Optimization by Run-time Specialization for Sparse-Matrix Vector Multiplication

Maria J. Garzaran
University of Illinois at Urbana-Champaign

Joint work with Sam Kamin (UIUC) and Baris Aktemur (Ozyegin University, Turkey)

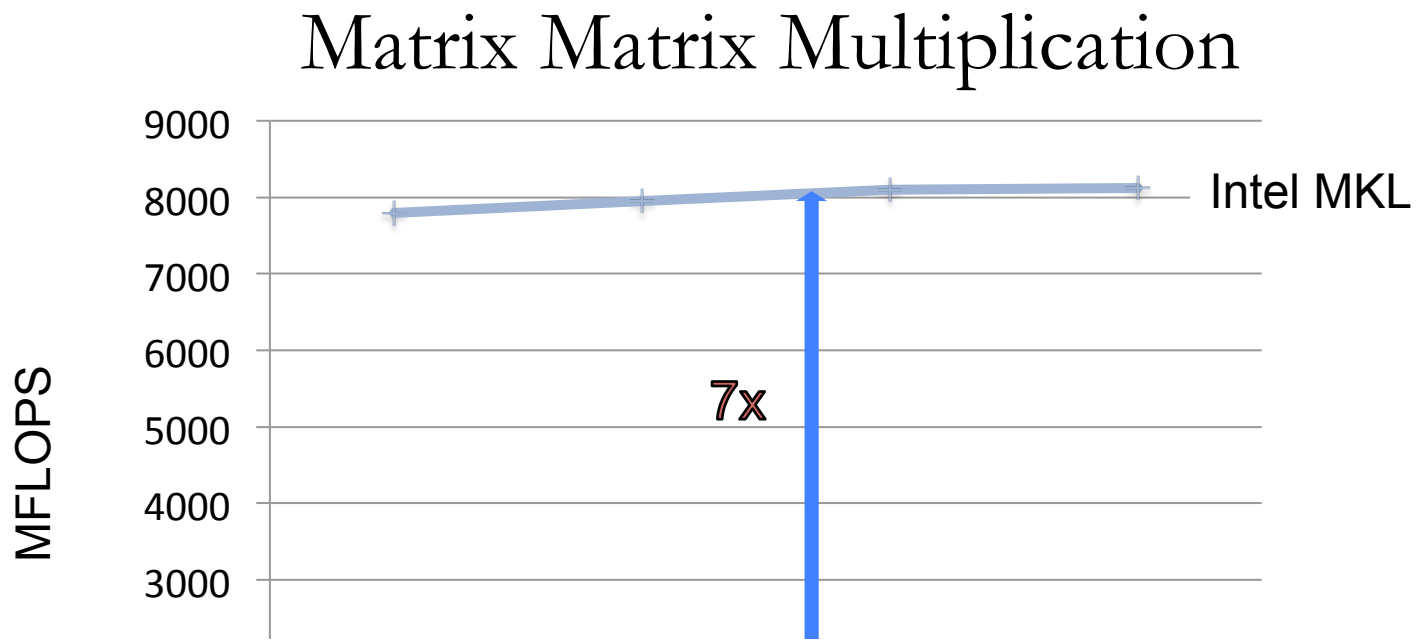
Why autotuning?

- Computers have complex hardware
  - e.g., branch prediction, hardware prefetch, multicores, …
- Generating highly efficient code is difficult
  - Compilers
    - Do not produce expected results
  - Manual
    - Increase cost
    - Low performance if not enough resources
- Must automate tuning

Compilers

- Compilers have limitations
  - Lack semantic information
    - fewer choices
  - Must target all class of applications
  - Must be reasonably fast

Compiler vs. Manual Tuning



[PACT'11] An Evaluation of Vectorizing compilers.
Joint project with Saeed Maleki, David Padua, Yaoquing Gao, and Tommy Wong.

[Forthcoming paper] On the Effectiveness of OpenACC compilers.
Joint project with Swapnil Ghike, Ruben Gran, and David Padua.

What is Autotuning?

- Strategy to generate highly efficient codes that adapt to
  - the target platform
  - input set
- Needs multiple versions of a given program
  - different compiler optimizations
  - different algorithms
  - different parameter values for a given algorithm
- Typically uses empirical search (executes each version in the target architecture) and select the fastest
  - can search using analytical models

Outline

- Input-independent
- Input-dependent
  - Pure algorithms
  - Specialization and Runtime code generation

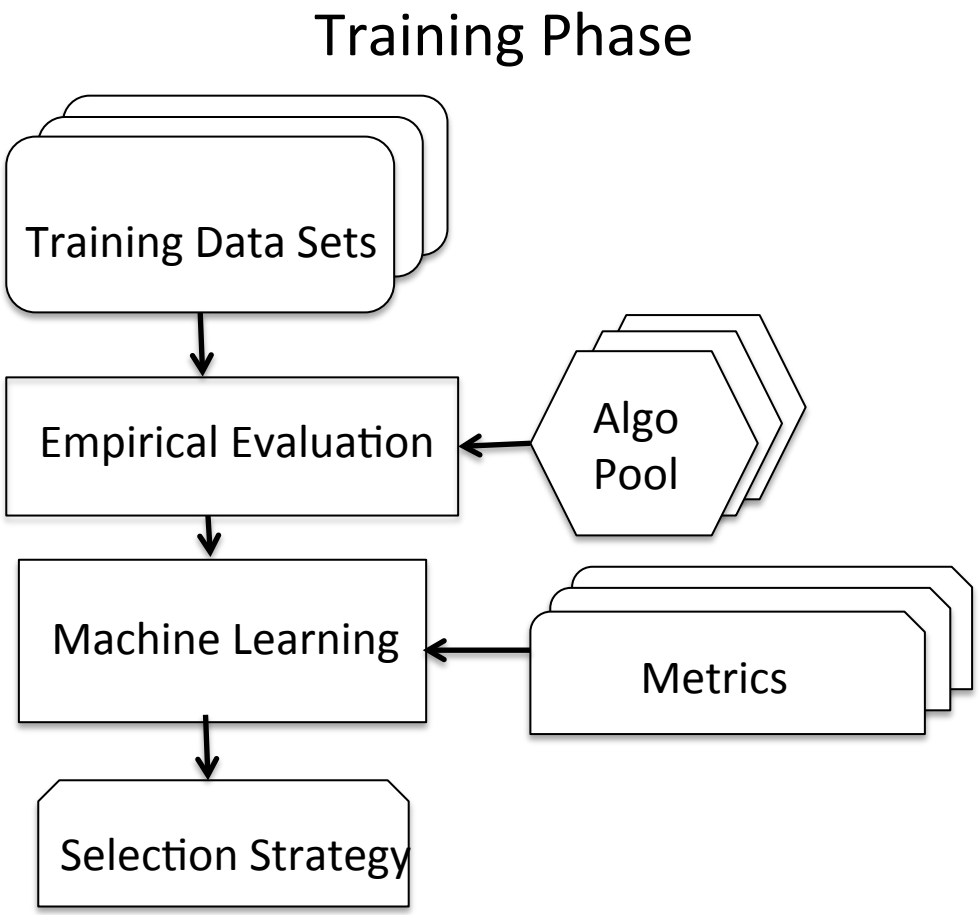
Outline

- Input-independent
- Input-dependent
  - Pure algorithms
  - Specialization and Runtime code generation

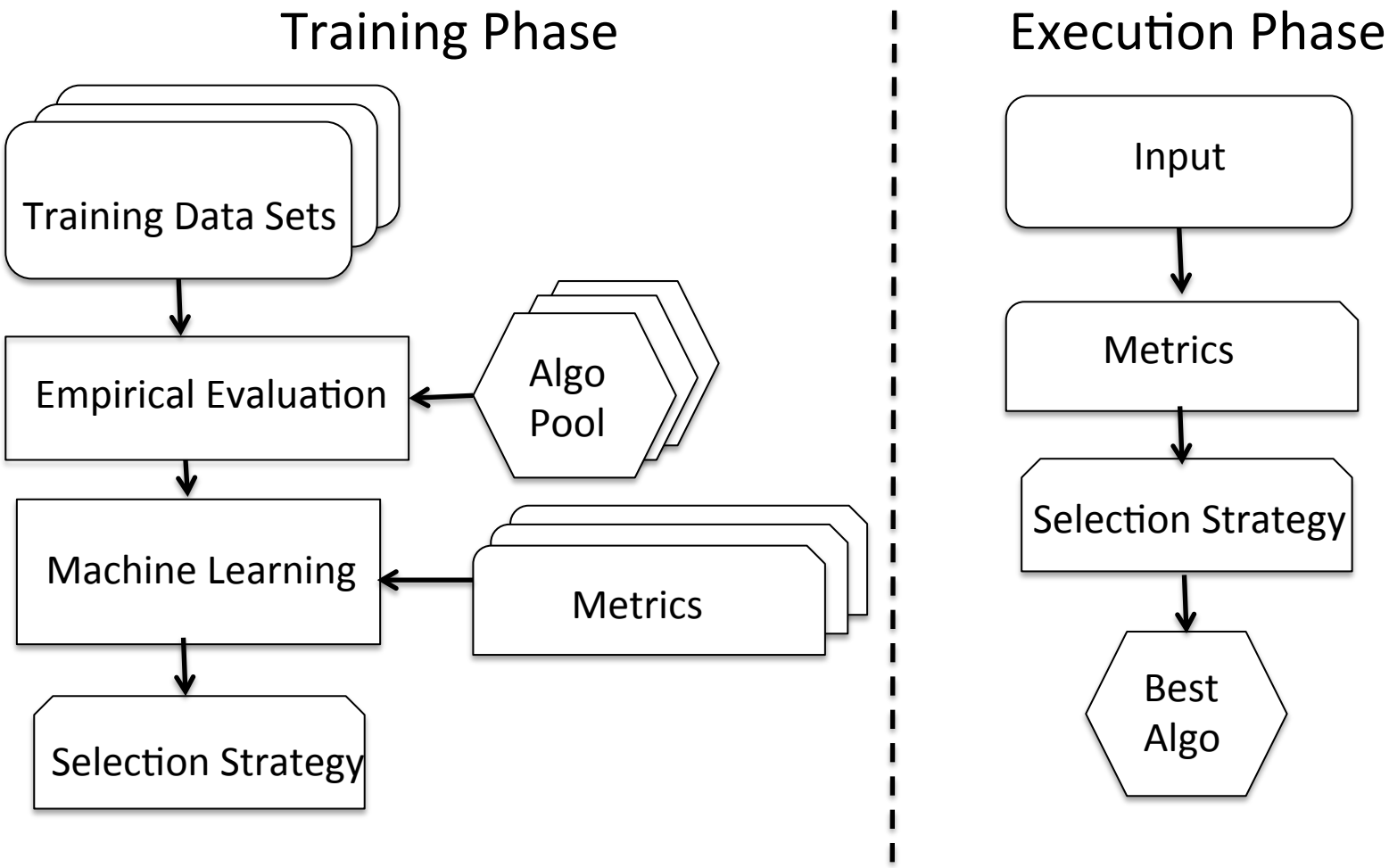
Input-dependent performance

- Performance depends on the characteristics of the input:
  - frequent pattern mining
  - sorting
  - sparse matrix-vector(s) multiplication
  - graph problems

Input-dependent Autotuning



Input-dependent Autotuning



Outline

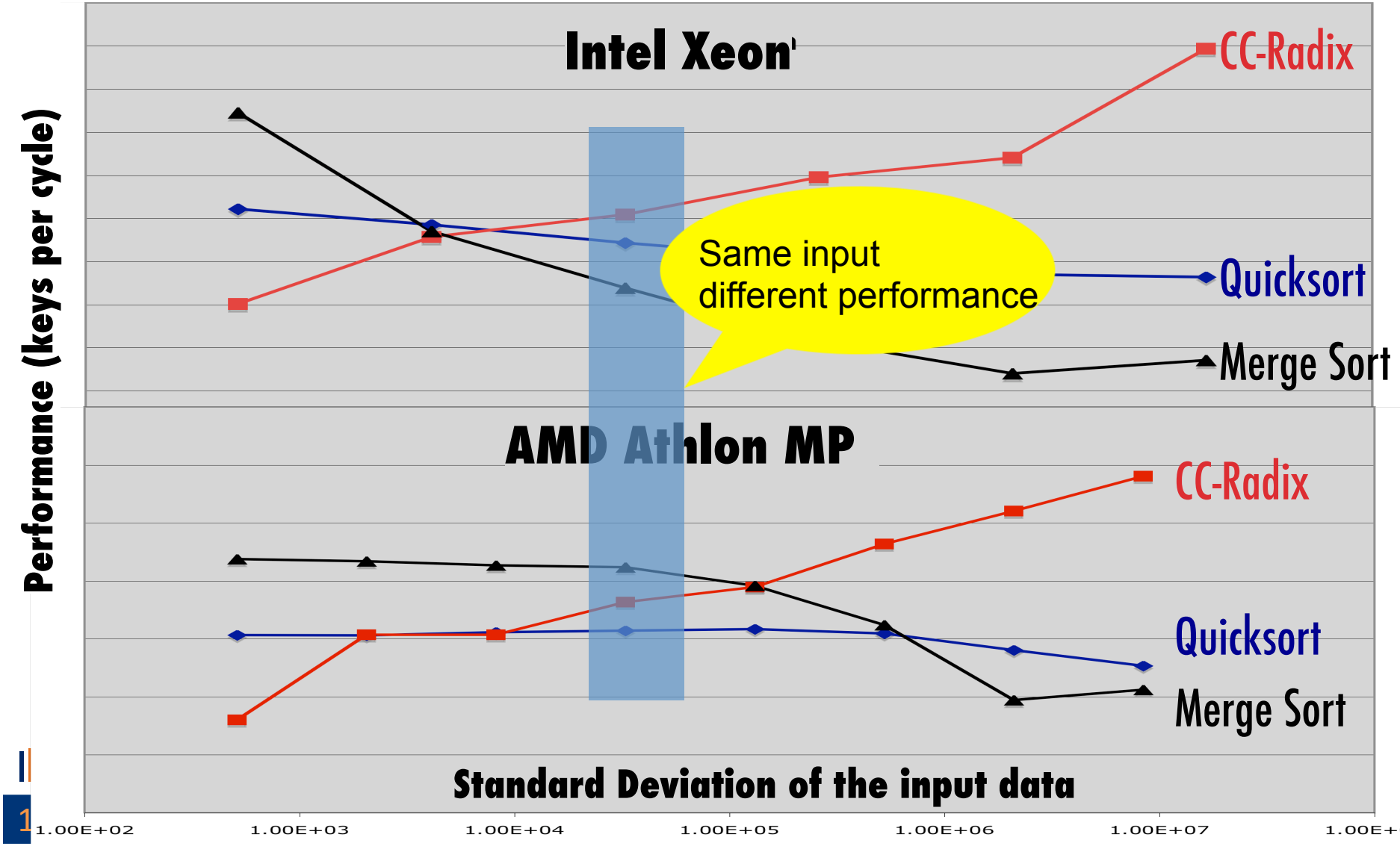
- Input-independent
- Input-dependent
  - Pure algorithms
    - Sorting
  - Specialization and Runtime code generation

Different Algorithms for Sorting

- Sorting is another interesting problem for autotuning
  - The compiler can do little about sorting
  - Performance depends on machine and input data characteristics.
- No single algorithm is the best for all input and platforms

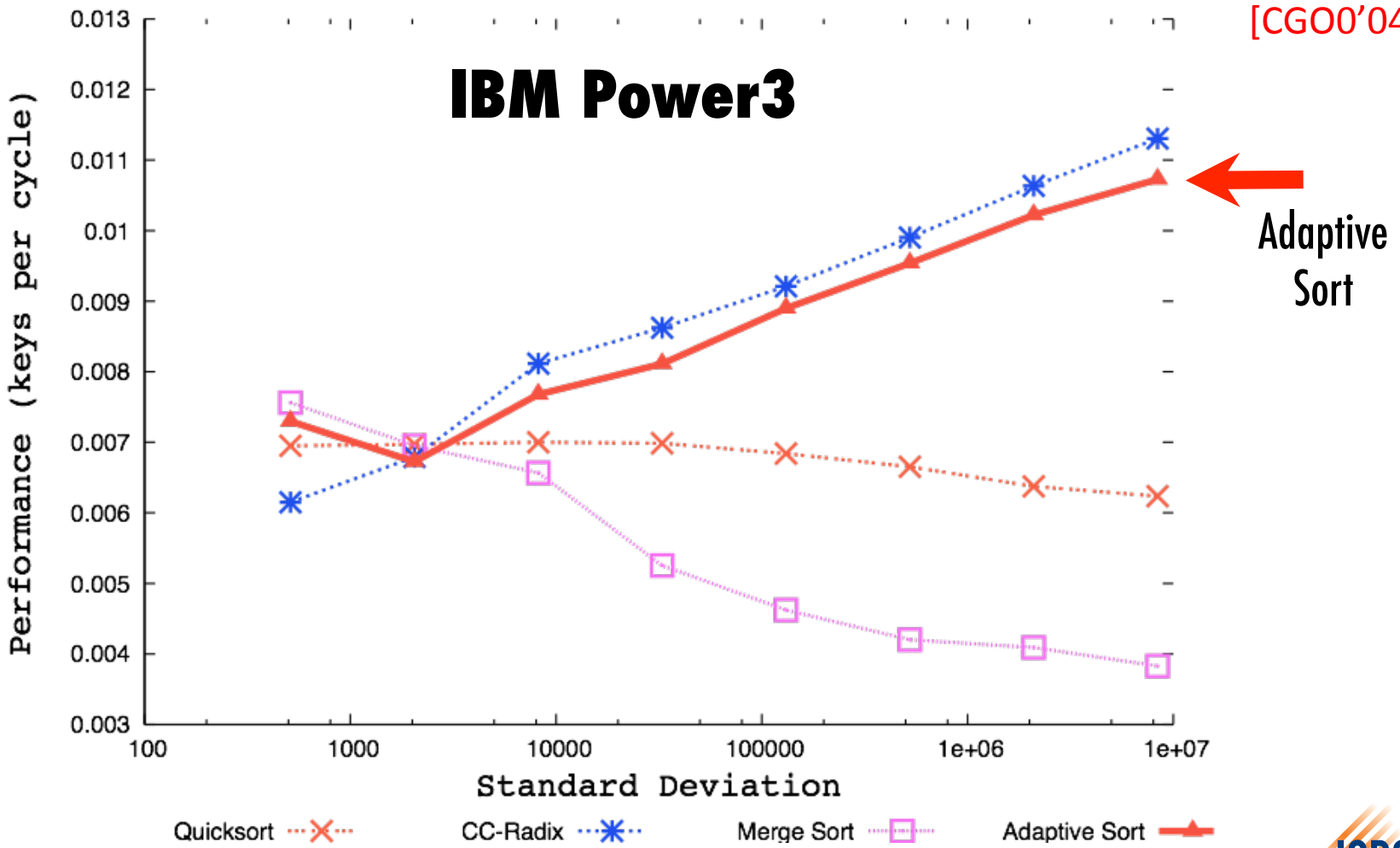
[CGO'04][CGO'5]
Joint project with Xiaoming Li and David Padua

Different Algorithms for Sorting



Selecting the best algorithm

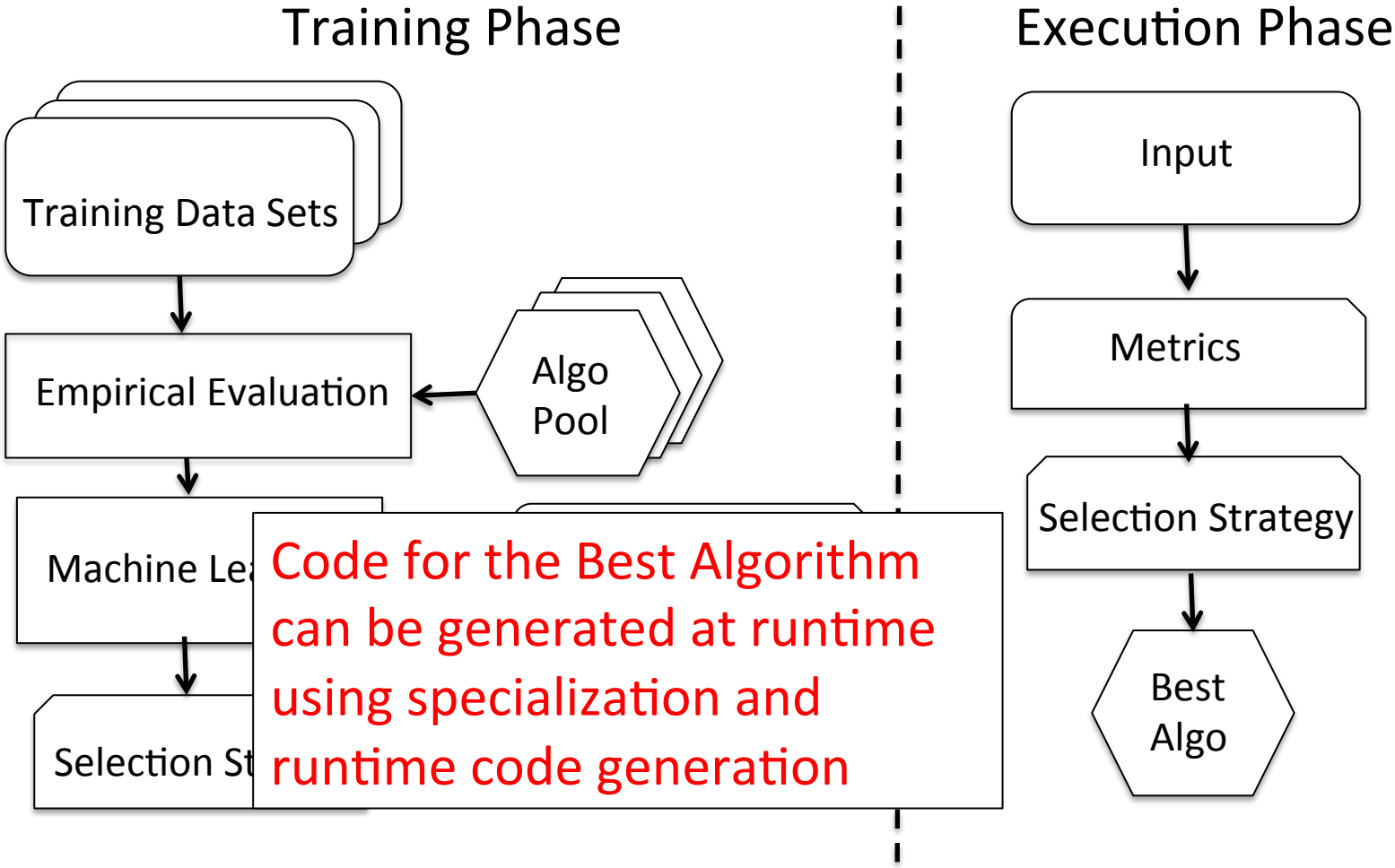
[CGO0'04]



Outline

- Input-independent
- Input-dependent
  - Pure algorithms
  - Specialization and Runtime code generation
    - Sparse Matrix-Vector Multiplication

Input-Dependent Autotuning



Code for the Best Algorithm can be generated at runtime using specialization and runtime code generation

Sparse Matrix-Vector Multiplication

- y = Ax, A sparse but known

- y = Ax is performed many times
  - Justifies one-time tuning effort

We generate code that is specialized for the matrix A
Code is specialized for the location of the nonzeros in A

Sparse Matrix-Vector Multiplication

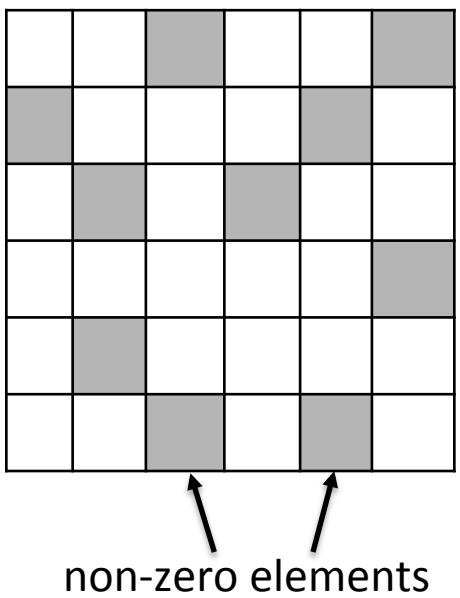
$$A = \begin{bmatrix} b & c & . & . \\ . & a & . & . \\ c & . & b & b \\ . & . & a & . \end{bmatrix}$$

CSR format

value	b c a c b b a
col_idx	0 1 1 0 2 3 3
row	0 2 3 6 7

Sparse Matrix Vector Multiplication (CSR)

Matrix A

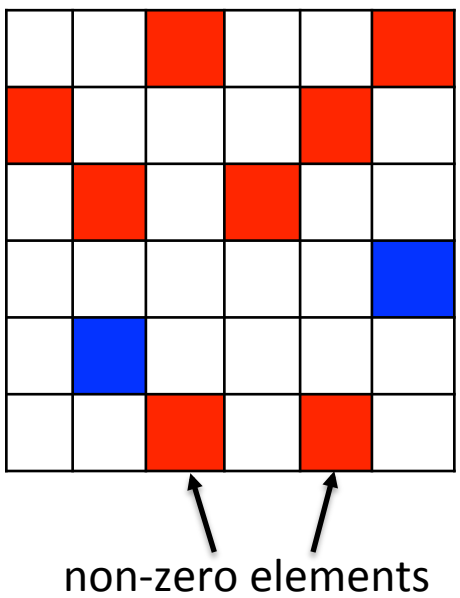



```
/* loop over rows */
for (i=0; i<m; i++) {
double y_i= y[i];
/*loop over non-zero elements in each row */
  for (jj = row_start[i]; jj<row_start[i+1];
        jj++, col_idx++, value++){
        y_i += value[0]*x[col_idx[0]];
  }
 y[i] = y_i;
}
```

indirect array accessing

Specialization 1: Number of Zeros

Matrix A

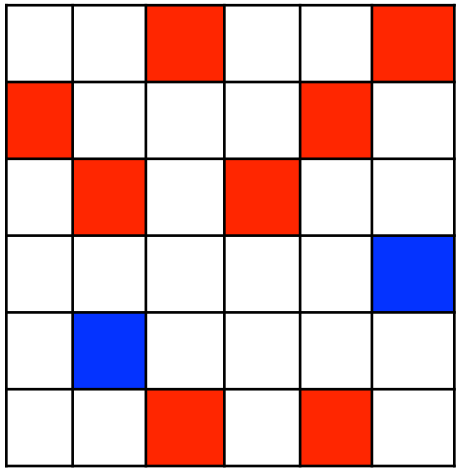



```
/* loop over rows */
for (i=0; i<m; i++) {
double y_i= y[i];
/*loop over non-zero elements in each row */
  for (jj = row_start[i]; jj<row_start[i+1];
        jj++, col_idx++, value++){
        y_i += value[0]*x[col_idx[0]];
  }
 y[i] = y_i;
}
```

indirect array accessing

Specialization 1: Number of Zeros

Matrix A



- Still indirect access
- Unrolled loop

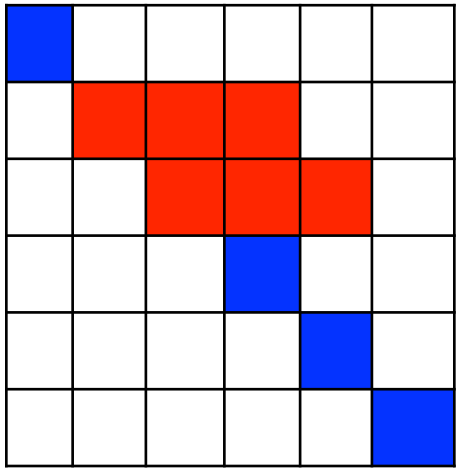
4 rows with 2 non-zeros

```
for (i=0; i<4; i++) {
    row = rows[a++];
    y[row] += value[b]*x[col_idx[b]];
    y[row] += value[b+1]*x[col_idx[b+1]];
    y[row] += y;
    b += 2;
}
for (i=0; i<2; i++) {
    row = rows[a++];
    y[row] += value[b]*x[cols[b]];
    b += 1;
}
```

completely unrolled inner loop

Specialization 2: Stencils

Matrix A



- No indirect access
- Unrolled loop

- Specialize code based on location of the non-zeros with respect to the diagonal

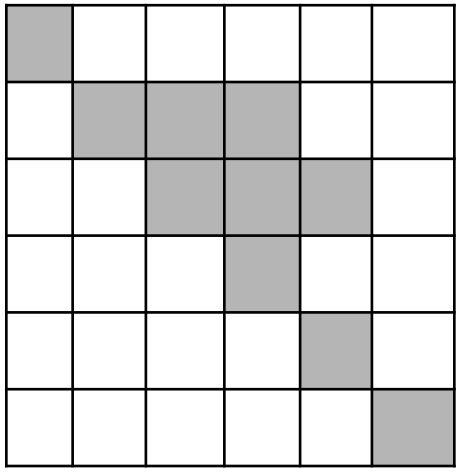
```
int stencil_1[1,2];
for (i=0; i<2; i++) {
    row = stencil_1[i];
    xx=x+row;
    w[row] += value[0]*xx[0];
    w[row] += value[1]*xx[1];
    w[row] += value[2]*xx[2];
    b += 3;
    }
for (i=0; i<4; i++) {
    w[row] += value[0]*xx[1];
    b += 1;}
```

no indirect access

inner loop is completely unrolled

Specialization 3: Unfolding

Matrix A



- Straight-line code that performs all the floating-point operations.

$w[1] \mathrel{+}= value_{1,2} * v[1]$

$w[2] \mathrel{+}= value_{2,2} * v[2] + value_{2,3} * v[3] + value_{2,4} * v[4];$

$w[3] \mathrel{+}= value_{3,3} * v[3] + value_{3,4} * v[4] + value_{4,5} * v[5];$

$w[4] \mathrel{+}= value_{4,4} * v[4];$

….

Code size is proportional to the number of non-zeros

Sparse Matrix-Vector Multiplication

- Many methods can be used
  - plain CSR
  - Unroll{i}
  - OSKI
  - Diagonal
  - NonZeros
  - Stencil
  - Unfolding
  - GenOSKI
  - Compressed Format

off-line

runtime code generation

Experimental results

- Have run experiments on 3 different platforms:
  - loome2: Intel Core i7
  - loome3: Intel Core i5
  - i2pc3: Intel Xeon E7
- Codes are compiled using clang with –O3.
- Codes run in parallel using OpenMP, 4 threads
- 53 matrices from Matrix Market and University of Florida Sparse Matrix Collection

Overall Speedups

The table shows average speedups with respect to MKL running in parallel with 4 threads

	Loome2	Loome3	i2pc3
Best	1.72	1.45	1.53
NonZeros	23 (1.18)	23 (1.31)	9 (1.45)
Stencil	18 (1.56)	14 (1.51)	18 (1.23)
Unfolding	9 (2.33)	10 (1.91)	20 (1.92)
GenOski	2 (1.14)	6 (1.27)	-
Compressed	1 (1.04)	-	-
	53	53	47

Specialization methods have higher speedups

Machine / Matrix	n	nz	Loome2		Loome3		i2pc3	
torso2	115K	1M	Stencil	1.72	Stencil	1.46	Unfolding	1.77
fidap011	16K	1M	CSR4	1.04	GenOski	1.21	MKL	1.0
s3dkq4m2	90K	2,2M	Stencil	1.55	Stencil	1.47	Stencil	1.35
engine	143K	2,4M	Unfolding	3.31	Unfolding	2.82	Unfolding	5.35

Method

Speedup over MKL

Specialization methods have higher speedups

Machine / Matrix	n	nz	Loome2		Loome3		i2pc3	
torso2	115K	1M	Stencil	1.72	Stencil	1.46	Unfolding	1.77
fidap011	16K	1M	NonZeros	1.04	GenOski	1.21	MKL	1.0
s3dkq4m2	90K	2,2M	Stencil	1.55	Stencil	1.47	Stencil	1.35
engine	143K	2,4M	Unfolding	3.31	Unfolding	2.82	Unfolding	5.35

	# Stencils	#Patterns	#Distinct Values
torso2	3,148	81	806653
fidap011	7,432	1684	211503
s3dkq4m2	1,131	380	74282
engine	84,195	108	1

How to predict the best method?

```
if (#nz || #distinct values) is small
    Unfolding
else if #stencils is small
    Stencil
      else if  #patterns is small
         GenOSKI
         else NonZeros
```

Status of the project

- Ongoing work
  - Designing more methods for code specialization
    - Compressed formats, hybrids?
  - Need fast runtime code generation
    - Currently using LLVM
    - Parallel code generation
  - Need to select the best specialization method
    - Identify the metrics to select the best method

Summary - I

- Code Specialization can produce speed-ups for most matrices and machines
- Novel way to autotune codes that adapt to the input:
  1. Algorithm selection
  2. Specialization and Runtime Code Generation
- Autotuning can be done in many problem domains

Summary - II

Autotuning produces:

+ highly efficient codes that adapt to the target machine and input

+ codes have longer life span,

but

- requires deep understanding of the algorithms
- codes are more difficult to debug

Questions?

garzaran@illinois.edu