Using Machine Learning to Improve Automatic Vectorization

Kevin Stock **Louis-Noël Pouchet** P. Sadayappan

The Ohio State University

January 24, 2012
HiPEAC Conference
Paris, France

Vectorization

Observations

- Short-vector SIMD is critical in current architectures
- Many effective automatic vectorization algorithms:
  - Loop transformations for SIMD (Allen/Kennedy, etc.)
  - Hardware alignment issues (Eichenberger et al., etc.)
  - Outer-loop vectorization (Nuzman et al.)
- But performance is usually way below peak!
  - Restricted profitability models
  - Usually focus on reusing data along a single dimension

Our Contributions

1. Vector code synthesizer for short-vector SIMD
   - Supports many optimizations that are effective for Tensors
   - SSE, AVX
2. In-depth characterization of the optimization space
3. Automated approach to extract program features
4. Machine Learning techniques to select at compile-time the best variant
5. Complete performance results on 19 benchmarks / 12 configurations

Considered Transformations

1. **Loop order**
   - Data locality improvement (for non-tiled variant)
   - Enable Load/Store hoisting
2. **Vectorized dimension**
   - Reduction loop, Stride-1 access
   - May require register transpose
3. **Unroll-and-jam**
   - Increase register reuse / arithmetic intensity
   - May be required to enable register transpose

Example

```
procedure IKJ(A_ki, B_jk, C_ij)

  for (i ← 0 ; i < M ; i++) do
    for (k ← 0 ; k < K ; k+ = 4) do
      a0[0 : 3] ← SPLAT(A[k + 0][i])
      a1[0 : 3] ← SPLAT(A[k + 1][i])
      a2[0 : 3] ← SPLAT(A[k + 2][i])
      a3[0 : 3] ← SPLAT(A[k + 3][i])
      for (j ← 0 ; j < N ; j+ = 4) do
        b0[0 : 3] ← B[j + 0][k : k + 3]
        b1[0 : 3] ← B[j + 1][k : k + 3]
        b2[0 : 3] ← B[j + 2][k : k + 3]
        b3[0 : 3] ← B[j + 3][k : k + 3]
        TRANSPOSE(b0, b1, b2, b3)
        c[0 : 3] ← C[i][j : j + 3]
        c[0 : 3]+ = a0[0 : 3] * b0[0 : 3]
        c[0 : 3]+ = a1[0 : 3] * b1[0 : 3]
        c[0 : 3]+ = a2[0 : 3] * b2[0 : 3]
        c[0 : 3]+ = a3[0 : 3] * b3[0 : 3]
        C[i][j : j + 3] ← c[0 : 3]
      end for
    end for
  end for
end procedure
```

Contraction

$$C_{ij} = \sum_k A_{ki} \cdot B_{jk}$$

- Vectorized along j
- B_{jk} transposed
- Each element of A_{ki} is splatted (broadcast) to all elements of a vector register

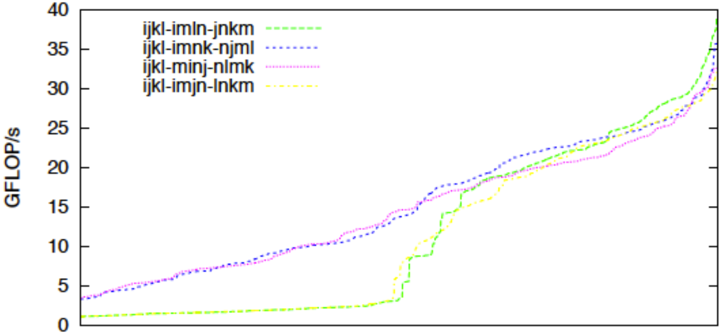
Observations

- The number of possible variants depends on the program
  - Ranged from 42 and 2497 in our experiments
  - It also depends on the vector size (SSE is 4, AVX is 8)
- We experimented with Tensor Contractions and Stencils
  - TC are generalized matrix-multiply (fully permutable)
  - Stencils

Experimental Protocol

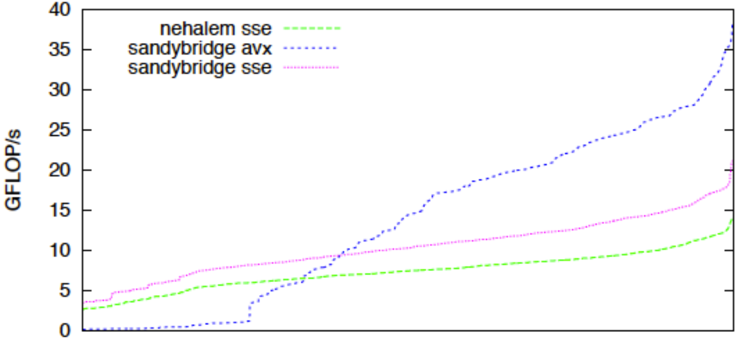
- Machines:
  - Core i7/Nehalem (SSE)
  - Core i7/Sandy Bridge (SSE, AVX)
- Compilers:
  - ICC 12.0
  - GCC 4.6
- Benchmarks:
  - Tensor Contractions ("generalized" matrix-multiply)
  - Stencils
  - All are L1-resident

Variability Across Programs



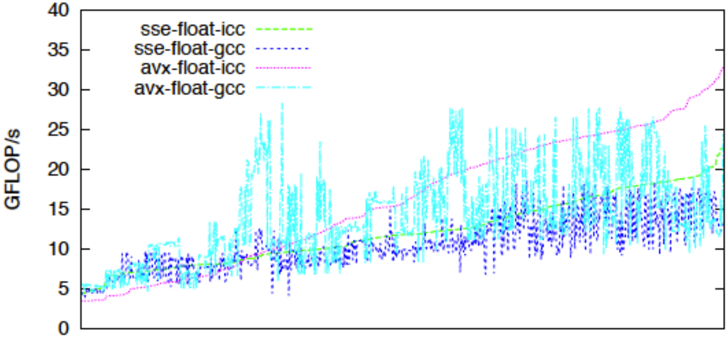
X axis: variants, sorted by increasing performance machine: Sandy Bridge / AVX / float

Variability Across Machines



X axis: variants, sorted by increasing performance

Variability Across Compilers



X axis: variants, sorted by increasing performance **for ICC**

Conclusions

1. The best variant depends on all factors:
   - Program
   - Machine (inc. SIMD instruction set)
   - Data type
   - Back-end Compiler
2. Usually a small fraction achieves good performance
3. Usually a minimal fraction achieves the optimal performance

Assembly Features: Objectives

Objectives: create a performance predictor

1. Work on the ASM instead of the source code
   - Important optimizations are done (instruction scheduling, register allocation, etc.)
   - Closest to the machine (without execution)
   - Compilers are (often) fragile
2. Compute numerous ASM features to be parameters of a model
   - Mix of direct and composite features
3. **Pure compile-time approach**

Assembly Features: Details

- **Vector operation count**
  - per-type count and grand total, for each type
- **Arithmetic Intensity**
  - Ratio FP ops / number of memory operations
- **Scheduling distance**
  - Count the distance between producer/consumer ops
- **Critical path**
  - Number of serial instructions

Static Model: Arithmetic Intensity

- Stock et al [IPDPS'10]: use arithmetic intensity to select variant
- Works well for some simple Tensor Contractions...
- **But fails to discover optimal performance** for the vast majority
- Likely culprits:
  - Features are missing (e.g., operation count)
  - The static model must be fine-tuned for each architecture

Machine Learning Approach

- Problem learn:
  - **PB1: Given ASM feature values, predict a performance indicator**
  - **PB2: Given the predicted performance rank by models, predict the final rank**
- Multiple learning algorithms evaluated (IBk, KStar, Neural networks, M5P, LR, SVM)
- Composition of models (weighted rank)
- Training on a synthesized set
- Testing on totally separated benchmark suites

Weighted Rank

- ML models often fail at predicting accurate performance value
- Better success at predicting the actual best variant
  - **Rank-Order** the variants, only the best ones really matter
  - Each model can give different answers
- Weighted Rank: combine the predicted **rank** of the variants
  - $(R_v^{IBK}, R_v^{K*}) \rightarrow WR_v$
  - Use linear regression to learn the coefficients

Experimental Protocol

- ML models: train 1 model per configuration (compiler $\times$ data type $\times$ SIMD ISA $\times$ machine)
- Use synthetic set for training
  - 30 randomly generated tensor contraction
  - Test set is fully disjoint
- Evaluate on distinct applications
  - CCSD: 19 tensor contractions (Couple Cluster Singles and Doubles)
  - 9 stencils operating on dense matrices
- Efficiency metric: 100% when the performance-optimal is achieved

Average Performance on CCSD (efficiency)

Config.	ICC/GCC	Random	St-m	IBk	KStar	LR	M5P	MLP	SVM	Weighted Rank
NSDG	0.42	0.64	0.82	0.86	0.85	0.83	0.81	0.84	0.83	0.86
NSDI	0.37	0.66	0.78	0.95	0.96	0.80	0.92	0.93	0.93	0.95
NSFG	0.31	0.53	0.79	0.91	0.86	0.64	0.86	0.80	0.63	0.90
NSFI	0.19	0.54	0.84	0.92	0.89	0.72	0.89	0.88	0.84	0.92
SADG	0.27	0.51	0.75	0.84	0.89	0.70	0.87	0.83	0.72	0.85
SADI	0.22	0.38	0.44	0.82	0.86	0.67	0.88	0.69	0.75	0.88
SAFG	0.21	0.49	0.65	0.81	0.82	0.68	0.81	0.81	0.67	0.81
SAFI	0.11	0.35	0.38	0.91	0.89	0.67	0.85	0.79	0.62	0.92
SSDG	0.43	0.67	0.86	0.88	0.85	0.83	0.78	0.85	0.75	0.87
SSDI	0.33	0.67	0.79	0.95	0.95	0.75	0.93	0.94	0.91	0.94
SSFG	0.33	0.53	0.82	0.88	0.87	0.63	0.88	0.78	0.63	0.88
SSFI	0.20	0.52	0.84	0.92	0.89	0.67	0.81	0.80	0.78	0.92
Average	0.28	0.54	0.73	0.88	0.88	0.71	0.85	0.83	0.75	0.89

Nehalem/**S**andybridge, **S**SE/**A**VX, **F**loat/**D**ouble, **I**CC/**G**CC

Average Performance on CCSD (GF/s)

Config.	Compiler min	Compiler avg	Compiler max	Weighted Rank min	Weighted Rank avg	Weighted Rank max	Improv.
NSDG	1.38GF/s	3.02GF/s	8.48GF/s	3.55GF/s	6.02GF/s	6.96GF/s	2.00×
NSDI	1.30GF/s	2.82GF/s	5.29GF/s	6.69GF/s	7.24GF/s	8.11GF/s	2.57×
NSFG	1.39GF/s	4.34GF/s	16.70GF/s	9.22GF/s	11.77GF/s	14.24GF/s	2.71×
NSFI	1.30GF/s	2.71GF/s	5.98GF/s	6.77GF/s	12.13GF/s	14.30GF/s	4.47×
SADG	2.31GF/s	4.55GF/s	11.63GF/s	10.35GF/s	14.26GF/s	17.88GF/s	3.13×
SADI	1.89GF/s	3.92GF/s	6.69GF/s	11.50GF/s	14.64GF/s	22.23GF/s	3.73×
SAFG	2.40GF/s	6.87GF/s	24.47GF/s	14.69GF/s	25.84GF/s	35.47GF/s	3.76×
SAFI	1.89GF/s	4.15GF/s	9.79GF/s	24.92GF/s	33.18GF/s	43.30GF/s	7.99×
SSDG	2.31GF/s	4.57GF/s	11.62GF/s	5.47GF/s	8.86GF/s	10.35GF/s	1.94×
SSDI	1.89GF/s	3.90GF/s	6.69GF/s	10.06GF/s	10.97GF/s	12.68GF/s	2.81×
SSFG	2.40GF/s	6.89GF/s	24.74GF/s	10.02GF/s	16.96GF/s	21.41GF/s	2.46×
SSFI	1.89GF/s	4.16GF/s	9.57GF/s	8.93GF/s	16.58GF/s	20.97GF/s	3.99×

Nehalem/**S**andybridge, **S**SE/**A**VX, **F**loat/**D**ouble, **I**CC/**G**CC

Average Performance on Stencils (efficiency)

Config.	ICC/GCC	Random	IBk	KStar	LR	M5P	MLP	SVM	Weighted Rank
NSDG	0.60	0.81	0.95	0.87	0.64	0.80	0.84	0.64	0.93
NSDI	1.05	0.94	0.95	0.95	0.96	0.93	0.94	0.94	0.95
NSFG	0.32	0.74	0.84	0.72	0.60	0.62	0.85	0.60	0.89
NSFI	0.41	0.94	0.95	0.95	0.96	0.93	0.93	0.95	0.96
SADG	0.41	0.80	0.85	0.82	0.68	0.75	0.74	0.68	0.86
SADI	0.79	0.93	0.92	0.92	0.92	0.93	0.94	0.93	0.92
SAFG	0.33	0.91	0.90	0.93	0.91	0.90	0.91	0.91	0.92
SAFI	0.41	0.95	0.96	0.96	0.94	0.95	0.93	0.94	0.96
SSDG	0.56	0.83	0.97	0.95	0.62	0.74	0.73	0.62	0.99
SSDI	1.03	0.97	0.97	0.97	0.97	0.97	0.96	0.96	0.97
SSFG	0.32	0.80	0.80	0.81	0.72	0.72	0.86	0.71	0.84
SSFI	0.42	0.95	0.96	0.96	0.96	0.96	0.95	0.96	0.96
Average	0.55	0.88	0.92	0.90	0.82	0.85	0.88	0.82	0.93

Nehalem/**S**andybridge, **S**SE/**A**VX, **F**loat/**D**ouble, **I**CC/**G**CC

Average Performance on Stencils (GF/s)

Config.	Compiler			Weighted Rank			Improv.
	min	avg	max	min	avg	max	
NSDG	2.17GF/s	3.35GF/s	4.12GF/s	3.48GF/s	5.34GF/s	6.91GF/s	1.59×
NSDI	4.26GF/s	5.59GF/s	6.65GF/s	4.33GF/s	5.24GF/s	6.97GF/s	0.94×
NSFG	3.20GF/s	3.78GF/s	4.45GF/s	7.22GF/s	10.50GF/s	12.52GF/s	2.77×
NSFI	2.76GF/s	4.20GF/s	5.10GF/s	8.85GF/s	9.97GF/s	12.26GF/s	2.37×
SADG	3.41GF/s	4.65GF/s	5.52GF/s	6.58GF/s	9.86GF/s	13.39GF/s	2.12×
SADI	6.44GF/s	7.89GF/s	9.02GF/s	7.90GF/s	9.23GF/s	11.49GF/s	1.17×
SAFG	4.40GF/s	5.05GF/s	6.13GF/s	11.36GF/s	14.44GF/s	19.08GF/s	2.86×
SAFI	4.17GF/s	5.85GF/s	7.02GF/s	10.41GF/s	13.74GF/s	16.07GF/s	2.35×
SSDG	3.41GF/s	4.66GF/s	5.52GF/s	6.19GF/s	8.44GF/s	10.26GF/s	1.81×
SSDI	6.48GF/s	7.87GF/s	8.88GF/s	6.21GF/s	7.61GF/s	9.97GF/s	0.97×
SSFG	4.36GF/s	5.02GF/s	6.14GF/s	9.51GF/s	13.41GF/s	16.05GF/s	2.67×
SSFI	4.17GF/s	5.86GF/s	7.02GF/s	12.38GF/s	13.48GF/s	16.01GF/s	2.30×

Nehalem/**S**andybridge, **S**SE/**A**VX, **F**loat/**D**ouble, **I**CC/**G**CC

Conclusions

Take-home message:

- Very significant improvement when using vector code synthesis
- Performance limitation of current compilers is in the decision heuristic
- Carefully crafted Machine Learning mechanisms provide good heuristics
  - Performance portability
  - Pure compile-time approach