Vectorization Efficiency Metrics

Zakhar A. Matveev, CJ Newburn

with input from Dmitry Prohorov, Hideki Saito
Metrics list
Vectorization Metrics

- Actual speed-up (could be: wall-clock, total, inclusive/exclusive):
  - \( S = \frac{\text{Time (Scalar Loop)}}{\text{Time (Vector Loop)}} \)
  - Efficiency = \( \frac{G}{\text{Max}_S} \), \( \text{Max}_S \leq \text{Max}_{VL} \)

- Gain/Efficiency:
  - \( G = \frac{\text{Scalar Loop Cost (cycles)}}{\text{Vector Loop Cost (cycles)}} \)
  - Efficiency = \( \frac{G}{\text{Max}_G} \), \( \text{Max}_G \leq \text{Max}_{VL} \)

How much faster the vector iterations are? Reflects quality of compiler code-generation.

*Should be equal to* \( S \) *in case of VPU-bound codes*
Vectorization Metrics

- **Path Reduction:**
  
  \[
  \text{Scalar Loop Path (# instructions)} / \text{Vector Loop Path (# instructions)}
  \]

  Gives a sense of the fraction of non-vector (overhead) instructions in a loop

- **Vector Utilization / Intensity** ("elements active") – only works on KNC

  \[
  \text{VPU_ELEMENTS_ACTIVE / VPU_INSTRUCTIONS_EXECUTED}
  \]

  Fraction of vector instructions that do work on vector registers. Reflect vector registers utilization

  If a mask bit is set for an element, it was presumed to be used. Drops for branchy if-else codes
Intensity/Utilization

Advantage:
- Dynamically “measures” fraction of vector instructions that do work (mask-aware)

Disadvantages:
- Only available on KNC (other metrics could be computed on IVB or Broadwell)
- More work/utilization doesn't mean more speed-up (if you care)
  - Some code may have good vector utilization, but scalar version could be faster than it!
  - Shifts/shuffles/“misc.”, prefetch instructions are counted “inappropriately”.
- Assumption: Mask bits are only set for elements in which useful work is done
  - This is false: Extra mask bits can be set, as long as there are no side effects.
- This is per binary loop, so separate values for peel/remainder..
Gain/Efficiency Estimate

Advantage:

- Accounts all nuances of vector vs. scalar assembly and maps it to speed-up

Disadvantages:

- Usually not available if you program in assembly/intrinsics.
- This is code generation performance model, not measurement. This doesn’t account dynamic mask values as well as other dynamic data (trip counts).

Advisor Gain/Efficiency:

- Recalculate (calibrate) Compiler Gain/Efficiency taking into account dynamic knowledge of trip counts, peel-remainder times.
- But limited to Xeon right now.
Tools to calculate metrics
Intensity/Utilization: VTune Amplifier XE 2016 Beta for KNC

Part of Intel Parallel Studio XE 2016
Gain/Efficiency: Intel Compiler (>=15.x)

Intel Compiler:
-O2 -opt-report5

Part of Intel Parallel Studio XE 2016
Gain/Efficiency: Advisor XE 2016 Beta (for Xeon only)

<table>
<thead>
<tr>
<th>Loops</th>
<th>Loop Type</th>
<th>Self Time</th>
<th>Vectorized Loops</th>
<th>Efficiency</th>
<th>Estimated Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>[loop in fCollisionBGK at lbpBGK.cpp:840]</td>
<td>Vectorized: Expand</td>
<td>0.020s</td>
<td>AVX</td>
<td>100%</td>
<td>2.05</td>
</tr>
<tr>
<td>[loop in fGetActive at lbpGET.cpp:152]</td>
<td>Vectorized: Expand</td>
<td>0.030s</td>
<td>AVX</td>
<td>36%</td>
<td>2.90</td>
</tr>
<tr>
<td>[loop in fGetOneMassSite at lbpGET.cpp:...</td>
<td>Vectorized: Expand</td>
<td>0.089s</td>
<td>AVX</td>
<td>36%</td>
<td>2.86</td>
</tr>
<tr>
<td>[loop in fGetEquilibriumF at lbpSUB.cpp:729]</td>
<td>Vectorized: Collapse</td>
<td>0.579s</td>
<td>AVX</td>
<td>25%</td>
<td>2.00</td>
</tr>
<tr>
<td>[loop in fGetEquilibriumF at lbpSUB.cpp:...</td>
<td>Vectorized (Body)</td>
<td>0.431s</td>
<td>AVX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[loop in fGetEquilibriumF at lbpSUB.cpp:...</td>
<td>Vectorized (Remainder)</td>
<td>0.087s</td>
<td>AVX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[loop in fGetEquilibriumF at lbpSUB.cpp:...</td>
<td>Remainder</td>
<td>0.061s</td>
<td>AVX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[loop in fPropagationSwap at lbpSUB.cpp:1 ...</td>
<td>Vectorized (Body)</td>
<td>1.259s</td>
<td>AVX</td>
<td>~14%</td>
<td>0.54</td>
</tr>
</tbody>
</table>

Part of Intel Parallel Studio XE 2016

**Advisor**: Survey Analysis Type
Some word on methodology..
WHAT to measure?

Actual Speed-up vs. Efficiency vs. Intensity. ??

- Measuring all and comparing results –
  is “useful enough” exercise already.

- Normally stick with at least one of them for workshop exercises.

Kernel vs. Sub-part vs. Workload ??

- Per-workload speed-up/efficiencies are lower than per-kernel (Amdahl's law)

- Both are important to understand, but don’t mix them up!

- For big HPC codes you rarely even look into everything. Define sub-set.

- Measuring / establishing proper baselines is very important/not-trivial itself
Some take-aways

Vectorization efficiency/gain

- Take it as input, but treat it as performance estimate
- Use Advisor if you want to overcome some of “static code-generation knowledge” limits

Vectorization intensity

- Take it as input, but don’t treat it as accurate:
  - Low intensity definitely means you have an issue. Otherwise – who knows.
  - If higher than expected, inspect code for masks that are all 1 even through conditionals
  - The VPU_ELEMENTS_ACTIVE won’t be available for anything other than KNC

Don’t compare apples with oranges (kernel and workload, etc)

Don’t mix up dimensional and non-dimensional metrics