Using Graph-Based Characterization for Predictive Modeling of Vectorizable Loop Nests

William Killian
PhD Preliminary Exam Presentation
Department of Computer and Information Science

Committee
John Cavazos and Xiaoming Li

January 20, 2015
Code Optimization
Problems with Optimizing Code

• New compilers
  – New optimizations
  – Extended language features

• New architectures
  – Old programming model won’t work well (GPUs)
  – New/improved capabilities (ISA, cache coherency)
  – Compilers don’t update with the architecture

• Old code
  – Legacy code expected to work
  – Maintenance of existing code
Workflow

Original Code

Execution Information

Compiler Generated Executable

Programmer Optimized Code

Compiler Optimized Code
Workflow with Optimizations

- Generated ASM
- Target Arch
- Manual optimizations
- Code Transformations
- Command-line flags
- Compiler Directives
- Built-in Heuristics
Optimizations

• Manual
  – Loop transformations – unrolling, fusion/fission
  – Data structure changes – Array of Structures → Structure of Arrays

• Compiler Directives
  – Source code hints to the compiler indicated by user
  – Usually used for local (scope or loop) optimizations
  – May automatically transform code (e.g. #pragma unroll 4)

• Command-Line flags
  – Optimizations applied to entire program (-O3, –funroll=4)
  – May specify target architecture and features permitted (e.g. –march=avx)
(Intel) SIMD Architecture Evolution

1980s
- Vector Machines

1997
- 64-bit
- MMX

1999-2008
- 128-bit
- SSE (SP)
- SSE2 (DP, INT)
- SSE3, SSSE3, SSE4

2011-2013
- 256-bit
- AVX (SP, DP)
- AVX2 (INT)

2013-2015
- 512-bit
- NVI – Xeon Phi
- AVX-512
(Intel) SIMD Architecture Evolution

- **1980s**
  - Vector Machines
  - 64-bit
  - MMX

- **1997**
  - 128-bit
  - SSE (SP)
  - SSE2 (DP, INT)
  - SSE3, SSSE3, SSE4

- **1999-2008**
  - 256-bit
  - AVX (SP, DP)
  - AVX2 (INT)

- **2011-2013**
  - 512-bit
  - NVI – Xeon Phi
  - AVX-512

- **2013-2015**

How can we choose the best optimizations to exploit vectorization?
Research Questions

Optimization Search Space:
With many types of vectorization optimizations, how do we choose which ones to apply?

Automation:
How can we automatically select optimizations, apply optimizations, and evaluate performance?

Automatic Performance Improvement:
How can we quickly select good vectorization optimizations that improve performance?
Research Questions

Optimization Search Space:
With many types of vectorization optimizations, how do we choose which ones to apply?

Automation:
How can we automatically select optimizations, apply optimizations, and evaluate performance?

Automatic Performance Improvement:
How can we quickly select good vectorization optimizations that improve performance?
Optimization Search Space

- Used source code directives to drive the optimization selection and modification.
- Guide the internal compiler vectorization heuristics to improve performance.
- Use 6 different optimizations that provide varying levels of guidance to the compiler.
- Exhaustive search space of directives on individual loop nests.
**Optimization Search Space**

- **Apply No Optimization**
  - Let the compiler perform default vectorization
- **#pragma vector always**
  - May generate slower code
  - Ignore speed-up factor predicted by internal model
- **#pragma ivdep**
  - May generate invalid code
  - Ignore built-in check for all unproven vector dependences
  - Proven vector dependences will not be vectorized
- **#pragma simd**
  - May generate invalid code
  - Ignore all dependencies and reductions
  - Can vectorize an entire loop nest (outer-loop vectorization)
  - Optional argument `vectorlength(n)`. Vector length states how many safe iterations can be done at once (n = 2, 4, 8)
Research Questions

Optimization Search Space:
With many types of vectorization optimizations, how do we choose which ones to apply?

Automation:
How can we automatically select optimizations, apply optimizations, and evaluate performance?

Automatic Performance Improvement:
How can we quickly select good vectorization optimizations that improve performance?
Version Generation Automation

• Creation of two utilities
  • autovec
    – Simplified directive language; provides support for permutation of optimizations
    – Source-to-source compiler
  • VALT – Vectorization And Loop Transformation
    – Provides developer with concise language to specify vectorization and loop optimization directives
    – Extension of autovec
    – Supports multiple backend compilers
## autovec language to Intel directives

<table>
<thead>
<tr>
<th>autovec directive</th>
<th>Intel-specific pragma</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma autovec permute</td>
<td>Generates each of the following version into new file</td>
</tr>
<tr>
<td>#pragma autovec vl(x)</td>
<td>#pragma simd vectorlength(x)</td>
</tr>
<tr>
<td>#pragma autovec ivdep</td>
<td>#pragma ivdep</td>
</tr>
<tr>
<td>#pragma autovec always</td>
<td>#pragma vector always</td>
</tr>
<tr>
<td>#pragma autovec none</td>
<td></td>
</tr>
</tbody>
</table>

**Permute was configured to generate:**

- No optimization
- #pragma vector always
- #pragma ivdep
- #pragma simd vectorlength(2)
- #pragma simd vectorlength(4)
- #pragma simd vectorlength(8)
VALT language grammar

\[
\langle \text{directive} \rangle ::= \#'\text{pragma}' 'VALT' \langle \text{clauselist} \rangle \\
\langle \text{clauselist} \rangle ::= \langle \text{clauselist} \rangle \ [\ [, ] \] \langle \text{clause} \rangle \\
| \langle \text{empty} \rangle \\
\langle \text{clause} \rangle ::= 'vector' '(' \langle \text{vectorlist} \rangle ')' \\
| 'depend' '(' \langle \text{dependopts} \rangle ')' \\
| 'vectorsize' '(' \langle \text{number} \rangle ')' \\
| 'loop' '(' \langle \text{looplist} \rangle ')' \\
\langle \text{vectorlist} \rangle ::= 'none' \\
| \langle \text{vectorlist} \rangle ',' \langle \text{vectoritem} \rangle \\
| \langle \text{vectoritem} \rangle \\
\langle \text{looplist} \rangle ::= \langle \text{looplist} \rangle ',' \langle \text{loopitem} \rangle \\
| \langle \text{loopitem} \rangle \\
\langle \text{loopitem} \rangle ::= 'unroll' [ '(' \langle \text{number} \rangle ')' ] \\
| 'jam' [ '(' \langle \text{number} \rangle ')' ] \\
| 'dist' \\
| 'nofusion' \\
\langle \text{dependopts} \rangle ::= 'ignore' \\
| 'default' \\
\langle \text{number} \rangle ::= [1-9][0-9]*
\]
## VALT language to Intel directives

<table>
<thead>
<tr>
<th>VALT directive</th>
<th>Intel-specific pragma</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma vector(default)</td>
<td>No code emitted</td>
</tr>
<tr>
<td>#pragma vector(none)</td>
<td>#pragma novector</td>
</tr>
<tr>
<td>#pragma vector(always)</td>
<td>#pragma vector always</td>
</tr>
<tr>
<td>#pragma vector(ignore)</td>
<td>#pragma ivdep</td>
</tr>
<tr>
<td>#pragma vector(aligned)</td>
<td>#pragma vector aligned</td>
</tr>
<tr>
<td>#pragma vector(temp)</td>
<td>#pragma vector temporal</td>
</tr>
<tr>
<td>#pragma vector(nontemp)</td>
<td>#pragma vector nontemporal</td>
</tr>
<tr>
<td>#pragma vectorsize(x)</td>
<td>#pragma simd vectorlength(x)</td>
</tr>
<tr>
<td>#pragma loop(unroll(x))</td>
<td>#pragma unroll(x)</td>
</tr>
<tr>
<td>#pragma loop(jam(x))</td>
<td>#pragma unroll_and_jam(x)</td>
</tr>
<tr>
<td>#pragma loop(nofusion)</td>
<td>#pragma nofusion</td>
</tr>
<tr>
<td>#pragma loop(dist)</td>
<td>#pragma distribute_point</td>
</tr>
</tbody>
</table>
Version Generation Workflow
Research Questions

Optimization Search Space:
With many types of vectorization optimizations, how do we choose which ones to apply?

Automation:
How can we automatically select optimizations, apply optimizations, and evaluate performance?

Automatic Performance Improvement:
How can we quickly select good vectorization optimizations that improve performance?
Machine Learning - Previous Solutions

- Stock et al. proposed using machine learning techniques to improve automatic vectorization
- Park et al. proposed using graph-based learning techniques to optimize programs at loop-nest granularity
Proposed Solution:

• Use graph-based learning techniques to choose vectorization optimizations for vectorizable loop nests

• Construct a graph-based speedup predictor that can predict a speedup when applying vectorization optimizations to a loop nest
Feature Extraction

- LLVM used to generate IR
- MinIR used to generate CFG
- Feature vector generated per basic block
  - Total # of Instructions
  - # of Add/Sub/Mul/Div
  - # of Load/Store
  - # of comparisons
  - # of conditional Branches
  - # of unconditional branches
Example Control Flow Graph for Loop Nest

- Total of 6 basic blocks
- One entry, one return
- Basic blocks may not contain code

```c
float aa[LEN2][LEN2];

for (int i = 0; i < LEN2; i++)
    for (int j = 0; j < i; j++)
        aa[i][j] = aa[j][i] + bb[i][j];
```
Machine Learning Model Construction

Machine Learning Algorithm

1 to N-1

bb0 -> bb1, bb2
bb1 -> bb3
...
bb4 -> bb2

1 to N-1

opt seq

speedups

Machine Learning Model
## Optimization Encoding

<table>
<thead>
<tr>
<th>Bit Configuration</th>
<th>Encoded Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>No Loop</td>
</tr>
<tr>
<td>100000</td>
<td>No Optimization Performed</td>
</tr>
<tr>
<td>110000</td>
<td>#pragma vector always</td>
</tr>
<tr>
<td>111000</td>
<td>#pragma ivdep</td>
</tr>
<tr>
<td>111100</td>
<td>#pragma simd vectorlength(2)</td>
</tr>
<tr>
<td>111110</td>
<td>#pragma simd vectorlength(4)</td>
</tr>
<tr>
<td>111111</td>
<td>#pragma simd vectorlength(8)</td>
</tr>
</tbody>
</table>
Machine Learning Algorithm

1. **Training data:**
   
   - $L = \text{set of all loop nests}$
   - $O = \text{set of all optimization sequences}$
   - $\text{speedup}(l, o)$ - observed speedup from applying $o$ to loop nest $l$
   
   $\text{scores} = \{(l, o, \text{speedup}(l, o)) | \forall l \in L, \forall o \in O, l_{\text{size}} = o_{\text{size}} \land \text{valid}(l, o)\}$

2. **Construct kernel similarity matrix** by computing similarity between all training data points (loop nest + optimization)

   - $\forall i \in [0, \|\text{scores}\|), \forall j \in [0, \|\text{scores}\|)$
   
   $\text{Sim}_{km}^{i,j} = \text{Sim}_{\text{loop}}^{i,j} \times \| \text{count1}(\text{scores}^i_o) - \text{count1}(\text{scores}^j_o) \|$ 

3. **Train on kernel matrix with speedup as production target**
Machine Learning Algorithm

- Graph kernel functions used to transform the training into a different, linearly-separable feature space.
  - Shortest path graph kernel
  - Used previously by Park et al. for speedup predictors
  - Similarity calculated by normalizing intersection kernel matrix

- Linear classifier is constructed that separates the points into multiple classes.

- Support vector machines (SVMs) used to construct predictive models from the kernel similarity matrix

- Predicts a speedup
Using Machine Learning Model for Unseen Program

SRC
Feature Extract
Machine Learning Model
opt seq
predicted speedups
EXPERIMENT SETUP
Benchmark Selection

**TSVC**
- 151 loop nests with varying access patterns, computations, and memory access types
- Originally used to evaluate how well a compiler can recognize patterns for vectorization
- Millisecond timing granularity with each loop-nest within a repeat loop

**PolyBench/C**
- 30 static control-flow micro-benchmarks from several scientific domains
- Modified to create different individual versions optimizing single loop nests (65 total loop nests)
- Clock-tick timing granularity across the entire kernel execution
## Machine Configuration

### Nehalem (NHM)
- Core i7 950
- 3.06GHz quad-core
- 8MB L3 cache
- 24GB DDR3-1333
- 128-bit vector width
- Up to SSE 4.2 ISA
- 45nm
- Q2 2009

### Haswell (HSW)
- Core i7 5930K
- 3.5GHz hex-core
- 15MB L3 cache
- 32GB DDR4-2133
- 256-bit vector width
- Up to AVX2 ISA
- 22nm
- Q4 2013

Processor dynamic frequency scaling was disabled for all experiments. We only analyzed speedup for cross-architecture comparison, not performance.
## Machine Configuration

**Nehalem (NHM)**
- Core i7 950
- 3.06GHz quad-core
- 8MB L3 cache
- 24GB DDR3-1333
- **128-bit vector width**
- **Up to SSE 4.2 ISA**
- 45nm
- Q2 2009

**Haswell (HSW)**
- Core i7 5930K
- 3.5GHz hex-core
- 15MB L3 cache
- 32GB DDR4-2133
- **256-bit vector width**
- **Up to AVX2 ISA**
- 22nm
- Q4 2013

Processor dynamic frequency scaling was disabled for all experiments. We only analyzed speedup for cross-architecture comparison, not performance.
Execution Configuration

- Each loop nest executed 10 times
- Ensured execution times within 1% (0.8% observed)
- Verified correctness of execution for each version by dumping live-out data (PolyBench loop nests) or checksum (TSVC loop nests)
- Average speedup recorded for each loop nest and optimization sequence pair
  - Used for exhaustive search performance and speedup predictor
Experiment Results

EXHAUSTIVE SEARCH SPACE SPEEDUP
TSVC Results

Nehalem

Haswell

TSVC Loop Nests (N = 151) sorted by increasing speedup
TSVC Cross-Architecture Analysis

Speedup normalized over `-O3 -xHOST`

TSVC Loop Nests (N=151) sorted by increasing speedup on Haswell

Correlation: $C = 0.8945$
PolyBench Results - Nehalem

Speedup normalized over '-O3 -xHOST'

Polybench Loop Nests (N=65) sorted by increasing speedup
PolyBench Results - Haswell

Polybench Loop Nests (N=65) sorted by increasing speedup

Speedup normalized over '-O3 -xHOST'
PolyBench Cross-Architecture Analysis

Correlation: C = 0.8894
Experiment Results

EXHAUSTIVE SEARCH SPACE VALID CODE GENERATION
Version Generation Example

TSVC s256 Speedups

Optimization Sequence

Speedup over -O3 -xHOST

Valid

Invalid
Version Generation Example
TSVC s126 Speedups
Version Generation Statistics

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Arch</th>
<th>Valid</th>
<th>Invalid</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSVC</td>
<td>Nehalem</td>
<td>1832</td>
<td>151</td>
<td>3</td>
</tr>
<tr>
<td>TSVC</td>
<td>Haswell</td>
<td>1826</td>
<td>155</td>
<td>5</td>
</tr>
<tr>
<td>PolyBench</td>
<td>Nehalem</td>
<td>5204</td>
<td>3826</td>
<td>0</td>
</tr>
<tr>
<td>Polybench</td>
<td>Haswell</td>
<td>5204</td>
<td>3826</td>
<td>0</td>
</tr>
</tbody>
</table>

Version Generation Across Architecture

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Valid Fastest</th>
<th>Invalid Fastest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nehalem</td>
<td>140</td>
<td>11</td>
</tr>
<tr>
<td>Haswell</td>
<td>143</td>
<td>8</td>
</tr>
</tbody>
</table>

Version Generation Performance for TSVC
Experiment Results

GRAPH-BASED SPEEDUP PREDICTOR
Evaluation Model

• Leave-One-Out Cross Validation
  – For a given loop nest, construct a model based on all other loop nests as training data
  – Compare predictor’s speedup to actual speedup
  – 151 models for TSVC, 65 models for PolyBench

• Evaluation Method
  – 1-shot : only consider top prediction
  – 3-shot : consider top three predictions
  – Top prediction is the optimization with best observed speedup
## TSVC Speedup Predictor - Nehalem

<table>
<thead>
<tr>
<th>loop</th>
<th>1-shot</th>
<th>3-shot</th>
<th>Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>s126</td>
<td>1.91</td>
<td>1.92</td>
<td>6.28</td>
</tr>
<tr>
<td>s221</td>
<td><strong>0.39</strong></td>
<td><strong>1.42</strong></td>
<td>1.42</td>
</tr>
<tr>
<td>s2251</td>
<td>2.03</td>
<td><strong>2.44</strong></td>
<td>2.44</td>
</tr>
<tr>
<td>s244</td>
<td><strong>0.46</strong></td>
<td>1.36</td>
<td>1.36</td>
</tr>
<tr>
<td>s256</td>
<td>0.98</td>
<td>0.99</td>
<td>7.92</td>
</tr>
<tr>
<td>s3112</td>
<td>1.99</td>
<td><strong>4.03</strong></td>
<td>4.03</td>
</tr>
<tr>
<td>s321</td>
<td><strong>0.50</strong></td>
<td>0.97</td>
<td>2.13</td>
</tr>
<tr>
<td>s424</td>
<td>0.99</td>
<td>1.94</td>
<td>2.88</td>
</tr>
<tr>
<td><strong>Arith. Mean</strong></td>
<td><strong>0.70</strong></td>
<td><strong>0.98</strong></td>
<td><strong>1.47</strong></td>
</tr>
<tr>
<td><strong>Geo. Mean</strong></td>
<td><strong>0.61</strong></td>
<td><strong>0.85</strong></td>
<td><strong>1.32</strong></td>
</tr>
</tbody>
</table>

74% of Optimal on Nehalem 3-shot
## TSV C Speedup Predictor - Haswell

<table>
<thead>
<tr>
<th>loop</th>
<th>1-shot</th>
<th>3-shot</th>
<th>Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>s126</td>
<td>1.74</td>
<td>1.84</td>
<td>5.95</td>
</tr>
<tr>
<td>s221</td>
<td>0.36</td>
<td>1.35</td>
<td>1.35</td>
</tr>
<tr>
<td>s2251</td>
<td>1.49</td>
<td>1.64</td>
<td>1.64</td>
</tr>
<tr>
<td>s244</td>
<td>0.32</td>
<td>1.18</td>
<td>1.30</td>
</tr>
<tr>
<td>s256</td>
<td>0.99</td>
<td>1.00</td>
<td>7.88</td>
</tr>
<tr>
<td>s3112</td>
<td>1.99</td>
<td>4.52</td>
<td>4.52</td>
</tr>
<tr>
<td>s321</td>
<td>0.44</td>
<td>1.00</td>
<td>1.65</td>
</tr>
<tr>
<td>s424</td>
<td>1.00</td>
<td>1.77</td>
<td>2.57</td>
</tr>
<tr>
<td>Arith. Mean</td>
<td>0.62</td>
<td>0.94</td>
<td>1.36</td>
</tr>
<tr>
<td>Geo. Mean</td>
<td>0.51</td>
<td>0.80</td>
<td>1.21</td>
</tr>
</tbody>
</table>

77% of Optimal on Haswell 3-shot
## PolyBench Speedup Predictor - Nehalem

<table>
<thead>
<tr>
<th>loop</th>
<th>1-shot</th>
<th>3-shot</th>
<th>Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>2mm-1</td>
<td>0.25</td>
<td>0.25</td>
<td>1.00</td>
</tr>
<tr>
<td>adi-4</td>
<td>0.99</td>
<td>1.32</td>
<td>1.40</td>
</tr>
<tr>
<td>correlation-1</td>
<td>1.00</td>
<td>1.22</td>
<td>1.22</td>
</tr>
<tr>
<td>covariance-1</td>
<td>1.37</td>
<td>1.68</td>
<td>1.68</td>
</tr>
<tr>
<td>dynprog-1</td>
<td>0.98</td>
<td>1.00</td>
<td>1.10</td>
</tr>
<tr>
<td>floyd-warshall</td>
<td>5.22</td>
<td>8.52</td>
<td>11.30</td>
</tr>
<tr>
<td>gemm</td>
<td>0.14</td>
<td>0.32</td>
<td>1.00</td>
</tr>
<tr>
<td>grammschmidt-3</td>
<td>1.00</td>
<td>7.38</td>
<td>8.17</td>
</tr>
<tr>
<td><em>Arith. Mean</em></td>
<td>0.97</td>
<td>1.21</td>
<td>1.46</td>
</tr>
<tr>
<td><em>Geo. Mean</em></td>
<td>0.85</td>
<td>0.98</td>
<td>1.24</td>
</tr>
</tbody>
</table>

84.44% of Optimal on Nehalem 3-shot
## PolyBench Speedup Predictor - Haswell

<table>
<thead>
<tr>
<th>loop</th>
<th>1-shot</th>
<th>3-shot</th>
<th>Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>2mm-1</td>
<td>0.67</td>
<td>0.68</td>
<td>1.00</td>
</tr>
<tr>
<td>adi-4</td>
<td>1.22</td>
<td>1.22</td>
<td>1.22</td>
</tr>
<tr>
<td>correlation-1</td>
<td>0.98</td>
<td>1.00</td>
<td>1.01</td>
</tr>
<tr>
<td>covariance-1</td>
<td>1.02</td>
<td>1.02</td>
<td>1.02</td>
</tr>
<tr>
<td>dynprog-1</td>
<td>0.99</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>floyd-warshall</td>
<td>25.88</td>
<td>25.88</td>
<td>28.86</td>
</tr>
<tr>
<td>gemm</td>
<td>0.12</td>
<td>0.38</td>
<td>1.00</td>
</tr>
<tr>
<td>grammschmidt-3</td>
<td>3.37</td>
<td>3.40</td>
<td>5.22</td>
</tr>
</tbody>
</table>

**Arith. Mean**

- 1.34
- 1.47
- 1.66

**Geo. Mean**

- 0.90
- 1.03
- 1.20

88.74% of Optimal on Haswell 3-shot
Threats to Validity

• Correctness of generated code
  – PolyBench - Analyzing live out data still may not verify correctness
  – TSVC - used a checksum computation. Invalid results still possible

• Speedup measurement
  – Execution performed on single user mode with timing at a kernel level
  – Speedup a “trend” for PolyBench – not representative of observable speedup for entire kernel

• Machine learning model
  – Optimization bit vector only defines a “level” of vectorization
  – SVM training parameters not explored
  – Generated model seems to find smaller variations between code – similar kernel matrices are generated
Contributions

• VALT directive compiler to simple code generation across different compiler backends

• autovec - exhaustive search code version generator

• Graph-based speedup predictor designed to predict the best vector optimizations to apply to a given loop nest

• Performance analysis of vectorizable micro-benchmarks that can carry across to similar types of kernels
Differences from Related Work

• Stock et al. work only targeted Tensor Contradiction and stencil kernels and didn’t use graph-based learning
  – Our approach works on many different types of code and uses graph-based features to construct the model

• Park et al. focused on a different set of optimizations, primarily targeting loop transformations, autoparallelization, and choosing whether or not to vectorize
  – Our approach explores the vectorization search space of loop nests, and allows us to potentially reach a more local maximum speedup given our optimization search space
Future Work

• Extend VALT to support multiple backends (PGI Compiler)
• Change how optimizations are represented
  – Annotate graph-based representation
  – Would eliminate encoding for maximum loop nest size
• Extend work to additional compilers
  – Newer versions of GCC (4.9+), PGI compiler
• Target wider vector size architectures
  – Xeon Phi (Knight’s Corner) – 512-bit vector width; limited ISA
  – Knight’s Landing and Skylake – AVX-512
Conclusion

• Provided automated and manual techniques for improving performance codes with vectorization optimizations

• Non-experts can use the utilities developed to automatically optimize codes to exploit vector hardware

• With the contributions presented, we
  – achieved up to a 30x speedup through exhaustive search
  – predicted within 88% of search space optimal using the proposed speedup predictor
Thank You!

QUESTIONS?