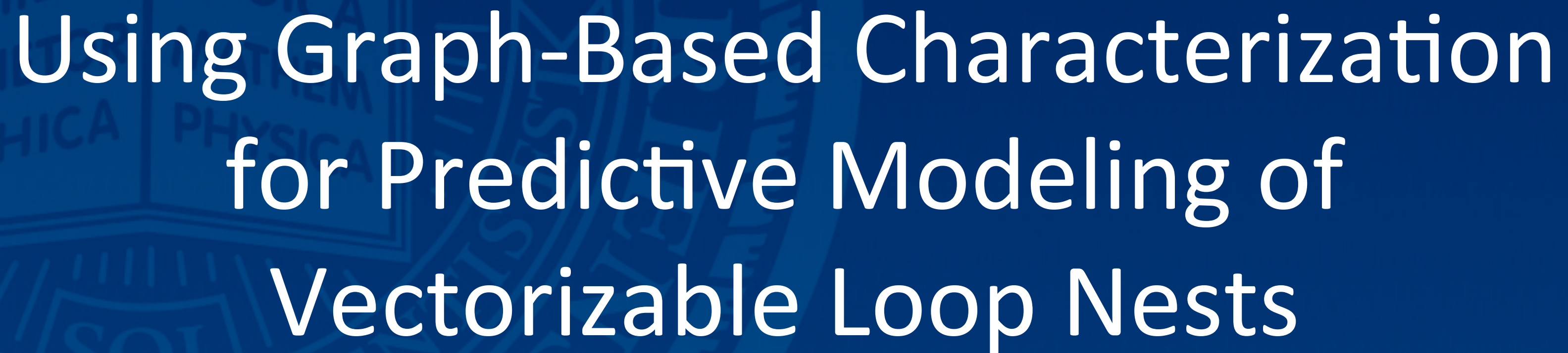


Using Graph-Based Characterization for Predictive Modeling of Vectorizable Loop Nests

William Killian

PhD Prelimary Exam Presentation

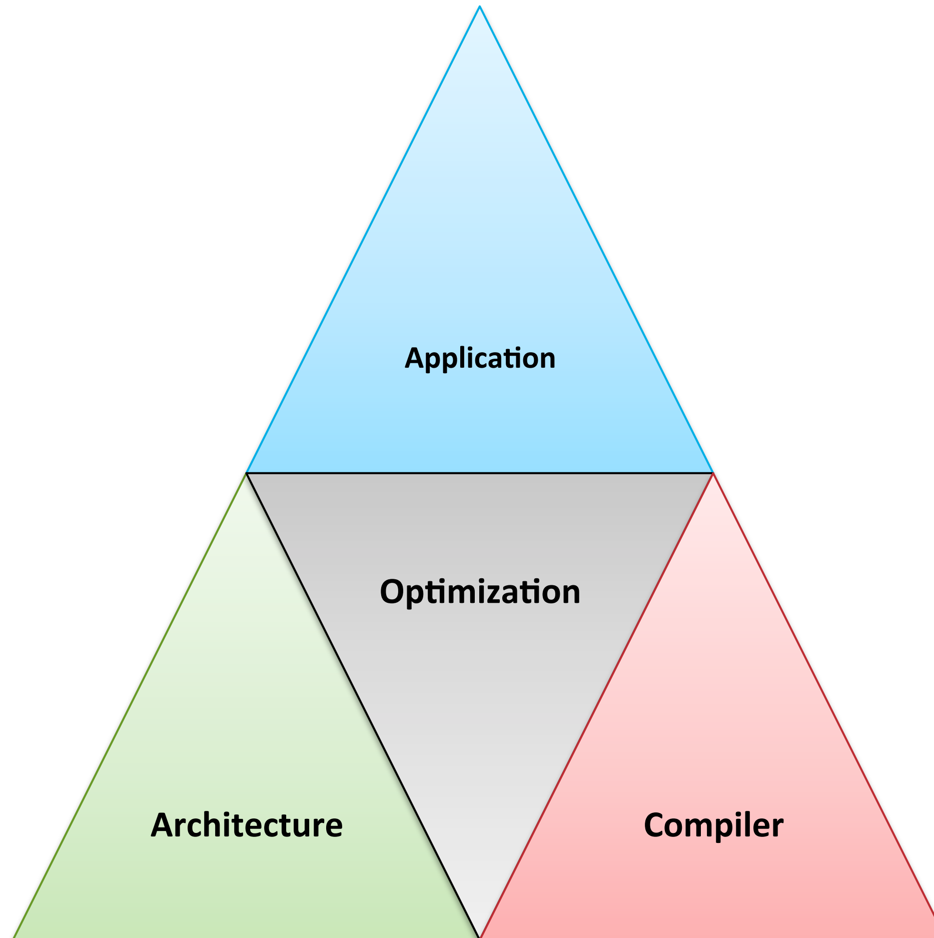
Department of Computer and Information Science

Committee

John Cavazos and Xiaoming Li

January 20, 2015

Code Optimization

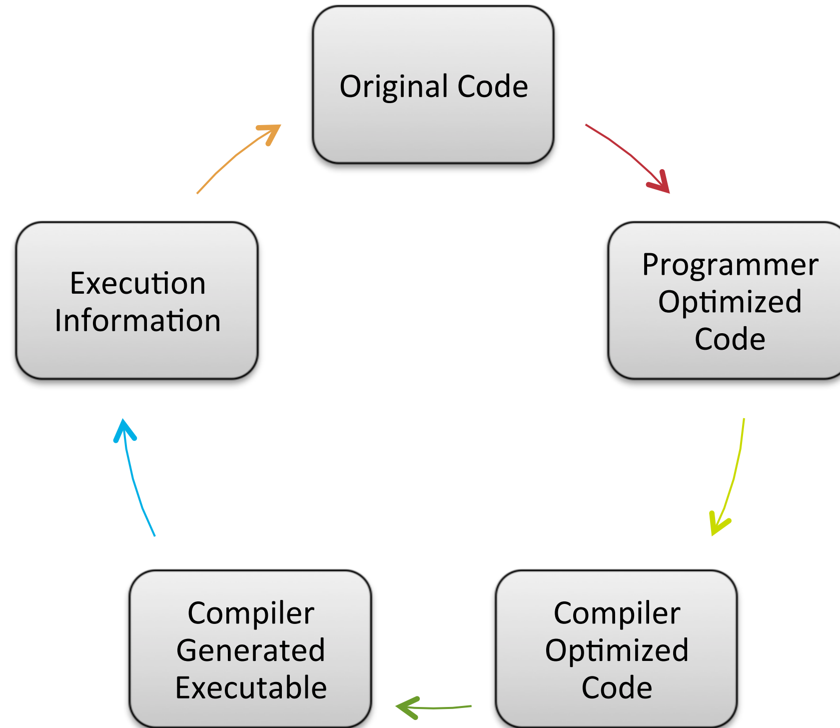


Problems with Optimizing Code

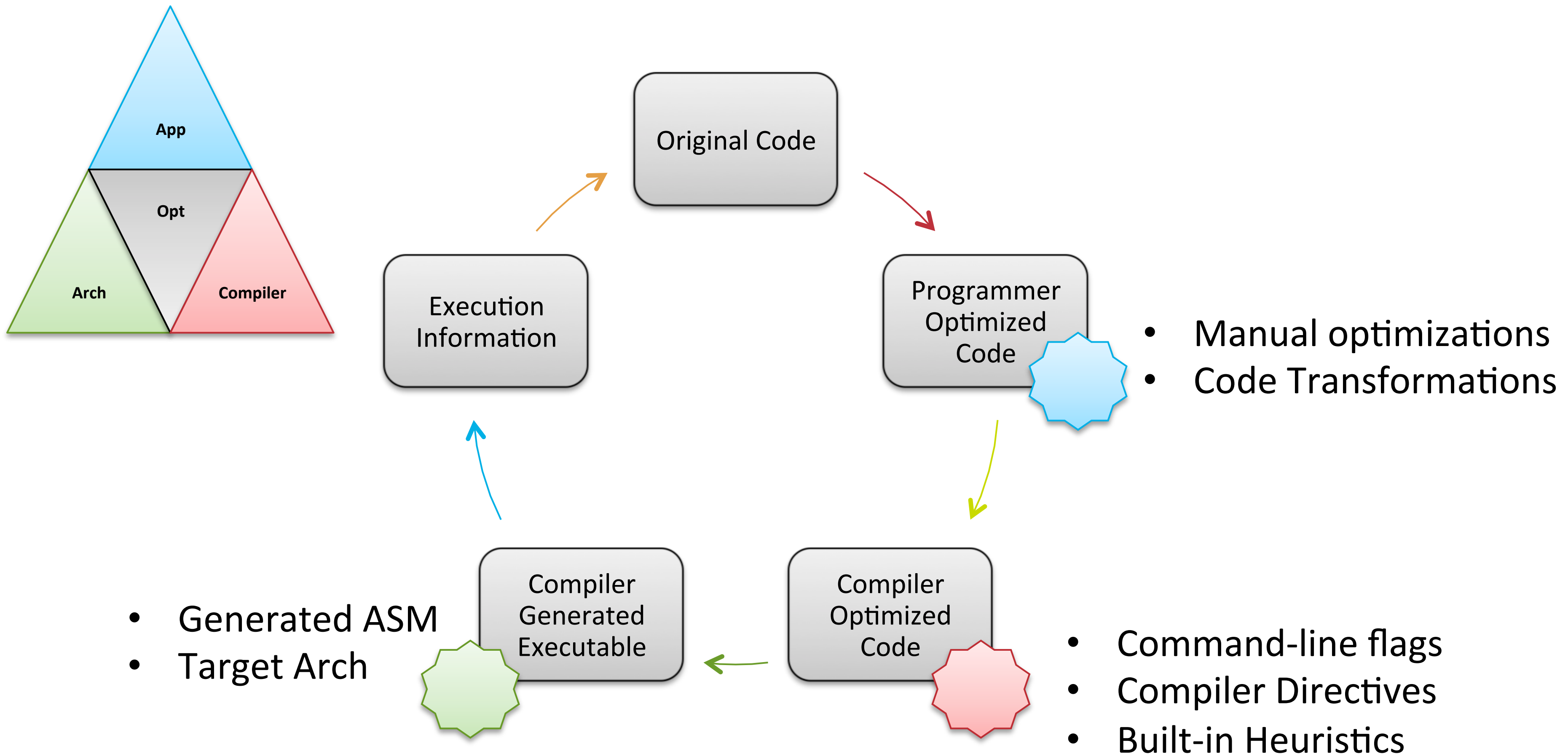
- New compilers
  - New optimizations
  - Extended language features
- New architectures
  - Old programming model won't work well (GPUs)
  - New/improved capabilities (ISA, cache coherency)
  - Compilers don't update with the architecture
- Old code
  - Legacy code expected to work
  - Maintenance of existing code

University of Delaware

Workflow



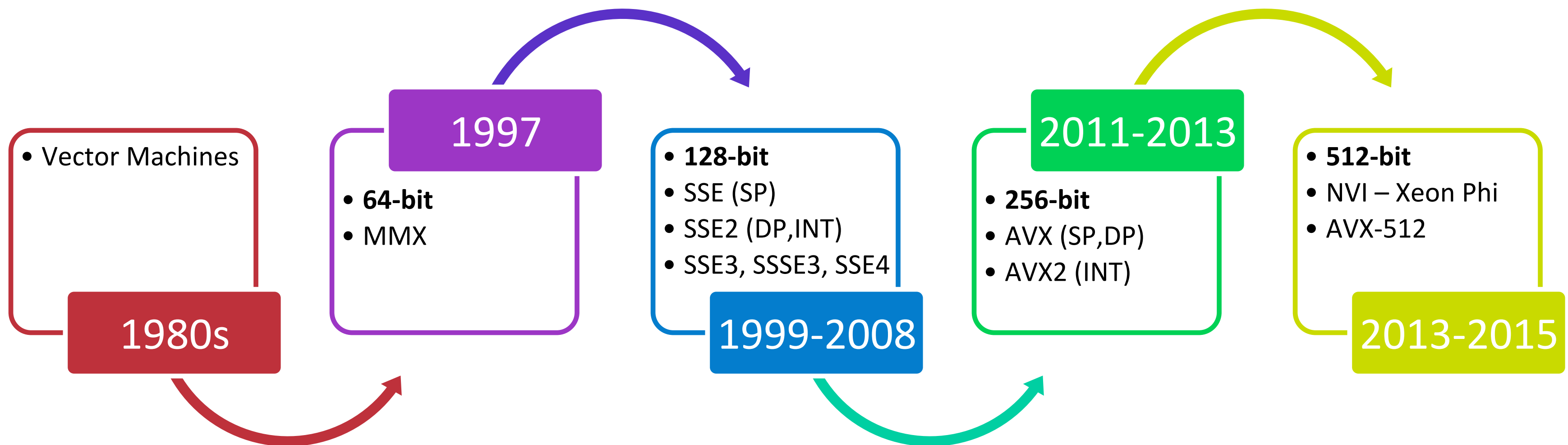
Workflow with Optimizations



Optimizations

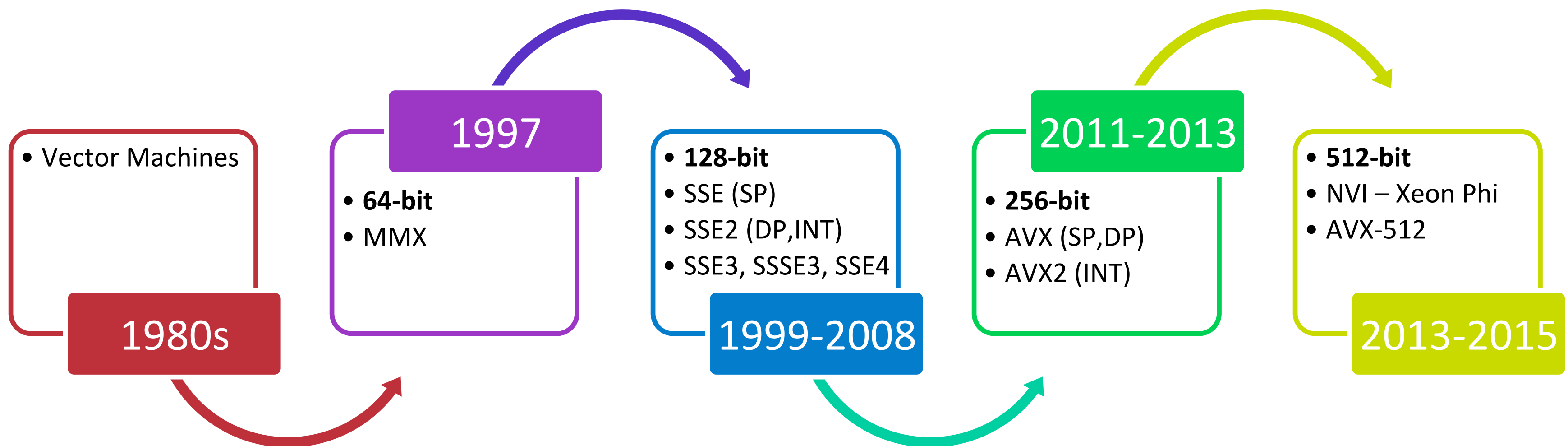
- Manual
  - Loop transformations – unrolling, fusion/fission
  - Data structure changes – Array of Structures → Structure of Arrays
- Compiler Directives
  - Source code hints to the compiler indicated by user
  - Usually used for local (scope or loop) optimizations
  - May automatically transform code (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



UNIVERSITY OF DELAWARE

(Intel) SIMD Architecture Evolution



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 – **V**ectoration **A**nd **L**oop **T**ransformation
  - Provides developer with concise language to specify vectorization and loop optimization directives
  - Extension of `autovec`
  - Supports multiple backend compilers

autovec language to Intel directives

autovec directive	Intel-specific pragma
`#pragma autovec permute`	Generates each of the following version into new file
`#pragma autovec vl(x)`	`#pragma simd vectorlength(x)`
`#pragma autovec ivdep`	`#pragma ivdep`
`#pragma autovec always`	`#pragma vector always`
`#pragma autovec none`	

Permute was configured to generate:

- No optimization
- `#pragma vector always`
- `#pragma ivdep`
- `#pragma simd vectorlength(2)`
- `#pragma simd vectorlength(4)`
- `#pragma simd vectorlength(8)`

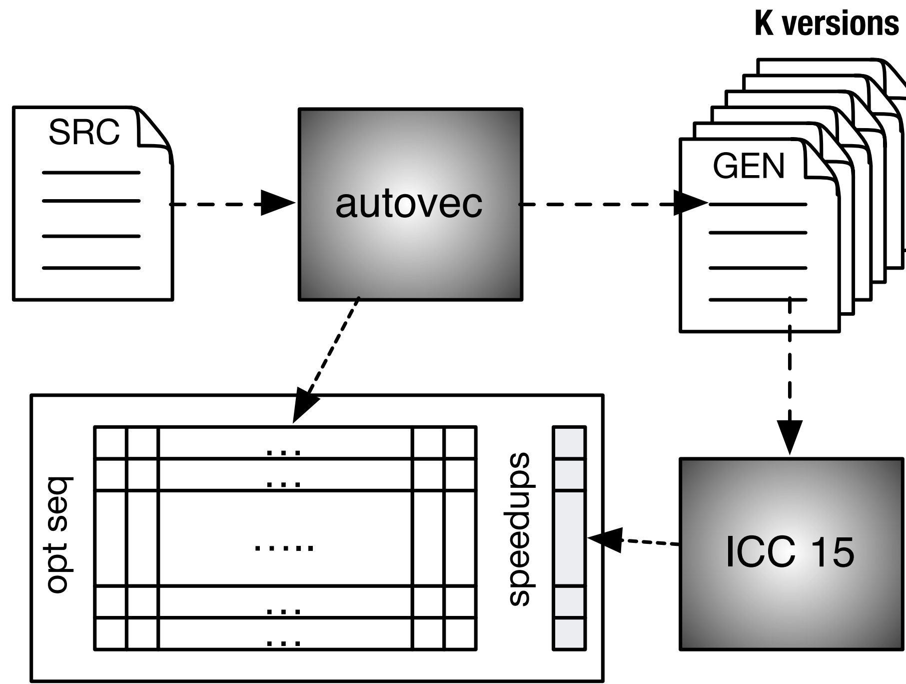
VALT language grammar

```
⟨directive⟩ ::= '#pragma' 'VALT' ⟨clauselist⟩

⟨clauselist⟩ ::= ⟨clauselist⟩ [[,]] ⟨clause⟩
    | ⟨empty⟩

⟨clause⟩ ::= 'vector' '(' ⟨vectorlist⟩ ')'
    | 'depend' '(' ⟨dependopts⟩ ')'
    | 'vectorsize' '(' ⟨number⟩ ')'
    | 'loop' '(' ⟨looplist⟩ ')'

⟨vectorlist⟩ ::= 'none'
    | ⟨vectorlist⟩ ',' ⟨vectoritem⟩
    | ⟨vectoritem⟩

⟨looplist⟩ ::= ⟨looplist⟩ ',' ⟨loopitem⟩
    | ⟨loopitem⟩

⟨vectoritem⟩ ::= 'default'
    | 'none'
    | 'always'
    | 'aligned'
    | 'unaligned'
    | 'nontemp'
    | 'temp'

⟨loopitem⟩ ::= 'unroll' [[ '(' ⟨number⟩ ')' ]]
    | 'jam' [[ '(' ⟨number⟩ ')' ]]
    | 'dist'
    | 'nofusion'

⟨dependopts⟩ ::= 'ignore'
    | 'default'

⟨number⟩ ::= [1-9][0-9]*
```

VALT language to Intel directives

VALT directive	Intel-specific pragma
#pragma vector(default)	No code emitted
#pragma vector(none)	#pragma novector
#pragma vector(always)	#pragma vector always
#pragma vector(ignore)	#pragma ivdep
#pragma vector(aligned)	#pragma vector aligned
#pragma vector(temp)	#pragma vector temporal
#pragma vector(nontemp)	#pragma vector nontemporal
#pragma vectorsize(x)	#pragma simd vectorlength(x)
#pragma loop(unroll(x))	#pragma unroll(x)
#pragma loop(jam(x))	#pragma unroll_and_jam(x)
#pragma loop(nofusion)	#pragma nofusion
#pragma loop(dist)	#pragma distribute_point

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?

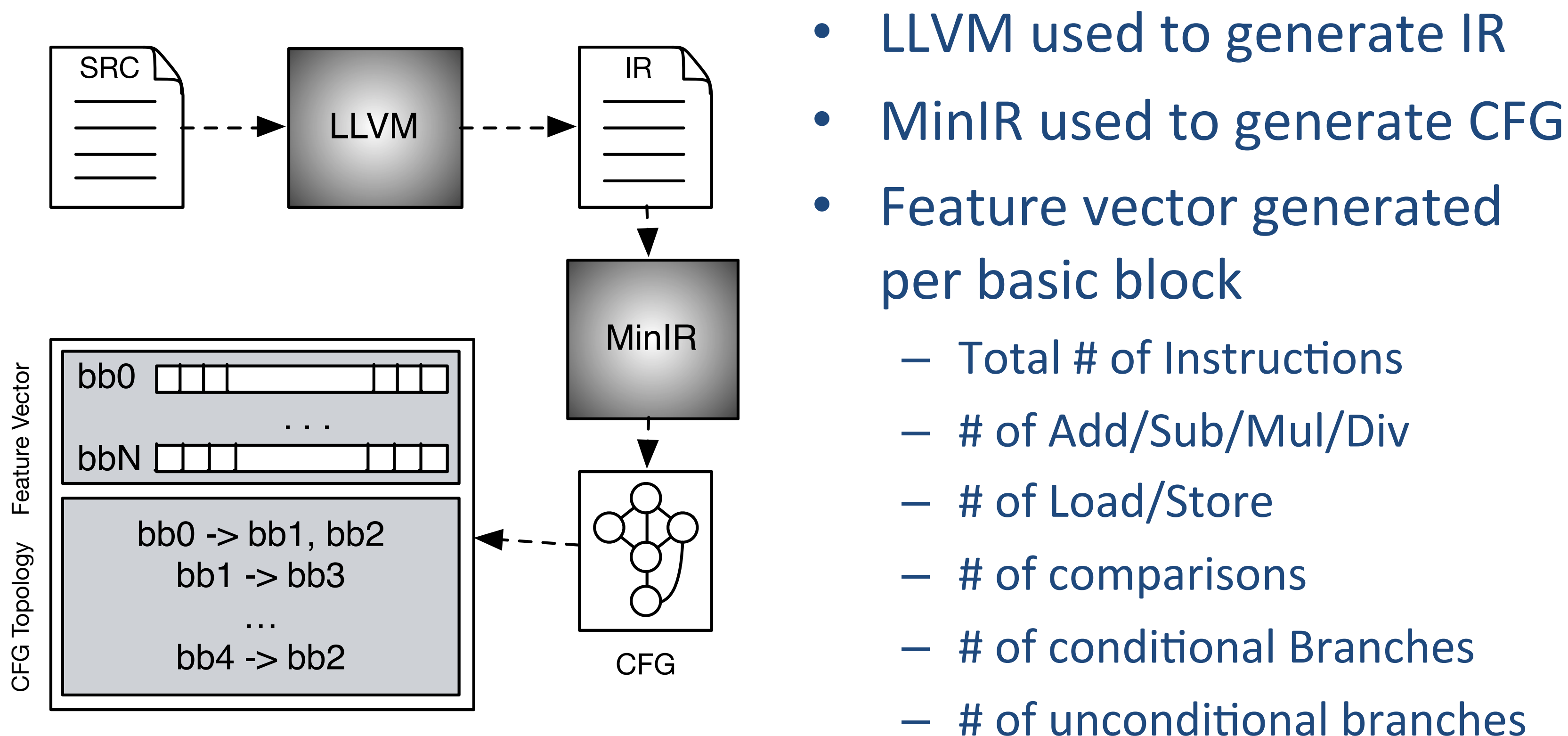
Machine Learning - Previous Solutions

- 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

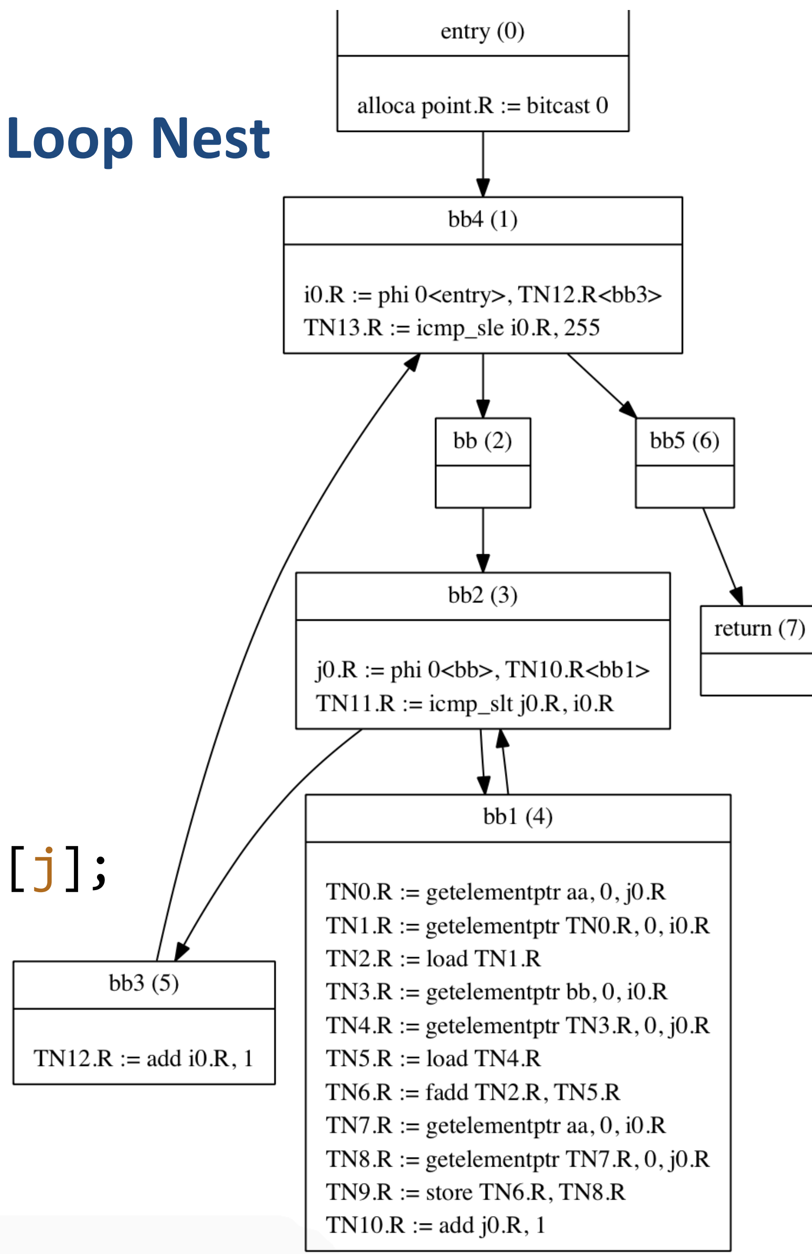


- LLVM used to generate IR
- MinIR used to generate CFG
- Feature vector generated per basic block
  - Total # of Instructions
  - # of Add/Sub/Mul/Div
  - # of Load/Store
  - # of comparisons
  - # of conditional Branches
  - # of unconditional branches

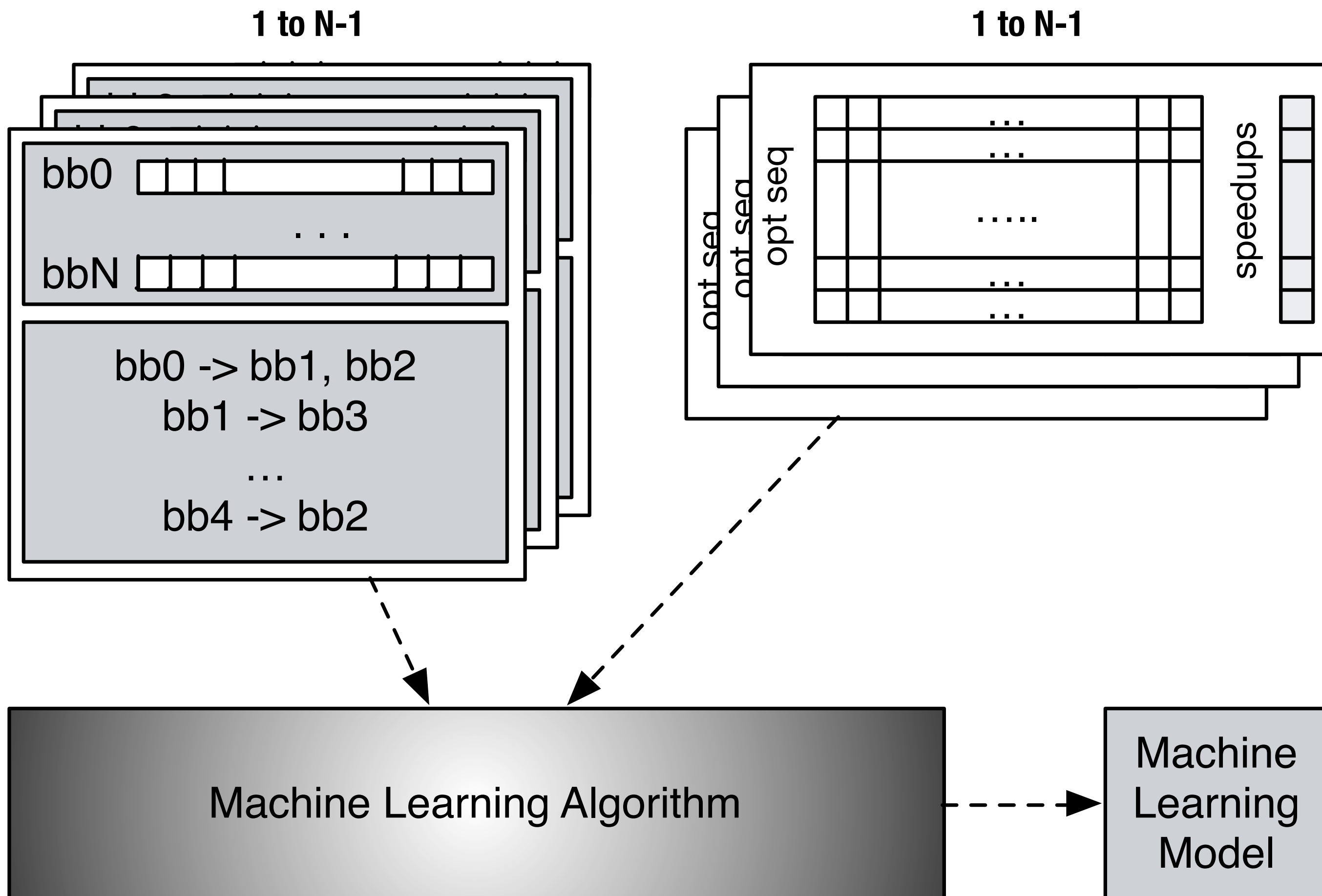
Example Control Flow Graph for Loop Nest

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

```
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



Optimization Encoding

Bit Configuration	Encoded Optimization
000000	No Loop
100000	No Optimization Performed
110000	`#pragma vector always`
111000	`#pragma ivdep`
111100	`#pragma simd vectorlength(2)`
111110	`#pragma simd vectorlength(4)`
111111	`#pragma simd vectorlength(8)`

Machine Learning Algorithm

1. Training data:

L = set of all loop nests

O = set of all optimization sequences

$\text{speedup}(l,o)$ - observed speedup from applying o to loop nest l

$$scores = \left\{ \left(l, o, \text{speedup}(l,o)\right) \middle| \forall l \in L, \forall o \in O, l_{size} = o_{size} \wedge \text{valid}(l,o) \right\}$$

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

$$\forall i \text{ in } [0, \|scores\|), \forall j \text{ in } [0, \|scores\|)$$

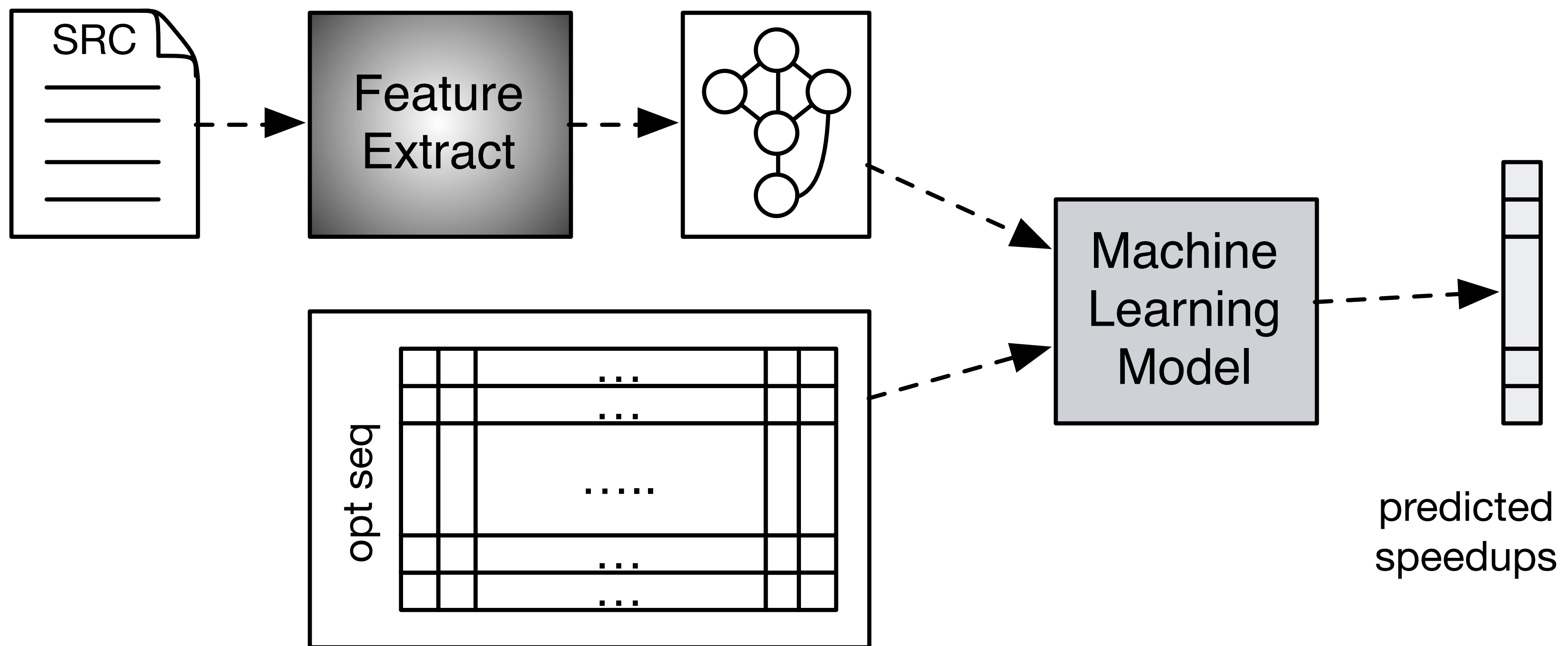
$$Sim_{km}^{i,j} = Sim_{loop}^{i,j} \times \| \text{count1}(scores_O^i) - \text{count1}(scores_O^j) \|$$

3. Train on kernel matrix with speedup as production target

Machine Learning Algorithm

- G*raph 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



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

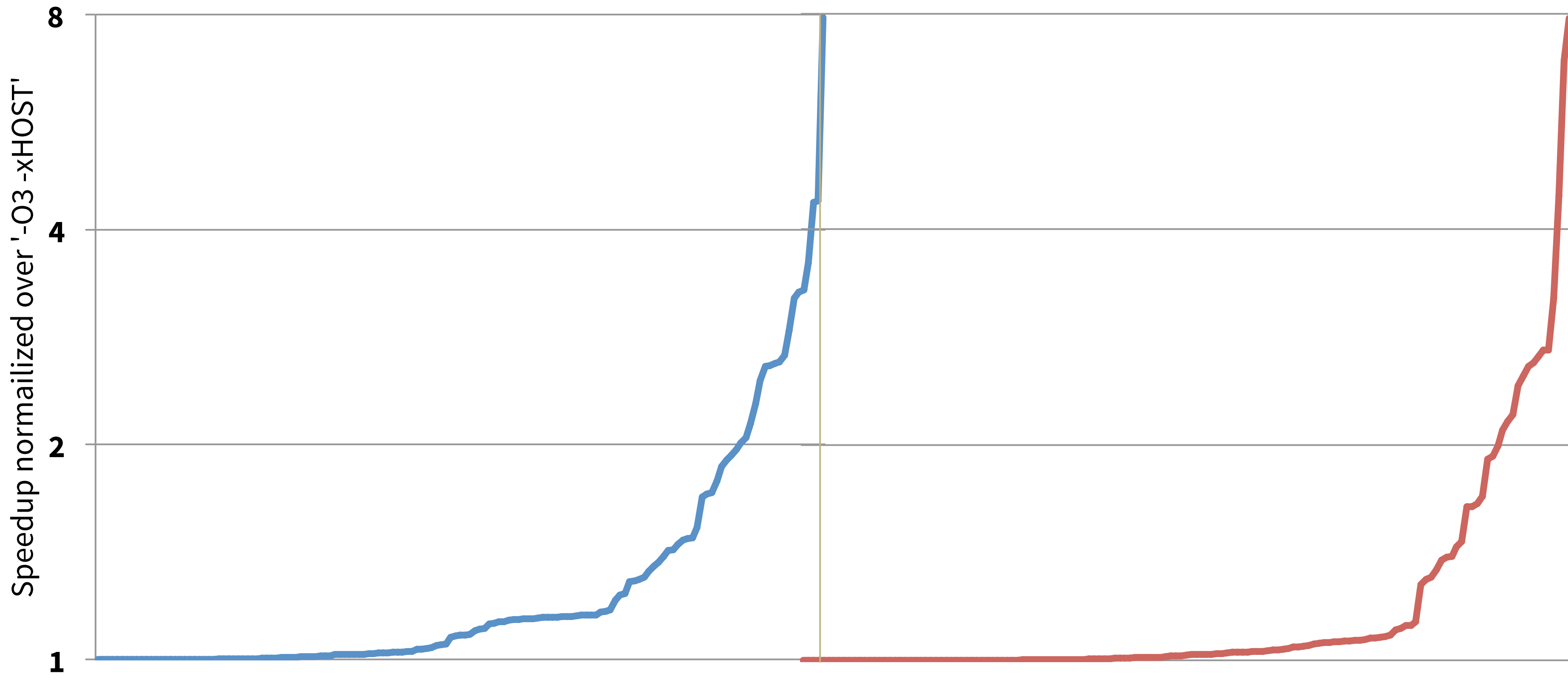
Execution Configuration

- 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

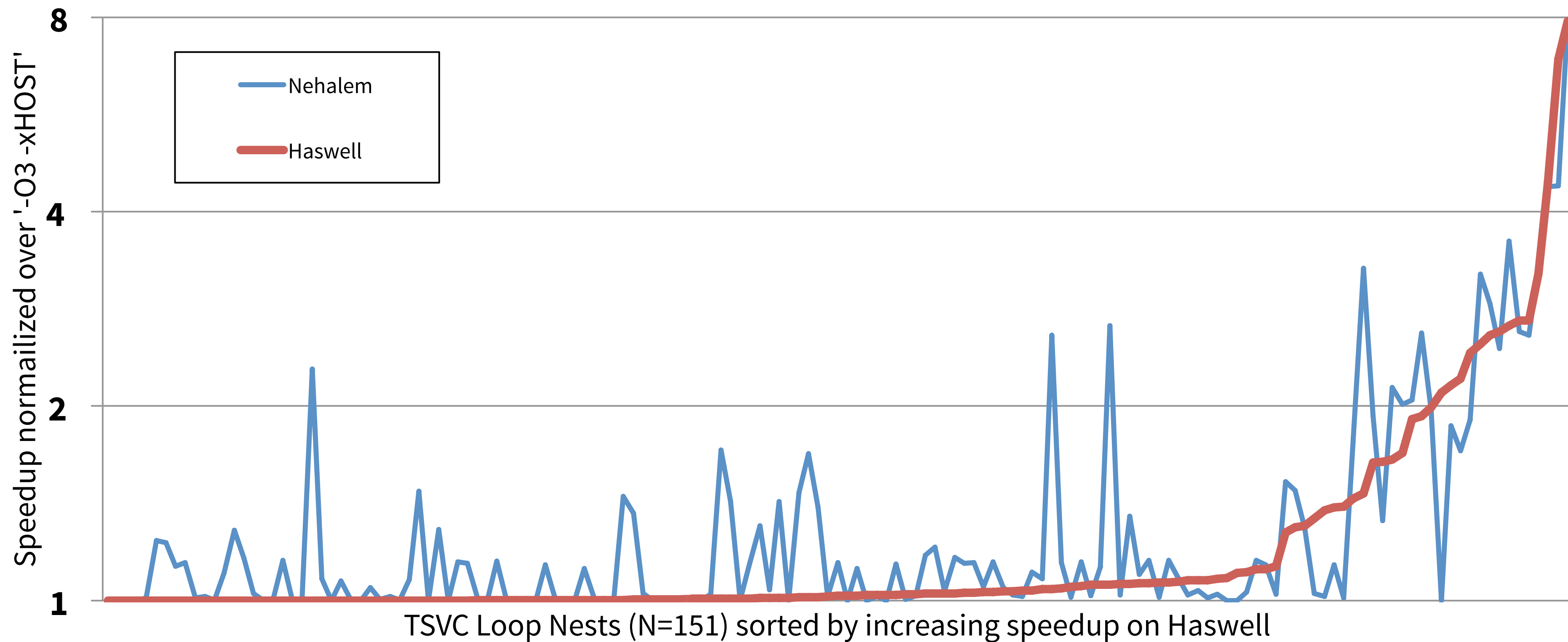
EXHAUSTIVE SEARCH SPACE SPEEDUP

TSVC Results



University of Delaware

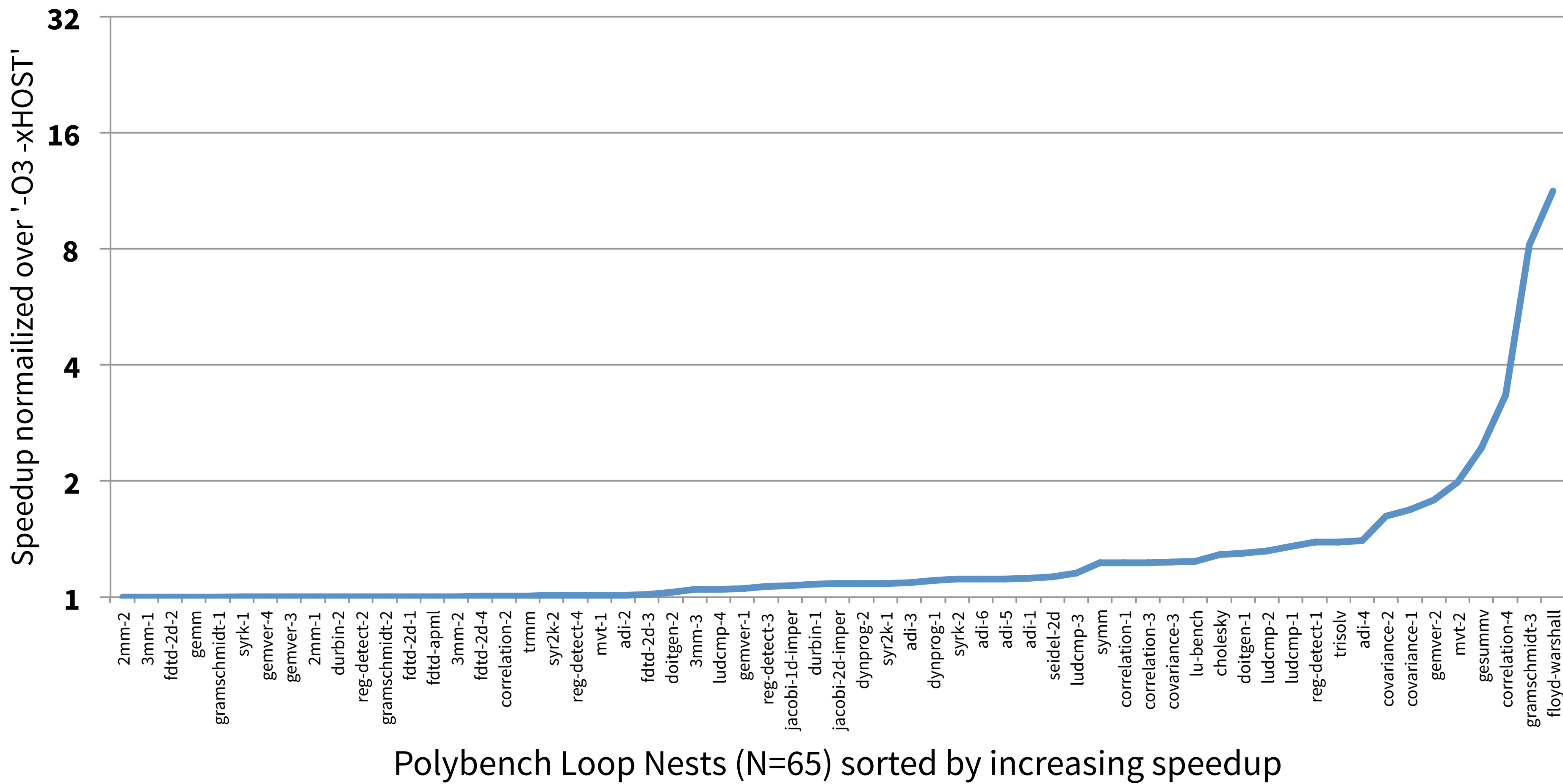
TSVC Cross-Architecture Analysis



Correlation: C = 0.8945

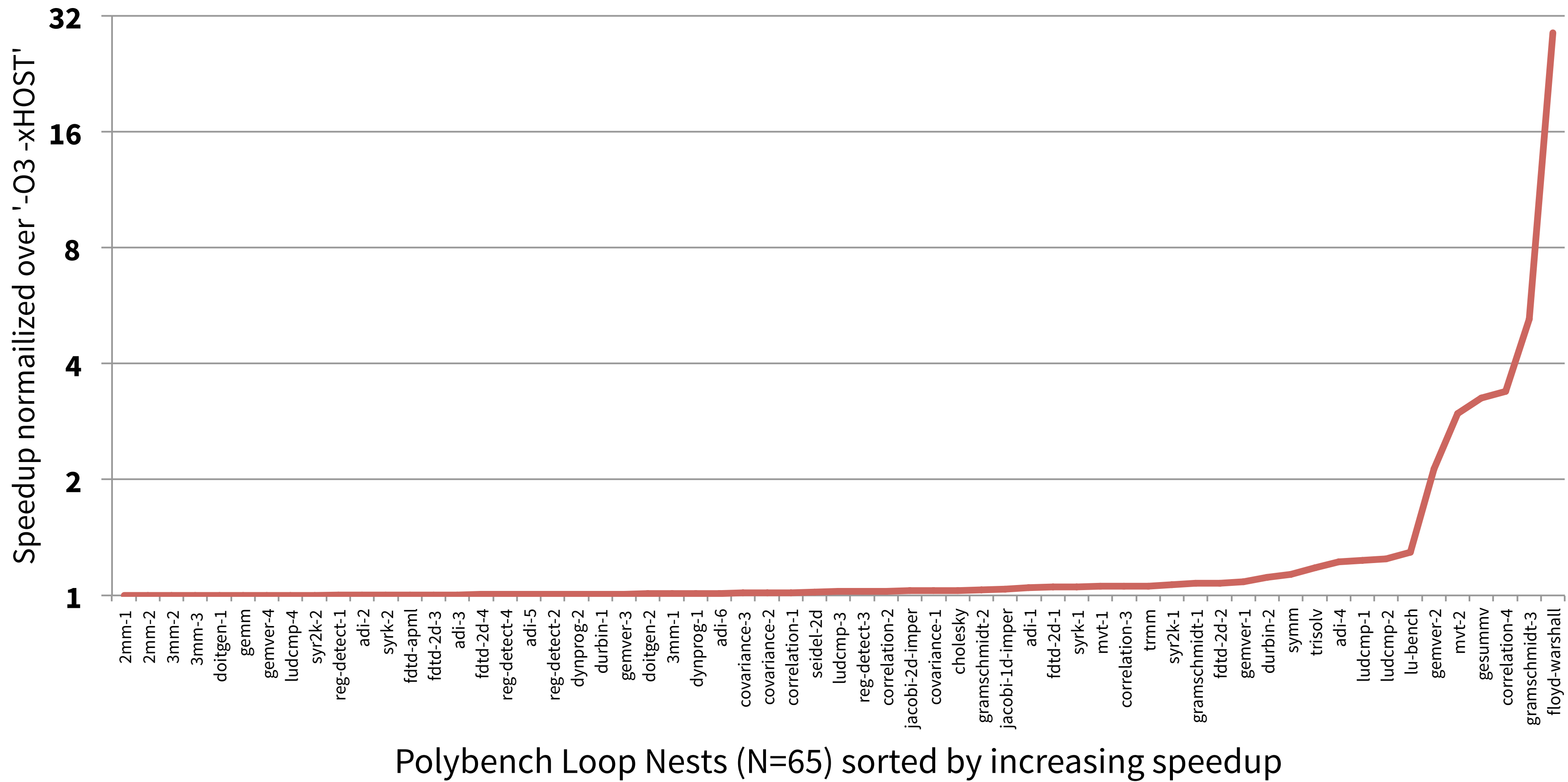
UNIVERSITY OF DELAWARE

PolyBench Results - Nehalem

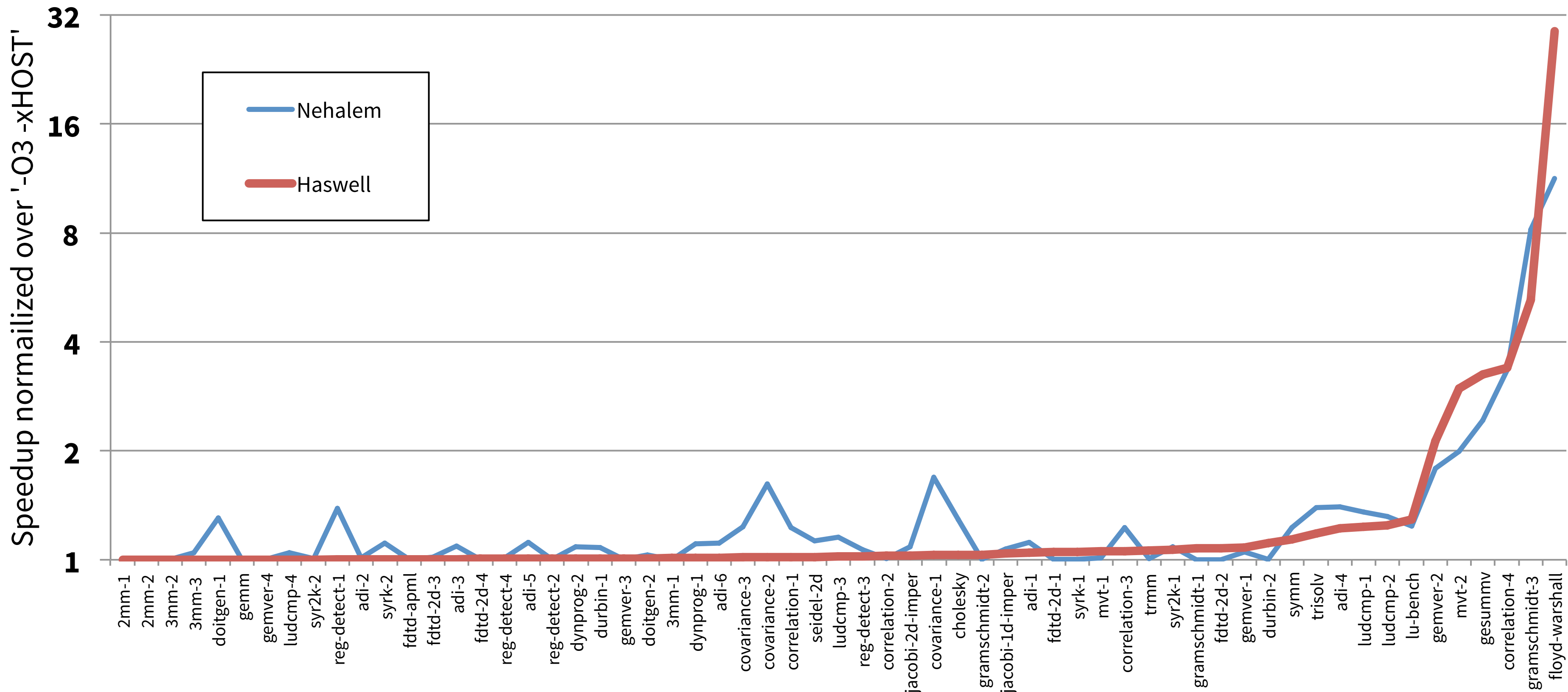


UNIVERSITY OF DELAWARE

PolyBench Results - Haswell



PolyBench Cross-Architecture Analysis



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

<u>Correlation:</u> C = 0.8894

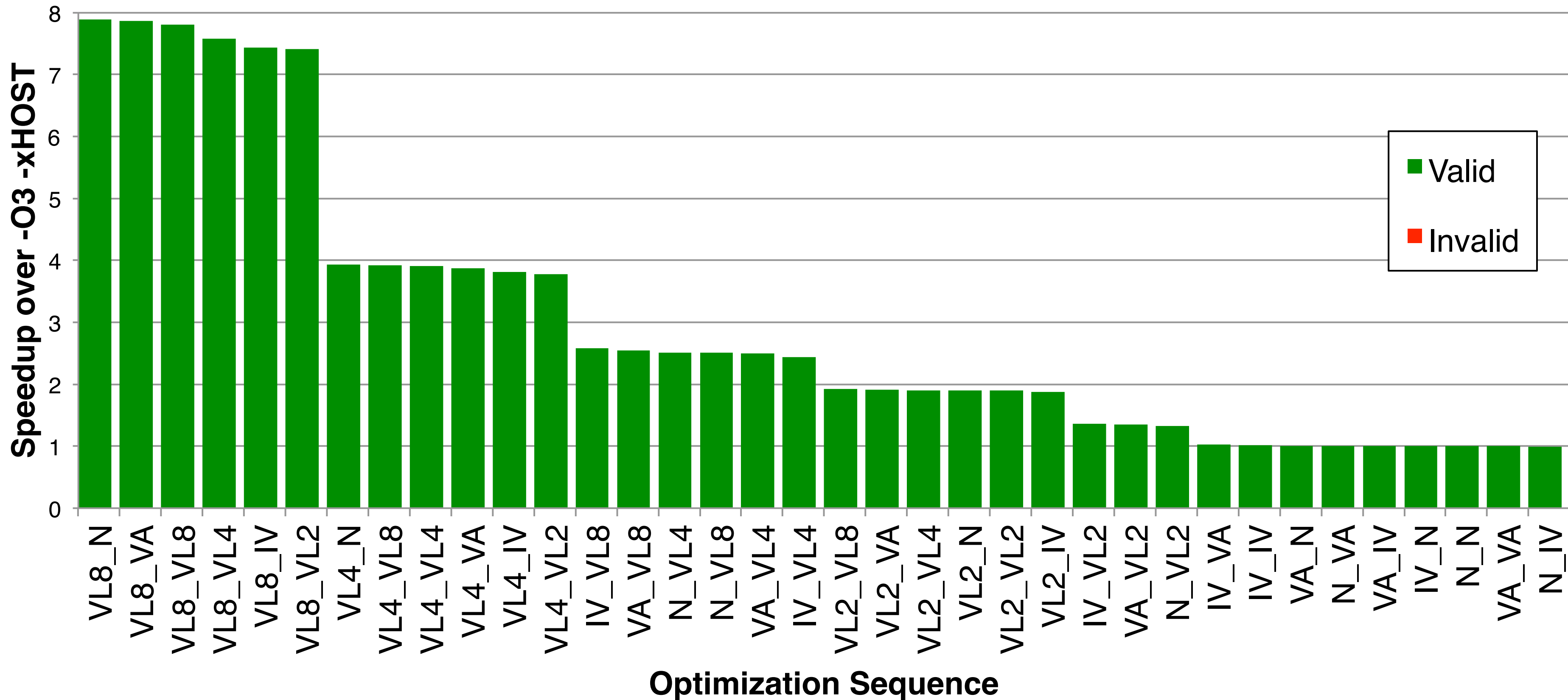
Experiment Results

EXHAUSTIVE SEARCH SPACE VALID CODE GENERATION

UNIVERSITY OF DELAWARE

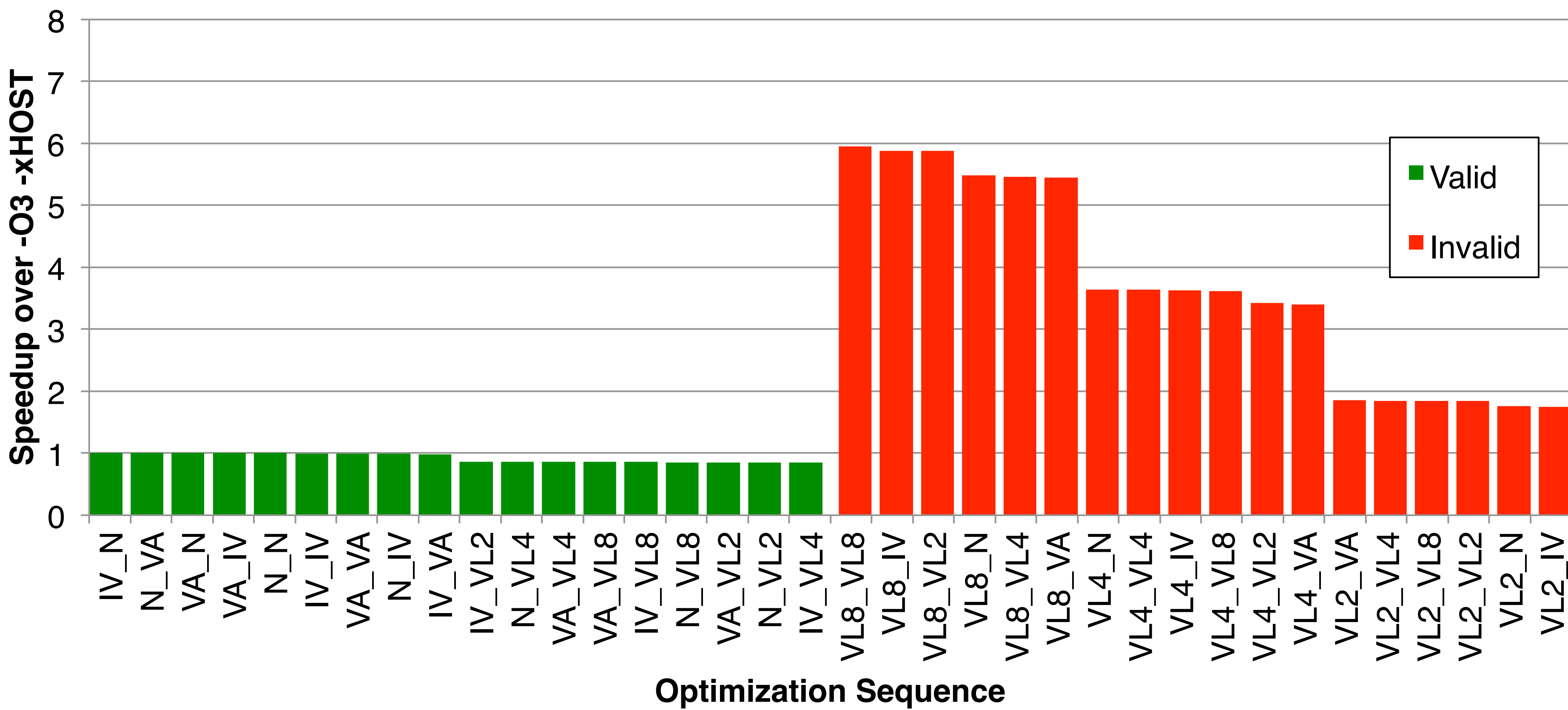
Version Generation Example

TSVC s256 Speedups



Version Generation Example

TSVC s126 Speedups



Version Generation Statistics

Benchmark	Arch	Valid	Invalid	Error
TSVC	Nehalem	1832	151	3
TSVC	Haswell	1826	155	5
PolyBench	Nehalem	5204	3826	0
Polybench	Haswell	5204	3826	0

Version Generation Across Architecture

Architecture	Valid Fastest	Invalid Fastest
Nehalem	140	11
Haswell	143	8

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

loop	1-shot	3-shot	Optimal
s126	1.91	1.92	6.28
s221	**0.39**	**1.42**	1.42
s2251	2.03	**2.44**	2.44
s244	**0.46**	**1.36**	1.36
s256	0.98	0.99	7.92
s3112	1.99	**4.03**	4.03
s321	**0.50**	0.97	2.13
s424	0.99	1.94	2.88
Arith. Mean	*0.70*	*0.98*	*1.47*
Geo. Mean	*0.61*	*0.85*	*1.32*

74% of Optimal on Nehalem 3-shot

TSVC Speedup Predictor - Haswell

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

77% of Optimal on Haswell 3-shot

PolyBench Speedup Predictor - Nehalem

loop	1-shot	3-shot	Optimal
2mm-1	**0.25**	**0.25**	1.00
adi-4	0.99	1.32	1.40
correlation-1	1.00	1.22	1.22
covariance-1	1.37	**1.68**	1.68
dynprog-1	0.98	1.00	1.10
floyd-warshall	5.22	**8.52**	11.30
gemm	**0.14**	**0.32**	1.00
grammschmidt-3	1.00	**7.38**	8.17
Arith. Mean	*0.97*	*1.21*	*1.46*
Geo. Mean	*0.85*	*0.98*	*1.24*

84.44% of Optimal on Nehalem 3-shot

PolyBench Speedup Predictor - Haswell

loop	1-shot	3-shot	Optimal
2mm-1	**0.67**	**0.68**	1.00
adi-4	**1.22**	**1.22**	1.22
correlation-1	0.98	**1.00**	1.01
covariance-1	**1.02**	**1.02**	1.02
dynprog-1	0.99	1.00	1.00
floyd-warshall	**25.88**	**25.88**	28.86
gemm	**0.12**	**0.38**	1.00
grammschmidt-3	3.37	3.40	5.22
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?