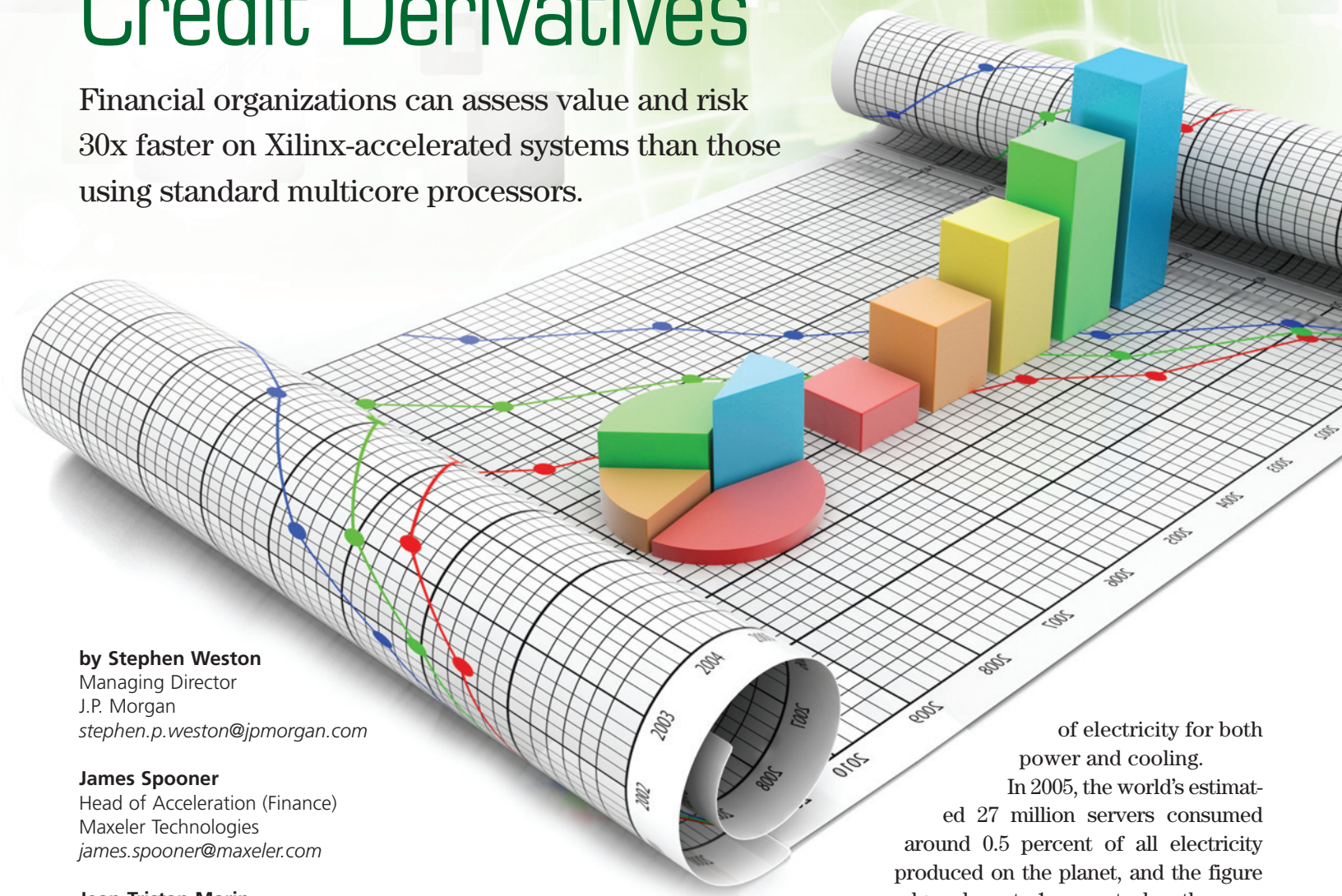
# FPGAs Speed the Computation of Complex Credit Derivatives

Financial organizations can assess value and risk 30x faster on Xilinx-accelerated systems than those using standard multicore processors.



**by Stephen Weston**
Managing Director
J.P. Morgan
*stephen.p.weston@jpmorgan.com*

**James Spooner**
Head of Acceleration (Finance)
Maxeler Technologies
*james.spooner@maxeler.com*

**Jean-Tristan Marin**
Vice President
J.P. Morgan
*jean-tristan.marin@jpmorgan.com*

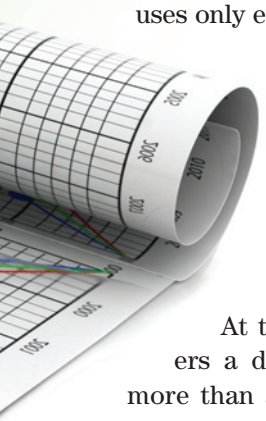
**Oliver Pell**
Vice President of Engineering
Maxeler Technologies
*oliver@maxeler.com*

**Oskar Mencer**
CEO
Maxeler Technologies
*mencer@maxeler.com*

Innovation in the global credit derivatives markets rests on the development and intensive use of complex mathematical models. As portfolios of complicated credit instruments have expanded, the process of valuing them and managing the risk has grown to a point where financial organizations use thousands of CPU cores to calculate value and risk daily. This task in turn requires vast amounts of electricity for both power and cooling.

In 2005, the world's estimated 27 million servers consumed around 0.5 percent of all electricity produced on the planet, and the figure edges closer to 1 percent when the energy for associated cooling and auxiliary equipment (for example, backup power, power conditioning, power distribution, air handling, lighting and chillers) is included. Any savings from falling hardware costs are increasingly offset by rapidly rising power-related indirect costs. No wonder, then, that large financial institutions are searching for a way to add ever-greater computational power at a much lower operating cost.

Toward this end, late in 2008 the Applied Analytics group at J.P. Morgan in London launched a collaborative project with the acceleration-solutions provider Maxeler Technologies that remains ongoing. The core of the project was the design and delivery of a MaxRack hybrid (Xilinx® FPGA and Intel CPU) cluster solution by Maxeler. The system has demonstrated more than 31x acceleration of the valuation for a substantial portfolio of complex credit derivatives compared with an identically sized system that uses only eight-core Intel CPUs.



The project reduces operational expenditures more than thirtyfold by building a customized high-efficiency high-performance computing (HPC) system. At the same time, it delivers a disk-to-disk speedup of more than an order of magnitude over comparable systems, enabling the computation of an order-of-magnitude more scenarios, with direct impact on the credit derivatives business at J.P. Morgan. The Maxeler system clearly demonstrates the feasibility of using FPGA technology to significantly accelerate computations for the finance industry and, in particular, complex credit derivatives.

## RELATED WORK

There is a considerable body of published work in the area of acceleration for applications in finance. A shared theme is adapting technology to accelerate the performance of computationally demanding valuation models. Our work shares this thrust and is distinguished in two ways.

The first feature is the computational method we used to calculate fair value. A common method is a single-factor Gaussian copula, a mathematical model used in finance, to model the underlying credits, along with Monte Carlo simulation to evaluate the expected value of the portfolio. Our model employs standard base correlation methodology, with a Gaussian copula for default correlation and a stochastic recovery process. We compute the expected (fair) value of a portfolio using a convolution approach. The combination of these methods presents a distinct computational, algorithmic and performance challenge.

In addition, our work encompasses the full population of live complex credit trades for a major investment bank, as opposed to relying on the usual approach of using synthetic data sets. To fully grasp the computing challenges, it's helpful to take a closer look at the financial products known as credit derivatives.

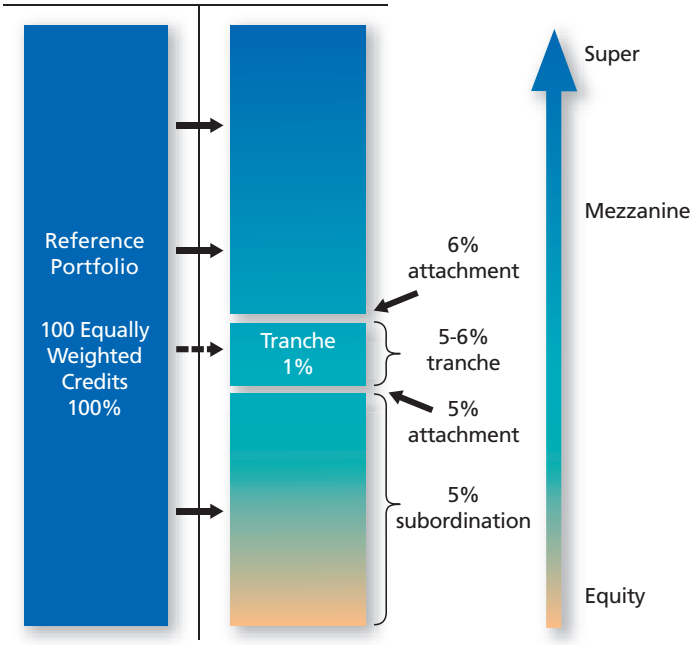
## WHAT ARE CREDIT DERIVATIVES?

Companies and governments issue bonds which provide the ability to fund themselves for a predefined period, rather like taking a bank loan for a fixed term. During their life, bonds typically pay a return to the holder, which is referred to as a "coupon," and at maturity return the borrowed amount. The holder of the bond, therefore, faces a number of risks, the most obvious being that the value of the bond may fall or rise in response to changes in the financial markets. However, the risk we are concerned with here is that when the bond reaches maturity, the issuer may default and fail to pay the coupon, the full redemption value or both.

Credit derivatives began life by offering bond holders a way of insuring against loss on default. They have expanded to the point where they now offer considerably greater functionality across a range of assets beyond simple bonds, to both issuers and holders, by enabling the transfer of default risk in exchange for regular payments. The simplest example of such a contract is a credit default swap (CDS), where the default risk of a single issuer is exchanged for regular payments. The key difference between a bond and a CDS is that the CDS only involves payment of a redemption amount (minus any recovered amount) should default occur.

Building on CDS contracts, credit default swap indexes (CDSIs) allow the trading of risk using a portfolio of underlying assets. A collateralized default obligation (CDO) is an extension of a CDSI in which default losses

Figure 1 – Tranched credit exposure




Source: J.P. Morgan

The modeling of behavior for a pool of credits becomes complex, as it is important to model the likelihood of default among issuers. Corporate defaults, for example, tend to be significantly correlated—the failure of a single company tends to weaken the remaining firms.

on the portfolio of assets are divided into different layers or "tranches," according to the risk of default. A tranche allows an investor to buy or sell protection for losses in a certain part of the pool.

Figure 1 shows a tranche between 5 percent and 6 percent of the losses in a pool. If an investor sells protection on this tranche, he or she will not be exposed to any losses until 5 percent of the total pool has been wiped out. The lower (less senior) tranches have higher risk and those selling protection on these tranches will receive higher compensation than those investing in upper (more senior, hence less likely to default) tranches.

CDOs can also be defined on nonstandard portfolios. This variety of CDO, referred to as "bespoke," presents additional computational challenges, since the behavior of a given bespoke portfolio is not directly observable in the market.

The modeling of behavior for a pool of credits becomes complex, as it is important to model the likelihood of default among issuers. Corporate defaults, for example, tend to be significantly correlated because all firms are exposed to a common or correlated set of economic risk factors, so that the failure of a single company tends to weaken the remaining firms.

The market for credit derivatives has grown rapidly over the past 10 years, reaching a peak of $62 trillion in January 2008, in response to a wide range of influences on both demand and supply. CDSes account for approximately 57 percent of the notional outstanding, with the remainder accounted for by products with multiple underlying credits. "Notional" refers to the amount of money underlying a credit derivative contract; "notional outstanding" is the total amount of notional in current contracts in the market.

```
for  i in 0 ... markets - 1
   for j in 0 ... names - 1
      prob = cum_norm ((inv_norm (Q[j]) - sqrt(rho)*M[i])/sqrt (1 - rho);
      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
```

Figure 2 – Pseudo code for the bespoke CDO tranche pricing

## A CREDIT DERIVATIVES MODEL

The standard way of pricing CDO tranches where the underlying is a bespoke portfolio of reference credits is to use the "base correlation" approach, coupled with a convolution algorithm that sums conditionally independent loss-random variables, with the addition of a "coupon model" to price exotic coupon features. Since mid-2007 the standard approach in the credit derivatives markets has moved away from the original Gaussian copula model. Two alternative approaches to valuation now dominate: the random-factor loading (RFL) model and the stochastic recovery model.

Our project has adopted the latter approach and uses the following methodology as a basis for accelerating pricing: In the first step, we discretize and compute the loss distribution using convolution, given the conditional survival probabilities and losses from the copula model. Discretization is a method of chunking a continious space into a discrete space and separating the space into discrete "bins," allowing algorithms (like integration) to be done numeri-

cally. We then use the standard method of discretizing over the two closest bins with a weighting such that the expected loss is conserved. We compute the final loss distribution using a weighted sum over all of the market factors evaluated, using the copula model.

Pseudo code for this algorithm is shown in Figure 2. For brevity, we've removed the edge cases of the convolution and the detail of the copula and recovery model.

## ACCELERATED CDO PRICING

Each day the credit hybrids business within J.P. Morgan needs to evaluate hundreds of thousands of credit derivative instruments. A large proportion of these are standard, single-name CDSes that need very little computational time. However, a substantial minority of the instruments are tranched credit derivatives that require the use of complex models like the one discussed above. These daily runs are so computationally intensive that, without the application of Maxeler's acceleration techniques, they could only be meaningfully carried out overnight. To complete the task, we used approximately 2,000 standard Intel cores. Even with such resources available, the calculation time is around four and a half hours and the total end-to-end runtime is close to seven hours, when batch preparation and results writeback are taken into consideration.

With this information, the acceleration stage of the project focused on designing a solution capable of dealing with only the complex bespoke tranche products, with a specific goal of exploring the HPC architecture design space in order to maximize the acceleration.

## MAXELER'S ACCELERATION PROCESS

The key to maximizing the speed of the final design is a systematic and disciplined end-to-end acceleration methodology. Maxeler follows the four stages of acceleration shown in Figure 3, from the initial C++ design to the final implementation, which ensures we arrive at the optimal solution.

In the analysis stage, we conducted a detailed examination of the algorithms contained in the original C++ model code. Through extensive code and data profiling with the Maxeler Parton profiling tool suite, we were able to clearly understand and map the relationships between the computation and the input and output data. Part of this analysis involved acquiring a full understanding of how the core algorithm performs in practice, which allowed us to identify the major computational and data movements, as well as storage costs. Dynamic analysis using call graphs of the running software, combined with detailed analysis of data values and

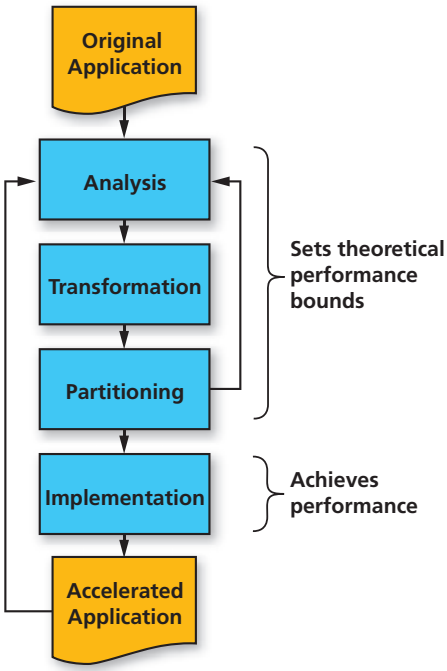



Figure 3 – Iterative Maxeler process for accelerating software

```
// Shared library and call overhead  1.2 %
for d in 0 ... dates - 1
   // Curve Interpolation  19.7%
   for j in 0 ... names - 1
        Q[j] = Interpolate(d, curve)
   end # for j
   for  i  in 0 ... markets - 1
      for j in 0 ...  names  - 1
       // Copula Section  22.0%
       prob = cum_norm   (( inv_norm (Q[j]) - sqrt (rho)*M[i])/sqrt (1 - rho);
       loss=  calc_loss (prob,Q2[j],RR[j],RM[j])*notional[j];
       n = integer(loss);
       lower = fractional(loss);
       for k in 0 ... bins - 1
          if j == 0
             dist[k] = k == 0 ? 1.0 : 0.0 ;
              // Convolution  51.4%
             dist[k] = dist[k]*(1-prob) +
                          dist[k-n]* prob  *(1 -L) +
                          dist[k-n-1]* prob  *L;
            // Reintegration,  5.1%
           if j == credits  - 1
              final_dist  [k] += weight[i] * dist[k];
         end # for k
      end # for j
   end # for i
end # for d
// Code outside main loop  0.5%
```

Figure 4 – Profiled version of original pricing algorithm in pseudo-code form

runtime performance, were necessary steps to identifying bottlenecks in execution as well as memory utilization patterns.

Profiling the core 898 lines of original source code (see Figure 4) focused attention on the need to accelerate two main areas of the computation: calculation of the conditional-survival probabilities (Copula evaluation) and calculation of the probability distribution (convolution).

The next stage, transformation, extracted and modified loops and program control structure from the existing C++ code, leading to code and data layout transformations that in turn enabled the acceleration of the core algorithm. We removed data storage abstraction using object orientation, allowing data to pass efficiently to the accelerator, with low CPU overhead. Critical transformations included loop unrolling, reordering, tiling and operating on vectors of data rather than single objects.

## PARTITIONING AND IMPLEMENTATION

The aim for the partitioning stage of the acceleration process was to create a contiguous block of operations. This block needed to be tractable to accelerate, and had to achieve the maximum possible runtime coverage, when balanced with the overall CPU performance and data input and output considerations.

The profiling process we identified during the analysis stage provided the necessary insight to make partitioning decisions.

The implementation of the partitioned design led to migrating the loops containing the Copula evaluation and convolution onto the FPGA accelerator. The remaining loops and associated computation stayed on the CPU. Within the FPGA design, we split the Copula and convoluter into separate code implementations that could execute and be parallelized independently of each other and could be sized according to the bounds of the loops described in the pseudo code.

In the past, some design teams have shied away from FPGA-based acceleration because of the complexity of the programming task. Maxeler's programming environment, called MaxCompiler, raises the level of abstraction of FPGA design to enable rapid development and modification of streaming applications, even when faced with frequent design updates and fixes.

Some design teams have shied away from FPGA-based acceleration because of the complexity of programming. MaxCompiler lets you implement the FPGA design in Java, without resorting to lower-level languages such as VHDL.

## IMPLEMENTATION DETAILS

MaxCompiler builds FPGA designs in a simple, modular fashion. A design has one or more "kernels," which are highly parallel pipelined blocks for executing a specific computation. The "manager" dynamically oversees the dataflow I/O between these kernels. Separating computation and communication into kernels and manager enables a high degree of pipeline parallelism within the kernels. This parallelism is pivotal to achieving the performance our application enjoys. We achieved a second level of parallelism by replicating the compute pipeline many times within the kernel itself, further multiplying speedup.

The number of pipelines that can be mapped to the accelerator is limited only by the size of the FPGAs used in the MaxNode and available parallelization in the application. MaxCompiler lets you implement all of the FPGA design efficiently in Java, without resorting to lower-level languages such as VHDL.

For our accelerator, we used the J.P. Morgan 10-node MaxRack configured with MaxNode-1821 compute nodes. Figure 5 sketches the system architecture of a MaxNode-1821. Each node has eight Intel Xeon cores and two Xilinx FPGAs connected to the CPU via PCI Express®. A MaxRing high-speed interconnect is also available, providing a dedicated high-band-

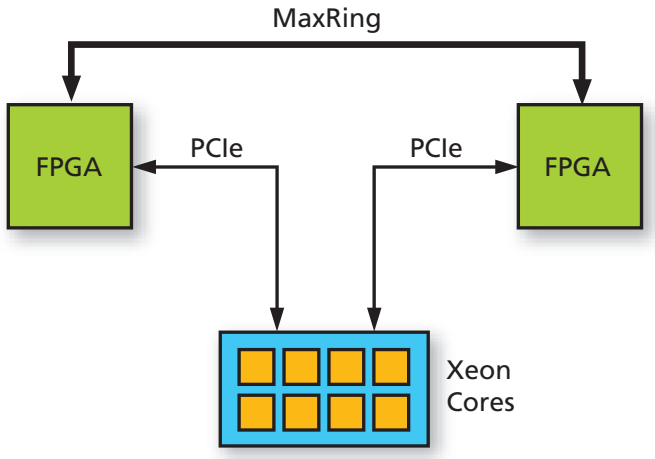



Figure 5 – MaxNode-1821 architecture diagram containing eight Intel Xeon cores and two Xilinx FPGAs

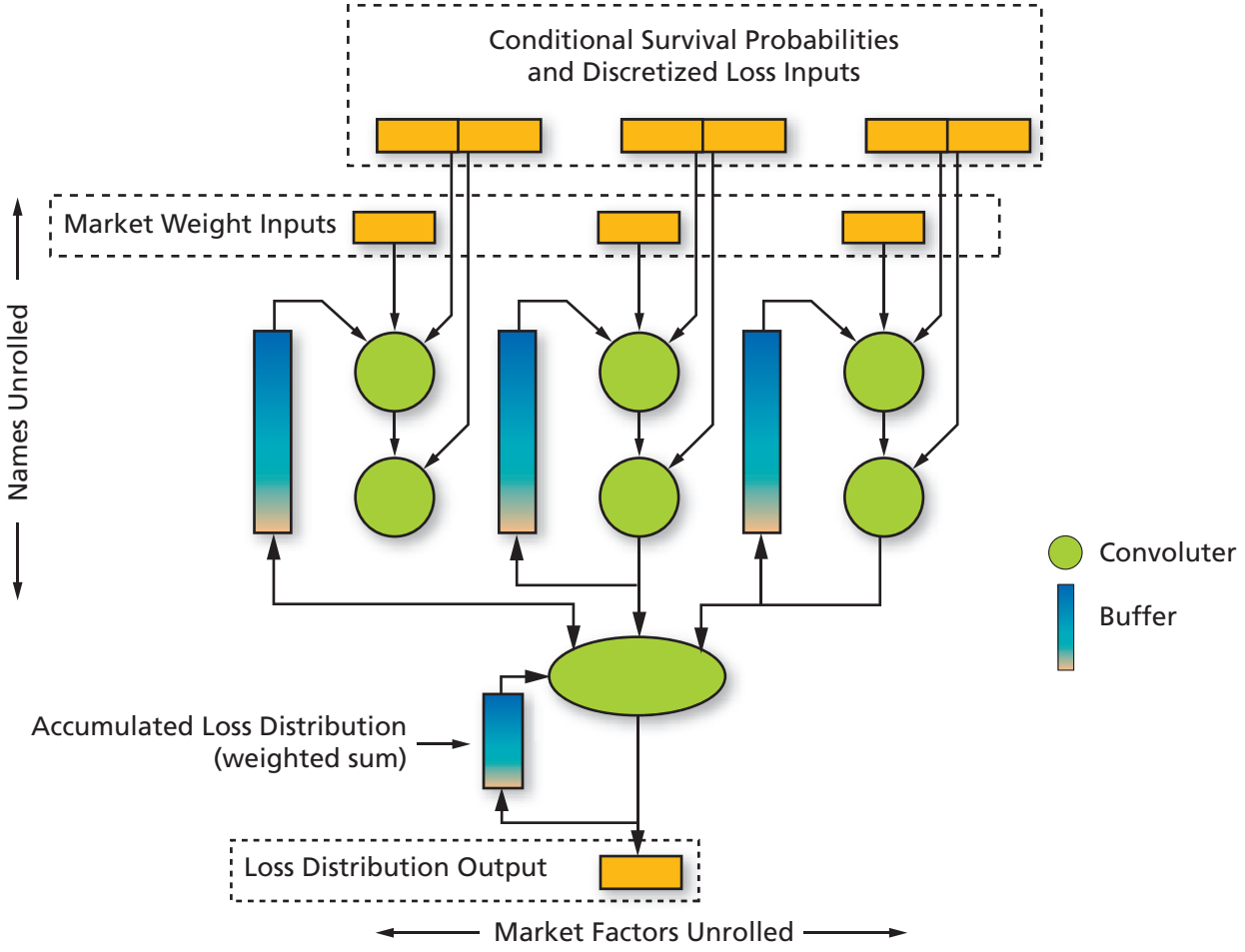



Figure 6 – Convoluter architecture diagram

width communication channel directly between the FPGAs.

One of the main focus points during this stage was finding a design that balanced arithmetic optimizations, desired accuracy, power consumption and reliability. We decided to implement two variations of the design: a "full-precision" design for fair-value (FV) calculations and a "reduced-precision" version for scenario analysis. We designed the full-precision variant for an average relative accuracy of $10^{-8}$ and the reduced-precision for an average relative accuracy of $10^{-4}$. These design points share identical implementations, varying only in the compile-time parameters of precision, parallelism and clock frequency.

We built the Copula and convolution kernels to implement one or more pipelines that effectively parallelize loops within the computation. Since the convolution uses each value the Copula model generates many times, the kernel and manager components were scalable to enable us to tailor the exact ratio of Copula or convolution resources to the workload for the design. Figure 6 shows the structure of the convolution kernel in the design.

When implemented in MaxCompiler, the code for the convolution in the inner loop resembles the original structure. The first difference is that there is an implied loop, as data streams through the design, rather than the design iterating over the data. Another important distinction is that the code now operates in terms of a data stream (rather than on an array), such that the code is now describing a streaming graph, with offsets forward and backward in the stream as necessary, to perform the convolution. Figure 7 shows the core of the code for the convoluter kernel.

## IMPLEMENTATION EFFORT

In general, software programming in a high-level language such as C++ is much easier than interacting directly with FPGA hardware using a low-level hardware description language. Therefore, in addition to performance, development and support time are increasingly significant components of overall effort from the software-engineering perspective. For our project, we measured programming effort as one aspect in the examination of programming models. Because it is difficult to get accurate development-time statistics and to measure the quality of code, we use lines of code (LOC) as our metric to estimate programming effort.

Due to the lower-level nature of coding for the FPGA architecture

```
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);
```

Figure 7 – MaxCompiler code for adding a name to a basket

```
// Shared library and call overhead  5%
for  d in 0 ...    dates  - 1
  // Curve Interpolation  54.5%
  for j in 0 ... names  - 1
       Q[j] = Interpolate(d, curve)
  end # for j
  for  i  in 0 ... markets  - 1
     for j in 0 ...  names  - 1
       // Copula Section  9%
       prob  = cum_norm  (( inv_norm  (Q[j]) - sqrt  (rho)*M[i])/sqrt(1 - rho);
       loss=  calc_loss  (prob,Q2[j],RR[j],RM[j])*notional[j];
       // (FPGA Data preparation and post processing)  11.2%
       n = integer(loss);
       lower = 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
end # for d
// Other overhead (object construction, etc)  19.9%
```

Figure 8 – Profiled version of FPGA take on pricing

when compared with standard C++, the original 898 lines of code generated 3,696 lines of code for the FPGA (or a growth factor of just over four).

Starting with the program that runs on a CPU and iterating in time, we transformed the program into the spatial domain running on the MaxNode, creating a structure on the FPGA that matched the data-flow structure of the program (at least the computationally intensive parts). Thus, we optimized the computer based on the program, rather than optimizing the program based on the computer. Obtaining the results of the computation then became a simple matter of streaming the data through the Maxeler MaxRack system.

In particular, the fact we can use fixed-point representation for many of the variables in our application is a big advantage, because FPGAs offer the opportunity to optimize programs on the bit level, allowing us to pick the optimal representation for the internal variables of an algorithm, choosing precision and range for different encodins  such as floating-point, fixed-point and logarithmic numbers.

We used Maxeler Parton to measure the amount of real time taken for the computation calls in the original implementation and in the accelerated software. We divided the total software time running on a single core by eight to attain the upper bound on the multicore performance, and compared it to the real time it took to perform the equivalent computation using eight cores and two FPGAs with the accelerated software.

For power measurements, we used an Electrocorder AL-2VA from Acksen Ltd. With the averaging window set to one second, we recorded a 220-second window of current and voltage measurements while the software was in its core computation routine.

As a benchmark for demonstration and performance evaluation, we used a fixed population of 29,250 CDO tranch-

es. This population comprised real bespoke tranche trades of varying maturity, coupon, portfolio composition and attachment/detachment points.

The MaxNode-1821 delivered a 31x speedup over an eight-core (Intel Xeon E5430 2.66-GHz) server in full-precision mode and a 37x speedup at reduced precision, both nodes using multiple processes to price up to eight tranches in parallel.

The significant differences between the two hardware configurations include the addition of the MAX2-4412C card with two Xilinx FPGAs and 24 Gbytes of additional DDR DRAM. Figure 8 shows the CPU profile of the code running with FPGA acceleration.

As Table 1 shows, the power usage per node decreases by 6 percent in the hybrid FPGA solution, even with a 31x increase in computational performance. It follows that the speedup per watt when computing is actually greater than the speedup per cubic foot.

Table 2 shows a breakdown of CPU time of the Copula vs. convolution computations and their equivalent resource utilization on the FPGA. As we have exchanged time for space when moving to the FPGA design, this gives a representative indication of the relative speedup between the different pieces of computation.

## THREE KEY BENEFITS

The credit hybrids trading group within J.P. Morgan is reaping substantial benefits from applying acceleration technology. The order-of-magnitude increase in available computation has led to three key benefits: computations run much faster; additional applications and algorithms, or those that were once impossible to resource, are now possible; and operational costs resulting from given computations drop dramatically.

The design features of the Maxeler hybrid CPU/FPGA computer mean that its low consumption of electricity, physically small footprint and low heat output make it an extremely attractive alternative to the traditional cluster of standard cores.

One of the key insights of the project has been the substantial benefits to be gained from changing the computer to fit the algorithm, not changing the algorithm to fit the computer (which would be the usual approach using standard CPU cores). We have found that executing complex calculations in customizable hardware with Maxeler infrastructure is much faster than executing them in software.

## FUTURE WORK

The project has expanded to include the delivery of a 40-node Maxeler hybrid computer designed to provide portfolio-level risk analysis for the credit hybrids trading desk in near real time. For credit derivatives, a second (more general) CDO model is currently undergoing migration to run on the architecture.

In addition, we have also applied the approach to two interest-rate models. The first was a four-factor Monte Carlo model covering equities, interest rates, credit and foreign exchange. So far, we've achieved an acceleration speedup of 284x over a Xeon core. The second is a general tree-based model, for which acceleration results are projected to be of a similar order of magnitude. We are currently evaluating further, more wide-ranging applications of the acceleration approach, and are expecting similar gains across a wider range of computational challenges.

| PLATFORM | IDLE | PROCESSING |
|---|---|---|
| Dual Xeon L5430 2.66 GHz Quad-Core 48-GB DDR DRAM | 185W | 255W |
| (as above) with MAX2-4412C Dual Xilinx SX240T FPGAs 24-GB DDR DRAM | 210W | 240W |

Table 1 – Power usage for 1U compute nodes when idle and while processing

| COMPUTATION | PERCENT TIME IN SOFTWARE | 6-INPUT LOOKUP TABLES | FLIP-FLOPS | 36-KBIT BLOCK RAMS | 18X25-BIT MULTIPLIERS |
|---|---|---|---|---|---|
| Copula Kernel | 22.0% | 30.05% | 35.12% | 11.85% | 15.79% |
| Convolution and Integration | 56.1% | 46.54% | 52.84% | 67.54% | 84.21% |

Table 2 – Computational breakdown vs. FPGA resource usage of total used