COMMERZBANK

# High Performance Computing Techniques in Finance

Andrea Odetti

Sanjeev Shukla

CORPORATES & MARKETS

## Introduction

- Focus on techniques to enhance and tune performance on a given machine
- Geared towards C++ but concepts should be language agnostic
- 2 main techniques presented:

  -Vectorised instructions

  -Efficient payoff languages

- We do NOT discuss distributed computing concepts

  -These are orthogonal to this presentation

# Content Of Vectorised Operations

# What are vectorised operations? Part I

- **SIMD** – **S**ingle **I**nstruction **M**ultiple **D**ata
- The same operation is performed upon multiple pieces of data in one “instruction”

```
double scalarExp( double x);
vector<double> vectorExp( const vector<double> & x ) ;
```

- The scalar version operates upon one input – vector operates upon multiple data
- The naïve implementation of vectorised version just loops over scalar version
- Of course proper implementations are cleverer and faster
- Best case is same speed as the scalar operation (and this is possible!)

# Acronyms everywhere…

- **SIMD** – **S**ingle **I**nstruction **M**ultiple **D**ata
  - -as explained
- **SSE** – **S**treaming **S**IMD **E**xtensions
  - -an explicit implementation of a SIMD instruction set
- **SSE2, SSE3, SSSE3, SSE4, SSE5**
  - -Enhancements to SSE
- **MMX**, **3DNow!**
  - -Early implementations of SIMD, now superseded by SSE and descendents
- **BLAS** – **B**asic **L**inear **A**lgebra **S**ubprograms
  - -An API specification of some basic operations

# What is BLAS?

- BLAS is a set of common linear algebra operations
- It is split into 3 levels:
  - -Level1 consists of vector-vector operations eg dotProduct
  - -Level2 consists of matrix-vector operations eg matrixTimesVector
  - -Level3 consists of matrix-matrix operations eg matrixTimesMatrix

- Since the basic elements are vectors and matrices…
- …any implementation of BLAS can benefit from use of vectorised instructions
- So from now on we only refer to vectorised instructions and assume this subsumes BLAS

# Example BLAS

- In the example below one can see the vector nature of BLAS straight away
- The explicit loop pointwise over vector elements is replaced by one simple function call

**Generic implementation**

```
double dotProduct(int n, double * x, double * y)
{
  double returnValue = 0.0 ;

  for(int i = 0 ; i < n ; ++i)
    returnValue += x[i]*y[i];

  return returnValue;
}
```

**BLAS implementation**

```
double dotProduct(int n, double * x, double * y)
{
  return BLAS::dotProduct(n,x,y);
}
```

# How to benefit from vectorised operations

- Analysis showed that a lot of time is spent in

  - -**Linear algebra** to generate Monte Carlo Paths for Brownian motion
    - Local volatility for a high dimensional trade: a lot of matrix multiplication to correlate gaussian random variables
    - Full factor BGM: a lot of vector operations for path construction

  - -Black Scholes formula to calibrate **Stochastic Volatility** and **Jump Diffusion**
    - For **each path NxM** Black Scholes Formulae are computed
    - N = number of times ; M = number of strikes ie N & M span the vol surface

# CPU Registers Part I

- **CPU Registers**

  -The CPU uses data registers to hold data(!)

  -Data might be integers, floats, bit sets

  -Access to these registers is extremely fast – the fastest memory available for access by the CPU

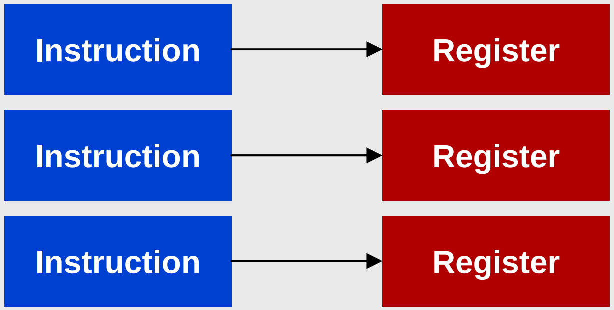
  -CPU instructions act on these registers (and possibly store results in them)

  -However registers are few and far between

  -Compilers deal with the job of allocating registers and moving data between main memory and the registers (or rather producing code which does this)

# CPU architecture methods

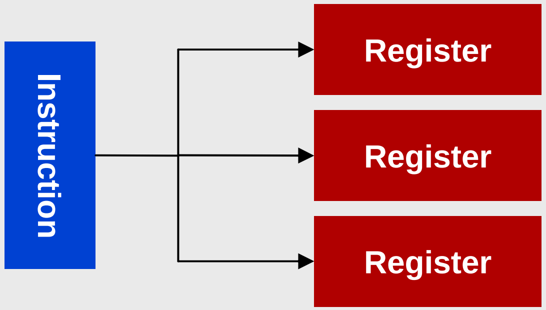
- **SISD – S**ingle **I**nstruction **S**ingle **D**ata




- Individual instructions act on individual data (held in registers)
- Implemented by the scalar FPU for example

# CPU architecture methods (cont.)

- **SIMD – S**ingle **I**nstruction **M**ultiple **D**ata




- The same instruction acts on multiple data (held in registers)
- First made popular with supercomputers in 80’s and 90’s for example

# CPU Registers Part II

- Registers come in various sizes eg 32 bit, 64 bit, 128 bit
- Vectorised operations in fact operate upon one register “packed” with data

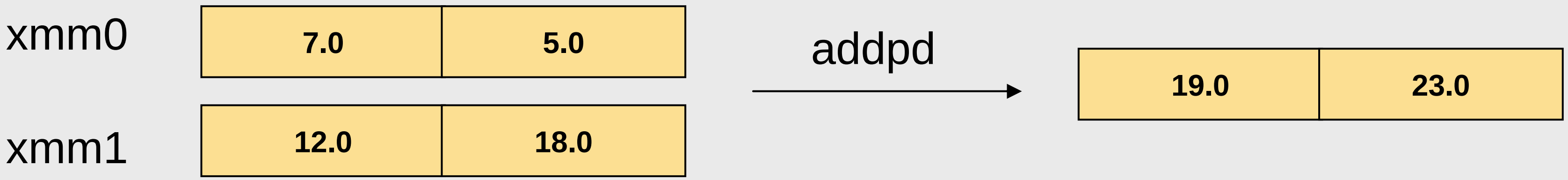
  -Eg a 128 bit register could be filled with:

| | | |
|---|---|---|
| - 2 doubles | (8 bytes each) | (8 bytes = quadword) |
| - 4 floats | (4 bytes each) | (4 bytes = dword) |
| - 2 long integers | (8 bytes each) | |
| - 4 integers | (4 bytes each) | |
| - 8 integers | (2 bytes each) | (2 bytes = word) |
| - 16 integers | (1 byte each) | |

# Anatomy of a vectorised instruction

- Consider the assembler instruction to add contents of the 128 bit registers xmm0 and xmm1 (populated with packed double data) and store the result in xmm0 (as packed double data)

  - addpd xmm0 , xmm1



# How to implement these instructions? I

- Standalone assembly
  - -Pros
    - - The fastest code and best flexibility possible short of writing machine code
  - -Cons
    - - Too complicated!
    - - Lots of expertise to write. Not reusable. Hard to maintain.

- Inline assembly embedded into C/C++
  - -Easy to use encapsulated functions. But Cons as before…
  - -Lots of expertise to write and hard to maintain

# How to implement these instructions? II

- Intrinsic Functions
  - -These are compiler/vendor dependent
  - -They are similar to inline functions in sense that code is embedded directly into point of use rather than a function call
  - -Better than inline though since the machine code is generated directly; often platform specific
  - -The SSE2 instruction set is available in the Visual Studio compiler as a set of intrinsic functions
  - -However the same problems remain – to code these requires similar knowledge of the instruction set

# How to implement these instructions? III

- 3rd Party Libraries
  - -Pros
    - In effect someone else has done the hard work for you using some combination of the methods mentioned in the previous slides
    - Maintenance is by the library vendor
    - Functions should be in a nice easy to use form
  - -Cons
    - Dependent on a black box solution from an external provider

## How to implement these instructions? (cont.)

- Example of inline assembly

  -Each of x and y contain 2 doubles and returns (x[1]+y[1], x[2]+y[2]) as 2 doubles

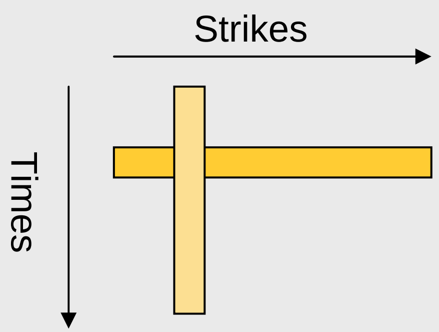
```
void add( double * x, double *y, double * returnValue )
{
  asm
  {
    movapd xmm0 , [x]
    movapd xmm1 , [y]
    addpd  xmm0 , xmm1
    movapd [returnValue] , xmm0
  }
}
```

# Worked example I

- Monte Carlo calibration of a model (such as a stochastic volatility model) requires valuation of the calibration products on each path.
- Such a calibration product may be a European Option
- The number of European Options needed to span the volatility surface can be large – can this benefit from vectorised operations?
- The number of European Options will be numberOfStrikes x numberOfTimes
- So we can vectorise in 2 possible ways:

  -Fix a strike and have a vector of times

  -Fix a time and have a vector of strikes
- Only trial and error will reveal quickest
- We choose to fix a time



# Worked example I (continued)

- We now need to consider how to vectorise a Black Scholes formula
- Central to the Black Scholes formula is evaluation of the cumulative normal distribution $\Phi(z)$.

□ $\Phi(z) = 0.5[1 + \mathrm{erf}(z/2^{1/2})]$

- Some vectorised libraries have an implementation of erf(z)
- Simple to extend and vectorise the Black Scholes formula for multiple strikes at fixed time and forward

**Generic implementation**

```
double cumNormDist(double z)
{
   return 0.5*(1.0+erf(z*ONE_OVER_SQRT_TWO));
}
```

**Vectorised implementation**

```
void cumNormDist(int n, double * z)
{
   BLAS::L1::scale(n,z,ONE_OVER_SQRT_TWO);
   SIMD::erf(n,z,z);
   SIMD::add(n,VECTOR_OF_ONES,z,z);
   BLAS::L1::scale(n,z,0.5);
}
```

# Worked example I: Results

## Specifications

-2 factor Stochastic Vol Model

-19 strikes for pricing model example

-15 strikes for calibration model example

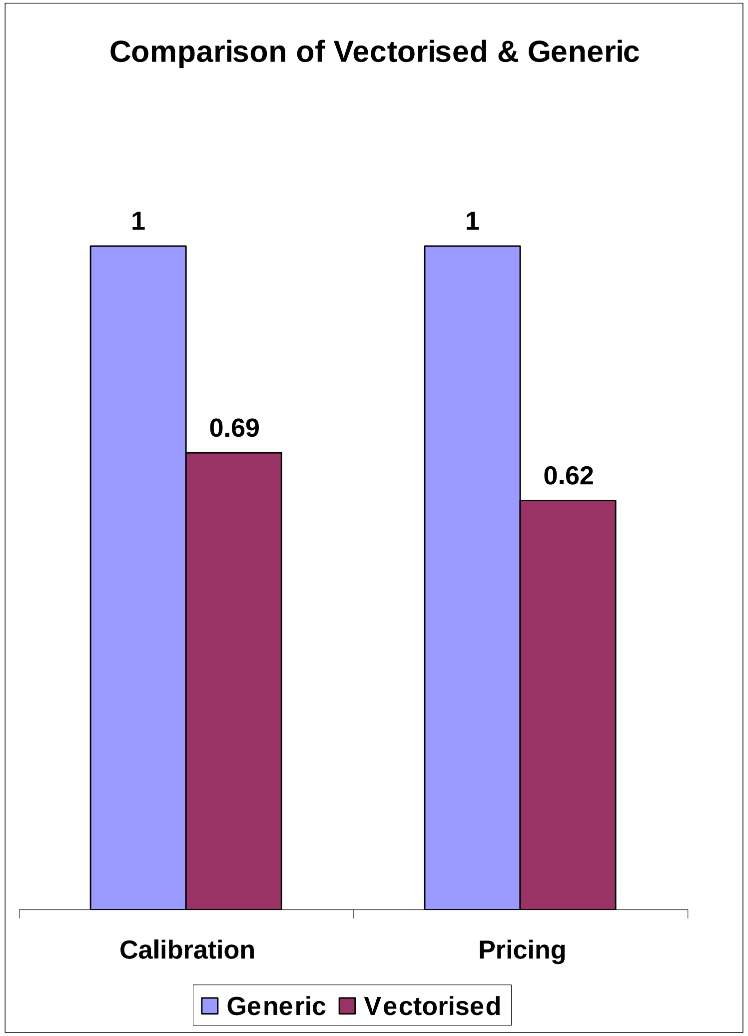
-(scales linearly in times)

## Comments

-Significant speed gains possible

-Mostly due to vectorised versions of complex mathematical functions rather than simple vectorisation functions such as BLAS

-Calibration gives slightly less improvement than Pricing since calibration involves many other overheads




Source: Financial Engineering, Commerzbank

# Worked example II

- A full factor BGM model has a high number of factors
- Each factor has lognormal type dynamics (with state dependent drift)
- Also a large number of matrix multiplications for correlated gaussians
- Prototypical dynamics for the $i^{th}$ Libor $L_i$ are of the form:

  $L_i(t) = \exp( drift_i(t) – 0.5\ sigma\_squared_i(t) + W_i(t) )$

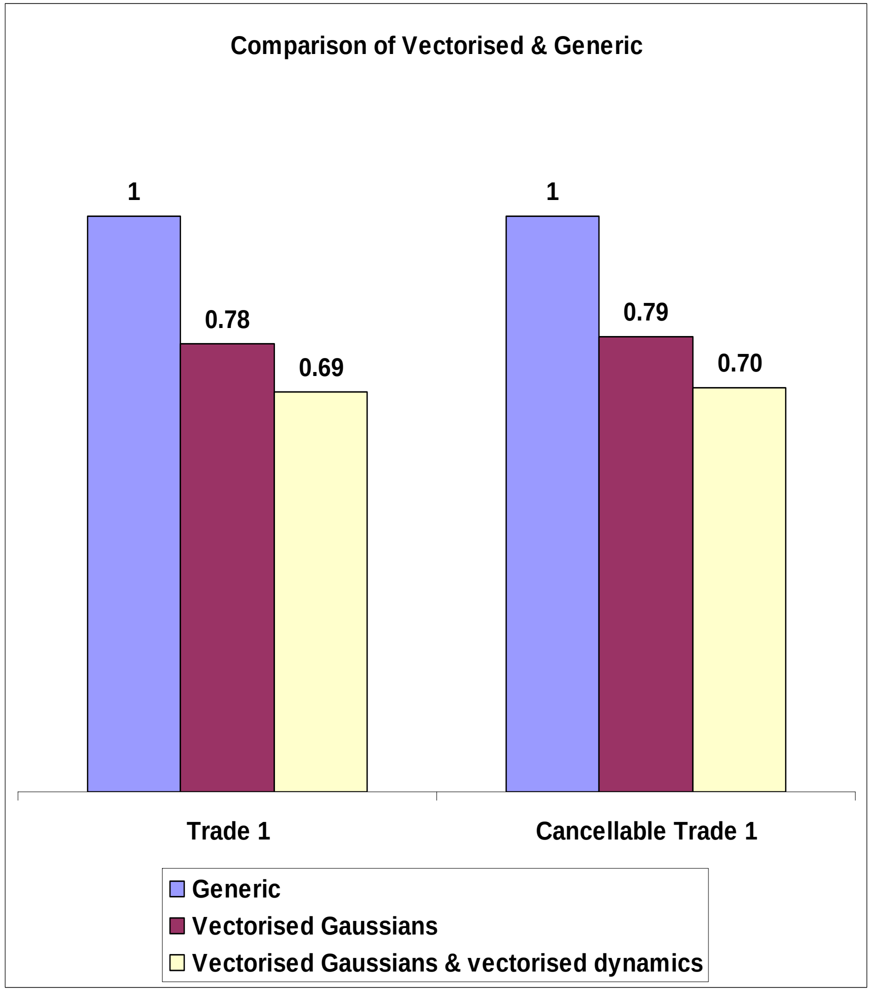
- Vectorisation proceeds as in example I but in two places:

  -Vectorise the matrix multiplications of the correlated gaussians

  -Vectorise the above dynamics using simple BLAS type routines for the addition and scaling of vectors and a vectorised version of exponential.

# Worked example II: Results

- **Specifications**
  - 40y BGM model, Libor_3M underlying
  - Trade 1 is a Ratchet Range Accrual, 6 month periods, with weekly observation frequency
- **Comments**
  - Again significant improvements observed
  - Approximately two thirds of overall improvement due to matrix multiplies
  - One third due to vectorised functions
  - Testing revealed simple vectorisation of dynamics made very little contribution to the last one third




**Source: Financial Engineering, Commerzbank**

## Conclusions

- Pros:
  - Faster! But how much is really code, CPU and problem dependent
- Cons:
  - Penalty for low dimensionality (use at least for size 8)
  - Can be harder to read & maintain
  - Not all math operations available: need of many temporary vectors

    (e.g. add 5 to every entry in a vector: need **vector of ones**)
  - Is it portable?
  - Mitigated by fact that many financial institutions work in controlled environments with fixed architectures and CPUs

- A useful tool but needs careful use and application!

# Content of Payoff Languages

# Payoff languages in a financial library

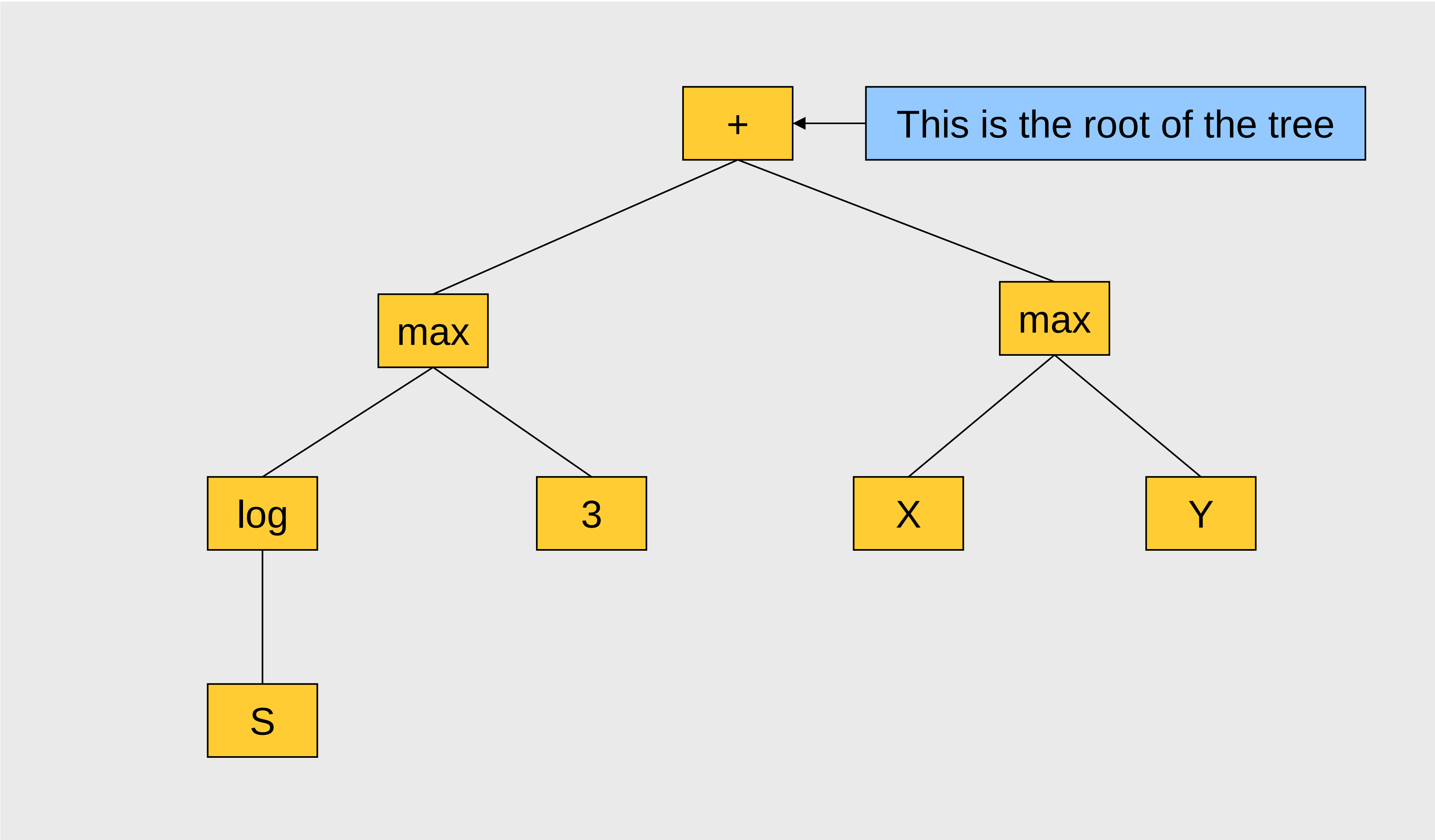
- From a string containing the textual description of a mathematical function, it is possible to dynamically (i.e. at runtime) generate a data structure representing it.

- Without limitation, we will confine ourselves to a case where there is only 1 payment, depending upon the values of an underlying called S and some extra variables X and Y

  - Payoff(S) = Max(Log(S), 3) + Max(X, Y)

- A very common implementation of this structure is a **tree**

# Abstract Syntax Tree



# Description of a node

- Node — Function
- Children — Arguments

- A node without children (i.e. a leaf) is a **number**
  - 5.6
  - X
  - Value of EuroStoxx50

- Every node has a function that returns its **value** (**after** valuing all arguments)

- To value the tree, just call **value** on the **root**

# Pros / Cons of AST

- Pros
  - Many

- Cons
  - It depends heavily on polymorphism
    - **virtual functions** (in C++)
    - Calls via **function pointer** (in C)

- For each path and for each node a virtual function is called
  - The pipeline stalls
  - BTB useless because target of jumps does not depend on **code location**, but on the **location in the tree**

    **B**ranch **T**arget **B**uffer is a map in the CPU [address of code -> destination of jump]

# But…

- The elements of the tree **do not change** once the tree is built (i.e. their dynamic types are **constant**)
  - If node types were path dependent, this approach would not be possible
- Given a **position in the tree**, the virtual function called is the **same** for each path
- In the following we are going to make more explicit the link between position in the tree and the function called
- Then we will be able to **tell the CPU** that information

# Reverse Polish Notation

- A tree is inherently written in Prefix notation
- We want to transform it to **Postfix** notation

- From

Max(Log(S), 3) + Max(X, Y)

- To

S, Log, 3, Max, X, Y, Max, +

- This can be obtained by **traversing in postorder** the tree.

# Postorder traversal

- Definition

  PostTraverse(Node a)

  {

      for each child c: PostTraverse(c)

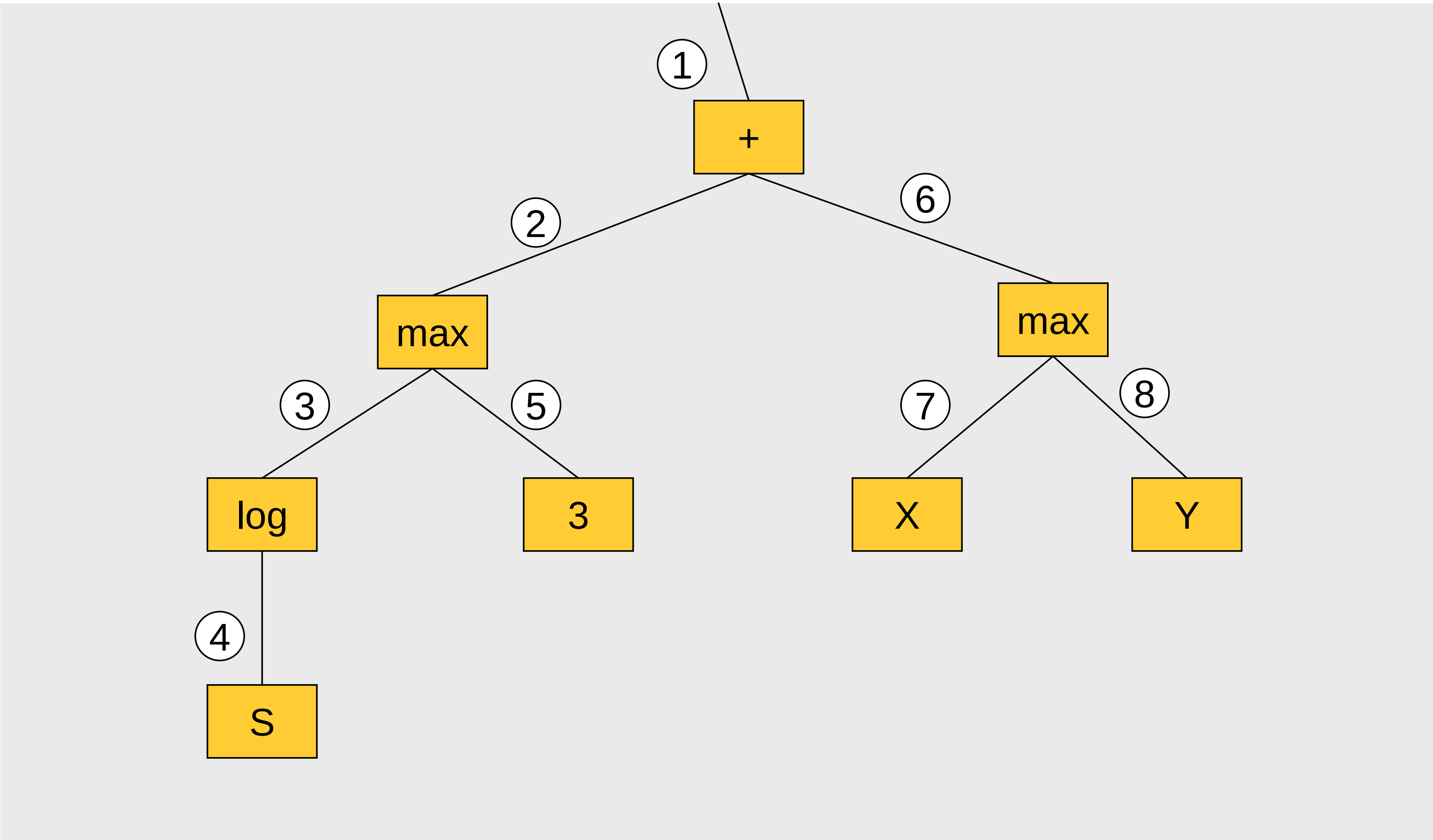
      Do something about yourself (e.g. **print function name**)

  }

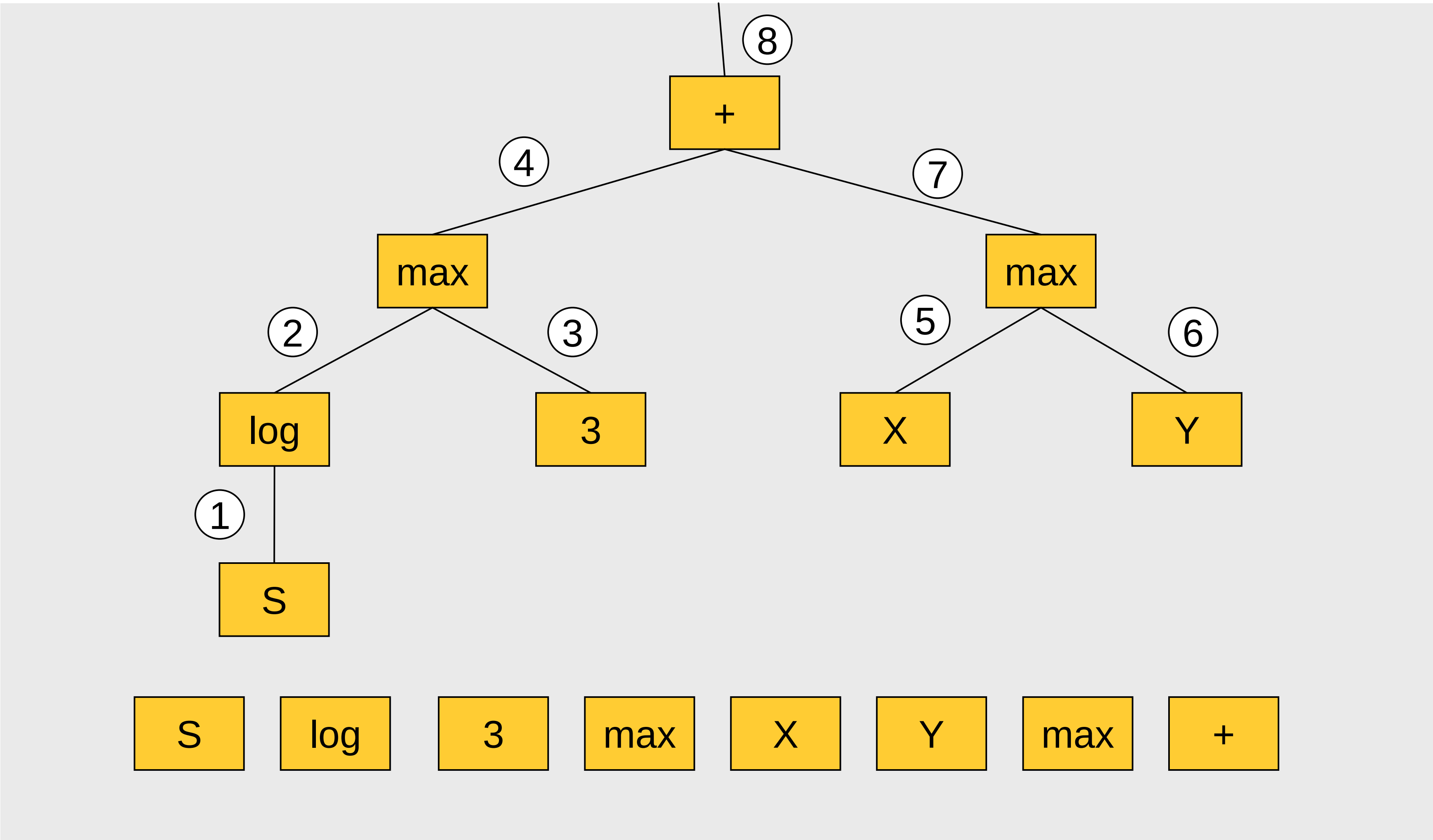
  - PostTraverse(root)

- This can be seen as writing **in a linear sequence** the names of the nodes in the order they **return from** the value function.

# Tree valuation: order of *calling* value



# RPN: order of *returning* from value



# Pseudo code

```
postTraverse(Node a, vector<Node> v)
{
        for each child c: postTraverse(c)
        v.push(this)
}


…..
vector<Node> linearTree
postTraverse(root)
…..
```

## RPN calculator: we need a stack

- In a tree valuation, a function **values** its arguments
- In postfix notation, a function is valued **after** its arguments.

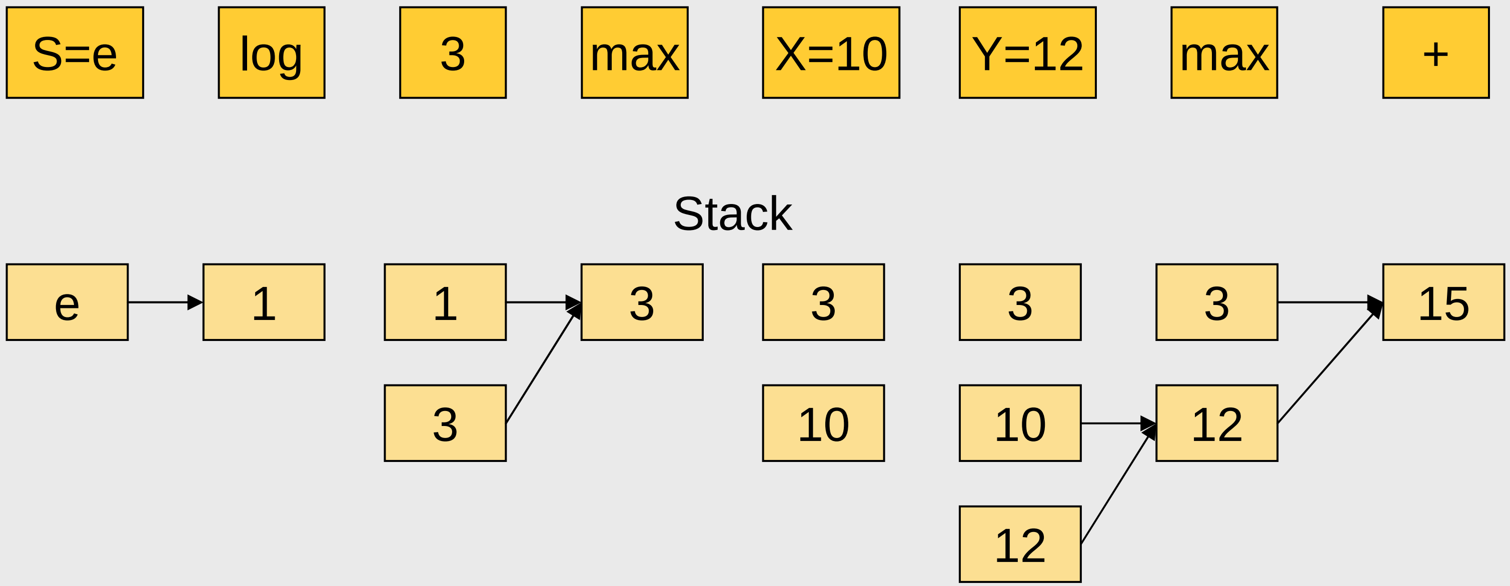
- When a node is *rpnValued* its arguments have to be **available**, **used** and **deleted**.
- A **stack** is the best candidate for this job

```
double add::treeValue()
{
    val1 = arg1.treeValue()
    val2 = arg2.treeValue()
    result = val1 + val2
    return result
}
```

```
void add::rpnValue(stack s)
{
    val1 = s.pop()
    val2 = s.pop()
    result = val1 + val2
    s.push(result)
}
```

# Example of stack based valuation

S, log, 3, max, X, Y, max, +




Every block must **pop ALL** (if any) its arguments and **push ONE** result

# RPN valuation: pseudo code

```
double valuePayoff(vector<Node> nodes)
{
        stack s
        for (int i = 0; i < nodes.size())
        {
                nodes[i].rpnValue(s)
        }
        assert(s.size() == 1)        << not present in the tree
        return s[0]
}
```

**rpnValue** is still a **virtual function**!

# However…

- The association of types (i.e. address of the virtual functions) with loop iteration is clear and more evident than in the tree

  - nodes[0] is always of type Stock
  - nodes[1] is always of type Log
  - nodes[2] is always of type DoubleConstant
  - …..

- How can we communicate it to the CPU?

# … we just unroll the loop!

- In order to tell the CPU about the type of the nodes we can simply unroll the loop and **static_cast** each node

```
double valuePayoff(vector<Node> nodes)
{
        stack s
        static_cast<Stock>  (nodes[0]).non_virtual_rpn_value(s)
        static_cast<Log>    (nodes[1]).non_virtual_rpn_value(s)
        …..
        static_cast<Add>    (nodes[7]).non_virtual_rpn_value(s)
        return s[0]
}
```

# We need to compile the code again

- But this can only done at **runtime** (when we have knowledge of the tree).

- There are at least 2 solutions:

  - **Write** C++ code to a file, call the **compiler** and dynamically **load** the DLL

  - Manually generate the **machine code**

- Can I find a compiler / assembler that I can **link** to my library?

# Machine code (in small doses)

- This is not as scary as it sounds because we only need to call functions like

  **static_cast<Stock>(nodes[0]).non_virtual_rpn_value(s)**

- Where the only differences are 2 pointers
  - The object's **this** (in Visual Studio passed in **ECX**)
  - The **address** of the function to call

| | | |
|---|---|---|
| FF 74 24 08 | push | dword ptr [esp+8] |
| B9 **50 3F 9A 1B** | mov | ecx,**1B9A3F50h** |
| E8 **20 C0 F0 11** | call | Stock::non_virtual_rpn_value (**11F0C020h**) |

| | | |
|---|---|---|
| FF 74 24 08 | push | dword ptr [esp+8] |
| B9 **70 44 9A 1B** | mov | ecx,**1B9A4470h** |
| E8 **20 B6 F0 11** | call | Log:: non_virtual_rpn_value (**11F0B620h**) |

| | |
|---|---|
| C3 | ret |

# Pros / Cons of the RPN & compiled code

- Cons
  - Extra complexity (whether calling a compiler or managing machine code)
  - Hard to handle functions that **conditionally value their arguments** (e.g. IF, logical operator, variable length loops)
  - Machine code: Harder to port to different CPUs and compilers
- Pros
  - It can coexist with tree valuation
  - **No virtual functions** call
  - The machine code is self contained
  - Compiler can **inline** most of the functions (+,-,max,log…..)
  - Given the limitations of the language, **there are no branch mispredictions**
  - Therefore more CPU resources available to the rest of the application
  - Potentially allows for more **parallelization** (e.g. ClearSpeed hardware)

# Example of IF statement

- Original expression:

3 + IF( X > 0, return X + Y, else Z – 6)

- RPN notation

3, | X, 0, >, X, Y, +, Z, 6, -, IF, | +

```
double if::treeValue()
{
    cond = arg_cond.treeValue()
    if (cond)
        return arg_true.treeValue()
    else
        return arg_false.treeValue()
}
```

```
void if::rpnValue(stack s)
{
    val_false = s.pop()
    val_true  = s.pop()
    cond      = s.pop()
    result = cond ? val_true : val_false
    s.push(result)
}
```

When we get to the IF block, both cases **have already been valued** (i.e. they are already in the stack)

# How to solve the IF case

- However we feel that an IF function is an almost essential feature of any payoff language

- There are a few solutions

1. We can treat IF as a **black box** and revert to the tree valuation

   We lose all benefits of the compilation

2. We can value **both arguments** and then select the correct one

   Best solution, especially for trivial cases. Code is still branch-free.

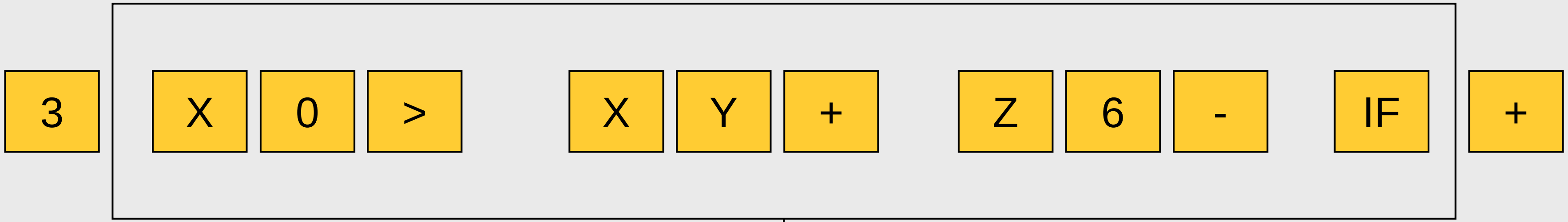
   Not possible when functions have **side effects**.

3. Emit more **complex** code

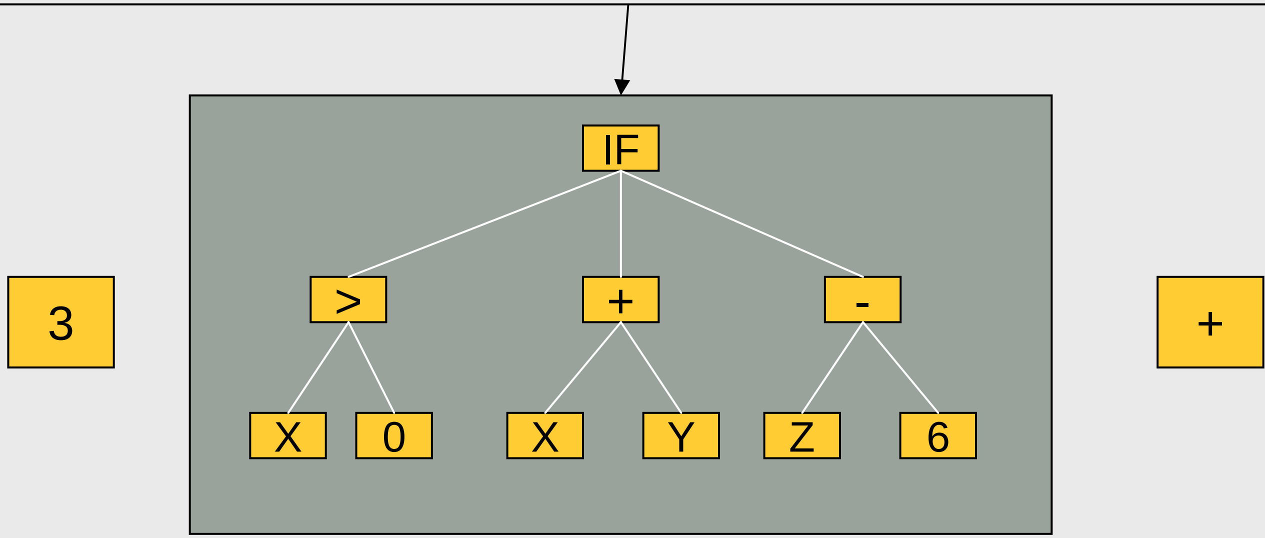
   Allows to handle more sophisticated cases (e.g. loops)

# IF as a black box

- As a fallback, for more complex cases we can **revert to the tree valuation**

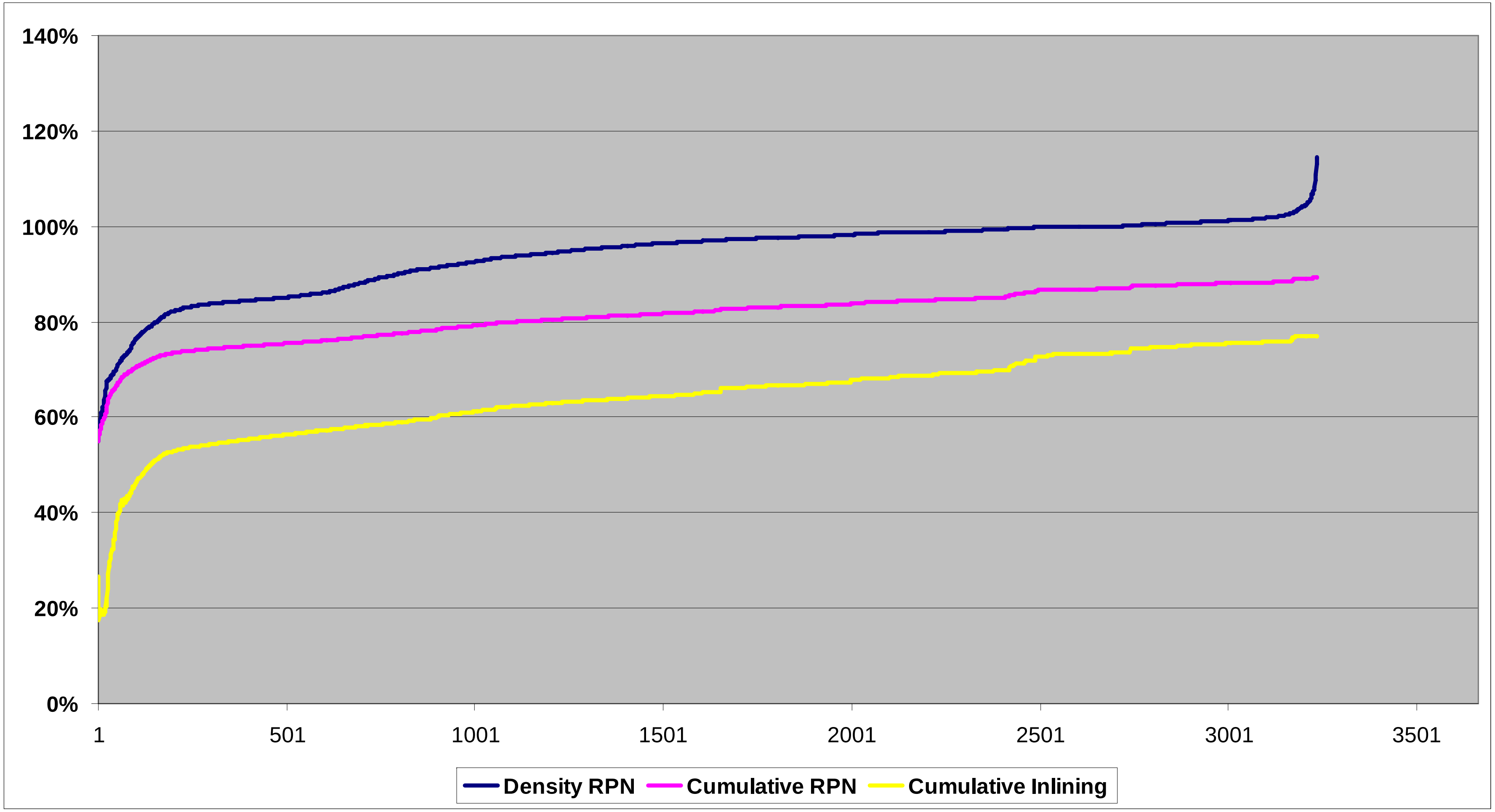



- Black box




- The same can be applied to **any other** complicated **function** (e.g. loops)

# Comparison Tree vs RPN valuation




Source: Financial Engineering, Commerzbank

# Results and conclusions

- We have implemented an internal compiler to **machine code**
- Initial tests on a large (~3000) set of equity trades have shown speed up of about 11%
- More (up to 25%) can be gained with **more aggressive inlining** of trivial operations
- **No change** to the pricing / risk / farm **infrastructure** since this solution is self contained in the library

- Very important to **encapsulate** complexity in order to keep code usable, readable and maintainable
- Easy to implement tree and RPN methods **side by side**

- Important to run over a **wide range of trades** to **profile** and **tune**

# Disclaimer

## Additional information available on request

This document has been created and published by the Corporates & Markets division of Commerzbank AG, Frankfurt/Main or the group companies mentioned in the document ("Commerzbank"). Commerzbank Corporates & Markets is the investment banking division of Commerzbank, integrating research, debt, equities, interest rates and foreign exchange.

This is a financial promotion/marketing communication (together “communication”). It is not “investment research” or “financial analysis” as these terms are defined in applicable regulations and has not been prepared by a research analyst. The views in this communication may differ from the published views of Commerzbank Corporates & Markets Research Department and the communication has been prepared separately of such department.

This communication may contain short term trading ideas. Any returns or future expectations referred to are not intended to forecast or predict future events. Any prices provided herein (other than those that are identified as being historical) are indicative only, and do not represent firm quotes as to either size or price.

This communication is for information purposes only. The information contained herein does not constitute the provision of investment advice. It is not intended to be nor should it be construed as an offer or solicitation to acquire, or dispose of, any of the financial instruments and/or securities mentioned in this communication and will not form the basis or a part of any contract.

Any information in this communication is based on data obtained from sources believed by Commerzbank to be reliable, but no representations, guarantees or warranties are made by Commerzbank with regard to the accuracy, completeness or suitability of the data.

The past performance of financial instruments is not indicative of future results. No assurance can be given that any financial instrument or issuer described herein would yield favourable investment results.

This communication is intended solely for distribution to professional customers and/or eligible counterparties of Commerzbank. It is not intended to be distributed to retail clients or potential retail clients. Neither Commerzbank nor any of its respective directors, officers or employees accepts any responsibility or liability whatsoever for any expense, loss or damages arising out of or in any way connected with the use of all or any part of this communication.

Commerzbank and/or its principals or employees may have a long or short position or may transact in financial instrument(s) and/or securities referred to herein or may trade in such financial instruments with other customers on a principal basis. The information may have been discussed between various Commerzbank personnel and such personnel may have already acted on the basis of this information (including trading for Commerzbank’s own account or communication of the information to other customers of Commerzbank). Commerzbank may act as a market maker in the financial instruments or companies discussed herein and may also perform or seek to perform investment banking services for those companies.

No part of this communication may be reproduced, distributed or transmitted in any manner without prior written permission of Commerzbank. This communicatuion or the manner of its distribution may be restricted by law or regulation in certain countries. Persons into whose possession this document may come are required to inform themselves about, and to observe any such restriction.

By accepting this communication, a recipient hereof agrees to be bound by the foregoing limitations.

This communication is issued by Commerzbank. Commerzbank AG, London Branch is authorised by the German regulator Bundesanstalt für Finanzdienstleistungsaufsicht (BaFin) and by the Financial Services Authority (FSA); regulated by the FSA for the conduct of business in the UK.

**Additional note to readers in the following countries:**

**Italy**: You should contact Commerzbank AG, London Branch if you wish to use our services to effect a transaction in any of the financial or other instruments mentioned in this communication.

**US**: not for distribution in United States

**Japan**: not for distribution in Japan

# Commerzbank Corporates & Markets Locations

## Main Offices

**Frankfurt**
DLZ - Gebäude 2
Händlerhaus
Mainzer Landstraße 153
60327 Frankfurt
Germany
Phone: +49 69 136 44440

**London**
60 Gracechurch Street
London, EC3V 0HR
United Kingdom
Phone: +44 20 7653 7000

## Branches

**Amsterdam**
Strawinskylaan 2501
1077 ZZ Amsterdam
Netherland
Phone: +31 205 574 911

**Budapest**
Széchenyi rakpart 8
H-1054 Budapest
Hungary
Phone: +361 374 8100

**Brussels**
Blvd Louis Schmidt 87
BE-1040 Brussels
Belgium
Phone: +32 2 743 1866

**Hong Kong**
21/F, Hong Kong Club Building
3a Chater Road
Hong Kong
China
Phone: +852 2842 9666

**Johannesburg**
5 Keyes Avenue
2196 Johannesburg
South Africa
Phone: +27 11 328 7600

**Luxembourg**
25, rue Edward Steichen
2540 Luxemburg
Luxembourg
Phone: +352 477 9111

**Madrid**
Ps. Castellana 110
28046 Madrid
Spain
Phone: +34 91 572 4700

**Milan**
Via Cordusio 2
20123 Milan
Italy
Phone: +39 02 725 961

**Moscow**
Kadashevskaya naberezhnya 14/2
119017 Moscow
Russia
Phone: +7 495 797 4800

**New York**
2 World Financial Center
31st Floor
New York, NY 10281
USA
Phone: +1 212 266 7200

**Paris**
23 rue de la Paix
75002 Paris
France
Phone: +33 1 4494 1700

**Prague**
Jugoslavska 1
120 21 Prague
Czech Republic
Phone: +420 221 193 111

**Singapore**
8 Shenton Way
Temasek Towers
Singapore 068811
Singapore
Phone: +65 63110 000

**Shanghai**
25F, World Plaza
855 Pudong South Road
200120 Shanghai
China
Phone: +86 21 5836 6666