Applying Vectorisation to CVA Aggregation
The Rise of Parallelism

- Moore’s Law continues to prevail
- Clock speeds have flat-lined
The Rise of Parallelism

- Number of cores increasing
- SIMD registers widening

<table>
<thead>
<tr>
<th>Up to Core(s)</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>12</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up to Threads</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>12</td>
<td>24</td>
<td>36</td>
</tr>
<tr>
<td>SIMD Width</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>Vector ISA</td>
<td>Intel® SSE3</td>
<td>Intel® SSE3</td>
<td>Intel® SSE4.2</td>
<td>Intel® AVX</td>
<td>Intel® AVX</td>
<td>Intel® AVX2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Intel® Xeon® Scalable Processor</th>
<th>~</th>
<th>~</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up to 28 cores</td>
<td>57-61</td>
<td>72</td>
</tr>
<tr>
<td>Up to 56 threads</td>
<td>228-244</td>
<td>288</td>
</tr>
<tr>
<td>SIMD Width 512</td>
<td>512</td>
<td>512</td>
</tr>
<tr>
<td>Vector ISA AVX-512</td>
<td>IMCI 512</td>
<td>Intel® AVX-512</td>
</tr>
</tbody>
</table>
The Rise of Parallelism

- ‘The Free lunch is over’
- Modern software must be built for parallelism
- The combination of both Threading and Vectorization provides the most dramatic performance gains
- **Intel® Xeon® Platinum 8180 Processor**: 28 cores x 16 SP values = 448x speedup
Vectorization

- Transformation of sequential code to exploit vector processing capabilities (SIMD) of modern processors
What Kind of Problem is Vectorizable?

- 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.
- So, first of all, look for a loop.
Issues Impacting Vectorization

1. Loop dependencies (read after write)

2. Pointer aliasing

3. Indirect memory access

4. Non straight-line code (function calls, conditional statements, etc)

```c
for(i=0;i<max;i++) {
    a[i] = a[i-1]*c[i];
}

void mul(double *in, double *out, int max){
    for (int i = 0; i < max; i++) {
        out[i] = in[i] * value;
    }
}
for(i=0;i<max;i++) {
    a[i] = b[ind[i]]*c[i];
}
for(i=0;i<end();i++) {
    if(Jump())
        i += CalcJump();
    a[i] = b[i]*c[i];
}
Implementing Vectorization

- There are a range of approaches for implementing vectorization

<table>
<thead>
<tr>
<th>Method</th>
<th>Control</th>
<th>Portability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Performance Libraries (MKL, IPP)</td>
<td>Easy</td>
<td>Low</td>
</tr>
<tr>
<td>Compiler: Auto-vectorization (no change of code)</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Compiler: Auto-vectorization hints (#pragma vector, ...)</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Explicit (user mandated) Vector Programming: OpenMP4.x, Intel Cilk Plus</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>SIMD intrinsic class (e.g.: F32vec, F64vec, ...)</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Vector intrinsic (e.g.: __m128_t, _mm_add_ps, ...)</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Assembler code (e.g.: [v]addps, [v]addss, ...)</td>
<td>Low</td>
<td>Low</td>
</tr>
</tbody>
</table>
6 Step Vectorization Program

1. Measure baseline performance
2. Determine hotspots
3. Check compiler report and Determine loop candidates
4. Analyse loop candidates
5. Implement Vectorization
6. Repeat
Step 1. Measure Baseline Performance

• The starting point is a reference release build.
  • A release build is important because the compiler will optimize your code
  • Use architecture specific compiler options
    – -xSSE4.1, -xCORE-AVX2, -xCORE-AVX512, -xHost for current architecture, -O3 is a good addition

• You need to have a baseline to measure how vectorization is improving performance
Step 2. Determine Hotspots

- Intel’s performance profiler VTune™ Amplifier XE shows the most time-consuming areas of code or “Hotspots” of your application.
Step 3. Determine Loop Candidates

- Compiler reports like *Intel's Compiler Optimization Report* can tell you which loops are suitable for vectorization.

- Generate compiler report
  - `-opt-report=5 opt-report-phase=vec`

- LOOP BEGIN at C:\Zips\benchmarks\vecif\vecif\vecif.cpp(14,2)
  remark #15344: loop was not vectorized: vector dependence prevents vectorization
  remark #15346: vector dependence: assumed ANTI dependence between counter (15:3) and counter (18:4)

- LOOP BEGIN at C:\Zips\benchmarks\vecif\vecif\vecif.cpp(27,7)
  remark #15301: SIMD LOOP WAS VECTORIZED
  remark #15478: estimated potential speedup: 3.500

- Non-vectorized loops may need your attention.
Step 4. Analyse specific hotspot code

- **Intel’s Advisor** helps focus effort for the maximum gain.
Step 5. Implement Recommendations

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

```c
#pragma simd
for (i=0; i<p; i++) {
    a[i] = b[i] * c[i];
    sum = sum + a[i];
}
```

```c
__declspec(vector)
float foo(float a, float b, float c, float d) {
    return a * b + c * d;
}

__declspec(vector(uniform(a),linear(i)))
void foo(float *a, int i);

-restrict compiler option
void mul(double *restrict in, double * restrict out, int max){
    for (int i = 0; i < max; i++) {
        out[i] = in[i] * value;
    }
}

#pragma simd reduction(+:sum)
for(i=0;i<p;i++) {
    a[i] = b[i] * c[i];
    sum = sum + a[i];
}
```
Step 6. Repeat

- The process is iterative and should be repeated till the desired performance is reached.
Applying Vectorization to CVA Aggregation

- The Finance domain provides many good candidates for vectorization
- Aggregation of Credit Value Adjustment (CVA) is a great example
- Usually done by Monte-Carlo simulation of exposures
- Simulation produces a multi-dimensional array [trades][paths][dates]
- We aggregate across each dimension to reduce to a single number: CVA
Netting

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

```java
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];
```

- The inner loop here is a classic candidate for compiler auto-vectorization.
- Elementwise addition, arrays of equal length.
- When using pointers, the compiler will require a hint that there are no dependencies.
Netting

- We may need to interpolate onto a new time grid

```c
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]];  
```

- Elementwise addition and multiplication.
- Arrays of different lengths. Indirect memory access means vectorization will be less efficient than simple case, but still possible to achieve gains
Collateral

- Variation Margin
  - Margin covers exposure in excess of threshold $K$
  - Variation Margin = $\text{Max}(\text{net exposure} - K, 0)$

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

- Collateralized Exposure

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

- Elementwise subtraction, arrays of equal length
- $\text{Max}$ function supports vectorization
Expected Exposure

- Total over paths then take average

```java
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;
```

- Inner loop auto-vectorizable
- Elementwise addition, arrays of equal length
- Elementwise division by a constant
CVA

- Integrate over time. Weight by probability of default.

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

- Elementwise multiplication and addition. Arrays of equal length
Q&A