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++) do
        a_{0}[0 : 3] ← SPLAT(A[k + 0][i])
        a_{1}[0 : 3] ← SPLAT(A[k + 1][i])
        a_{2}[0 : 3] ← SPLAT(A[k + 2][i])
        a_{3}[0 : 3] ← SPLAT(A[k + 3][i])
    for (j ← 0 ; j < N ; j++) do
        b_{0}[0 : 3] ← B[j + 0][k : k + 3]
        b_{1}[0 : 3] ← B[j + 1][k : k + 3]
        b_{2}[0 : 3] ← B[j + 2][k : k + 3]
        b_{3}[0 : 3] ← B[j + 3][k : k + 3]
        TRANSPOSE(b_{0}, b_{1}, b_{2}, b_{3})
        c[0 : 3] ← C[i][j + 3]
        c[0 : 3] += a_{0}[0 : 3] * b_{0}[0 : 3]
        c[0 : 3] += a_{1}[0 : 3] * b_{1}[0 : 3]
        c[0 : 3] += a_{2}[0 : 3] * b_{2}[0 : 3]
        c[0 : 3] += a_{3}[0 : 3] * b_{3}[0 : 3]
        C[i][j + 3] ← c[0 : 3]
    end for
end for
end procedure

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

- 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
  - \((R^{IBK}_v, R^{K*}_v) \rightarrow WR_v\)
  - Use linear regression to learn the coefficients
Experimental Protocol

- ML models: train 1 model per configuration (compiler × data type × SIMD ISA × 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)

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

**Nehalem/Sandybridge, SSE/AVX, Float/Double, ICC/GCC**
## Average Performance on CCSD (GF/s)

<table>
<thead>
<tr>
<th>Config.</th>
<th>Compiler min</th>
<th>Compiler avg</th>
<th>Compiler max</th>
<th>Weighted Rank min</th>
<th>Weighted Rank avg</th>
<th>Weighted Rank max</th>
<th>Improv.</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSDG</td>
<td>1.38GF/s</td>
<td>3.02GF/s</td>
<td>8.48GF/s</td>
<td>3.55GF/s</td>
<td>6.02GF/s</td>
<td>6.96GF/s</td>
<td>2.00×</td>
</tr>
<tr>
<td>NSDI</td>
<td>1.30GF/s</td>
<td>2.82GF/s</td>
<td>5.29GF/s</td>
<td>6.69GF/s</td>
<td>7.24GF/s</td>
<td>8.11GF/s</td>
<td>2.57×</td>
</tr>
<tr>
<td>NSFI</td>
<td>1.30GF/s</td>
<td>2.71GF/s</td>
<td>5.98GF/s</td>
<td>6.77GF/s</td>
<td>12.13GF/s</td>
<td>14.30GF/s</td>
<td>4.47×</td>
</tr>
<tr>
<td>SADI</td>
<td>1.89GF/s</td>
<td>3.92GF/s</td>
<td>6.69GF/s</td>
<td>11.50GF/s</td>
<td>14.64GF/s</td>
<td>22.23GF/s</td>
<td>3.73×</td>
</tr>
<tr>
<td>SAFG</td>
<td>2.40GF/s</td>
<td>6.87GF/s</td>
<td>24.47GF/s</td>
<td>14.69GF/s</td>
<td>25.84GF/s</td>
<td>35.47GF/s</td>
<td>3.76×</td>
</tr>
<tr>
<td>SAFI</td>
<td>1.89GF/s</td>
<td>4.15GF/s</td>
<td>9.79GF/s</td>
<td>24.92GF/s</td>
<td>33.18GF/s</td>
<td>43.30GF/s</td>
<td>7.99×</td>
</tr>
<tr>
<td>SSDG</td>
<td>2.31GF/s</td>
<td>4.57GF/s</td>
<td>11.62GF/s</td>
<td>5.47GF/s</td>
<td>8.86GF/s</td>
<td>10.35GF/s</td>
<td>1.94×</td>
</tr>
<tr>
<td>SSDI</td>
<td>1.89GF/s</td>
<td>3.90GF/s</td>
<td>6.69GF/s</td>
<td>10.06GF/s</td>
<td>10.97GF/s</td>
<td>12.68GF/s</td>
<td>2.81×</td>
</tr>
<tr>
<td>SSFG</td>
<td>2.40GF/s</td>
<td>6.89GF/s</td>
<td>24.74GF/s</td>
<td>10.02GF/s</td>
<td>16.96GF/s</td>
<td>21.41GF/s</td>
<td>2.46×</td>
</tr>
</tbody>
</table>

**Nehalem/Sandybridge, SSE/AVX, Float/Double, ICC/GCC**
### Average Performance on Stencils (efficiency)

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

**Nehalem/Sandybridge, SSE/AVX, Float/Double, ICC/GCC**
Average Performance on Stencils (GF/s)

<table>
<thead>
<tr>
<th>Config.</th>
<th>Compiler Weighted Rank</th>
<th>Improv.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>min</td>
<td>avg</td>
</tr>
<tr>
<td>NSDG</td>
<td>2.17GF/s</td>
<td>3.35GF/s</td>
</tr>
<tr>
<td>NSDI</td>
<td>4.26GF/s</td>
<td>5.59GF/s</td>
</tr>
<tr>
<td>NSFG</td>
<td>3.20GF/s</td>
<td>3.78GF/s</td>
</tr>
<tr>
<td>NSFI</td>
<td>2.76GF/s</td>
<td>4.20GF/s</td>
</tr>
<tr>
<td>SADI</td>
<td>6.44GF/s</td>
<td>7.89GF/s</td>
</tr>
<tr>
<td>SAFG</td>
<td>4.40GF/s</td>
<td>5.05GF/s</td>
</tr>
<tr>
<td>SAFI</td>
<td>4.17GF/s</td>
<td>5.85GF/s</td>
</tr>
<tr>
<td>SSDG</td>
<td>3.41GF/s</td>
<td>4.66GF/s</td>
</tr>
<tr>
<td>SSDI</td>
<td>6.48GF/s</td>
<td>7.87GF/s</td>
</tr>
<tr>
<td>SSFG</td>
<td>4.36GF/s</td>
<td>5.02GF/s</td>
</tr>
<tr>
<td>SSFI</td>
<td>4.17GF/s</td>
<td>5.86GF/s</td>
</tr>
</tbody>
</table>

Nehalem/Sandybridge, SSE/AVX, Float/Double, ICC/GCC
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