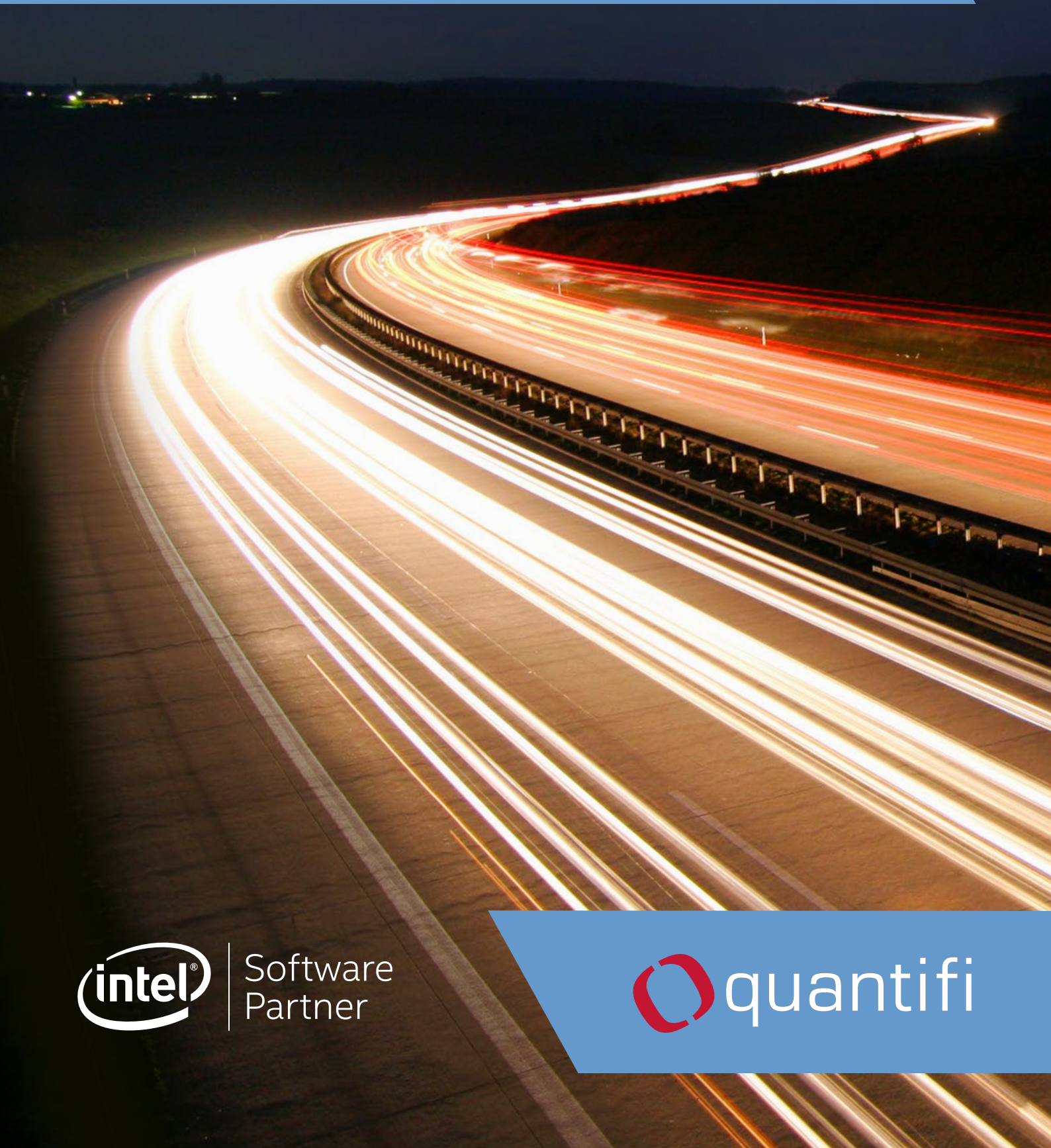
WHITEPAPER

VECTORISATION PERFORMANCE WITH QUANTIFI



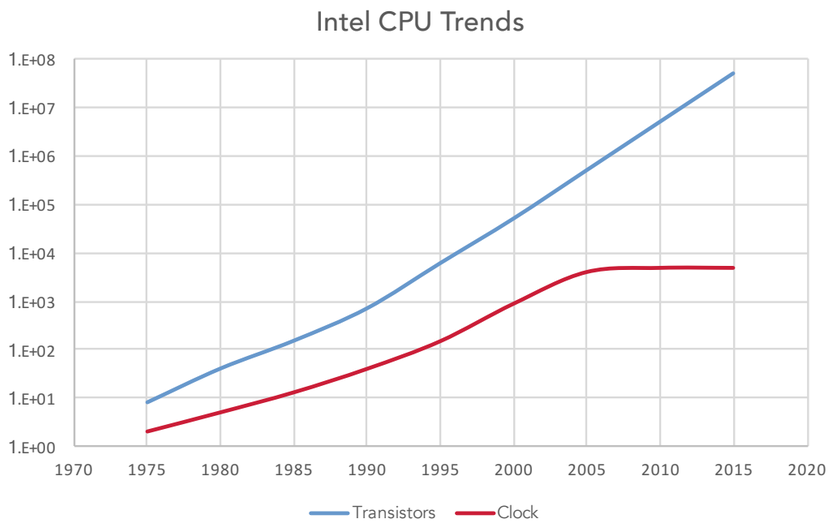
intel® Software Partner

quantifi

The rise of parallelism

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

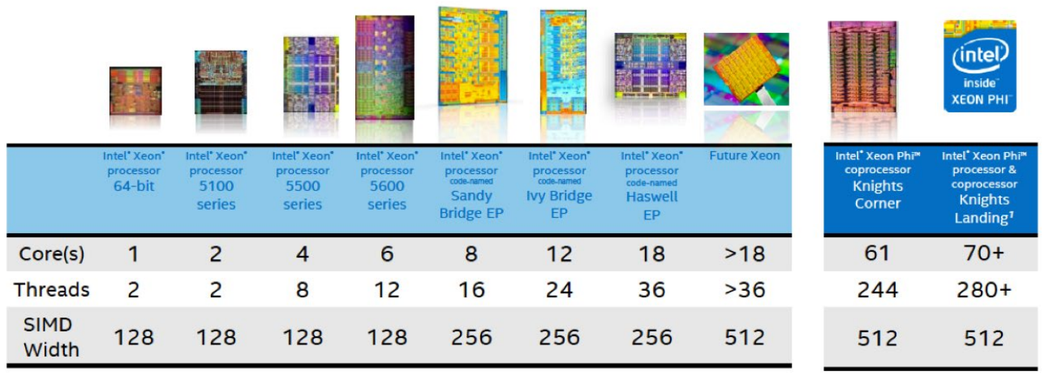


Source: Data from Intel

Since that time, a great deal of focus has been given to engineering applications that are capable of exploiting the growing number of CPU cores by running multi-threaded or grid-distributed calculations. This type of parallelism has become a routine part of designing performance critical software.

At the same time as the multi core chip design has given rise to task parallelism in software design, chipmakers have also been increasing the power 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 'vectorisation'.

The most recent processors have many cores/threads and the ability to implement single instructions on an increasingly large data set (SIMD width).



	Intel® Xeon® processor 64-bit	Intel® Xeon® processor 5100 series	Intel® Xeon® processor 5500 series	Intel® Xeon® processor 5600 series	Intel® Xeon® processor code-named Sandy Bridge EP	Intel® Xeon® processor code-named Ivy Bridge EP	Intel® Xeon® processor code-named Haswell EP	Future Xeon	Intel® Xeon Phi™ coprocessor Knights Corner	Intel® Xeon Phi™ processor & coprocessor Knights Landing†
Core(s)	1	2	4	6	8	12	18	>18	61	70+
Threads	2	2	8	12	16	24	36	>36	244	280+
SIMD Width	128	128	128	128	256	256	256	512	512	512

Source: Intel

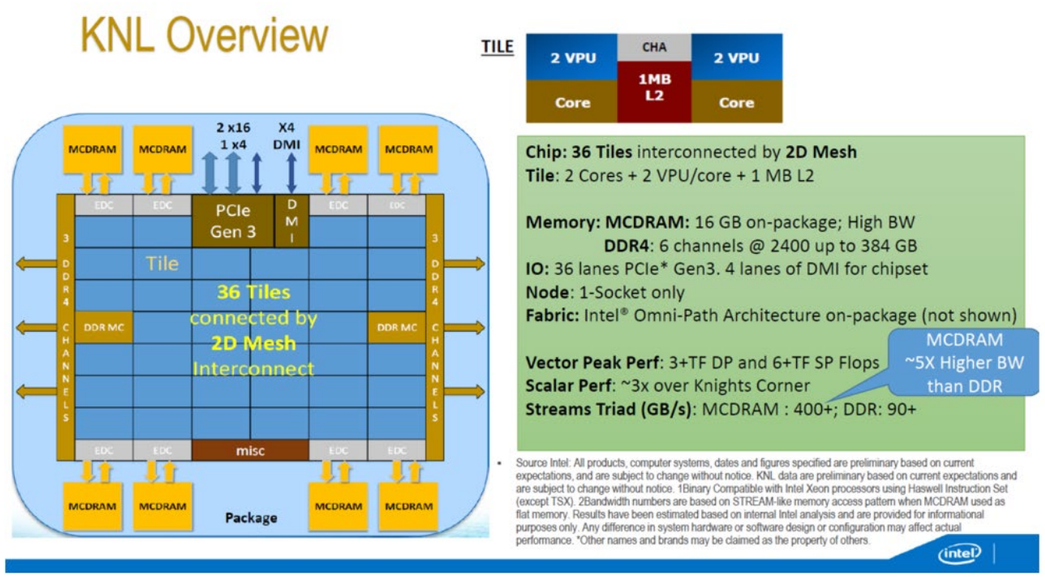
A key driver of these architectural change was the power/performance dynamic of the alternative architectures.

- Wider SIMD - Linear increase in transistors and power
- Multi core - Quadratic increase in transistors and power
- Higher clock frequency - Cubic increase power

SIMD provides a way to increase performance using less power.

The first widely deployed desktop SIMD was with Intel's MMX extensions to the x86 architecture in 1996.

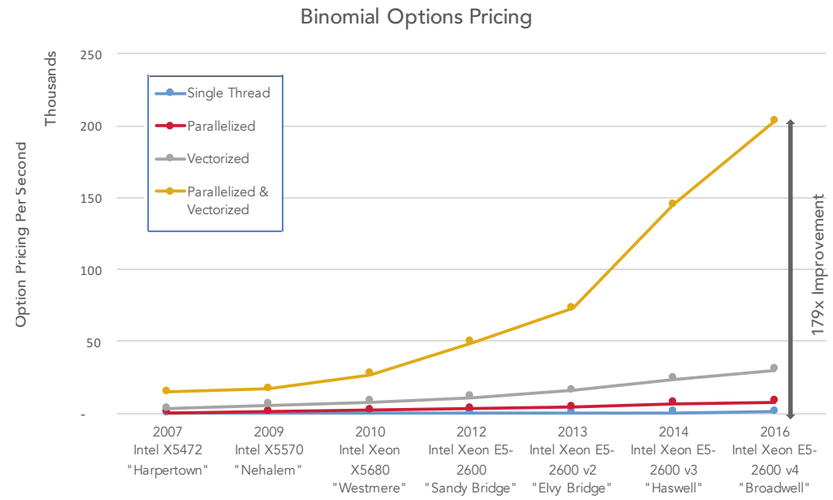
Intel's latest generation of Xeon Phi processors uses Intel's new 14nm manufacturing process, has over 70 cores on a 2D mesh structure, 4 threads per core, and can operate on 512 bit vectors (SIMD length).



Source: Intel

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

Modern software must leverage both Threading and Vectorisation to get the highest performance possible from the latest generation of processors.



Source: Data from Intel

The above results are for a binomial options pricing example. Most existing code is either serial or implements Threading or Vectorisation only. The combination of both Threading and Vectorisation provides dramatic improvements and the scale of those improvements is growing with each new generation of hardware.

Modern software must leverage both Threading and Vectorisation to get the highest performance possible from the latest generation of processors.

Vectorisation

Why Vectorise?

Vectorisation is the process of converting an algorithm from operating on a single value at a time to operating on a set of values (vector) at one time.

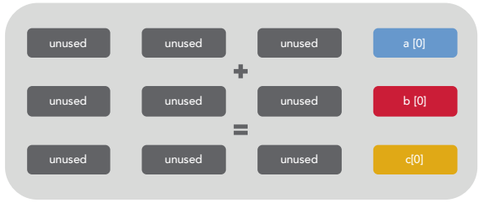
Modern CPUs provide direct support for vector operations where a single instruction is applied to multiple data (SIMD). For example a CPU with a 512 bit register could hold 16 32-bit single precision values and do a single calculation 16 times faster than executing a single instruction at a time. Combine this with threading and multi-core CPUs leads to orders of magnitude performance gains.

The following is code to add two vectors:

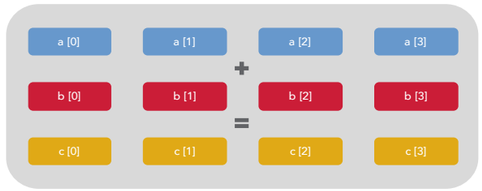
```
for (i = 0; i < 4; i++)
  c[i] = a[i] + b[i];
```

In a serial calculation, the individual vector (array) elements are added in sequence.

The additional register space in modern CPUs is unused.



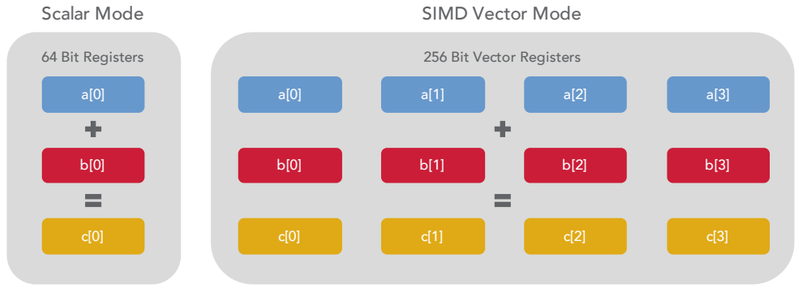
In a vectorised calculation, all elements of the vector (array) can be added in one calculation step.



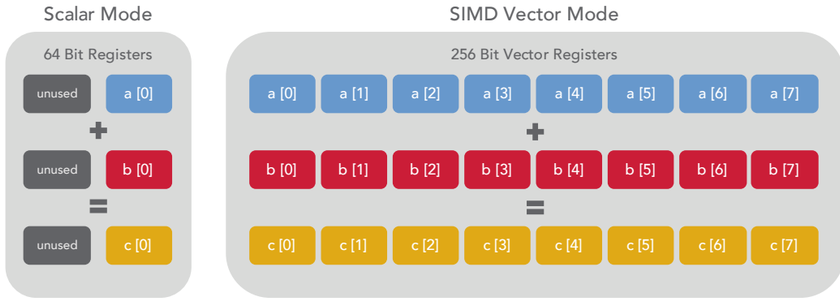
Traditional processor registers have room to hold only a single value at a time, and traditional processor instructions operate on these single values, or pairs of single values sequentially. The addition of special wide CPU registers for holding vectors of numerical values, along with instruction set extensions to support Single Instruction, Multiple Data (SIMD) operations has enabled a new type of parallelism in software known as vectorisation. Vectorisation allows a CPU to operate on multiple pieces of data at the same time. Take for example a simple loop that adds together each element of two arrays:

```
for (i = 0; i < count; i++)
  c[i] = a[i] + b[i];
```

Traditional scalar instruction sets would need to process each addition one step at a time. Whereas SIMD vector instructions can perform several additions in parallel.



In the above example, a 64 bit data register can hold just one 64 bit double precision floating point number, and traditional add instruction adds one pair of doubles. In contrast, the AVX2 instruction set architecture extensions, introduced in 2013 and now widespread on commodity hardware includes a 256 bit vector register. This allows space for 4 double precision floating point numbers, performing 4 addition operations simultaneously. If double precision accuracy is not required, single precision floating point numbers can be used, allowing room for 8 values, and 8 concurrent additions. This highlights an additional difference with the scalar register, which will simply waste the additional space and continue to perform only a single addition at a time.



The width of vector registers continues to grow, Intel has introduced AVX-512, providing 512 bit vector registers on their latest Knights Landing processors – again doubling the number of parallel operations possible. As the trend towards wider vector registers continues, vectorised code will stand to gain additional performance benefits automatically.

What kind of problem is vectorisable?

Not all code can take advantage of vectorisation. The problem set must be amenable to a vectorised solution. Vectorisation works best on problems that require the same simple operation to be performed on each element in a data set. So, first of all, look for a loop. The prototypical example is used above - the addition of each element in an array.

```
for (i = 0; i < count; i++)
  c[i] = a[i] + b[i];
```

But many other primitive operators can also be vectorised. The kinds of matrix transformation seen in linear algebra are usually a good candidate for vectorisation. The good news is that the Finance domain provides many problem sets that are suitable.

Issues that impact vectorisation

Issues that impact Vectorising your code

1. Loop Dependencies (Avoid read-after-write)

```
for (i = 1; i < end; i++)
  f[i] = f[i-1] + b[i-1];
```

2. Indirect Memory Access (Use loop index directly. Seek unit loop stride)

```
for (i = 0; i < end; i++)
  c[idxC[i]] = a[i] + b[i];
```

3. Non 'Straight line' code (function calls, conditions, unknown loop count)

```
for (i = 0; i < CalcEnd(); i++)
{
  if (DoJump())
    i += CalcJump();
  c[i] = a[i] + b[i];
}
```

When preparing a code base for vectorisation, a survey of the target code should be conducted to identify potential issues. There are a number of features that potentially prevent vectorisation or reduce the efficiency of vectorised code. Look for the following:

Loop dependencies

This describes cases where the result of one iteration of a loop depends on another iteration.

```
for (i = 1; i < end; i++)
  f[i] = f[i-1] + b[i-1];
```

The case to avoid is 'Read-after-write', where a variable is written in one iteration and read in a subsequent iteration. These cannot be vectorised. It may be possible to vectorise some 'Write-after-read' cases.

```
for (i = 1; i < end; i++)
  f[i-1] = f[i] + b[i];
```

Indirect memory access

This is usually identifiable by the presence of an indexing function or array of indexes.

```
for (i = 0; i < end; i++)
  c[idxC[i]] = a[i] + b[i];
```

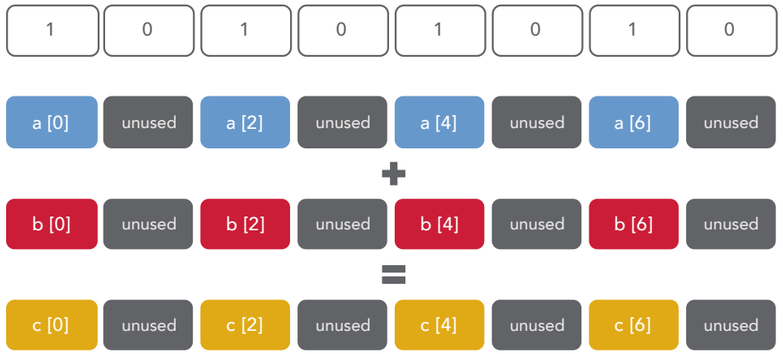
Where possible indirect indexing should be avoided. The loop index should be used directly in array subscripts. It may be possible to implement a vectorised version of a loop with indirect or sparse indexing using SIMD scatter/gather operators, or the newer compress/expand instructions.

Branches/conditional statements

If your loop contains conditionals such as if or switch statements, this may prevent the compiler from automatically vectorising the code.

```
for (i = 0; i < end; i++)
  if (i%2 == 0)
    c[i] = a[i] + b[i];
```

The compiler will attempt to generate a logical mask for the vector operations. If this fails, there may be a way to manually implement such a logical mask. Note that even if this is successful, the vectorised code will be less efficient than code that operates on all elements in the vector in terms of number of parallel operations. The AVX512F instruction set also introduces new expand and contract operations that may assist in vectorisation of conditional loops.



Function calls

```
for (i = 0; i < end; i++)
  DoSomething(i);
```

The vectorised loop should contain 'straight line' code. The only function calls permitted are functions that can be inlined by the compiler, or a prescribed set of 'intrinsic' mathematical functions shown in the table below.

acos	acosh	asin	asinh	atan	atan2	atanh	cbrt
ceil	cos	cosh	erf	erfc	erfinv	exp	exp2
fabs	floor	fmax	fmin	log	log10	log2	pow
round	sin	sinh	sqrt	tan	tanh	trunc	

Careful attention should be paid to any custom function calls within the loop to ensure they can be in-lined. Alternatively, the developer can apply __declspec(vector) attribute to tell compiler to create both scalar and vector version of the function" to the end of the "Function Calls" section.

Pointer aliasing

Because of the flexibility of pointers in c/c++, the compiler cannot determine, without help, whether there may be dependencies between iterations when using pointers. If pointers are used, it may be necessary to add special hints or keywords for the compiler

```
void add(double *a, double *b, double *c)
{
  for (i = 0; i < count; i++)
    c[i] = a[i] + b[i];
}
```


Unknown loop count

The compiler needs to know when a loop will exit in order to safely push multiple iterations onto the vector processing unit for parallel execution. Therefore, the loop count should be invariant within the loop and the loop should not contain any conditional breaks.

```
for (i = 0; i < checkEnd(); i++)
  c[i] = a[i] + b[i];
```

Outer loops

Only the innermost loop in a set of nested loops is always targeted for vectorisation. It is possible to attempt to vectorise outer loops either with compiler flags or by explicitly marking up the code with #pragma SIMD. However, the vector operations of a nested loop should preferably be placed within the innermost loop. It is acceptable to use the outer loop index within the inner loop.

```
for (i = 0; i < iEnd; i++) {
  for (j= 0; j < jEnd; i++) {
    c[i][j] = a[i] + b[i];
  }
}
```

It may be worth considering reordering or inverting a loop to place the vector operations within the inner loop.

Optimisation

Loop structure

Understanding the structure of a vectorised loop can assist in optimisation. A typical vectorised loop consists of:

- Main vector body
- Optional Peel part. Used for code with unaligned data.
- Optional Remainder part. Used at end of loop when iterator count does not match vector length.

The goal should be to eliminate any peel/remainder part and execute the entire loop in the main vector body. This leads to the following recommendations:

Align data

Data addresses should be aligned to 32 byte boundaries. This can be ensured by using special memory allocation functions. The C++ 0x standard introduces the alignas specifier to assist with this goal. You may need to tell the compiler that data is aligned by adding __assume_aligned() compiler hint.

Align iterator count

If the number of loop iterations is divisible by the vector length, all loop iterations can be executed in the vector processing unit, avoiding the less efficient remainder part. It may even be beneficial to pad the loop with extra operations to maintain the correct trip count.

Memory Access

The efficiency of the vector instruction set extensions relies on the ability to load multiple array elements into the wide vector register. This relies on having all the elements contiguous in memory.

Bandwidth and Latency

A loop that accesses contiguous sections of memory will generate a lower number of cache misses and will therefore tend to be bound by memory bandwidth. As bandwidth between the CPU and cache and between the caches and main memory increases with newer hardware, performance of bandwidth bound code should automatically be boosted.

A loop that accesses non-contiguous memory addresses will tend to cause more cache misses. In these circumstances, the performance is bound by the latency of the request. Although larger caches and faster memory can reduce cache misses and improve latency, this is still a less desirable situation.

Loop Stride

Loop stride refers to the increments used in the loop iterator. There are 3 possibilities:

- Unit stride
- Constant stride
- Irregular stride

Unit Stride

Clearly unit stride is the best for performance as it naturally works over data that is contiguous in memory. This should be the preferred access pattern wherever possible.

Constant Stride

The iterator moves in constant increments. The compiler may have a couple of responses to this situation. If the stride value is small, it may continue to load the contiguous array into the register and use a bit mask to execute only on the subset of elements. As previously noted, the use of masks reduces the number of operations that can complete within each instruction cycle. If the stride value is large, it is common for the compiler to gather/scatter the data in order arrange the data into a vectorisable sequence.

The presence of constant stride in a loop may be a clue that data ordering is not optimal. In this case it is worth considering whether an Array of Structures to Structure of Arrays transformation can be applied. More generally, despite reducing encapsulation, the Structure of Arrays layout should be preferred for performance of vectorised code.

```
// Array of Struct
struct Pixel {
        float r;
        float g;
        float b;
        float x;
        float y;
      }
Pixel ArrayOfStruct[ ];
```

```
// Struct of Arrays
struct Pixels{
        float* r;
        float* g;
        float* b;
        float* x;
        float* y;
      }
Pixels StructOfArrays;
```

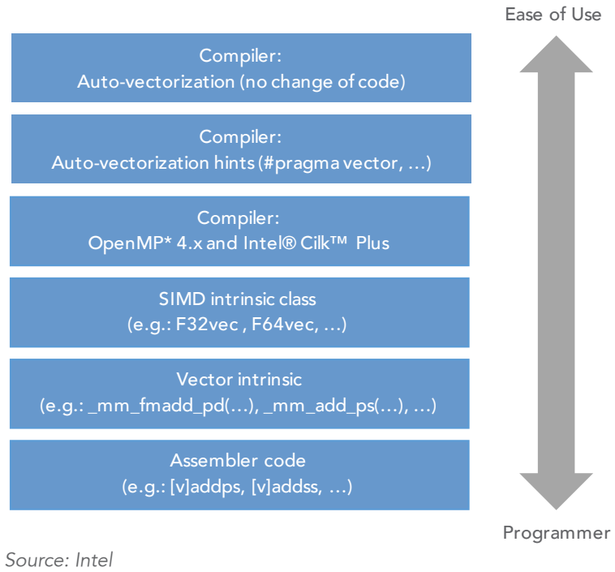
Irregular Stride

With an irregular stride, data is accessed in an unpredictable manner. The only option for the compiler here is to use the gather/scatter instructions to repack the data into contiguous memory. The developer may also attempt to implement some pre-processing of the data to achieve the same result, however the gather/scatter instructions introduced in AVX-512 are likely more efficient. Development effort may be better spent refactoring the code to arrange the data in a more optimal order.

Implementing Vectorisation

Alternatives

There are a range of alternatives and tools for implementing Vectorisation. They vary in terms of complexity, flexibility and future compatibility.



Source: Intel

Intel's 6 Step Program for Vectorisation

The simplest way to implement vectorisation is to start with Intel's 6-step process. This process leverages Intel tools to provide a clear path to transforming existing code into modern, high-performance software leveraging multicore and manycore processors.

Step 1. Measure baseline release build performance

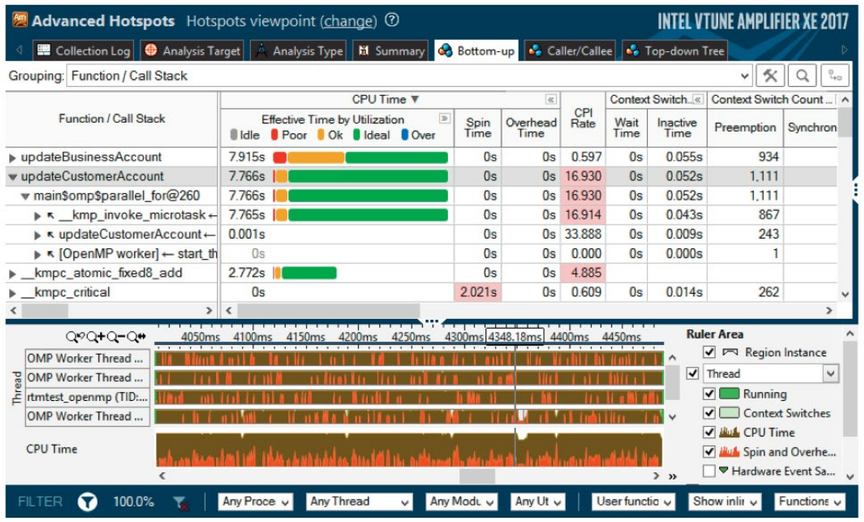
The starting point is a reference release build. A release build is important because:

1. The compiler will optimise your code
2. You need to have a baseline to measure how vectorisation is improving performance

Ideally, you should set a goal for performance to know when you are done.

Step 2. Determine hotspots

Tools like Intel's performance profiler VTune™ Amplifier XE can be used to profile your application to find the most time-consuming areas of code or "Hotspots." Identifying Hotspots helps focus effort on the areas of optimisation that will generate the most benefit.



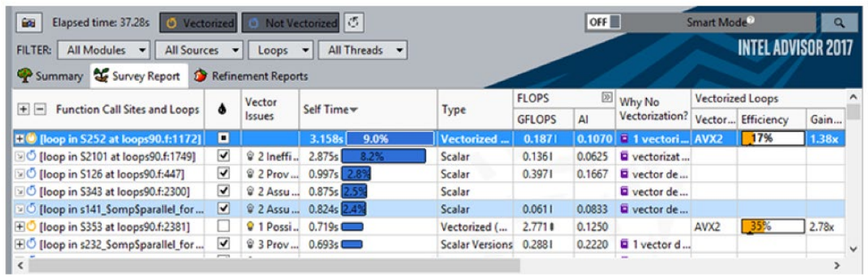
Intel VTune Amplifier XE

Step 3. Determine loop candidates

Compiler reports like Intel's Compiler Optimisation Report can tell you which loops are suitable for vectorisation. Loops in hotspots that are not automatically vectorisable maybe able to be modified using various techniques to allow them to be vectorised.

Step 4. Analyse specific hotspot code to measure performance gains

Tools like Intel's Advisor can be used to measure potential benefits from vectorisation of specific code to help focus effort for the maximum gain.



Intel Advisor

Step 5. Implement Vectorisation Recommendations

Implement recommendations for vectorising code using re-ordering of code, compiler hints or other methods.

Step 6. Repeat

The process is iterative and should be repeated till the desired performance is reached.

Applying Vectorisation to CVA Aggregation

As noted, the Finance domain provides many good candidates for vectorisation. A particularly good example is the aggregation of Credit Value Adjustment (CVA) and other measures of counterparty risk. The most common general purpose approach to calculation of 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 randomised simulated 'paths', we can estimate the distribution of forward values, giving both the expected and extreme 'exposures'. The simulation step results in a 3-dimensional array of exposures, the dimensions are [trades][paths][dates]. The task of calculating CVA from these exposures occurs in several steps: Netting, Collateralisation, 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.

The inner loop here is clearly a classic candidate for vectorisation and provided care is taken to arrange the data in appropriate order, the compiler will take care of this automatically.

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

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]]*
        beta[bIdx[t][d]];
```

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

A Copy into properly aligned arrays.

Then all operations become matrix addition or Hadamard product.

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

B Rely on improved scatter/gather produced by compiler auto-vectorisation.

In our testing, it appears 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 vectorisation 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 path wise values are independent.

Collateral

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

Variation Margin = Max(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 recognisable vectorised 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 collateralised 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 Period of Risk

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 uncollateralised exposure. To introduce MPR into our model, we compare the netExposure at date t, to the collateral that can 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 tmargin = t – Δt. Then our variation margin amount is determined using these interpolated exposures.

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


The second step, calculating collateralised exposures, remains unchanged and still uses the net exposures at the original exposure dates.

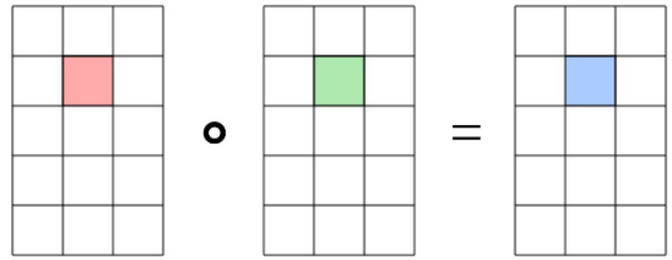
Again, note here that the results along each path are independent, so there is an opportunity to process multiple paths in concurrent threads.

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 element-wise operations on the matrix.



CVA

Our target risk measure. Integrate over time. Weight by default prob.

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

References

Vectorization, Kirill Rogozhin, Intel, March 2017
Vectorization of Performance Dies for the Latest AVX SIMD, Kevin O'Leary, Intel, Aug 2016,
A Guide to Vectorization with Intel® C++ Compilers, Intel, Nov 2010,
Vectorization Codebook, Intel, Sep 2015,
The Free Lunch Is Over - A Fundamental Turn Toward Concurrency in Software, Herb Sutter, March 2005
Recipe: Using Binomial Option Pricing Code as Representative Pricing Derivative Method, Shuo-li, Intel, June 2016

Quantifi is a specialist provider of risk, analytics and trading solutions. Our award-winning suite of integrated pre and post-trade solutions allows market participants to better value, trade and risk manage their exposures and responds more effectively to changing market conditions. Quantifi is trusted by the world's most sophisticated financial institutions, including five of the six largest global banks, two of the three largest asset managers, leading hedge funds, insurance companies, pension funds, and other financial institutions across 40 countries. Renowned for our client focus, depth of experience, and commitment to innovation, Quantifi is consistently first-to-market with intuitive, award-winning solutions.

enquire@quantifisolutions.com | www.quantifisolutions.com

EMEA +44 (0) 20 7248 3593 NA +1 212 784 6815 APAC +61 (02) 9221 0133

Follow us on 

Intel is a world leader in the design and manufacturing of essential product and technologies that power the cloud and an increasingly smart, connected world. Intel delivers computer, networking, and communications platforms to a broad set of customers including original equipment manufacturers (OEMs), original design manufacturers (ODMs), cloud and communications service providers, as well as industrial, communications and automotive equipment manufacturers. We are expanding the boundaries of technology through our relentless pursuit of Moore's Law and computing breakthroughs that make amazing experiences possible. We were incorporated in California in 1968 and reincorporated in Delaware in 1989.

www.intel.com

View our related webinar:
https://www.quantifisolutions.com/applying-vectorisation-to-cva-aggregation-video