1. Flow analysis of Motor Bike and Car.
2. Gauss Siedel Solver used for solving $AXx = b$.
3. Transform $(L + D + U) * x = b$ 
   $=> (L + D) * x = b - U * x$.

Figure: Motor Bike CFD*

*Source: Openfoam tutorial

Gauss Siedel iteration : $(L + D) * x^{new} = b - U * x^{old}$
What’s unique about my tuning work

- The OpenFOAM® (Open Field Operation and Manipulation) CFD Toolbox is a free, open source CFD software package which has a large user base across most areas of engineering and science, from both commercial and academic organizations.

- **Application domain:** Computational Fluid Dynamics

- **Execution mode:** Symmetric MPI, Cluster

- **The following tools were used in the analysis and optimization cycles:**
  - Intel vTune Analyzer, Intel Trace Analyzer and Collector, idb debugger, Intel compiler parallelization reports
Insights

• **What I learned:**
  - Effective load distribution Xeon & Xeon Phi in symmetric mode give better performance. This is due to difference in CPU frequency between Xeon:Xeon Phi and total number of cores present in both of them
  - Reduction in neighboring communication in unstructured mesh between Xeon & Xeon Phi gives better performance
  - IO penalty is high on Xeon Phi as compare to Xeon
  - System tuning like Huge Page = 2MB, compiler flags gives signification improvements.

• **What I recommend:** Effective use of tools to diagnose performance problems.

• **Most effective optimization: Vectorization Intrinsics**
  - Changes to get efficient vectorization and avoid compiler overhead due to VGATHER & VSCATTER instructions
  - Unrolling
  - Prefetching
  - Explored decomposition algorithm change for weighted decomposition support

• **Surprises:** Oversubscription works better in native Xeon Phi mode when IO is turned off.
Performance – Motorbike Case

- **Compelling Performance:**
  - Haswell optimized: 164s
  - Haswell + Xeon Phi optimized: 119s (164/119 = 1.38 X)

- **Competitive performance:**
  - There are no published numbers for GPU's for OpenFOAM benchmark application.

- **Results:**
  - 1.38 X Speedup due to Xeon Phi addition w.r.t Xeon optimized result
Issues with GaussSeidelSmooother Loop

- Inefficient use of cacheline
- Scatter/Gather overhead
- Low trip count, on average ~4, not good for vectorization
- Outer loop is not parallelizable because of dependencies
- Indirect Referencing:
  - Inefficient use of cacheline
  - Scatter/Gather overhead
- Overhead of scalar “DIV” operation inside loop
- Unaligned data structures throughout the application
Identify vector assembly overhead

for (facei=fStart; facei<fEnd; facei++)
psii -= upperPtr[facei]*psiPtr[uParam[facei]];

vgatherdpdq (%r14,%zmm14,8), %k4, %zmm15
vgatherdpdq (%r13,%zmm11,8), %k1, %zmm16
vgatherdpsl (%r13,%zmm10,4), %k3, %zmm14
vgatherdpsl (%r12,%zmm0,4), %k2, %zmm11

... (Multiple gathers, due to indirect reference, unrolling and peel & remainder loops)

1. The FOR loop overhead is high as it has only a small trip count in most of the cases.

2. By default : vectorized and unrolled by 4

3. Compiler performing reduction operation seems costlier as visible in assembly.

4. Peel and remainder loops are introducing overheads since trip count is < 8 for 99% of the time.

These issues suggest that manual vectorization using intrinsics might be beneficial.

Vtune profiling & ICC compiler vec-report leads to bottleneck identification.
<table>
<thead>
<tr>
<th></th>
<th>Time (s)</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>First Loop</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Baseline</td>
<td>486</td>
<td>By default vectorized</td>
</tr>
<tr>
<td>Intrinsic</td>
<td>186</td>
<td></td>
</tr>
<tr>
<td><strong>Speedup 2.6 X</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Second Loop</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Baseline</td>
<td>221</td>
<td>By default non-vectorized</td>
</tr>
<tr>
<td>Baseline + pragma ivdep</td>
<td>412</td>
<td>Enabling vectorization degrades performance</td>
</tr>
<tr>
<td>Intrinsic</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td><strong>Speedup 1.7 X</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: Time taken from Vtune by aggregating assembly

Vtune profiling & ICC compiler vec-report/opt-report used for detailed analysis
### Prefetching using Intrinsics: Vtune Snapshot

#### Baseline:

<table>
<thead>
<tr>
<th>Source</th>
<th>CPU Time: Total</th>
<th>Cache Usage: Self</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>L1 Misses</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L1 Misses</td>
</tr>
<tr>
<td>409</td>
<td>6.446s</td>
<td>0</td>
</tr>
<tr>
<td>411</td>
<td></td>
<td>58.748s</td>
</tr>
</tbody>
</table>

```c
// Multiply the field by coefficients and add into the result
forall(faceCells, elemI)
{
    result[faceCells[elemI]] = coeffs[elemI]*scalarReceiveBuf[elemI];
}
```

#### Optimized:

<table>
<thead>
<tr>
<th>Source</th>
<th>CPU Time: Total</th>
<th>Cache Usage: Self</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>L1 Misses</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L1 Misses</td>
</tr>
<tr>
<td>414</td>
<td>0.558s</td>
<td>0</td>
</tr>
<tr>
<td>416</td>
<td></td>
<td>31.381s</td>
</tr>
<tr>
<td>417</td>
<td></td>
<td>14.208s</td>
</tr>
</tbody>
</table>

```c
register const label len = faceCells.size();

#pragma no prefetch

forall (register label elemI=0; elemI<len; elemI++)
{
    _mm_prefetch((char *)&result[faceCells[elemI+4]], _MM_HINT_T1);
    result[faceCells[elemI]] = coeffs[elemI]*scalarReceiveBuf[elemI];
    _mm_prefetch((char *)&result[faceCells[elemI+8]], _MM_HINT_T0);
    _mm_prefetch((char *)&coeffs[elemI+64], _MM_HINT_T1);
    _mm_prefetch((char *)&coeffs[elemI+32], _MM_HINT_T0);
    _mm_prefetch((char *)&scalarReceiveBuf[elemI+64], _MM_HINT_T1);
    _mm_prefetch((char *)&scalarReceiveBuf[elemI+32], _MM_HINT_T0);
}
```
Huge memory pages (2MB) are often necessary for memory allocations on the coprocessor. With 2MB pages, TLB misses and page faults may be reduced, and there is a lower allocation cost.

### Without huge pages vs huge pages : difference analysis using VTune

<table>
<thead>
<tr>
<th>Function / Call Stack</th>
<th>CPU Time: Difference</th>
<th>CPU Time: r005ge</th>
<th>CPU Time: r004ge</th>
</tr>
</thead>
<tbody>
<tr>
<td>[libOpenFOAM.so]</td>
<td>5818.818s</td>
<td>5818.818s</td>
<td></td>
</tr>
<tr>
<td>func@0xe7325</td>
<td>4912.119s</td>
<td>4912.119s</td>
<td></td>
</tr>
<tr>
<td>[vmlinux]</td>
<td>911.528s</td>
<td>2354.130s</td>
<td>1442.602s</td>
</tr>
</tbody>
</table>

### Hardware Metrics

- Clockticks: 8,319,150,000,000 - 9,175,320,000,000 = -856,169,999,999
- Instructions Retired: 3,033,800,000,000 - 3,247,510,000,000 = -213,709,999,999
- CPI Rate: 2.742 - 2.825 = -0.083
- L1 Misses: 56,477,050,000 - 49,516,600,000 = 6,960,450,000
- L1 Hit Ratio: 0.956 - 0.966 = -0.010
- Estimated Latency Impact: 98.744 - 137.408 = -38.664
- L1 TLB Miss Ratio: 0.010 - 0.019 = -0.010
- L2 TLB Miss Ratio: Not changed, 0.000
- L1 TLB Misses per L2 TLB Miss: 774.667 - 422.932 = 351.735
- Vectorization Intensity: 2.750 - 2.878 = -0.129
- L1 Compute to Data Access Ratio: 1.768 - 2.145 = -0.377
- L2 Compute to Data Access Ratio: 46.503 - 81.565 = -35.063
Optimization gains depend on the order in which they are applied.
Thank You!

Promise what we deliver.
Deliver what we promise. That’s certainty.
### Functions improved – Vtune counters

<table>
<thead>
<tr>
<th>Function</th>
<th>Baseline</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L1 Misses</td>
<td>L1 Hit Ratio</td>
</tr>
<tr>
<td>GaussSeidelSmother::smooth</td>
<td>6194650000</td>
<td>0.963597</td>
</tr>
<tr>
<td>lduMatrix::Amul</td>
<td>2143750000</td>
<td>0.958503</td>
</tr>
<tr>
<td>lduMatrix::sumA</td>
<td>566300000</td>
<td>0.958976</td>
</tr>
<tr>
<td>lduMatrix::sumMagOffDiag</td>
<td>266000000</td>
<td>0.967213</td>
</tr>
<tr>
<td>GAMGSolver::agglomerateMatrix</td>
<td>280000000</td>
<td>0.937792</td>
</tr>
<tr>
<td>fvc::surfaceSum</td>
<td>154000000</td>
<td>0.966514</td>
</tr>
<tr>
<td>processorFvPatchField::updateInterfaceMatrix</td>
<td>616000000</td>
<td>0.92781</td>
</tr>
<tr>
<td>processorGAMGInterfaceField::updateInterfaceMatrix</td>
<td>700000000</td>
<td>0.952449</td>
</tr>
</tbody>
</table>

L1 hit ratio has improved throughout, and L1 misses have decreased.