Technology in banking – a problem in scale and complexity

Stanford University, 11 May 2011

Peter Richards and Stephen Weston

Our talk today

JPMorgan Chase & Co.

The business challenges in global banking within JPMorgan Chase encompass many areas of computer science – with the added dimension of scale.

This introductory talk will examine the scope of challenges that are currently being faced by the technology team at JPMorgan Chase.

Specifically, abstraction in both application and data environments, security and control and application resiliency in a continuous environment.

A case study will provide depth and context.

Structure of the talk

JPMorgan Chase & Co.

Scale at JPMorgan Chase

Complexity at JPMorgan Chase

Challenges

A case study

Conclusions

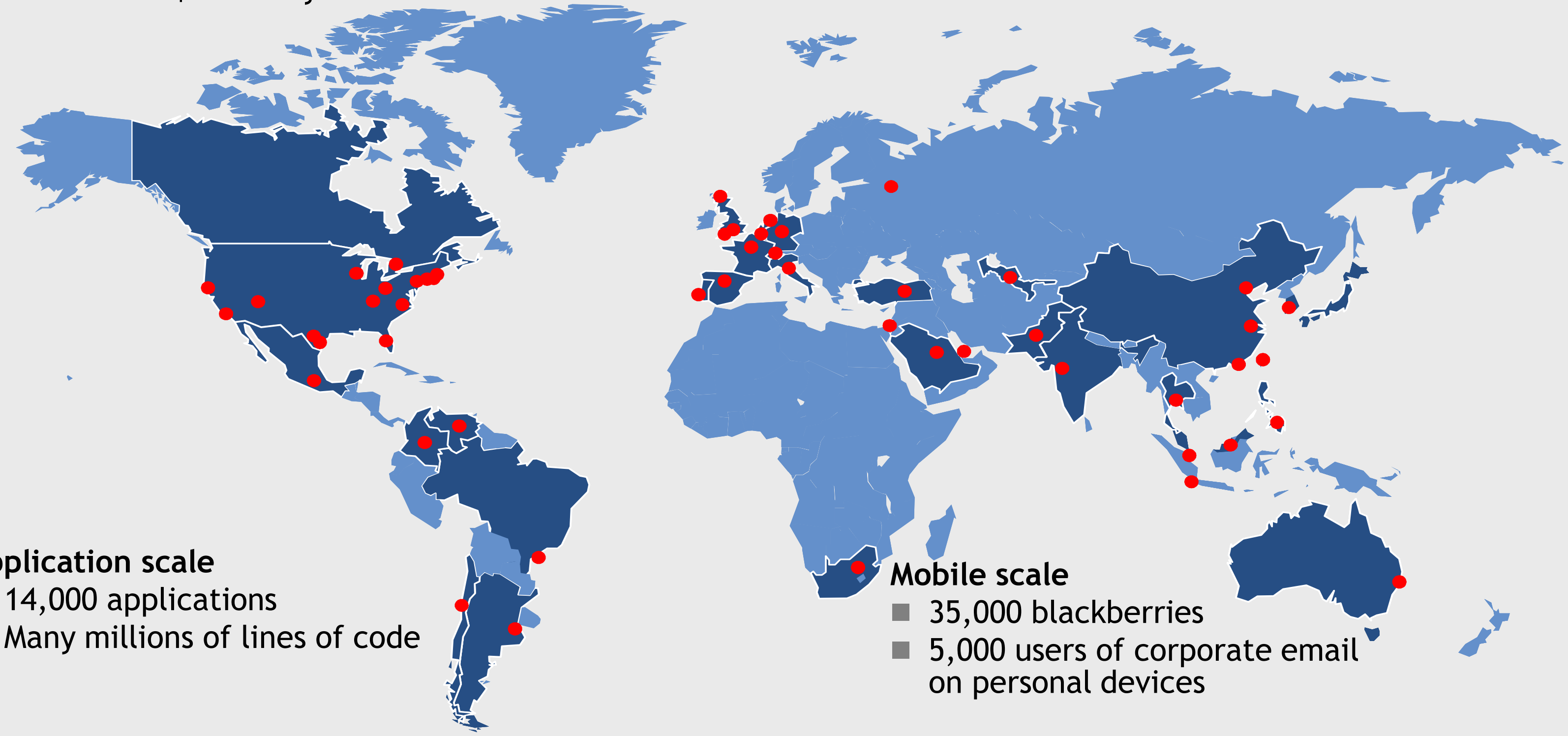
Q & A

If what you have heard is of interest…

JPMorgan Chase has a global technology footprint

Business scale
- JPMC Headcount: 240,000+
- 24.5MM checking accounts; 5,500 branches; 15,500 ATMs
- 139 million cards in circulation
- $1.8Tn Assets Under Supervision
- Clears over $5Tn daily

Infrastructure scale
- > 950K+ sqft datacenter space, 42 Datacenters
- 50K+ servers
- 150PB of storage
- 300,000 endpoints (desktops, laptops, VDI)
- 1.4m network ports



Application scale
- 14,000 applications
- Many millions of lines of code

Mobile scale
- 35,000 blackberries
- 5,000 users of corporate email on personal devices

Challenges

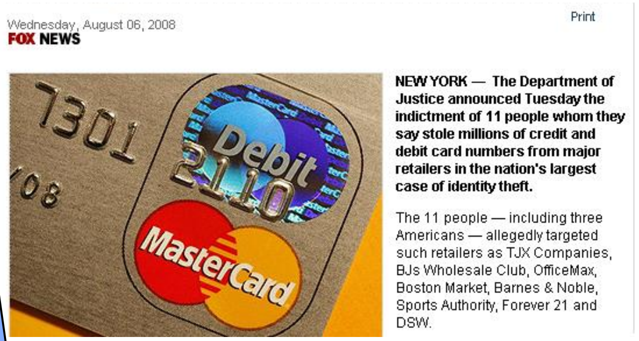
JPMorgan Chase & Co.

- Today, we are talking about challenges posed by scale and complexity and how they are faced by our teams in technology and quantitative research
- Over 30y profitability has gone up by a factor of 30 with only twice as many people (12% growth rate)
- Mostly due to gains in productivity due to technology paired with smart algorithms that take advantage of that technology
- Our dependence on technology and automation through algorithms is growing as markets become increasingly electronic

Challenges (just a few!)

JPMORGAN CHASE & CO.

Tele-presence to leverage product knowledge specialists across 5,000 branches



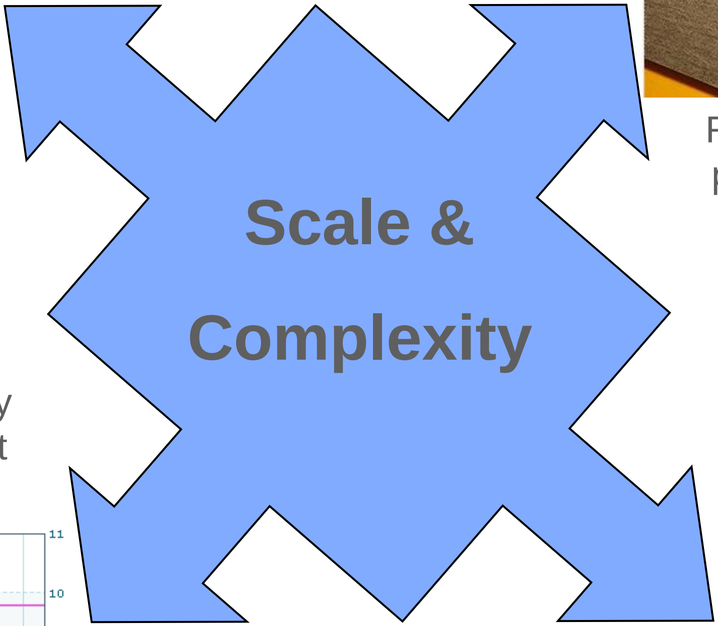
Wednesday, August 06, 2008
FOX NEWS

Print

NEW YORK — The Department of Justice announced Tuesday the indictment of 11 people whom they say stole millions of credit and debit card numbers from major retailers in the nation's largest case of identity theft.

The 11 people — including three Americans — allegedly targeted such retailers as TJX Companies, BJs Wholesale Club, OfficeMax, Boston Market, Barnes & Noble, Sports Authority, Forever 21 and DSW.

Real-time fraud detection to protect our 27m credit card customers



Optimising data transformation, storage and transmission for 160Pb of data

Low-latency/high frequency trading & risk management for $Tn of trades

R (9.8)
R (8.29)
S (7.09)
S (6.17)
11
10
9
8
7
6
Dec
Jan
Feb
Mar
Apr
May
🟩 20 Day Moving Average
🟥 50 Day Moving Average

Case study

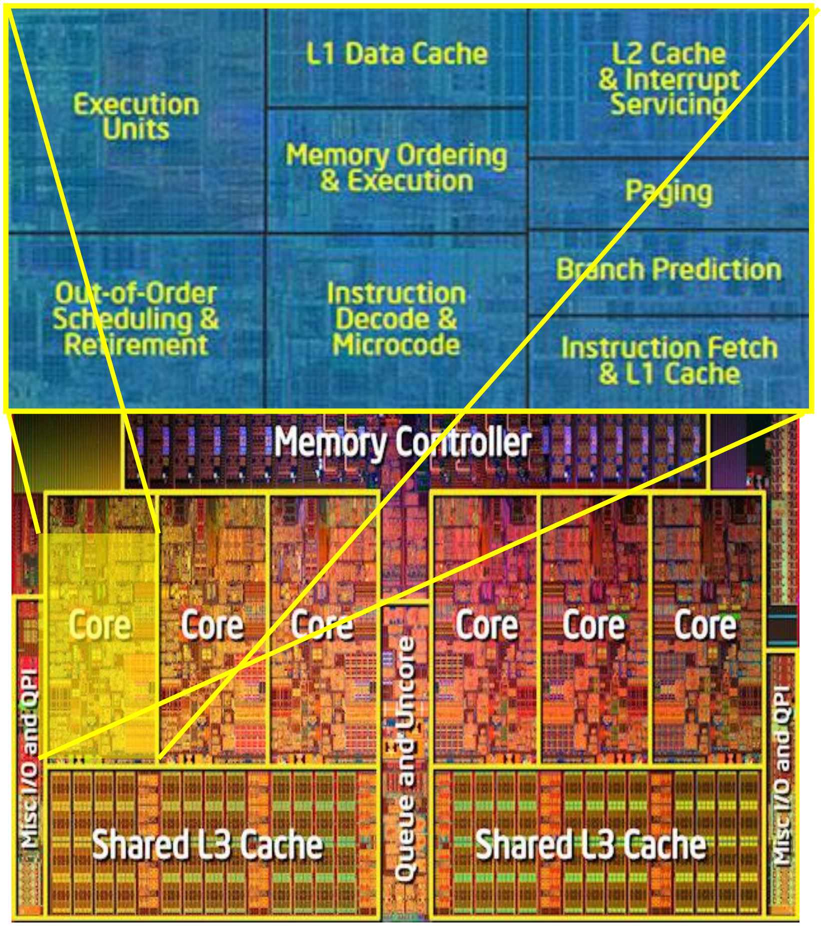
J.P.Morgan

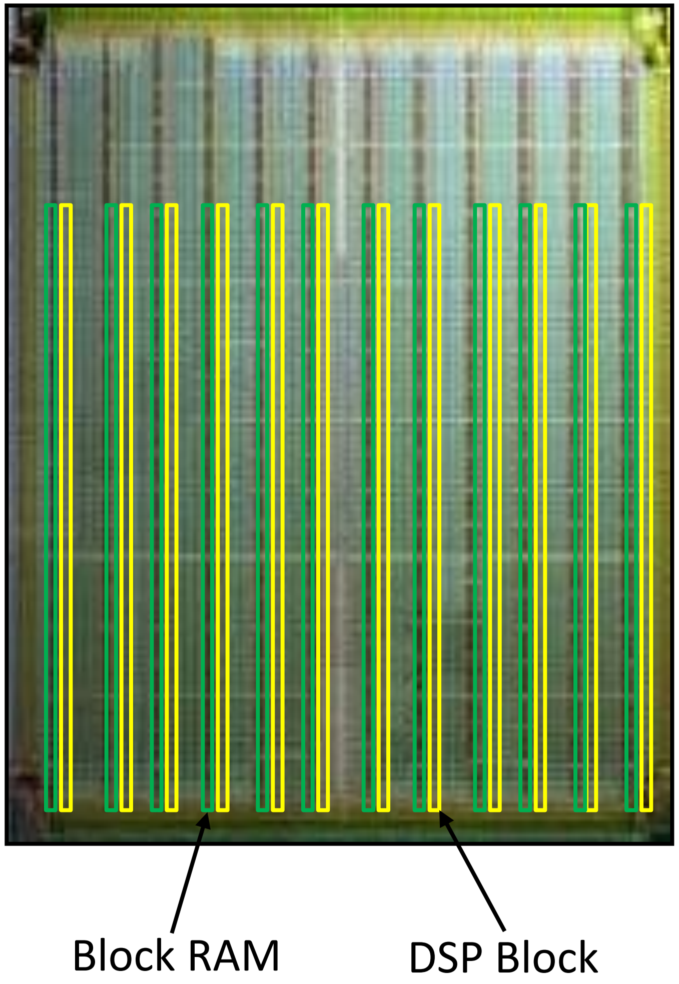
- We are now going to turn to a case study that highlights J.P.Morgan's commitment to meeting these challenges by harnessing its capabilities and capacity for innovation
- We will be focusing on a small part of a multi-year collaborative project between J.P.Morgan and a small but talent-packed firm called Maxeler Technologies started by a present and former Stanford allumni

CPU v FPGA - how the silicon is used

J.P.Morgan

Intel 6-Core X5680 “Westmere”





FPGA-101

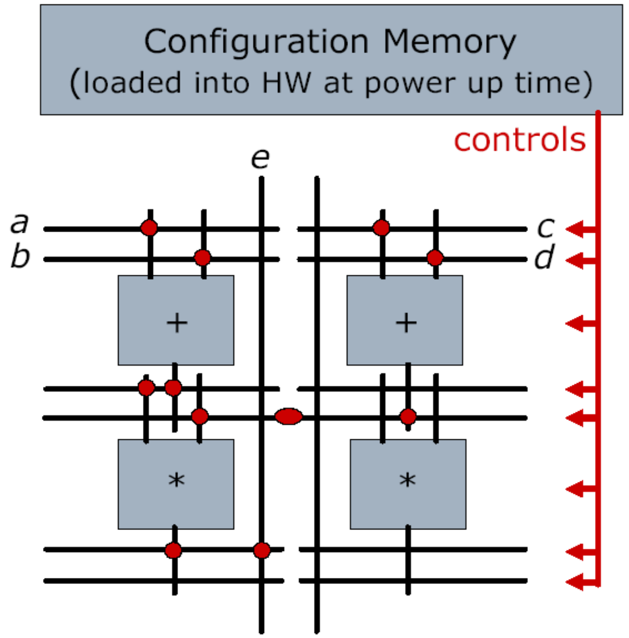
- Field Programmable Gate Arrays (FPGAs) are programmable semiconductor devices that are based around a matrix of configurable logic blocks connected via programmable interconnects.
- Compared to normal microprocessors, where the design of the device is fixed by the manufacturer, FPGAs can be programmed and precisely configured to compute the exact algorithm(s) required by any given application.
- This makes FPGAs very powerful and versatile because **it enables the programmer to change the computer to fit the algorithm** (which enables huge performance gains), compared the other way round in Intel world.
- **So, the calculations per Watt of power consumed, per metre^3 are between 100 and 1000 times better for an FPGA compared to a standard Intel core.**

FPGA-102

J.P.Morgan

- An algorithm is implemented as a special configuration of a general purpose electric circuit.
- Connections between prefabricated wires are programmable
- Function of calculating elements is itself programmable
- FPGAs are two dimensional matrix-structures of configurable logic blocks (CLBs) surrounded by input/output blocks that enable communication with the rest of the environment.

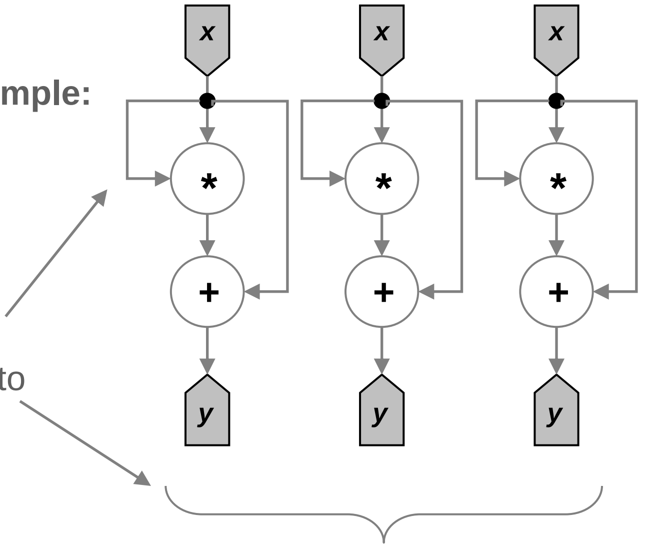
A slightly more complex example:
$e=(a+b)*(c+d)$



A very simple example:

$f(x) = x^2 + x$

Moving from a single calculation to fine-grained parallelism

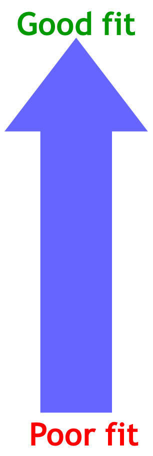


Migrating algorithms from C++ to FPGAs involves doing a Fourier Transform from time domain execution to spatial domain execution in order to maximise computational throughput – **it's a paradigm shift to stream computing that provides acceleration of up to 1,000x compared to an Intel CPU.**

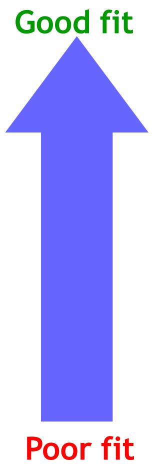
Suitability for migration

J.P.Morgan

The general conclusion from our initial review of existing research was:



GPUs	FPGAs
No interdependence in the data flow and the computation can be done in parallel – e.g. survival curve bootstrapping	Computation involves lots of detailed low-level hardware control operations which can not be efficiently implemented in a high level language – e.g. bespoke tranche valuation
Applications contain a lot of parallelism but involve computations which cannot be efficiently implemented on GPUs	A certain degree of complexity is required and the implementation can take advantage of data streaming and pipelining
Applications have a lot of memory accesses and have limited parallelism	Applications that require a lot of complexity in the logic and data flow design – e.g. Gaussian elimination



Source: "Accelerating compute intensive applications with GPUs and FPGAs", Che, Li, Shaeffer, Skadron and Lach, University of Virginia, 2008.

So, based on the above findings we decided on the following acceleration approach for credit hybrids:

- FPGAs – use for tranche evaluation.
- GPUs – use for default/survival curve generation.

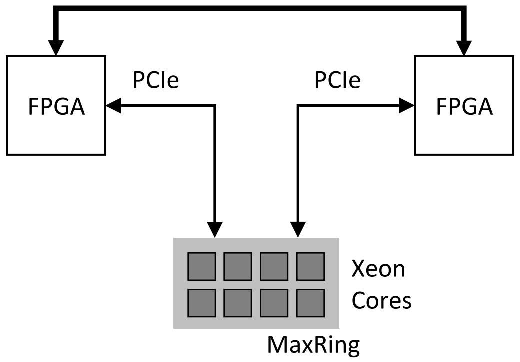
But, it is worth bearing in mind that FPGAs can also be made to run simple algorithms (e.g. survival curve generation) very fast.

Hybrid computer design

J.P.Morgan

Specifications:

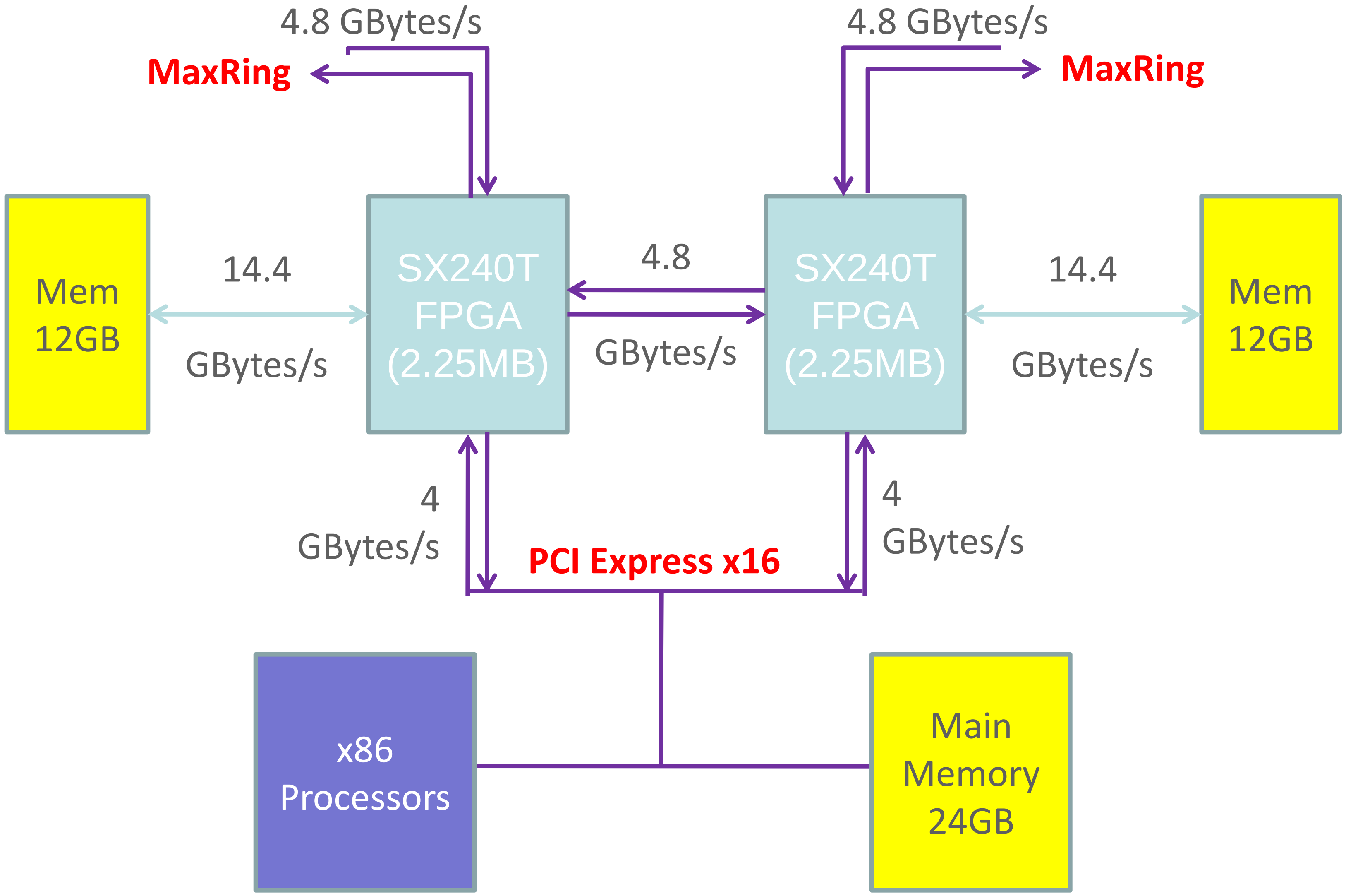
Compute	8x 2.66GHz Xeon Cores 2x SX240T FPGAs
Interconnect	PCI-Express Gen. 1 MaxRing Gigabit Ethernet
Storage	2x 1TB Hard disks
Memory	48GB DRAM
Form Factor	1U
Machine	40U



MAX2 Node Architecture

Hybrid compute node design

Bandwidth on MAX2 cards

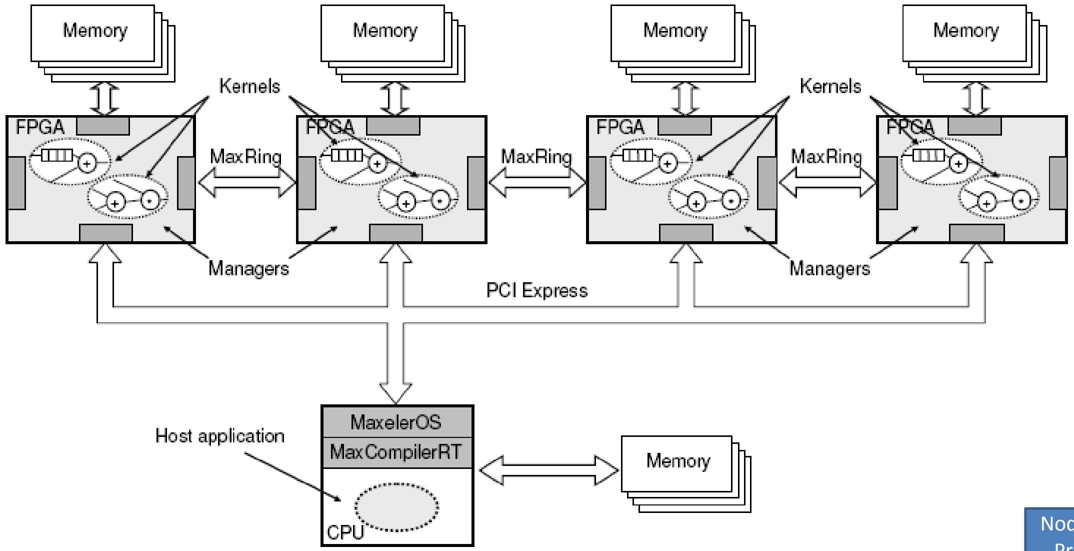


Hybrid compute node architecture

J.P.Morgan

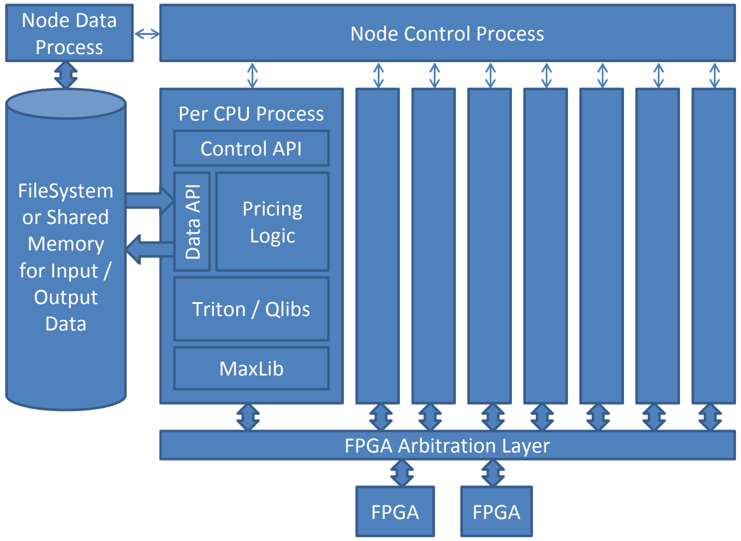
Python API is used to specify pricing streams and manage data I/O.

How the FPGA machine is called:



- FPGAs are statically scheduled, so computation and data I/O both have predictable performance.
- Our approach is based on hardware platform ***and*** MaxelerOS configuration options.
- Since computation + data movement costs are known, performance can be predicted very accruately for specified hardware architecture

Compute node architecture

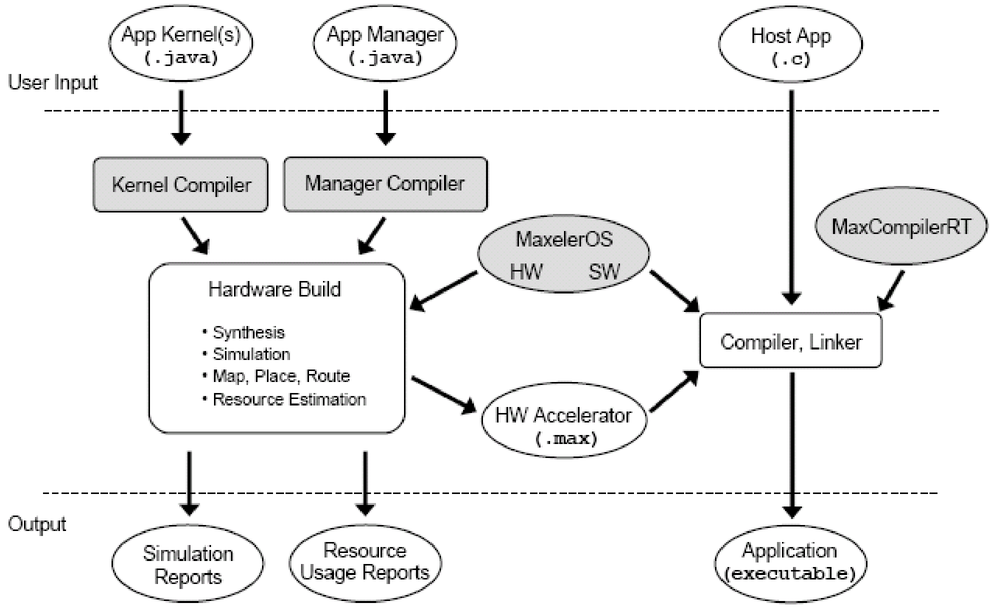


1

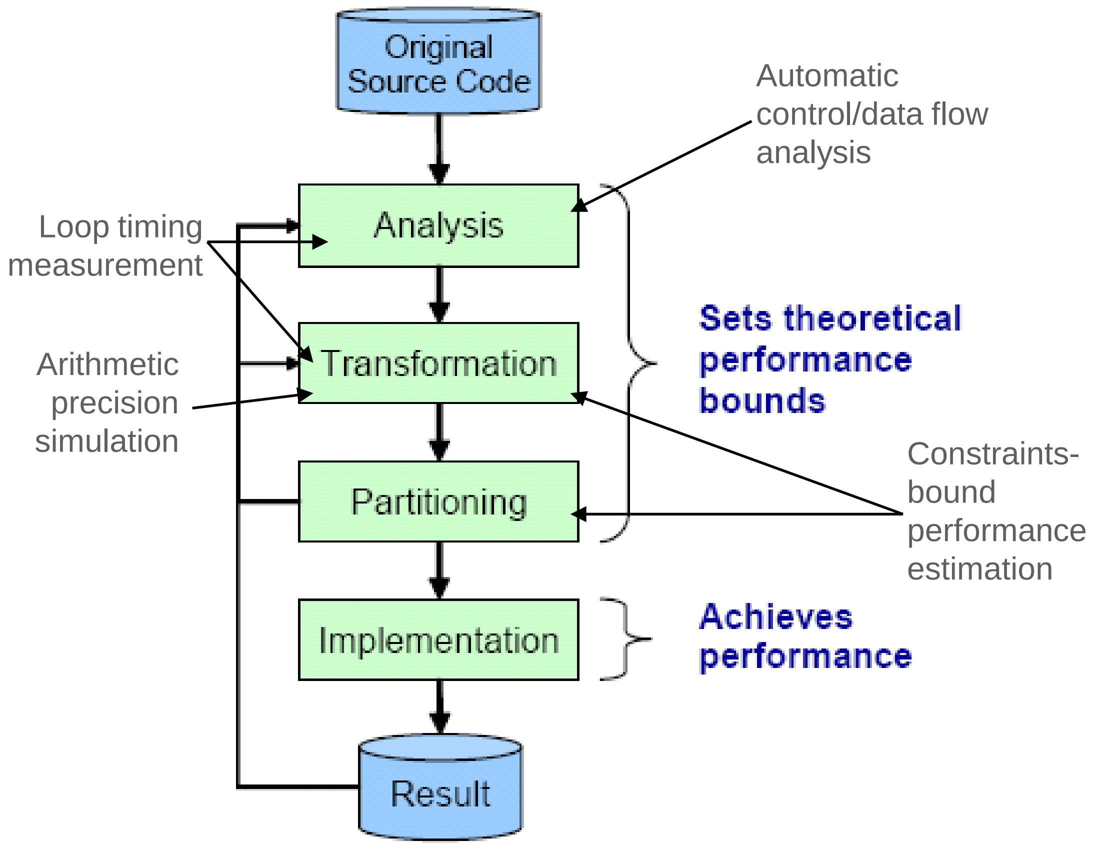
Manager-kernel interaction

J.P.Morgan

- Traditional HPC processors are designed to optimize latency, but not throughput.
- Structured arrays such as FPGAs, offer complementary throughput acceleration.
- Properly configured, an array can provide 10x-100x improvement in arithmetic or memory bandwidth by streaming.
- Some distinctions:
  - Memory limited vs. compute limited
  - Static vs. dynamic control structure
  - Homogeneous v core + accelerator at the node



Iterative acceleration process


Original Source Code
Analysis
Transformation
Partitioning
Implementation
Result
Loop timing measurement
Arithmetic precision simulation
Automatic control/data flow analysis
Sets theoretical performance bounds
Constraints-bound performance estimation
Achieves performance

- When designing the kernels it is critical to allocate time early in the design schedule to decide what logic to implement in software that runs on the CPU and what to implement on the FPGA.
- The MaxParton tool is used to profile and partition the source code for acceleration prediction.
- Clearly the timing-critical portions of the design should go in the faster FPGA while non-critical portions should go in the processor; however this partitioning is not always so black and white and the final design is usually the result of iteration.
- The iterative process covers algorithm, architecture, and arithmetic level of optimisation.

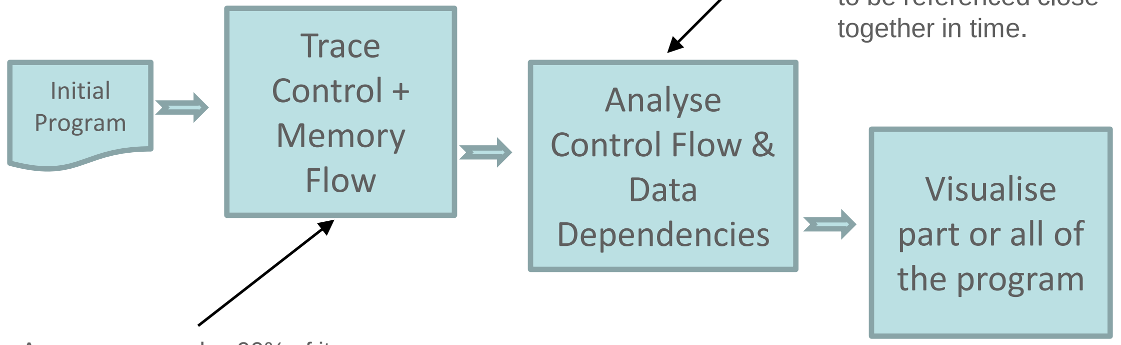
Iterative acceleration process

J.P.Morgan

Partitioning source code is carried out using the MaxParton tool suite consisting of the following tools:

- Multithreaded lightweight timer
- Arithmetic precision simulator
- Automatic control / data flow analyser
- Constraints-bound performance estimator

- MaxParton enables the developer to
  - Carry out run-time analysis of compiled software
  - Automatically analyse impenetrable object-oriented C++ software.
  - Traces data dependencies between functions and libraries
- Using MaxParton is a three stage process
  - Trace
  - Analyse
  - Visualise


Initial Program
Trace Control + Memory Flow
Analyse Control Flow & Data Dependencies
Visualise part or all of the program

Analyse locality:
Temporal locality: recently accessed items are likely to be accessed in the near future.
Spatial locality: items whose addresses are near one another tend to be referenced close together in time.

A program spends ~90% of its time in 10% of its code

Iterative acceleration process

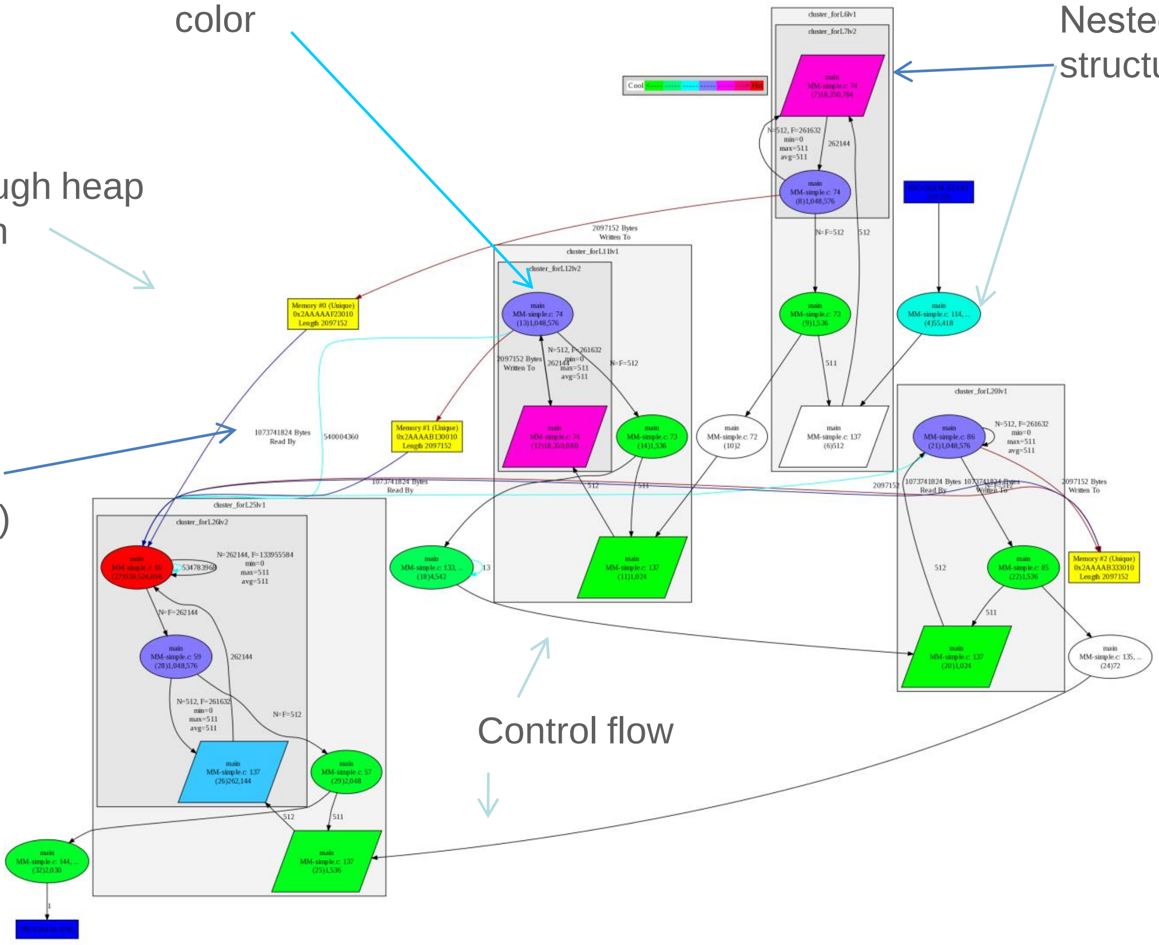
Hot-spots identified by color

Nested 'for' loop structures

Dependency through heap memory allocation

Data flow (size in bytes)

Control flow



cluster_forL6lv1
cluster_forL7lv2
main MM-simple.c 74 (7)18,350,764
Cool
main MM-simple.c 74 (8)1,048,576
N=512, F=261632 min=0 max=511 avg=511
262144
2097152 Bytes Written To
cluster_forL11lv1
cluster_forL12lv2
Memory #0 (Unique) 0x2AAAAAF23010 Length 2097152
main MM-simple.c 74 (13)1,048,576
main MM-simple.c 73 (9)1,536
main MM-simple.c 114, .. (4)55,418
1073741824 Bytes Read By
540004360
Memory #1 (Unique) 0x2AAAAB130010 Length 2097152
main MM-simple.c 74 (12)18,350,080
main MM-simple.c 73 (14)1,536
main MM-simple.c 72 (10)2
main MM-simple.c 137 (6)512
cluster_forL20lv1
main MM-simple.c 86 (21)1,048,576
cluster_forL25lv1
cluster_forL26lv2
main MM-simple.c 80 (27)539,524,096
N=262144, F=133955584 min=0 max=511 avg=511
main MM-simple.c 133, .. (18)4,542
main MM-simple.c 137 (11)1,024
main MM-simple.c 85 (22)1,536
Memory #2 (Unique) 0x2AAAAB333010 Length 2097152
main MM-simple.c 59 (28)1,048,576
main MM-simple.c 137 (20)1,024
main MM-simple.c 135, .. (24)72
main MM-simple.c 137 (26)262,144
main MM-simple.c 57 (29)2,048
main MM-simple.c 144, .. (32)2,030
main MM-simple.c 137 (25)1,536

Iterative acceleration process

J.P.Morgan

MaxParton allows the developer to:

- Evaluate partitioning of Storage in:
  - Host memory (~96GB)
  - FPGA BRAM Memory (2-4.7MB)
  - DDR Memory (12-24GB)
- Assign bandwidths based on platform:
  - FPGA BRAM Memory (~ 9 TB/s)
  - DDR Memory (~40 GB/s)
  - PCI-Express (1-4 GB/s)
  - MaxRing (1-4 GB/s)
- Evaluate placing computation on:
  - CPU
  - FPGA

- To determine and predict highest performance partitioning option subject to Amdahl's Law:

$$OverallSpeedup = \frac{1}{(1-F)+\frac{F}{S}}$$

where F = fraction of the code enhanced and S = the speedup of enhanced fraction. You will notice that the best overall speedup that can be achieved according to this Law is 100% of the speedup of the enhanced fraction as you approach 100% of code enhanced.

Valuation of tranched CDOs

J.P.Morgan

- So, how is the tranche valuation code mapped into FPGA code?
- The base correlation with stochastic recovery model is used to value tranche based products.
- At its core, the model involves two key computationally intensive loops:
  - Constructing the conditional survival probabilities using a Copula
  - Constructing the probability of loss distribution using convolution.

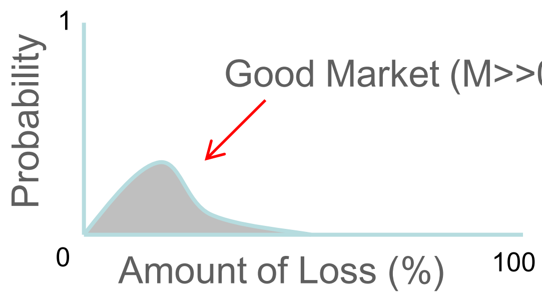
Unconditional Survival Probability for this Name

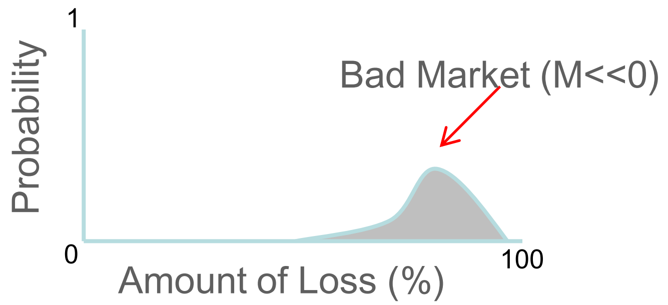
Market factor

$$g_\rho(p_i, u) = \mathcal{N}\left(\frac{\mathcal{N}^{-1}(p_i) - \sqrt{\rho}\,M}{\sqrt{1-\rho}}\right)$$

Conditional Survival Probability for this Name

Correlation





Valuation of tranched CDOs

J.P.Morgan

After removing the C++ class hierarchy, the “flattened” C-code looks like this:

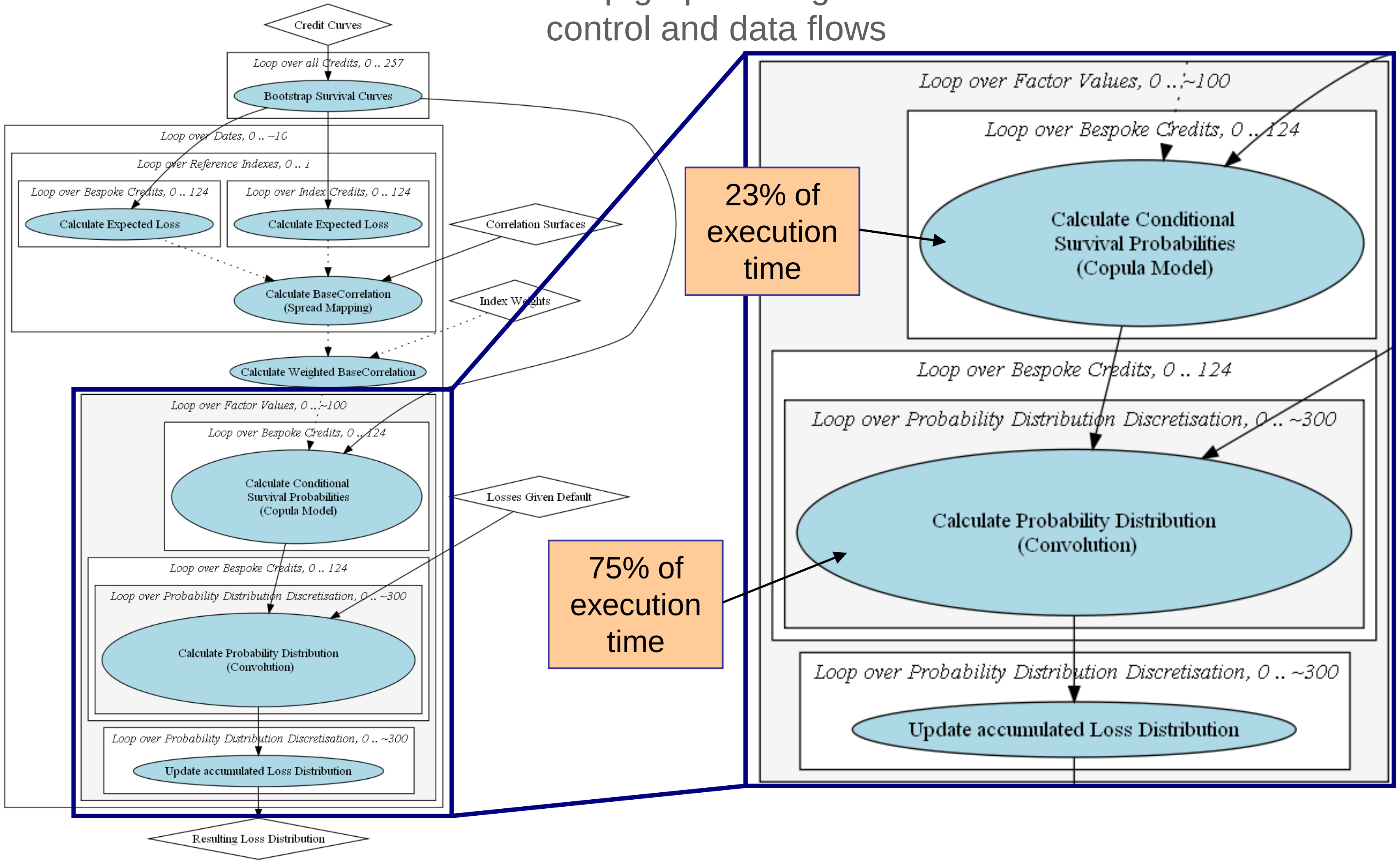
```
for i in 0 ... markets-1
  for j in 0 ... names-1
    prob=cum_norm((inv_norm(Q[j])-sqrt(p)M)/sqrt(1-p);
    loss=calc_loss(prob,Q2[j],RR[j],RM[j])*notional[j];
    n = integer(loss);
    L = fractional(loss);
    for k in 0 ... bins-1
      if j == 0
        dist[k] = k == 0 ? 1.0 : 0.0;

      dist[k] = dist[k]*(1-prob) +
                dist[k-n]*prob*(1-L) +
                dist[k-n-1]*prob*L;
      if j == credits -1
        final_dist[k] += weight[i] * dist[k];

    end # for k
  end # for j
end # for i
```

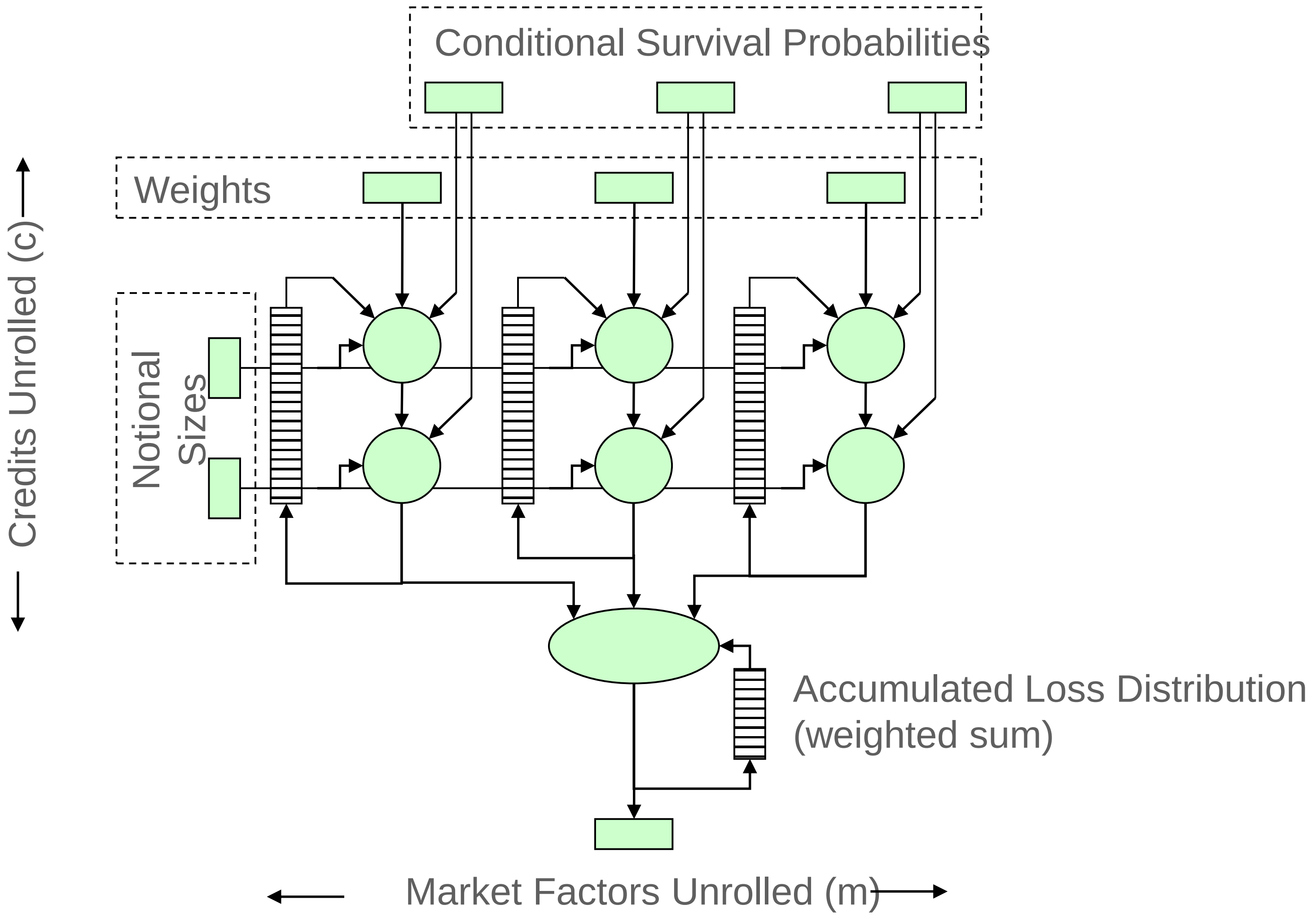
Kernel design

Loop graphs are generated from MaxParton to visualise control and data flows


Credit Curves
Loop over all Credits, 0 .. 257
Bootstrap Survival Curves
Loop over Dates, 0 .. ~10
Loop over Reference Indexes, 0 .. i
Loop over Bespoke Credits, 0 .. 124
Calculate Expected Loss
Loop over Index Credits, 0 .. 124
Calculate Expected Loss
Correlation Surfaces
Calculate BaseCorrelation (Spread Mapping)
Index Weights
Calculate Weighted BaseCorrelation
Loop over Factor Values, 0 .. ~100
Loop over Bespoke Credits, 0 .. 124
Calculate Conditional Survival Probabilities (Copula Model)
Losses Given Default
Loop over Bespoke Credits, 0 .. 124
Loop over Probability Distribution Discretisation, 0 .. ~300
Calculate Probability Distribution (Convolution)
Loop over Probability Distribution Discretisation, 0 .. ~300
Update accumulated Loss Distribution
Resulting Loss Distribution
23% of execution time
75% of execution time

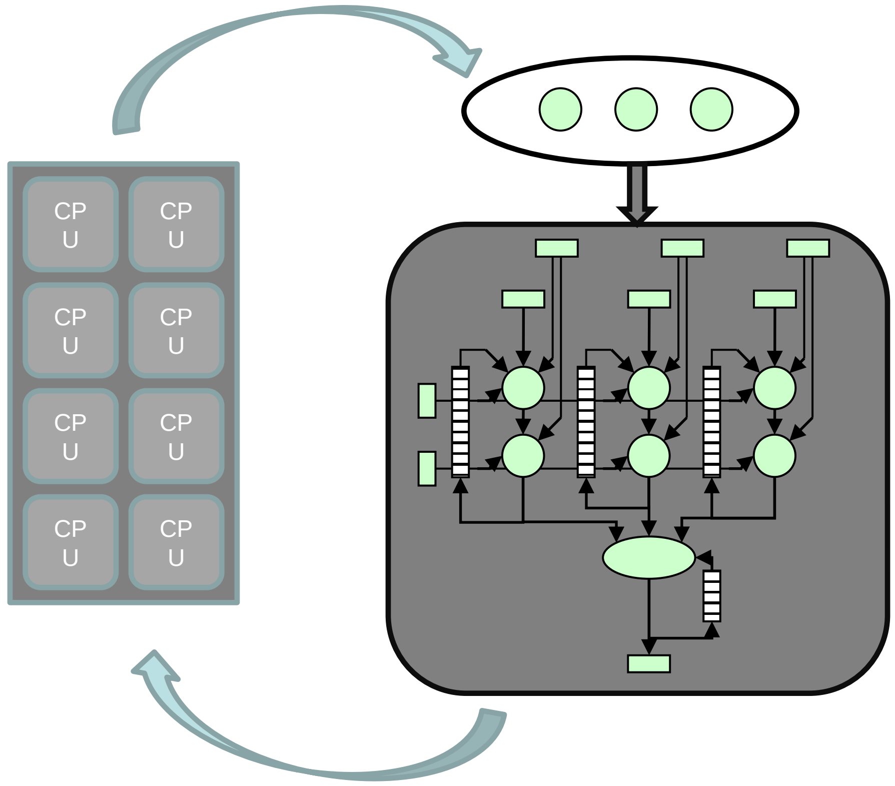
Loop graph output identify two kernels for bitstream design – copula and convolution

Convoluter Design


Conditional Survival Probabilities
Weights
Notional Sizes
Credits Unrolled (c)
Accumulated Loss Distribution (weighted sum)
Market Factors Unrolled (m)

Architecture

The architecture of the bitstream is derived directly from loop graph, enabling relatively straight forward construction of the Java code which is then compiled



Original code

The first task in the code migration is to remove all use of classes, templates and other C++ features in order to simplify parallelisation because **abstraction kills parallelism** (Prof Paul Kelly, Imperial College)

Flattened C code:

```
for i in 0 ... markets-1
  for j in 0 ... names-1
    prob=cum_norm((inv_norm(Q[j])-sqrt(p)M)/sqrt(1-p);
    loss=calc_loss(prob,Q2[j],RR[j],RM[j])*notional[j];
    n = integer(loss);
    L = fractional(loss);
    for k in 0 ... bins-1
      if j == 0
        dist[k] = k == 0 ? 1.0 : 0.0;

      dist[k] = dist[k]*(1-prob) +
                dist[k-n]*prob*(1-L) +
                dist[k-n-1]*prob*L;
      if j == credits -1
        final_dist[k] += weight[i] * dist[k];

    end # for k
  end # for j
end # for i
```

MaxCompiler Java code:

```
HWVar d = io.input("inputDist", _distType);
HWVar p = io.input("probNonzeroLoss", _probType);
HWVar L = io.input("lowerProportion", _propType);
HWVar n = io.input("discretisedLoss", _operType);
HWVar lower = stream.offset(-n-1,-maxBins,0,d);
HWVar upper = stream.offset(-n,-maxBins,0,d);
HWVar o = ((1-p)*d + L*p*lower + (1-L)*p*upper);
io.output("outputDist", _distType, o);
```

Setting the build level instructs MaxCompiler on which stage to run the build process to:

FULL_BUILD: runs complete build process

COMPILE_ONLY: stops after generating the VHDL output

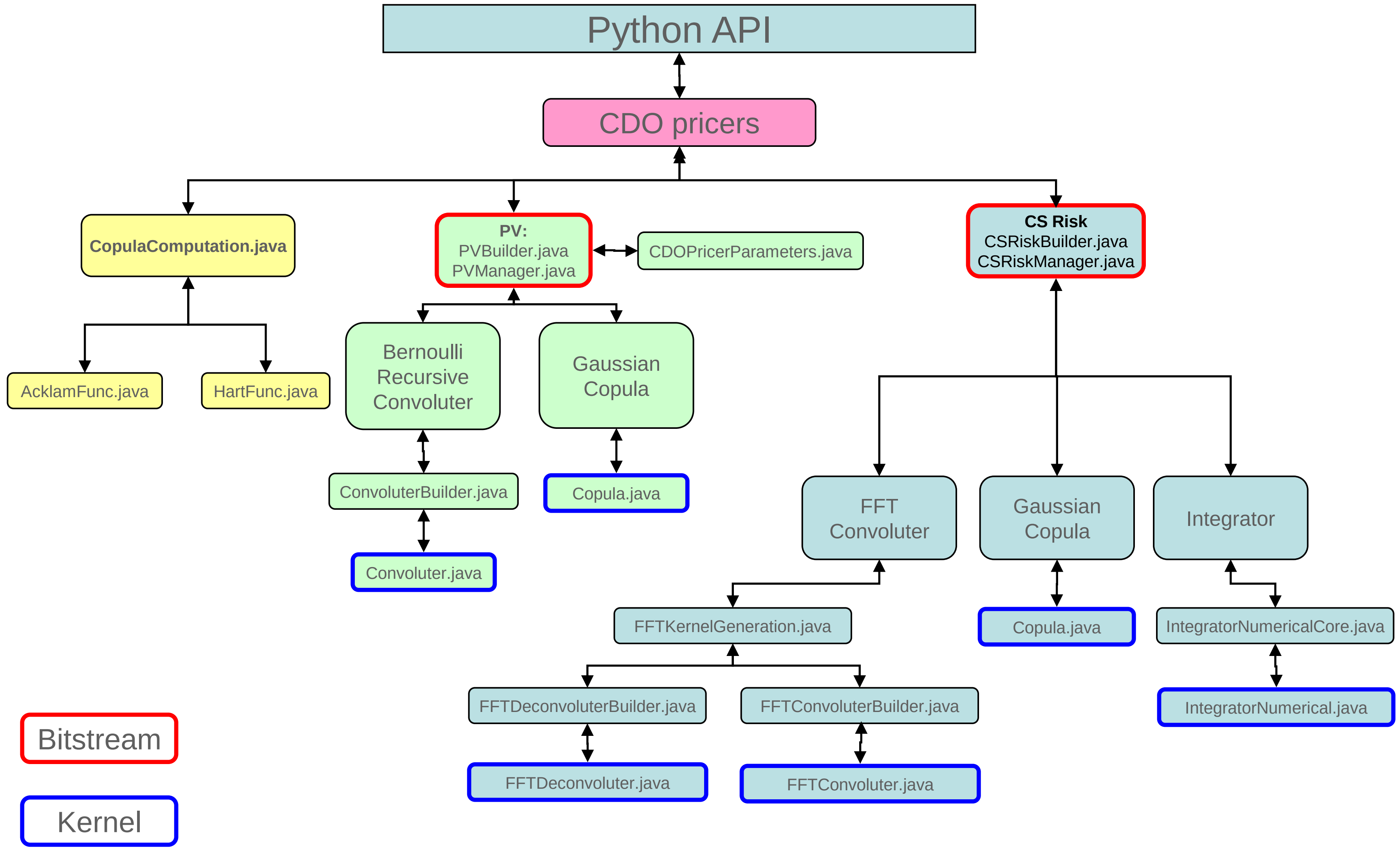
SYNTHESIS: only compiles the VHDL output

MAP: will map the synthesised design to components on the FPGA

PAR: will place and route the design for the FPGA, producing a bitstream

Code design

J.P.Morgan



Code design

J.P.Morgan

- Inside the convolution, a Fourier Transform is used to evaluate the integral.
- Choice of algorithm to implement the Fourier Transform is critical.
- Different algorithms have different characteristics, which need to be considered in relation to the number of available multipliers (known as DSPs) available on the FPGA.
- The following table shows just how variable Fourier Transform algorithms are in terms of their requirements to perform multiplications and additions:

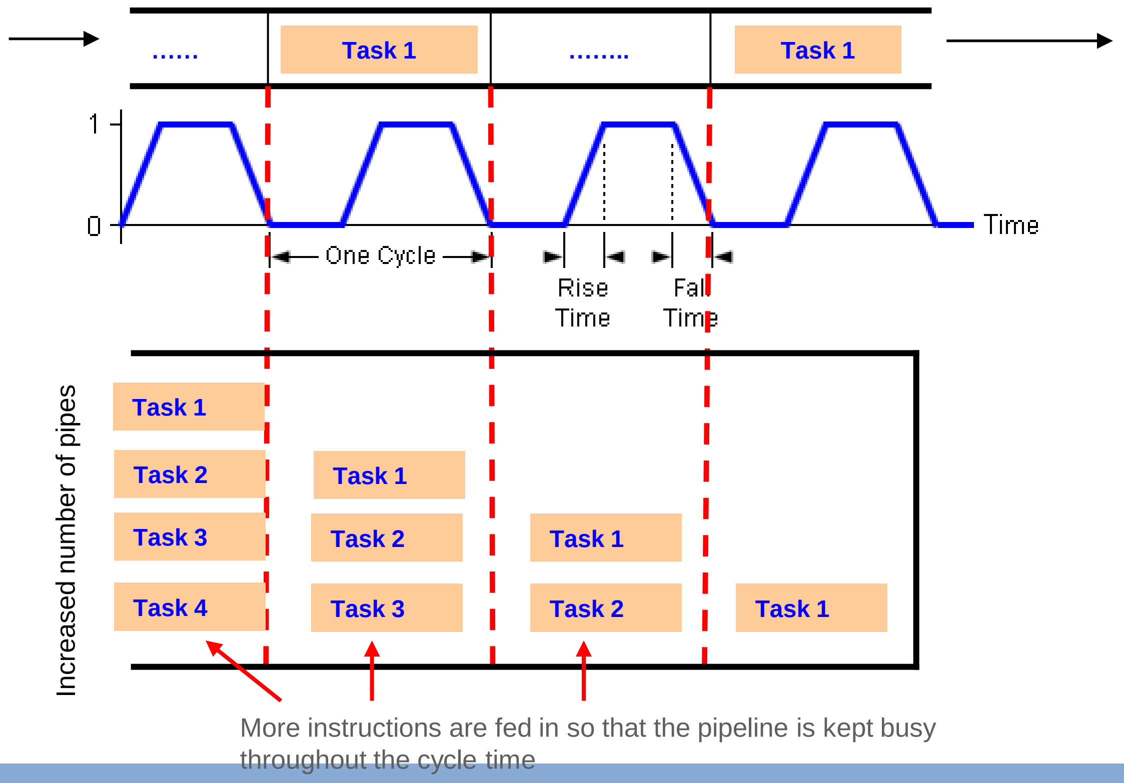
Algorithm	**Multiplications per output grid point**	**Additions per output grid point**
Direct computation of discrete Fourier Transform: 1,000 x 1,000	8,000	4,000
Basic Cooley-Tukey FFT: 1,024 x 1,024	40	60
Hybrid Cooley-Tukey/ Winograd FFT: 1,000 x 1,000	40	72.8
Winograd FFT: 1,008 x 1,008	6.2	91.6
Nussbaumer-Quandalle FFT: 1,008, x 1,008	4.1	79

Notice a factor of ~140x difference in performance between the best and the worst performance

Code design – efficiency is key

J.P.Morgan

- Having designed the basic kernels and the structure of their code, the next stage to the acceleration process is to work out how to fit the maximum number of calculation pipes on to the FPGA chip.
- Pipelining involves overlapping multiple instructions during execution with the objective of minimising the overall run time of the computation.
- Pipelining therefore involves dividing the computation into stages so that each stage can completed as part of an instruction in parallel.
- The stages are then connected together to form a pipe.
- Instructions enter at one end, progress through the stages and exit at the other end.



More instructions are fed in so that the pipeline is kept busy throughout the cycle time

- Note that pipelining does not decrease the time taken for execution of an individual instruction, instead is increases instruction throughput.
- The key to success with pipelining is to ensure that an instruction remains in the pipeline for the minimum amount of time but that its use is maximised during the duration of its stay.

Building the PV manager

J.P.Morgan

```
package com.maxeler.clients.jpmorgan.cdopricers.pv;

import com.maxeler.clients.jpmorgan.cdopricers.pv.bernoullirecursiveconvoluter.Convoluter;
import com.maxeler.clients.jpmorgan.cdopricers.pv.gaussiancopula.Copula;
import com.maxeler.maxcompiler.v1.managers.BuildConfig;
import com.maxeler.maxcompiler.v1.managers.BuildConfig.Level;
import com.maxeler.maxcompiler.v1.managers._BuildOnlyManager;
import com.maxeler.maxcompiler.v1.kernelcompiler.Kernel;
import com.maxeler.maxeleros.managercompiler.facade.ManagerDesign;
import com.maxeler.maxeleros.managercompiler.facade.Stream;
import com.maxeler.maxeleros.managercompiler.facade.blocks.KernelBlock;
import com.maxeler.maxeleros.platforms.BoardCapabilities;


public class PVManager extends ManagerDesign
{
    static protected Hashtable<String, Integer> m_addrTable;
    static protected Hashtable<String, Integer> m_maskTable;
    public CdoPricerParams _params;


    public PVManager(_BuildOnlyManager manager, BoardCapabilities capabilities, String name)
    {
          super(manager, capabilities, name);
    }
    public void setupKernels(Kernel copula, Kernel convoluter, CdoPricerParams params)
    public static void buildHardware(_BuildOnlyManager bm, String name, CdoPricerParams params)
}
```

Building the kernels

J.P.Morgan

```
public void setupKernels(Kernel copula, Kernel convoluter, CdoPricerParams params)
{
      KernelBlock copula_core = addKernel(copula);
      KernelBlock conv_core = addKernel(convoluter);

      config.setNumberOfPCIExpressLanes(4);

      Stream perCredit = addStreamFromHost("perCreditInputBus");
      copula_core.getInput("perCreditInputBus").connect(perCredit);

      for (int i = 0; i < params.getCreditUnrollFactor(); i++)
      {
          conv_core.getInput("credit_input" + i).connect(copula_core.getOutput("copulaOutput" + i));
          conv_core.getInput("prop_lower_input" + i).connect(copula_core.getOutput("propLowerOutput" +i));
      }
      Stream pcieToHost addStreamToHost("pcieToHost");
      pcieToHost.connect(conv_core.getOutput("output_pcie"));
}
```

Use PCIe x4 to reduce latency when streaming – critical for small runs

PCIe going into Copula

Copula to convoluter streams

Building the FFT kernel

J.P.Morgan

- Recall from the kernel design slide that the results of the marginal probability distribution generated by the Copula are fed into the convolution, which is in turn used to calculate the integral which finally produces the accumulated loss distribution.
- Our approach is therefore to build a MultiPiped kernel for convolution, where we build MultiPipe explicitly from the credit data, allowing us to take advantage of various optimisation opportunities.
- We are calculating: $p_0 + p_1 tp_1 + p_2 tp_2$

  where $tp_1 = e^{(2\pi i n k / N)}$ and $tp_2 = e^{(2\pi i (n+1) k / N)}$
- We generate the kernels in pairs, sharing multiplies in order to save resources.
- The bins have been re-arranged such that each pair of adjacent (complex) bins have a phase relationship which is a function of 90', which means we can use muxes to rotate the complex value accordingly.
- We do this rotation after the multiply by p_1 or p_2, halving the number of DSPs needed for kernel generation.
- The tp_1 / tp_2 values for the first 90' of the unit circle are stored in a lookup table (getExp()) and are indexed by $n * k \% (N / 4)$, $(n+1) * k \% (N / 4)$
- Once multiplied they are fed into muxes with the computed phase selecting the correct input.
- To compute the phase for a given pipe, we compute $n * k / (N/4)$ and $(n+1) * k / (N/4)$

Building the FFT kernel

J.P.Morgan

First, we need to define a map that will return tp_1 and tp_2:

```
private Map<String, KComplex> getExp(int i, HWVar tp1Addr, HWVar tp2Addr)
{
        int romSize = _params.getFFTSize() / 4;
        Bits[] expTable = new Bits[romSize];
        for (int nk = 0; nk < romSize; nk++)
        {
             double x = 2 * Math.PI * nk / (_params.getFFTSize());
             expTable[nk]  = _types._kernType.getContainedType().encodeConstant(new
             KComplexType.ConstantValue(Math.cos(x), Math.sin(x)));
        }

        DualPortMemOutputs<KComplex> out = mem.romDualPort(
             tp1Addr,
             tp2Addr,
             _types._kernType.getContainedType(),
             expTable);

        LinkedHashMap<String, KComplex> tpMap = new LinkedHashMap<String,
        KComplex>();
        tpMap.put("tp1", out.getOutputA());
        tpMap.put("tp2", out.getOutputB());
        return tpMap;
}
```

Exponential lookup table
We store the values of

$$e^{(2\pi i n k / N)}$$

for $nk < N/4$ in lookup table in order to compute the kernels for convolution

Create the Mapped ROM

Return result for tp_1 / tp_2

Building the FFT kernel

Set up the kernel generator:

```
public class FFTKernelGeneration extends KernelLib
{
      private final FFTConvoluterTypes _types;
      private final CdoPricerParams _params;
      private final Kernel _design;
      public FFTKernelGeneration(Kernel design, FFTConvoluterTypes types, CdoPricerParams params)
      {
            super(design);
            _types = types;
            _params = params;
            _design = design;
      }
```

Building the FFT kernel

J.P.Morgan

…finally, build the multi-pipe kernel:

```
KMultiPipe<KComplex> makeKernel (KStruct credit, HWVar currentStep)
{
      KMultiPipe<KComplex> kernel = _types._kernType.newInstance(_design);
      HWVar prob = ((HWVar)credit.get("prob_non_zero_value"));
      HWVar propLower = ((HWVar)credit.get("proportion_lower"));
      HWVar n = (HWVar)credit.get("operator_size");

      if (!_types._isFixedPoint) optimization.pushEnableBitGrowth(false);
      HWVar p0_orig = (1 - prob).cast(_params.getFpgaRealType());
      HWVar p2_orig = (prob * propLower).cast(_params.getFpgaRealType());
      HWVar p1_orig = (prob - p2_orig).cast(_params.getFpgaRealType());

      if (!_types._isFixedPoint) optimization.popEnableBitGrowth();
      KComplex p1tp1 = null, p2tp2 = null;
      KMultiPipe<KComplex> new_p1tp1 = _types._kernType.newInstance(_design);
      KMultiPipe<KComplex> new_p2tp2 = _types._kernType.newInstance(_design);
      HWVar nk = null;
      HWVar nkm = null;
      HWVar nkd = null;
      HWVar np1km = null;
      HWVar np1kd = null;
      HWVar phaseN = null, phaseNp1 = null;
      HWVar p0 = null;
      HWVar p1 = null;
      HWVar p2 = null;

      ….. Continued….
```

`makeKernel()` - builds a MultiPiped kernel for convolution. We build MultiPipe explicitly here from the credit data, which takes advantage of various optimisation opportunities

Building the FFT kernel

J.P.Morgan

```
….. Continued….
for (int i = 0; i < _types._kernType.getNPipes() ; ++i)
{
      if ((i % 4) == 0)
      {
            p0 = optimization.limitFanout(p0_orig, 4096);
            p1 = optimization.limitFanout(p1_orig, 4096);
            p2 = optimization.limitFanout(p2_orig, 4096);
      }
      FixOperatorFactory phaseCalcOpFactory = new FixOperatorFactory();
      phaseCalcOpFactory.setMultDSPUsage(DSPUsage.LOW);
      phaseCalcOpFactory.setPreventCoregenUsage(true);

      if (i % 2 == 0)
      {
            HWVar k = currentStep + (1 + i / 2);
            optimization.pushEnableBitGrowth(true);
            if (i == 0)
                  nk = n * k;
            Else
                  nk = nk + n;
            HWVar np1k = nk + k;
            optimization.popEnableBitGrowth();
            int modBits = MathUtils.bitsToAddress(_params.getFFTSize() / 4);
            int sliceBits = 2;
            nkm = nk.cast(hwUInt(modBits));
            nkd = nk.slice(modBits, sliceBits).cast(hwUInt(sliceBits));
            np1km = np1k.cast(hwUInt(modBits));
            np1kd = np1k.slice(modBits, sliceBits).cast(hwUInt(sliceBits));
            phaseN = nkd;
            phaseNp1 = np1kd;
            Map<String, KComplex> expMap = getExp(i / 2, nkm, np1km);
            KComplex tp1 = expMap.get("tp1");
            KComplex tp2 = expMap.get("tp2");

      …..continued….
```

Fanout reducing registers Re-register $p_0 - 2$ every few pipes

Building the FFT kernel

J.P.Morgan

```
….. Continued….
      optimization.pushEnableBitGrowth(false);
      p1tp1 = (tp1 * p1).cast(_types._kernType.getContainedType());
      p2tp2 = (tp2 * p2).cast(_types._kernType.getContainedType());
      optimization.popEnableBitGrowth();
      }
      else
      {
            // Mapped on the unit circle
            optimization.pushEnableBitGrowth(true);
            nkd = (nkd + n);
            np1kd = (np1kd + n + 1);
            optimization.popEnableBitGrowth();
            phaseN = nkd.slice(0, 2).cast(hwUInt(2));
            phaseNp1 = np1kd.slice(0, 2).cast(hwUInt(2));
      }
      new_p1tp1.connect(i, ArithOpt.complexRotate(p1tp1, phaseN));
      new_p2tp2.connect(i, ArithOpt.complexRotate(p2tp2, phaseNp1));
}
new_p1tp1.watch("newp1tp1");
new_p2tp2.watch("newp2tp2");
KComplex p0Complex = _types._kernType.getContainedType().newInstance(this);
p0Complex.setReal(p0);
p0Complex.setImaginary(p0Complex.getImaginary().getType().newInstance(this, 0.0));
KMultiPipe<KComplex> p0CompMP = _types._kernType.newInstance(this, p0Complex);
optimization.pushEnableBitGrowth(false);
kernel = TriAdd.add3(p0CompMP, new_p1tp1, new_p2tp2).cast(_types._kernType);
optimization.popEnableBitGrowth();
return kernel.watch("kernel");            }
}
```

These are the kernels in the lower left section of the k*n plane. We need to map these to the other 7 sections

Building the hardware

J.P.Morgan

```
public static voidbuildHardware(_BuildOnlyManager bm, String name, CdoPricerParams
    params)
{
      BoardCapabilities caps = params.getBoard();
      PVManager manager = new PVManager(bm, caps, name);
      Copula copula = new Copula(bm, params);
      Convoluter convoluter = new Convoluter(bm, params);

      manager.setupKernels(copula, convoluter, params);
      manager.config.setStreamClockFrequency(params.getFrequency());
      manager.config.setApplicationRevisionIds(params.getAppId(), params.getRevId());

      BuildConfig build_config = new BuildConfig(Level.FULL_BUILD);
      build_config.setBuildEffort(params.getMPPREffort());
      build_config.setMPPRCostTableSearchRange(params.getMPPRStartCT(),params.getMPPREndCT());
      build_config.setMPPRParallelism(params.getMPPRThreads());
      build_config.setEnableTimingAnalysis(true);
      build_config.setMPPRRetryNearMissesThreshold(params.getMPPRRetryThreshold());
      bm.setBuildConfig(build_config);
      bm.setRootEntity(manager.build());
      bm.build();
}
```

Resource usage

MaxCompiler provides detailed information on how much of the available FPGA resources have been used by any given kernel. The information on this slide is for the PV kernel which combines the Copula and Convolution-Integration kernels, running across 100 pipes at 200Mhz on a single FPGA chip:

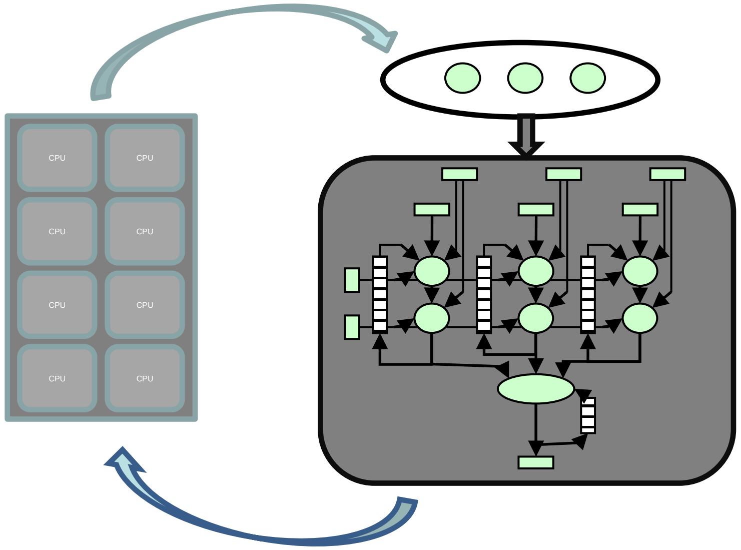
Computation	Software
Copula Kernel	22.00%
Convolution and Integration	56.10%

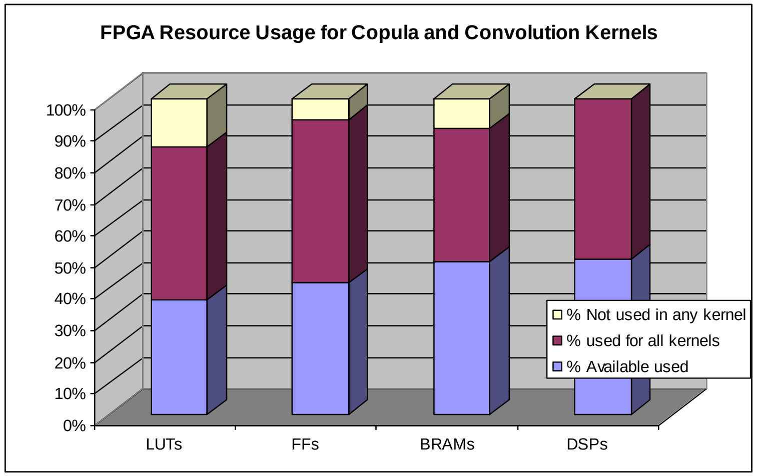
Summary of FPGA Resources

LUTs	FFs	BRAMs	DSPs	Comment
149,760	149,760	516	1,056	Total available resources for FPGA
85,391	106,778	484	1,020	Total resources used
57.02%	71.30%	93.80%	96.59%	% of available
65,078	94,567	396	1,020	Total for all designs
76.21%	88.56%	81.82%	100.00%	% of available
20,313	12,211	88	-	Resources not associated with any design
23.79%	11.44%	18.18%	0.00%	%

Detail for each kernel

LUTs	FFs	BRAMs	DSPs	Comment
37,987	53,622	276	800	BernoulliRecursiveConvoluter - total
44.49%	50.22%	57.02%	78.43%	%
23,466	36,802	270	800	BernoulliRecursiveConvoluter - user
27.48%	34.47%	55.79%	78.43%	%
14,047	14,399	6	-	BernoulliRecursiveConvoluter - scheduling
16.45%	13.48%	1.24%	0.00%	%
474	2,421	-	-	BernoulliRecursiveConvoluter - other kernel
0.56%	2.27%	0.00%	0.00%	%
27,091	40,945	120	220	CopulaFix - total
31.73%	38.35%	24.79%	21.57%	%
24,179	27,828	119	220	CopulaFix - user
28.32%	26.06%	24.59%	21.57%	%
2,771	11,951	1	-	CopulaFix - scheduling
3.25%	11.19%	0.21%	0.00%	%
141	1,166	-	-	CopulaFix - other kernel
0.17%	1.09%	0.00%	0.00%	%





Handling errors

J.P.Morgan

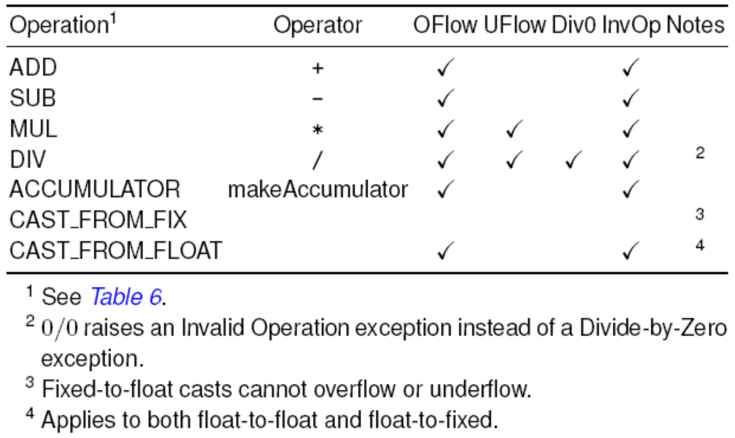
During execution, errors can arise in three ways:

- API calls
  - Python API wraps all errors in try-catch
- Triton library calls
  - Triton exception handling will pass errors
- MaxCompiler allows the user to optimise the numerical behaviour of kernel operations through two features:
  - Numeric exceptions (such as overflow) which allow the user to see numeric **exceptions** that occurred for **which operations**.
  - Doubt – which is a feature unique to MaxCompiler which allows the developer to see which **data** have been affected by a **numeric exception**.
- Together these features allow the developer to detect and recover from all numeric exceptions generated in a Kernel.

Handling errors

Handling errors in kernel operations

- Arithmetic operations in Kernel designs have the capability of raising numeric exceptions.
- Numeric exceptions cost extra logic on the device, so are disabled by default.
- Enabling numeric exceptions is helpful during the design process to debug any numerical issues.
- Numeric exceptions are raised in similar circumstances to a CPU but the Kernel always continues processing, raising a flag to indicate that a numeric exception has occurred.
- For floating-point numbers, the type of numeric exceptions that can be raised closely follow the IEEE 754 standard.
- For fixed-point numbers, overflow and divide-by-zero exceptions can be raised.
- The following table summarises the errors supported by MaxCompiler:



Operation[1]	Operator	OFlow	UFlow	Div0	InvOp	Notes
ADD	+	✓			✓	
SUB	−	✓			✓	
MUL	*	✓	✓		✓	
DIV	/	✓	✓	✓	✓	2
ACCUMULATOR	makeAccumulator	✓			✓	
CAST_FROM_FIX						3
CAST_FROM_FLOAT		✓			✓	4

[1] See *Table 6*.
[2] $0/0$ raises an Invalid Operation exception instead of a Divide-by-Zero exception.
[3] Fixed-to-float casts cannot overflow or underflow.
[4] Applies to both float-to-float and float-to-fixed.

Accuracy

J.P.Morgan

- MaxCompiler supports floating-point data streams both in IEEE 754 standard formats (half-, single- anddouble-precision) and with user-specified sizes of mantissa and exponent.
- A floating-point type is parameterized with mantissa and exponent bit-widths in MaxCompiler using the function hwFloat:
- `HWFloat hwFloat(int exponent bits, int mantissa bits)`
- Double precision is thus `hwFloat(11, 53),`with an 11 bit exponent and 53 bit mantissa.
- As the exponent and mantissa can be defined at compile time, it is therefore possible to build bitstreams with varying degrees of accuracy as required.
- This is useful, since double precision accuracy is not an absolute requirement throughout every part of a computation.
- As accuracy is reduced, performance increases and FPGA resource use declines.
- Having the ability to build bitstreams with varying degrees of accuracy is extremely useful, as lower precision bitstreams can be used for scenario analysis, where it can be acceptable to trade-off speed in favour of absolute accuracy.
- The potential speedups can be as much as 20% for every decimal place of accuracy sacrificed.
- Several lower accuracy bitstreams have been built for PV and can be run when as an when speed is preferred over accuracy.

Testing

J.P.Morgan

Test bench steps:

- Code coverage – how much RTL has been simulated?
  - Statements – were all executed?
  - Branch – were all branches taken?
  - Condition – were all conditions tested?
  - Expression – were all parts of concurrent assignments tested?
  - Finite state machine – were all states and transitions tested?
- Test planning – improve the speed of verification – have a plan!
- Assertions – catching bugs at source
  - Use of the **assert** statement
  - Multi-cycle assertions
  - Placing assertions
- Transaction-level Simulation – create tests and check results
- Self-checking test bench - automation of transaction-level testing
- Automatic stimulus -
- Functional coverage

Debugging

J.P.Morgan

- The primary tool for debugging kernels in simulation is a **`watch`**.
- Watches allow the developer to see what is going on inside the kernel by tracking the value of any **`HWVar`** that has been tagged for **`watch`** during every cycle that the kernel is running.
- Debugging a kernel involves adding the **`watch`** method to any number of target streams.
- Debug output is generated by running the target kernel in simulation mode, which causes a .csv file to be generated containing data for every variable upon which a watch has been placed, e.g.

```
HWVar x = io.input(”x”, hwFloat(8, 24));
x.watch(”x”) ;
// Data
HWVar x prev = stream.offset(x, −1);
HWVar x next = stream.offset(x, 1);
```

- Recall that FPGAs are statically scheduled, so there is no need for dynamic debugging, so .csv output is adequate for finding and fixing bugs.

Code validation

J.P.Morgan

One of the concerns raised around migrating models to work on FPGAs is the degree to which the resulting calculation is an accurate representation of the original model.

Assume that we can measure the predictive error of the migrated code like this:

$$e = e_1 + e_2 + e_3$$

$$e_1 = \left(y_{C++} - y_{FPGA}\right)$$

Confirming e1 is small is fundamental **Test**

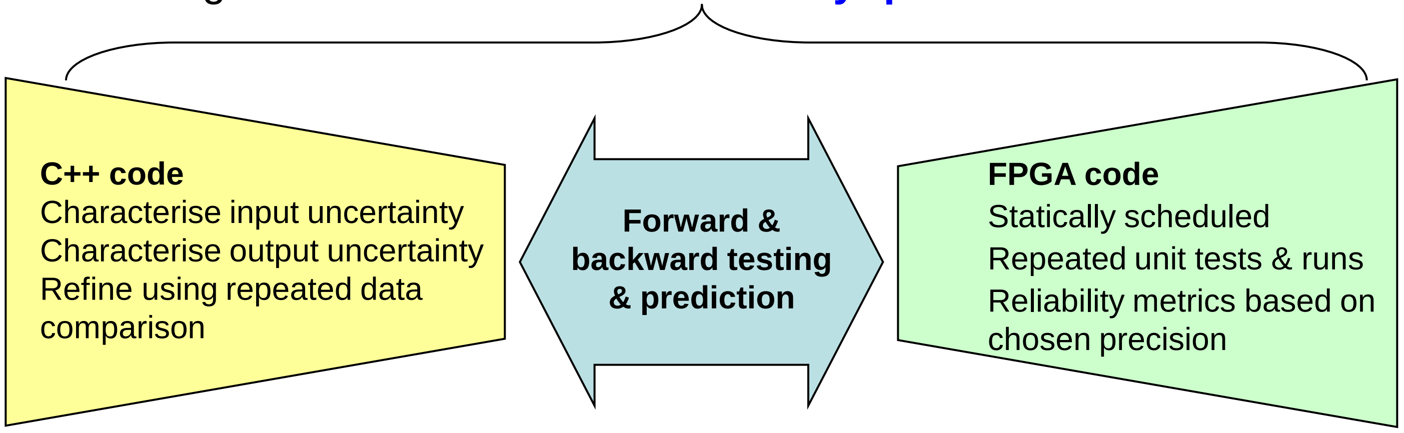
$$e_2 = \left(y_{FPGA} - y_{\mathrm{Pr}oteus}\right)$$

Confirming e2 is small is verification problem **Test**

$$e_3 = \left(y_{\mathrm{Pr}oteus} - y_{\mathrm{Pr}ecise}\right)$$

Confirming e3 is small is a validation problem **Test!**

Then the remaining concern is therefore **uncertainty quantification**:



C++ code
Characterise input uncertainty
Characterise output uncertainty
Refine using repeated data comparison

Forward & backward testing & prediction

FPGA code
Statically scheduled
Repeated unit tests & runs
Reliability metrics based on chosen precision

Acceleration of tranche risk

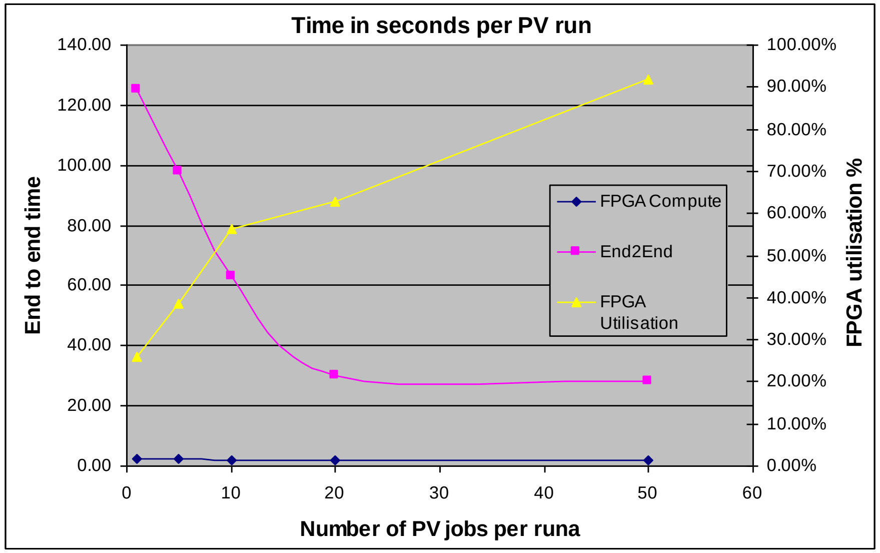
J.P.Morgan

- Most advanced thread of acceleration work – ~2 years effort.
- Migration of the production model (base correlation with stochastic recovery) used to price and calculate risk for: vanilla tranches, bespoke tranches, n-th to default and CDO^2 (together accounting for ~98% of compute).
- Currently a single FPGA prices a single complex trade 134x faster than a single CPU.
- End-to-end time to price global credit hybrids portfolio once, reduced to ~125secs with pure FPGA time of ~2 secs to price ~30,000 tranches and total compute time of ~30 secs.
- End to end time for pointwise credit deltas on global credit hybrids portfolio reduced to ~238 secs with pure FPGA time of ~12 secs, using a 40-node FPGA machine.
- Running multiple trading/risk scenarios for desk (example shown below of 5 multi-name default scenarios affecting 122 names in different combinations) – total end-to-end time of ~320 secs, results accurate to within $5 across global portfolio. Not previously possible to run such scenarios multiple times within a single trading day.

Acceleration of tranche risk

J.P.Morgan

- We can also run complex scenarios such as one that defaults all of the 2,000+ names in the portfolio (ordered in terms of expected loss) in increasing groups (i.e. name 1, names 1 + 2, names 1 + 2 + 3 etc…) and run it for both market and zero recovery (a total of 4,032 PV jobs) – never previously computationally feasible using standard Intel cores
- The most interesting result from the exercise is that we are gaining an understanding of the shape of the curve that describes the performance trade off as the following graph shows…



Number of Scenarios	FPGA Compute	End2End	FPGA Utilisation
1	2.57	125.21	25.99%
5	2.35	98.02	38.54%
10	2.06	66.68	56.24%
20	1.86	30.88	62.63%
50	1.80	28.27	91.97%

Postscript

J.P.Morgan

- One of the key strategic results of JP Morgan’s work with Maxeler is that JP Morgan has adapted its technology strategy from one of “build or buy” to one of “build or buy or acquire”
- JP Morgan has taken a 20% stake in Maxeler – a key example of its commitment of using innovation to achieve strategic competitive advantage

Any questions?

JPMorgan Chase & Co.

Q & A