

- FX Swaps
- Catastrophy Bonds
- Bermudan Options
- Inflation Swaps
- Interest Rate Swaps
- Variance Certificates
- Plain Vanilla Options
- Indexed Linked Bonds
- Credit Default Swaps (CDS)
- Rainbow Options
- Chooser Options
- Himalyan Options
- FX Options
- Forward Rate Agreements
- Callable Range Accrual Notes

How do Banks and Society Keep Track?

Algebraic Properties, simple reasoning

Many Problems

- Financial contracts need management
  fixings, ***decisions***, ***corporate actions***, …
- Banks must **report daily** on their total value of assets
- Banks must **control risk** (counterparty risk, currency risk, …)
- Banks need to know about future cash flows, …

A Solution:

- Specify financial contracts in a **domain specific language**
- Use a **functional programming language** (e.g., ML)

Financial industry has already recognized the value of FP:

- LexiFi (ICFP'00 paper by Peyton-Jones, Eber, Seward)
- SimCorp A/S (uses LexiFi technology)
- Jane Street Capital (focus on electronic trading)
- Societe Generale, Credit Suisse, Standard Chartered
- Contract ”Pay-off” specifications are often functional in style

Constructing Contract Management Software in Standard ML

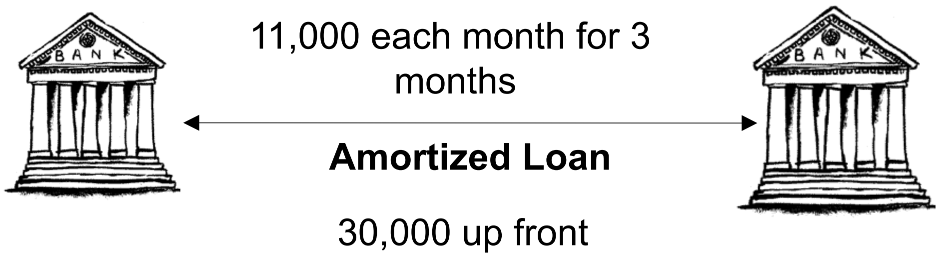
Basics:

```
(* Currencies *)
datatype currency = EUR | DKK

(* Observables *)
datatype obs =
           Const of real
         | Underlying of string * Date.date
         | Mul of obs * obs
         | Add of obs * obs
         | Sub of obs * obs
         | Max of obs * obs
```

The Contract Language as a Standard ML Datatype

```
(* Contracts *)
datatype contract =
         One of currency
       | Scale of obs * contract
       | All of contract list
       | Acquire of Date.date * contract
       | Give of contract

(* Shorthand notation *)
fun flow(d,v,c) = Acquire(d,Scale(Const v,One c))
val zero = All []
```

Example Financial Contracts in an Embedded DSL




```
(* Simple amortized loan *)
val ex1 =
    let val coupon = 11000.0
        val principal = 30000.0
    in All [Give(flow(?"2011-01-01",principal,EUR)),
            flow(?"2011-02-01",coupon,EUR),
            flow(?"2011-03-01",coupon,EUR),
            flow(?"2011-04-01",coupon,EUR)]
    end

(* Cross currency swap *)
val ex2 =
    All [Give(
           All[flow(?"2011-01-01",7000.0,DKK),
               flow(?"2011-02-01",7000.0,DKK),
               flow(?"2011-03-01",7000.0,DKK)]),
         flow(?"2011-01-01",1000.0,EUR),
         flow(?"2011-02-01",1000.0,EUR),
         flow(?"2011-03-01",1000.0,EUR)]
```

Notice: flows in Different currencies

A More Complex Example...

```
(* Call option on "Carlsberg" stock *)
val equity = "Carlsberg"
val maturity = ?"2012-01-01"
val ex4 =
    let val strike = 50.0
        val nominal = 1000.0
        val obs =
            Max(Const 0.0,
                Sub(Underlying(equity,maturity),
                    Const strike))
    in Scale(Const nominal,
             Acquire(maturity,Scale(obs,One EUR)))
    end
```

Meaning: Acquire at maturity the amount (in EUR), calculated as follows (*P* is price of Carlsberg stock at maturity):

nominal * max(0, *P* – *strike*)

What can we do with the contract definitions?

- Report on the **expected future cash flows**
- Perform **management operations**:
  - Advancement (simplify contract when time evolves)
  - Corporate action (stock splits, merges, catastrophic events, …)
  - Perform fixing (simplify contract when an underlying becomes known)
- Report on the **value (price) of a contract**
- …

Expected Future Cash Flows

```
(* Future cash flows *)
fun noObs _ = raise Fail "noObs"
val _ = println "\nEx1 - Cash flows for simple amortized loan:"
val _ = println (cashflows noObs ex1)

Ex1 - Cash flows for simple amortized loan:
2011-01-01 Certain   EUR ~30000.0000000
2011-02-01 Certain   EUR 11000.0000000
2011-03-01 Certain   EUR 11000.0000000
2011-04-01 Certain   EUR 11000.0000000

val _ = println "\nEx2 - Cash flows for cross-currency swap:"
val _ = println (cashflows noObs ex2)

Ex2 - Cash flows for cross-currency swap:
2011-01-01 Certain   EUR 1000.00000000
2011-01-01 Certain   DKK ~7000.00000000
2011-02-01 Certain   EUR 1000.00000000
2011-02-01 Certain   DKK ~7000.00000000
2011-03-01 Certain   EUR 1000.00000000
2011-03-01 Certain   DKK ~7000.00000000
```

Contract Management and Contract Simplification

```
(* Stock option cash flows assuming underlying stock price of 79.0 *)
val _ = println "\nEx4 - Cash flows on stock option (Strike:50,Price:79):"
val _ = println (cashflows (fn _ => Const 79.0) ex4)

(* Contract management *)
val ex5 = fixing(equity,maturity,83.0) ex4
val _ = println "\nEx5 - Call option with fixing 83"
val _ = println ("ex5 = " ^ pp ex5)
val ex6 = fixing(equity,maturity,46.0) ex4
val _ = println "\nEx6 - Call option with fixing 46"
val _ = println ("ex6 = " ^ pp ex6)
```

Output:

```
Ex4 - Cash flows on stock option (Strike:50,Price:79):
2012-01-01 Uncertain EUR 29000.0000000

Ex5 - Call option with fixing 83
ex5 = Scale(33000.0000000,One(EUR))

Ex6 - Call option with fixing 46
ex6 = zero
```

Simple Contract Valuation (pricing)

```
(* Valuation (Pricing) *)
structure FlatRate = struct
  fun discount d0 d amount rate =
      let val time = real(Date.diff d d0) / 360.0
      in amount * Math.exp(~ rate * time)
      end
  fun price d0 (R : currency -> real)
               (FX: currency * real -> real) t =
      let val flows = cashflows0 noE t
      in List.foldl (fn ((d,cur,v,_),acc) =>
                         acc + FX(cur,discount d0 d v (R cur)))
                     0.0 flows
      end
end


fun FX(EUR,v) = 7.0 * v
  | FX(DKK,v) = v
fun R EUR = 0.04
  | R DKK = 0.05

val p1 = FlatRate.price (?"2011-01-01") R FX ex1
val p2 = FlatRate.price (?"2011-01-01") R FX ex2
val _ = println("\nPrice(ex1) : DKK " ^ Real.toString p1)
val _ = println("\nPrice(ex2) : DKK " ^ Real.toString p2)
```

Doesn't know how to Deal with underlyings!

Output:

```
Price(ex1) : DKK 19465.9718165

Price(ex2) : DKK 17.3909947790
```

Stochastic Models

General stochastic Black-Scholes models for pricing may be implemented using Monte-Carlo Simulation.

We would like to utilize parallel hardware but avoid writing the models in low-level CUDA or OpenCL!

Is there a DSL for writing models?

- Combinators for manipulating curves...

Other tricks:

- Use Automatic Differentiation (AD) techniques to get the “Greeks” for free without using expensive finite difference methods (FINCAD F3).

Project: Compiling APL

APL is in essence a functional language

APL has arrays as its primary data structure

APL "requires a special keyboard"!

> APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums.
>
> Edsger Dijkstra

Examples:

```
a ← ⍳ 8            ⍝ array [1..8]
b ← +/ a           ⍝ sum of elements in a
f ← {2+ω×ω}        ⍝ function x^2 + 2
c ← +/ f ¨a        ⍝ apply f to all elements of a and sum the elements
```

Reduce

Map

Compiling APL – An Example

APL Code:

```
diff ← {1↓ω−¯1⌽ω}
signal ← {¯50⌈50⌊50×(diff 0,ω)÷0.01+ω}
+/ signal ι 100000
```

Notice: The APL Compiler has removed all notions of arrays!

Generated C Code:

```
double kernel(int n14) {
  double d13 = 0.0;
  for (int n82 = 0; n82 < 100000; n82++) {
    d13 = (max(-50.0,min(50.0,(50.0*(i2d(((1+n82)-((n82<1) ? 0 : n82)))/
              (0.01+i2d((1+n82)))))))+d13);
  }
  return d13;
}
```

Can We Do Better?

Consider again **map** and **reduce**:

Example: map (add 1) [1, 2,.., n] $\rightarrow$ [2, 3, ..., n+1]

In General: map f [a_1, a_2, ..., a_n] $\rightarrow$ [f(a_1), f(a_2), ..., f(a_n)]

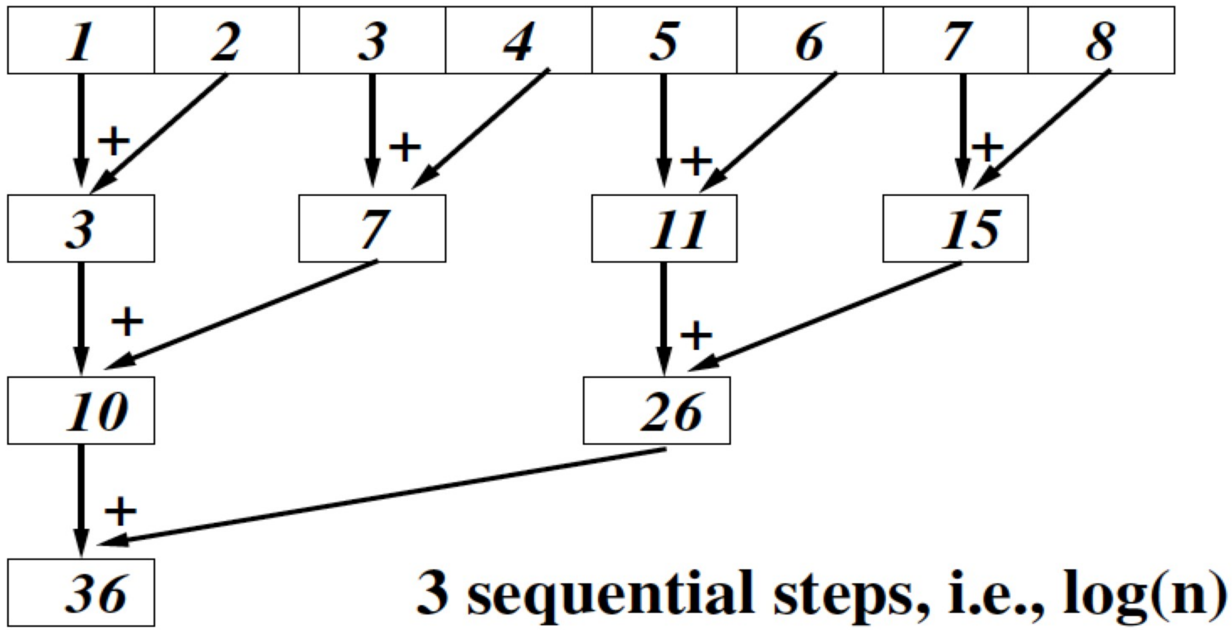
Example: reduce (+) [1, 2, 3, 4, 5] $\rightarrow$ 1+2+3+4+5 = 15

In General: reduce $\odot$ [a_1, a_2, ..., a_n] $\rightarrow$ $a_1 \odot a_2 \odot \ldots \odot a_n$

Requires $\odot$ to be an associative binary operator, i.e.,
$(a_1 \odot a_2) \odot a_3 = a_1 \odot (a_2 \odot a_3)$
Example: addition, i.e., $(1 + 2) + 3 = 1 + (2 + 3) = 6$

map is inherently parallel. How about reduce?

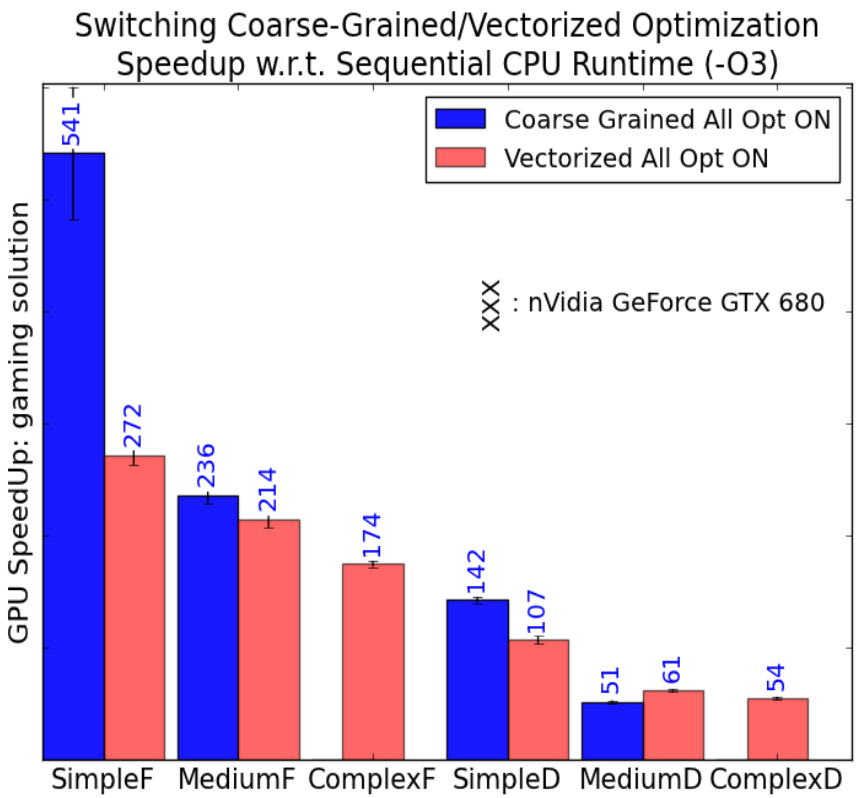
Computing **Reduce** in Parallel



By making use of **map** and **reduce** and a few other combinators (e.g., **scan**), a high degree of parallelism can be obtained.

Project: Pricing Financial Contracts on GPGPUs

Experiments made by HIPERFIT postdocs C. Oancea and C. Andreetta:



Three kinds of contracts: Simple, Medium, Complex

F/D: Floats/Doubles

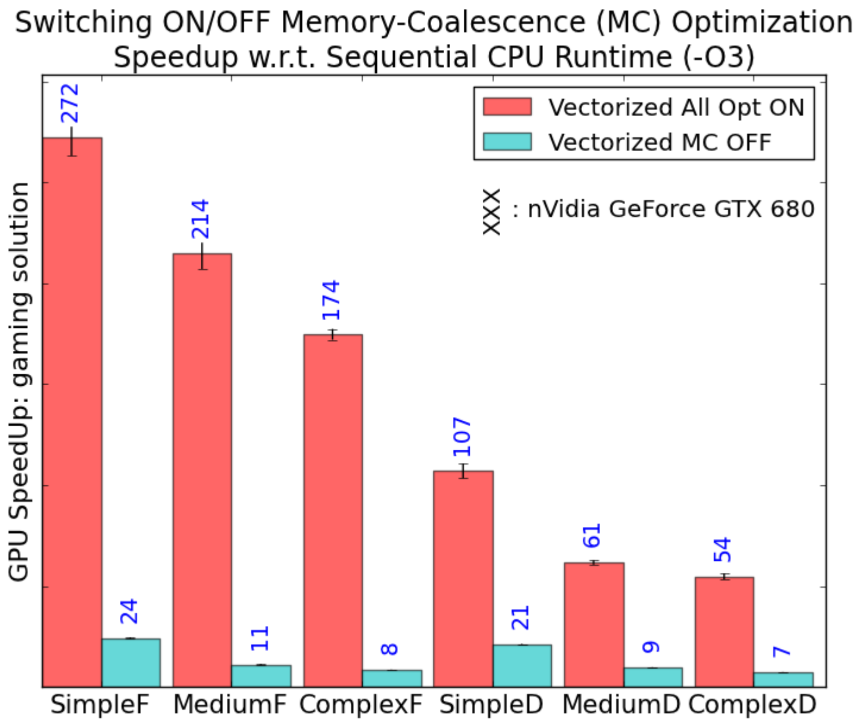
Coarse Grained: One outer map (use of map fusion)

Vectorized: Map distribution

Note: Complete setup with parallel sobol sequence generation, brownian bridge, payoff function...

GPGPUs Requires New Kinds of Optimizations

Accessing global memory in a “coalesced way” may lead to dramatic speedups!



Coalesced: consecutive threads must access consecutive global memory slots…

Often a change of algorithm is needed for ensuring coalescing (e.g., matrix transposition)…

Some Conclusions

Functional programming:

- Is **declarative**: **Focuses on what** instead of how
- Is **value oriented** (functional, persistent data structures)
- Eases **reasoning** (formal as well as informal)
- Eases **parallel processing**

Open Problems

Modern computation model is highly parallel:

- Computation occurs everywhere simultaneously
- Grand challenge: How to program it?
  (What is a good programming model?)

We need a **cost-model** that can predict relative parallel performance (and scalability) of algorithms in the presence of memory hierarchies.

How can model-builders benefit from **tomorrow's** parallel hardware without knowing about the latest OpenCL/CUDA programming techniques?