# Vectorization: The Rise of Parallelism

CEO Rohan Douglas and Development Manager – Risk Architecture Jamie Elliott of Quantifi focus on optimizing the latest generation of hardware to meet the demand for higher-performance risk and analytics.

New challenges in the financial markets driven by changes in market structure and regulations and accounting rules like Basel III, EMIR, Dodd–Frank, MiFID II, Solvency II, IFRS 13, IRFS 9, and FRTB have increased demand for higher-performance risk and analytics. Problems like XVA require orders of magnitude more calculations for accurate results. This demand for higher performance has put a focus on how to get the most out of the latest generation of hardware.

## The rise of parallelism

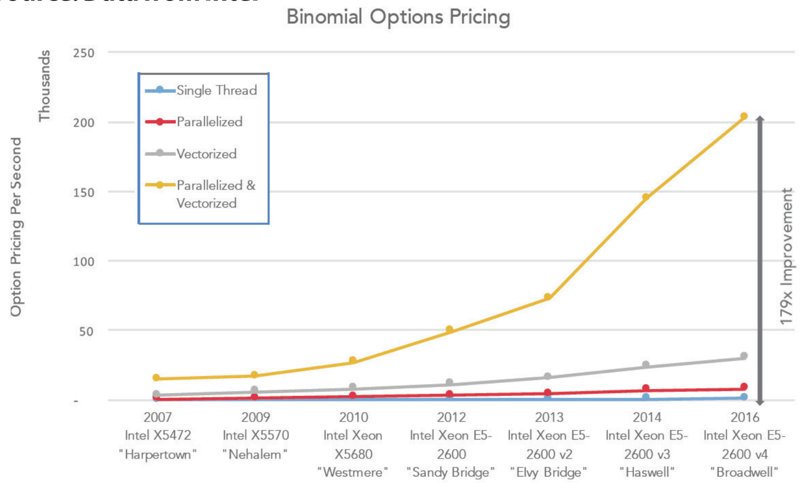
For the past decade, Moore's law has continued to prevail, but while chipmakers have continued to pack more transistors into every square inch of silicon, the focus of innovation has moved away from greater clock speeds and toward multicore and manycore architectures.

As Herb Sutter famously observed in 2005, for developers this architectural shift meant the end of the "free lunch," where existing software automatically ran faster with each new generation of hardware. Traditional applications based on a single serial thread of instructions no longer see performance gains from new hardware as CPU clock rates have flat-lined.

At the same time as multicore chip design has given rise to task parallelism in software design, chipmakers have also been increasing use of a second type of parallelism, instruction-level parallelism. Alongside the trend to increase core count, the width of SIMD (single instruction, multiple data) registers has been steadily increasing. The software changes required to exploit instruction-level parallelism are known as "vectorization." SIMD provides a way to increase performance using less power. The most recent processors have many cores/threads and the ability to implement single instructions on an increasingly large data set (SIMD width).

The first widely deployed desktop SIMD was with Intel's MMX extensions to the x86 architecture in 1996. Intel's latest generation of Xeon Phi processors – codenamed Knights Landing – use Intel's new 14-nm manufacturing process, have over 70 cores on a 2D mesh structure, four threads per core, and can

**Figure 1: Results for a binomial options pricing example. Source: Data from Intel**



operate on 512-bit vectors (SIMD length).

The results shown in Figure 1 are for a binomial options pricing example. Most existing code is either serial or implements threading or vectorization only. The combination of both threading and vectorization provides dramatic improvements, and the scale of those improvements is growing with each new generation of hardware. Modern software must leverage both threading and vectorization to get the highest performance possible from the latest generation of processors.

## Implementing vectorization

Not all code can take advantage of vectorization. Vectorization works best on problems that require the same simple operation to be performed on each element in a data set. The kinds of matrix transformation seen in linear algebra are usually a good candidate for vectorization.

The good news about implementing vectorization is that Intel provides a rich set of tools and a well-defined six-step process that provides a clear path to transforming existing code into modern, high-performance software leveraging multicore and manycore processors.

## Applying vectorization to CVA aggregation

The most common general-purpose approach to calculation of credit value adjustment (CVA) is based on a Monte Carlo simulation of the distribution of forward values for all derivative trades with a counterparty. The evolution of market prices over a series of forward dates is simulated, then the value of each derivative trade is calculated at that forward date using the simulated market prices. This gives us a "path" of projected values over the lifetime of each trade. By running a large number of these randomized simulated "paths," we can estimate the distribution of forward values, giving both the expected and extreme "exposures." The simulation step results in a three-dimensional array of exposures, the dimensions being [trades][paths][dates]. The task of calculating CVA from these exposures occurs in several steps: netting, collateralization, integration over paths, integration over dates.

### *Netting*

When a counterparty defaults, all positions with the counterparty must be closed out. If the appropriate legal agreements are in place, positive and negative individual trade exposures may offset each other, and a single net amount can be agreed. Mathematically, allowing netting just means that trade exposures are additive.

Computing the net exposures is a simple sum of trade exposures for each path and date.

```
for (t = 0; t < tradeCount; t++)
 for (p = 0; p < pathCount; p++)
  for (d = 0; d < dateCount; d++)
   netExposure[p][d] += tradeExposure[t][p][d];
```

This is equivalent to a sequence of matrix additions:

$$A+B=\begin{pmatrix}A_{11} & A_{12} & \cdots & A_{1m}\\ A_{21} & A_{22} & \cdots & A_{2m}\\ \vdots & \vdots & \ddots & \vdots\\ A_{n1} & A_{n2} & \cdots & A_{nm}\end{pmatrix}+\begin{pmatrix}B_{11} & B_{12} & \cdots & B_{1m}\\ B_{21} & B_{22} & \cdots & B_{2m}\\ \vdots & \vdots & \ddots & \vdots\\ B_{n1} & B_{n2} & \cdots & B_{nm}\end{pmatrix}$$

$$=\begin{pmatrix}A_{11}+B_{11} & A_{12}+B_{12} & \cdots & A_{1m}+B_{1m}\\ A_{21}+B_{21} & A_{22}+B_{22} & \cdots & A_{2m}+B_{2m}\\ \vdots & \vdots & \ddots & \vdots\\ A_{n1}+B_{n1} & A_{n2}+B_{n2} & \cdots & A_{nm}+B_{nm}\end{pmatrix}$$

The inner loop here is clearly a classic candidate for vectorization and provided care is taken to arrange the data in appropriate order, the compiler will take care of this automatically. In the simplest case, all the trade exposures share a common set of dates, and we wish to compute net exposures for this same set of common dates. A more complex extension of this problem comes when either the trades each have a unique set of dates, or the dates we wish to net on are not the same as the original dates. In either case, we need to first interpolate the trade exposures on a new common time grid, in order to then net them together. We use simple linear interpolation, which amounts to a weighted sum of the pair of exposures on the dates surrounding the interpolation date.

```
for (t = 0; t < tradeCount; t++)
 for (p = 0; p < pathCount; p++)
  for (d = 0; d < dateCount; d++)
   netExposure[p][d] += tradeExposure[t][p][aIdx[t][d]]*
    alpha[aIdx[t][d]] + tradeExposure[t][p][bIdx[t][d]]*
```

The netExposure array is now different from the size of the source arrays, and this introduces indirect memory access. As discussed, indirect references can have a negative impact on the efficiency of vectorization. We examined two options for combating this problem:

(a) Copy into properly aligned arrays. Then all operations become matrix addition or Hadamard product:

$$A\circ B=\begin{pmatrix}A_{11} & A_{12} & \cdots & A_{1m}\\ A_{21} & A_{22} & \cdots & A_{2m}\\ \vdots & \vdots & \ddots & \vdots\\ A_{n1} & A_{n2} & \cdots & A_{nm}\end{pmatrix}\circ\begin{pmatrix}B_{11} & B_{12} & \cdots & B_{1m}\\ B_{21} & B_{22} & \cdots & B_{2m}\\ \vdots & \vdots & \ddots & \vdots\\ B_{n1} & B_{n2} & \cdots & B_{nm}\end{pmatrix}=$$

$$\begin{pmatrix}A_{11}B_{11} & A_{12}B_{12} & \cdots & A_{1m}B_{1m}\\ A_{21}B_{21} & A_{22}B_{22} & \cdots & A_{2m}B_{2m}\\ \vdots & \vdots & \ddots & \vdots\\ A_{n1}B_{n1} & A_{n2}B_{n2} & \cdots & A_{nm}B_{nm}\end{pmatrix}$$

(b) Rely on improved scatter/gather produced by compiler auto-vectorization.

▷

In our testing, it appears that the scatter/gather solution remains more efficient. The cost of iterating over the arrays and copying data into well-aligned arrays turns out to be greater than the consequent performance improvement in the main body of the loop. This remains a subject of active research.

Developers should always be looking for cases where vectorization and multi-threading can complement each other. One example would be the opportunity to perform interpolation of each trade exposure concurrently on multiple threads, then perform the netting of interpolated values as a final step. An alternative is to distribute the paths across multiple threads, as at this stage the pathwise values are independent.

## Collateral

Trading parties often agree to mitigate counterparty risk by requiring collateral to be posted to cover losses in the event of default. Collateral agreements can take a variety of forms, and here we present a simplified version with some of the most common features. From the example, it can be seen that vectorization can easily be applied.

## Variation margin

This is an agreement that one or both parties must provide collateral to offset the market value of a trade or trades. The margin may cover the portfolio value in excess of a threshold $K$. By making $K = 0$, the full value of the portfolio at the margin call date will be covered:

$$\text{Variation margin} = \text{Max}(\text{net exposure} - K, 0)$$

Once the net exposures have been aggregated from the trade exposures, we can compute the variation margin for each exposure:

```
for (p = 0; p < pathCount; p++)
 for (d = 0; d < dateCount; d++)
  vm[p][d]= Max(netExposure[p][d] - K,0)
```

Returning to the list of "intrinsic" mathematical functions with compiler-recognizable vectorized versions, we can see that a max operator is available.

Since we are interested only in positive exposures, amounts that can be lost after the default of the counterparty, our collateralized exposure then becomes:

```
for (p = 0; p < pathCount; p++)
 for (d = 0; d < dateCount; d++)
  collateralizedExposure[p][d] = Max(netExposure[p][d] - vm[p][d],0)
```

## Margin of risk period

The above formulas assume that collateral is posted instantaneously. In reality, margin call frequency will be at most once a day and there will be a lag between the margin call and the delivery of collateral. During this margin period of risk (MPR), the market value of the portfolio may increase, creating additional uncollateralized exposure. To introduce MPR into our model, we compare netExposure at date t to the collateral that we can be sure has already been delivered by t. That means determining the collateral based on a margin call date at some lag Δt relative to t. This becomes a use case for the interpolation of exposures discussed above. We must first interpolate the exposures onto a new time grid, where each new date $t_{margin} = t - \Delta t$. Then our variation margin amount is determined using these interpolated exposures:

```
vm[p][d]= Max(netExposureAtMarginDate[p][d] - K,0)
```

The second step, calculating collateralized exposures, remains unchanged and still uses the net exposures at the original exposure dates. Again, note here that the results along each path are independent, so there is an opportunity to process multiple paths in concurrent threads.

## *Expected exposure*

Our first risk measure. Integrate over paths. Total then average:

```
for (p = 0; p < pathCount; p++)
 for (d = 0; d < dateCount; d++)
  ee[d] += collateralizedExposure[p][d];
for (d = 0; d < dateCount; d++)
  ee[d] /= pathCount;
```

Again, this is all straightforward elementwise operations on the matrix.

## *CVA*

Our target risk measure – integrated over time and weighted by default probability:

```
for (d = 0; d < dateCount; d++)
  CVA += ee[d] * defaultProb[d] * lgd[d];
```

Software design must adapt to take advantage of these new processor technologies. Multi-threading and vectorization are powerful tools on their own, but only by combining them can performance be maximized.

quantifi

### FURTHER READING

Li, S. 2016. *Recipe: Using Binomial Option Pricing Code as Representative Pricing Derivative Method*. Intel, June.
O'Leary, K. 2016. *Vectorization of Performance Dies for the Latest AVX SIMD*. Intel, August.
Rogozhin, K. 2017. *Vectorization*. Intel, March.
Sabahi, M. 2010. *A Guide to Vectorization with Intel® C++ Compilers*. Intel, November.
Sutter, H. 2005. The free lunch is over – a fundamental turn toward concurrency in software. *Dr. Dobb's Journal* 30(3), March.
*Vectorization Codebook*, Intel, September 2015.