Unification of static analyses and runtime measurements for improving vectorization

Ashay Rane, Rakesh Krishnaiyer, Chris Newburn,
James Browne, Leo Fialho and Zakhar Matveev

4th August, 2014
Petascale Tools Workshop

Overview of this work

Goal: To increase the applicability and efficiency of vectorization by:

1. Understand compiler vectorization messages.
2. Find what information is the compiler missing.
3. Gather and analyze runtime measurements.
4. Feed runtime information back to compiler.

Why vectorization?

- Increased SIMD vector lengths, hence perf boost.
- Improves energy efficiency of the processor.
- Inherent limitations for compiler because of lack of runtime information.
- Lots of headroom available to improve vectorization.

Time taken by non-vectorized loops

Application	Time
heartwall	07.43%
euler	12.42%
kmeans	19.54%
backprop	32.52%
leukocyte	35.01%
lavaMD	37.42%
srad_v1	48.45%
pre_euler_double	71.60%
pre_euler	75.94%
euler_double	78.99%
streamcluster	85.58%

Causes of poor vectorization

Limited information available at compile-time, hence compiler assumed:

- Inter-iteration dependence.
- Varying trip count (non-countable loop).
- Temporal array references.
- Mis-aligned loads and stores.

Reasons for poor vectorization

Example: Rodinia LavaMD.

- Hot function `kernel_cpu(box* b, fp* qv, ...)` defined in `kernel_cpu.c`.
- Compiler does not know caller arguments when compiling `kernel_cpu.c`.
- Assumes pointers `b` and `qv` may overlap in memory.
- Concludes possible existence of vector dependence.

Reasons for poor vectorization

Example: NAS CG.

- Unknown loop trip count: `for (k = rowstr[j]; k < rowstr[j+1]; k++) { }`
- Double indirection in loop body: `suml += a[k]*p[colidx[k]];` .
- Compiler generates gather and scatter instructions for each iteration.

Reasons for poor vectorization

Example: NBody.

- Operates on dynamically allocated (`malloc()`ed) arrays
- Memory allocator may allocate objects in any way that it desires.
- Compiler cannot guarantee alignment of objects to cache-line boundary.

Our contributions - MACVEC tool

1. What information does the compiler need?
2. How to measure without high overhead?
3. How to feed information back to compiler?

Tool (MACVEC) workflow

1. Profile application for hotspots using production inputs.
2. Parse compiler vectorization reports to find loops not fully vectorized.
3. Instrument hot-loops that are not fully vectorized.
4. Gather measurements, analyze results and generate recommendations.
5. Verify validity of the recommended changes.
6. Implement changes, measure performance gains.

Tool (MACVEC) workflow

1. Profile application for hotspots using production inputs.
2. Parse compiler vectorization reports to find loops not fully vectorized.
3. Instrument hot-loops that are not fully vectorized.
4. Gather measurements, analyze results and generate recommendations.
5. Verify validity of the recommended changes.
6. Implement changes, measure performance gains.

Automated step

Manual step

Dynamic profiling measurements

- Loop trip counts.
- Array access strides.
- Alignment of arrays.
- Overlapping pointers.
- Non-temporal or streaming stores.
- Branch path outcomes.

Measurement collection overhead

Measurement	**Overhead (geo. mean)**
Trip count	1.08x
Strides	1.05x
Alignment	1.12x
Pointer overlap	1.07x
Branch outcomes	1.07x

Rule-based recommendations

Loop trip count

Precondition:

- Loop trip count less than threshold (1024).

Recommendation: `#pragma loop_count(n)`

Rule-based recommendations

Stride

Precondition:

- Non-unit but fixed-length strides for specific data structures.

Recommendation: Convert from array-of-structs to struct-of-arrays refs.

Rule-based recommendations

Stride

Precondition:

- Code to be compiled for Intel Xeon Phi.
- Fixed-length strides that are more than 4 cache lines apart.

Recommendation: `#pragma prefetch array`, `-opt-gather-scatter-unroll`.

Rule-based recommendations

Alignment

Precondition:

- All arrays aligned to cache-line boundary.
- Loop is vectorizable.

Recommendation: `#pragma vector aligned`.

Rule-based recommendations

Non-temporal stores

Precondition:

- Low reuse for arrays used in loop body.
- Loop is vectorizable.

Recommendation: `#pragma vector nontemporal` .

Rule-based recommendations

Streaming stores

Precondition:

- Arrays are written but never read back.
- Arrays are accessed with unit stride, no mask register.
- Low reuse for specific array.

Recommendation: `-opt-streaming-stores=always`.

Rule-based recommendations

Pointer-overlap checks

Precondition:

- Span of memory accessed using pointers does not overlap with other pointer accesses.

Recommendation: `restrict` keyword.

Rule-based recommendations

Branch path analysis

Precondition:

- Branch evalutes to always true or always false.

Recommendation: `__builtin_expect()`.

Results: running time improvements

Validation applications	**Xeon**	**Xeon Phi**
NBody	0.93x	1.45x
STREAM Copy	1.06x	1.00x
STREAM Scale	1.41x	1.32x
STREAM Add	1.30x	1.29x
STREAM Triad	1.29x	1.30x

Results: running time improvements

Small benchmarks	Xeon	Xeon Phi
NAS CG	1.06x	2.18x
LavaMD	2.19x	8.99x
SRAD	0.99x	1.09x

Results: running time improvements

Full applications	**Xeon**	**Xeon Phi**
LBM	1.06x	1.20x
Lulesh	1.03	1.00x
MILC	1.10x	1.60x

Safety of recommended changes

- Are recommendations independent of standard compiler optimizations?
- Will recommendations be applicable across multiple program inputs?

- Seven of the nine recommendations are guaranteed to be safe.
- O(1) runtime checks guarantee safety for remaining recommendations.

Summary

- Identified some key metrics necessary to improve vectorization.
- Combined static and dynamic information to generate recommendations.
- MACVEC will be available in the next release of PerfExpert.