Highly Scalable Multi-Objective Test Suite Minimisation Using Graphics Card

Shin Yoo, Mark Harman
CREST, University College London, UK
Shmuel Ur
University of Bristol, UK

It is all good improving SBSE for practitioners, but...

I have waited for hours while my SBSE experiment was running multiple times for statistical analysis.

I have waited for hours while my SBSE experiment was running multiple times for statistical analysis.

- Statistical Analysis? What is that?

I have waited for hours while my SBSE experiment was running multiple times for statistical analysis.

- Statistical Analysis? What is that?

- GECCO 2011 GPGPU Competition
  - Full Implementation of an Estimation of Distribution Algorithm on a GPU, S. Poulding, J. Staunton, N. Burles
- Genetic Programming on GPU
  - A SIMT Interpreter for Genetic Programming, W. B. Langdon

Outline

- Motivation
- What is GPGPU?
- Reformulating Test Suite Minimisation/MOEA for GPU
- Empirical Results
- Lessons Learned

Motivation

- Computational Characteristics of Population-based EA
  - Single fitness function (most of the time)
  - Many individual to evaluate
  - Single Instruction Multiple Data (SIMD) Architecture
  - Can be expensive!

Motivation

- CPU-based Parallelism
- Multi-core CPUs do improve situation, but not much
- Clusters are expensive and difficult to manage
- Concurrent programming is hard

GPGPU

- General Purpose computation on Graphics Processing Units: uses graphics card to run massively parallel programs
- Modern graphics cards have hundreds of small cores, originally intended to be used to run shader algorithms
- Single algorithm, many different triangles: sounds familiar?

GPGPU

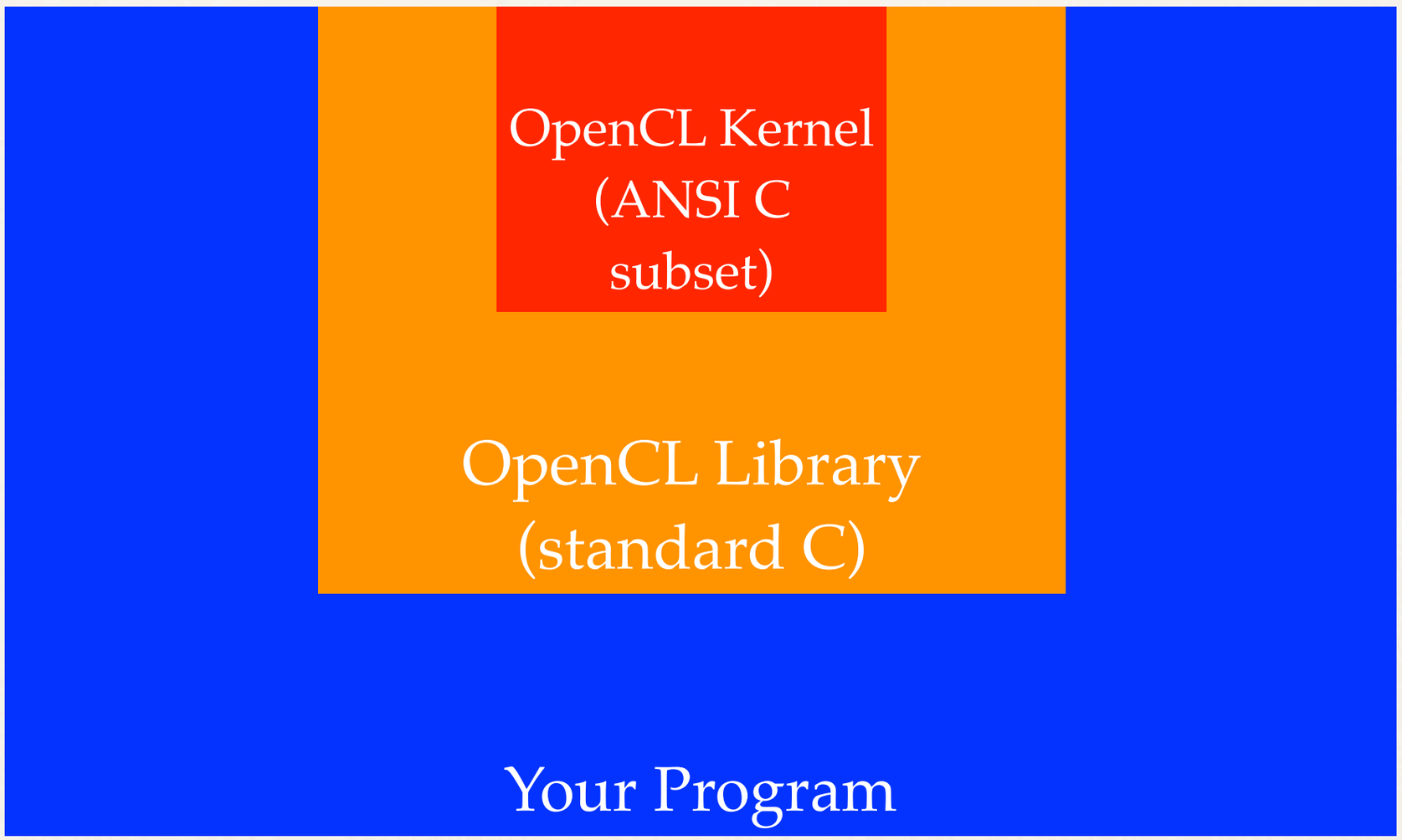
- Emerged during early 2000
  - Mapping Computational Concepts to GPUs, Mark Harris, ACM SIGGRAPH, 2005
- Now a must-have ingredient of any significant parallelisation

Tianhe, currently ranked at 2nd fastest supercomputer with 2.6 petaflops
14,337 Intel CPU + 7,168 NVidia GPU

GPGPU: Frameworks

- CUDA (Compute Unified Device Architecture) by NVidia
  - Works for NVidia Cards
  - Supports OpenCL too
- OpenCL (Open Computing Language) by Khronos
  - Device agnostic
  - Included in Mac OS X 10.6 and onwards

OpenCL: Language



OpenCL: Kernel Code

- Kernels are the smallest executable unit for OpenCL
- Compiled at runtime
- Many useful built-in functions and types, such as vector arithmetic. For example,
  - `float`*n*: float vector (*n* = 2, 4, 8, 16)
  - `float16 a = vload(0, ptr); float16 b = sqrt(a);`

OpenCL: Kernel Code

```
__kernel void multiply(__global float* C, __global
float* A, __global float* B, int wA, int wB)
{
    // Get the thread ids for this tile
    int tx = get_global_id(0);
    int ty = get_global_id(1);

    // value stores the element
    // that is computed by the thread
    float value = 0;
    for (int k = 0; k < wA; ++k)
    {
        value += A[ty * wA + k] * B[k * wB + tx];
    }

    // Write the matrix to device memory each
    // thread writes one element
    C[ty * wB + tx] = value;
}
```

OpenCL: Kernel Code

```
__kernel void multiply(__global float* C, __global
float* A, __global float* B, int wA, int wB)
{
    // Get the thread ids for this tile
    int tx = get_global_id(0);
    int ty = get_global_id(1);

    // value stores the element
    // that is computed by the thread
    float value = 0;
    for (int k = 0; k < wA; ++k)
    {
        value += A[ty * wA + k] * B[k * wB + tx];
    }

    // Write the matrix to device memory each
    // thread writes one element
    C[ty * wB + tx] = value;
}
```

memory modifiers

OpenCL: Kernel Code

```
__kernel void multiply(__global float* C, __global
float* A, __global float* B, int wA, int wB)
{
    // Get the thread ids for this tile
    int tx = get_global_id(0);
    int ty = get_global_id(1);

    // value stores the element
    // that is computed by the thread
    float value = 0;
    for (int k = 0; k < wA; ++k)
    {
        value += A[ty * wA + k] * B[k * wB + tx];
    }

    // Write the matrix to device memory each
    // thread writes one element
    C[ty * wB + tx] = value;
}
```

memory modifiers

built-in functions

OpenCL: Wrappers

- JavaCL: http://code.google.com/p/javacl/
  - Completely Object-oriented design
  - Nice integration with Java API
  - Thanks, Olivier :)
- PyOpenCL: http://mathema.tician.de/software/pyopencl

A Peek into JavaCL

```
CLContext context = JavaCL.createBestContext();
CLProgram program = context.createProgram(myKernelSource).build();
CLKernel multiplyKernel = program.createKernel(
        "myKernel",
        new float[] { u, v },
        context.createIntBuffer(Usage.Input, inputBuffer, true),
        context.createFloatBuffer(Usage.Output, resultsBuffer, false)
);
...
multiplyKernel.enqueueNDRange(queue, new int[]{bWidth, aHeight}, new
int[]{16, 16}, new CLEvent[]{});
queue.finish();
...
fitnessBuffer = fitnessMem.readBytes(queue, 0, aHeight * bWidth * 4,
new CLEvent[]{}).order(ByteOrder.LITTLE_ENDIAN);
coverageBuffer = coverageMem.readBytes(queue, 0, populationSize * 4,
new CLEvent[]{}).order(ByteOrder.LITTLE_ENDIAN);
scoreBuffer = scoreMem.readBytes(queue, 0, populationSize * 4, new
CLEvent[]{}).order(ByteOrder.LITTLE_ENDIAN);
queue.finish();
```

Test Suite Minimisation

- The Problem: Your regression test suite is too large.
- The Idea: There must be some redundant test cases.
- The Solution: Minimise (or reduce) your regression test suite by removing all the redundant tests.

Test Suite Minimisation

Things to tick off (branches, statements, DU-paths, etc)

	r0	r1	r2	r3	Time
t0	1	1	0	0	2
t1	0	1	0	1	3
t2	0	0	1	1	7
t3	0	0	1	0	3

Cost of ticking things off

Your tests

Now the problem becomes the following: what is the subset of rows (i.e. tests) that, when combined, will cover (i.e. put ‘1’ on) the most of the columns?

Reformulation

$$A' = \begin{pmatrix} a_{1,1} & \dots & a_{1,m} \\ a_{2,1} & \dots & a_{2,m} \\ & \dots & \\ a_{l,1} & \dots & a_{l,m} \\ cost(t_1) & \dots & cost(t_m) \end{pmatrix}$$

A: the test suite

a_{ij} = # element i has been executed by test case j

$$B = \begin{pmatrix} b_{1,1} & \dots & b_{1,n} \\ b_{2,1} & \dots & b_{2,n} \\ & \dots & \\ b_{m,1} & \dots & b_{m,n} \end{pmatrix}$$

B: the population

$b_{jk} = 1$ if test case j has been selected by individual k;

0 otherwise

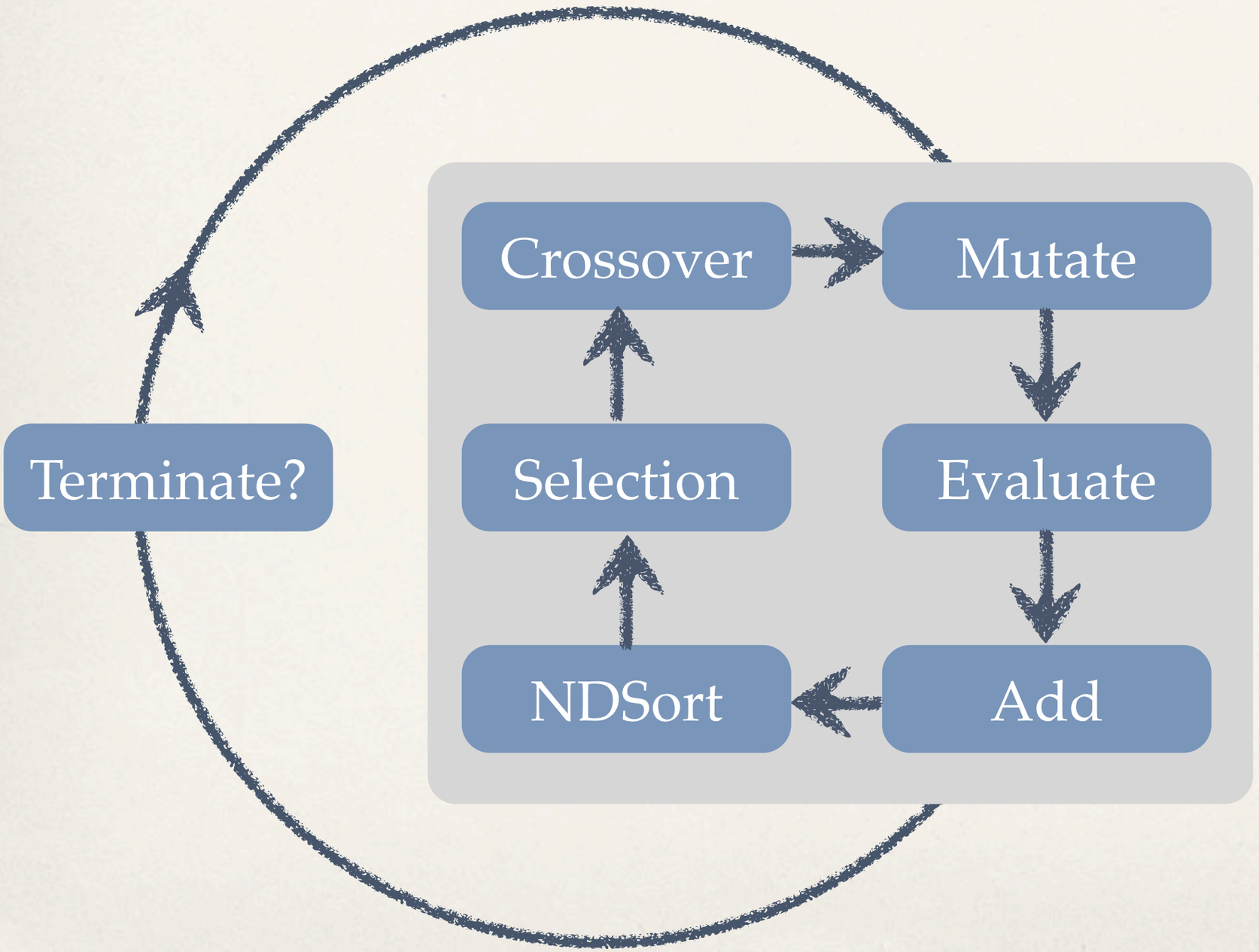
Reformulation

$$C' = AB = \begin{pmatrix} c_{1,1} & \dots & c_{1,n} \\ c_{2,1} & \dots & c_{2,n} \\ & \dots & \\ c_{l,1} & \dots & c_{l,n} \\ cost(p_1) & \dots & cost(p_n) \end{pmatrix}$$

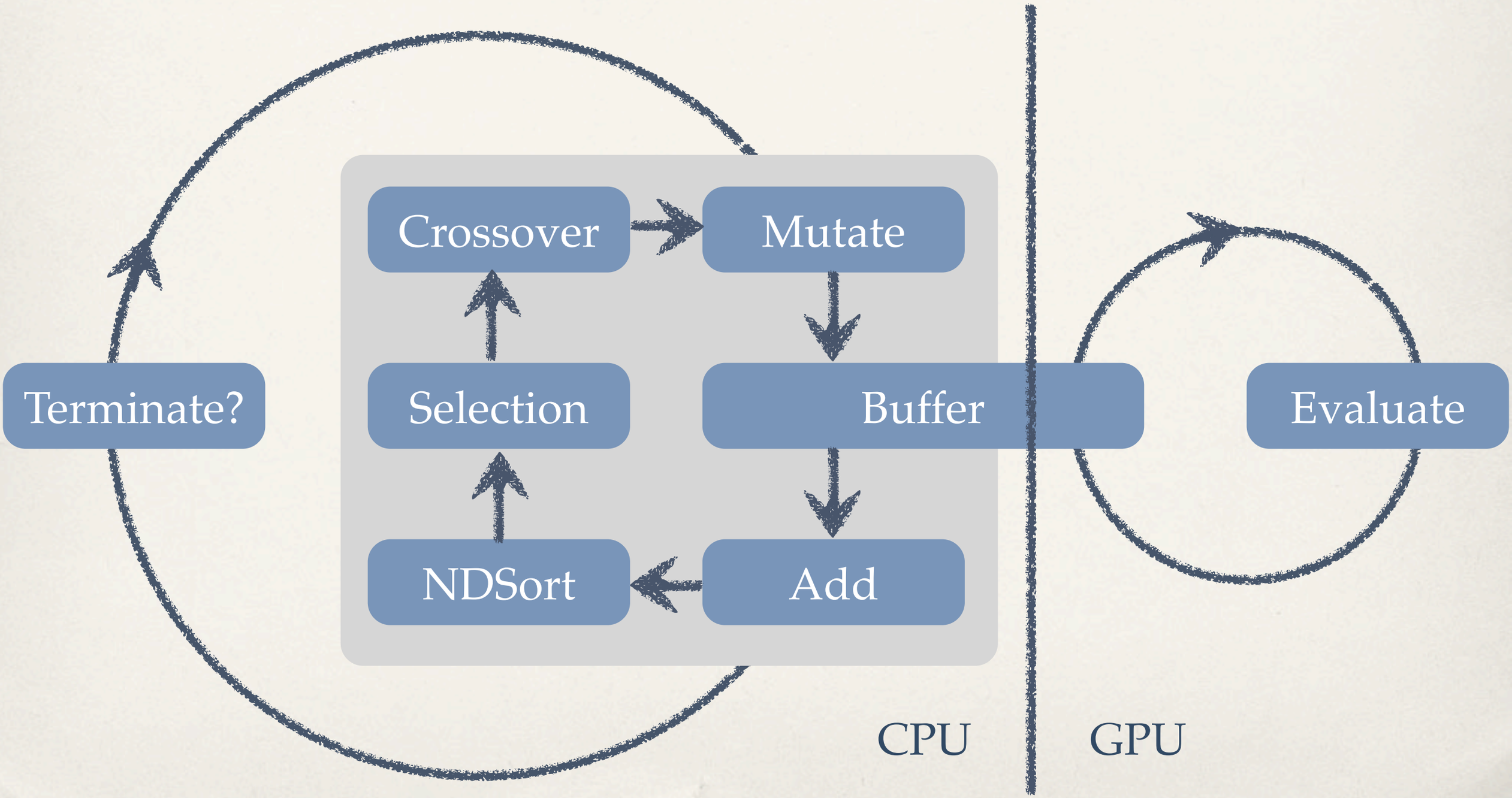
c_{ij} = # element *i* has been executed by test cases selected in individual *k*

$cost(p_k)$ = sum of execution cost of tests selected by individual k

Algorithm: NSGA-II



Algorithm: NSGA-II



Experimental Setup

- NSGA-II from jMetal, a Java MOEA library
- We have compared 5 different computational methods:
  - Baseline with OOP model: each individual is evaluated separately using a few method calls
  - JOMP with 1/2/4 threads: matrix-based approach on CPU with multi-threads
  - GPGPU: matrix-based approach on GPU, only fitness evaluation is performed on GPU

Experimental Setup

- Population size: 256 (1 individual per thread)
- Iteration: 250
- Single-point crossover with probability 0.9
- Single bit-flip mutation

Experimental Setup

- Hardware
  - Intel Core i7 2.8GHz (quad-core), 4GM RAM
  - ATI Radeon HD4850 (not the most powerful card right now, $100)
- Software
  - OS X 10.6.5 (Darwin Kernel 10.6.0 x64)
  - JRE 1.6.0_22

Subjects

Program	Program Size (LOC)	Test Suite Size
printtokens	188	315-319
		4,130
printtokens2	199	4,115
schedule	142	224-227
		2,650
schedule2	142	2,710
tcas	65	1,608
totinfo	124	1,052
replace	242	5,545
space	3,268	154-160
flex	3,956	103
gzip	2,007	213
sed	1,789	370
bash	6,167	1,061
ibm	61,770	181

- Both smaller test suites and the entire test input in SIR were considered for printtokens and schedule
- IBM middleware size represents functional checkpoints

Table 1: Speed–up results for NSGA-II

Subject	S_{JOMP1}	S_{JOMP2}	S_{JOMP4}	S_{GPU}
`printtokens-1`	0.83	1.21	1.54	2.14
`printtokens-2`	0.83	1.23	1.56	2.20
`printtokens-3`	0.82	1.21	1.53	2.13
`printtokens-4`	0.84	1.22	1.54	2.19
`schedule-1`	0.97	1.22	1.40	1.56
`schedule-2`	0.96	1.22	1.41	1.46
`schedule-3`	0.96	1.22	1.39	1.45
`schedule-4`	0.95	1.20	1.37	1.43
`printtokens`	0.76	1.24	1.44	4.52
`schedule`	0.69	1.08	1.26	3.38
`printtokens2`	0.72	1.18	1.37	4.38
`schedule2`	0.71	1.09	1.27	3.09
`tcas`	0.84	1.10	1.30	1.94
`totinfo`	0.90	1.28	1.61	2.50
`flex`	1.58	2.76	4.19	6.82
`gzip`	1.19	2.15	3.31	8.00
`sed`	1.02	1.87	3.04	10.28
`space-1`	1.77	3.22	5.10	10.51
`space-2`	1.86	3.34	5.19	10.88
`space-3`	1.80	3.27	5.16	10.63
`space-4`	1.76	3.25	5.12	10.54
`replace`	0.73	1.23	1.44	5.26
`bash`	1.54	2.90	4.87	25.09
`haifa`	3.01	5.55	9.04	24.85

Table 1: Speed–up results for NSGA-II

Subject	S_{JOMP1}	S_{JOMP2}	S_{JOMP4}	S_{GPU}
`printtokens-1`	0.83	1.21	1.54	2.14
`printtokens-2`	0.83	1.23	1.56	2.20
`printtokens-3`	0.82	1.21	1.53	2.13
`printtokens-4`	0.84	1.22	1.54	2.19
`schedule-1`	0.97	1.22	1.40	1.56
`schedule-2`	0.96	1.22	1.41	1.46
`schedule-3`	0.96	1.22	1.39	1.45
`schedule-4`	0.95	1.20	1.37	1.43
`printtokens`	0.76	1.24	1.44	4.52
`schedule`	0.69	1.08	1.26	3.38
`printtokens2`	0.72	1.18	1.37	4.38
`schedule2`	0.71	1.09	1.27	3.09
`tcas`	0.84	1.10	1.30	1.94
`totinfo`	0.90	1.28	1.61	2.50
`flex`	1.58	2.76	4.19	6.82
`gzip`	1.19	2.15	3.31	8.00
`sed`	1.02	1.87	3.04	10.28
`space-1`	1.77	3.22	5.10	10.51
`space-2`	1.86	3.34	5.19	10.88
`space-3`	1.80	3.27	5.16	10.63
`space-4`	1.76	3.25	5.12	10.54
`replace`	0.73	1.23	1.44	5.26
`bash`	1.54	2.90	4.87	25.09
`haifa`	3.01	5.55	9.04	24.85

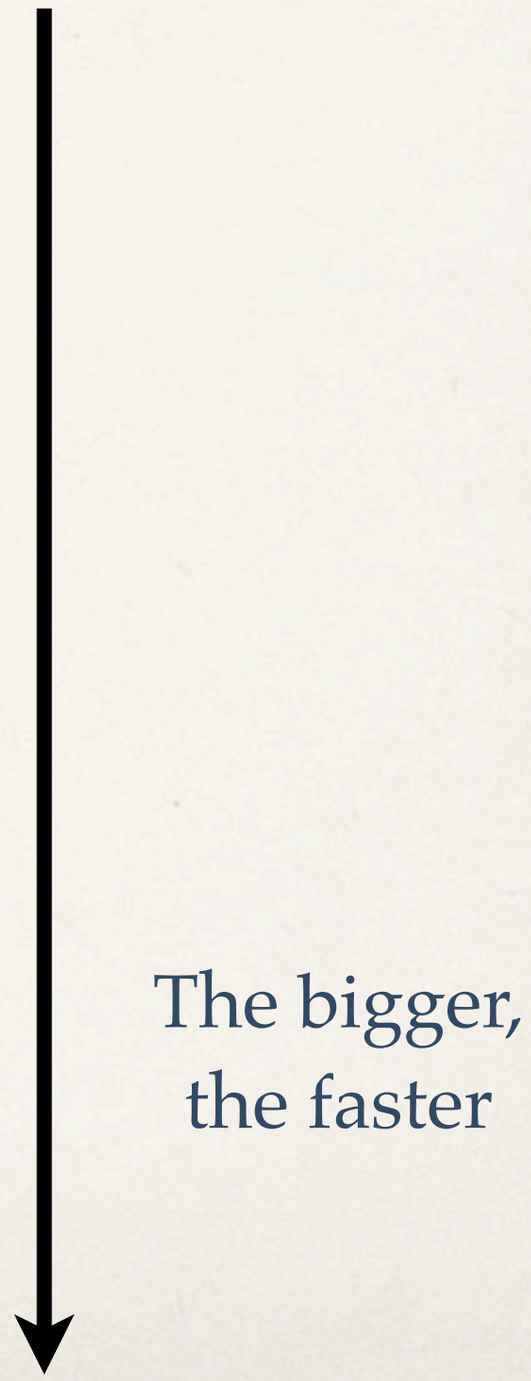


Table 1: Regression Analysis for NSGA-II

Config	Model	α	β	γ	R^2
JOMP1	$S_p \sim z$	1.56e-07	-	1.00e+00	0.4894
	$S_p \sim \log z$	2.01e-01	-	-1.34e+00	0.3423
	$S_p \sim l + m$	3.27e-05	-1.13e-04	1.17e+00	0.7060
	$S_p \sim \log l + m$	2.69e-01	-4.83e-05	-4.79e-01	0.8487
	$S_p \sim l + \log m$	3.12e-05	-1.78e-01	2.15e+00	0.7600
	$S_p \sim \log l + \log m$	2.62e-01	-6.83e-02	-6.15e-02	0.8509
JOMP2	$S_p \sim z$	3.24e-07	-	1.58e+00	0.5009
	$S_p \sim \log z$	4.78e-01	-	-4.05e+00	0.4606
	$S_p \sim l + m$	6.64e-05	-1.82e-04	1.87e+00	0.6367
	$S_p \sim \log l + m$	6.00e-01	-2.84e-05	-1.83e+00	0.9084
	$S_p \sim l + \log m$	6.35e-05	-3.07e-01	3.58e+00	0.6836
	$S_p \sim \log l + \log m$	5.96e-01	-4.04e-02	-1.59e+00	0.9086
JOMP4	$S_p \sim z$	5.80e-07	-	2.15e+00	0.5045
	$S_p \sim \log z$	8.72e-01	-	-8.13e+00	0.4814
	$S_p \sim l + m$	1.16e-04	-3.42e-04	2.70e+00	0.6199
	$S_p \sim \log l + m$	1.08e+00	-5.93e-05	-4.00e+00	0.9322
	$S_p \sim l + \log m$	1.11e-04	-5.49e-01	5.74e+00	0.6611
	$S_p \sim \log l + \log m$	1.08e+00	-5.50e-02	-3.72e+00	0.9313
GPU	$S_p \sim z$	2.25e-06	-	4.13e+00	0.7261
	$S_p \sim \log z$	3.45e+00	-	-3.66e+01	0.7178
	$S_p \sim l + m$	3.62e-04	-1.63e-04	5.33e+00	0.4685
	$S_p \sim \log l + m$	3.53e+00	7.79e-04	-1.66e+01	0.8219
	$S_p \sim l + \log m$	3.62e-04	-1.34e-01	5.98e+00	0.4676
	$S_p \sim \log l + \log m$	3.85e+00	1.69e+00	-2.82e+01	0.8713

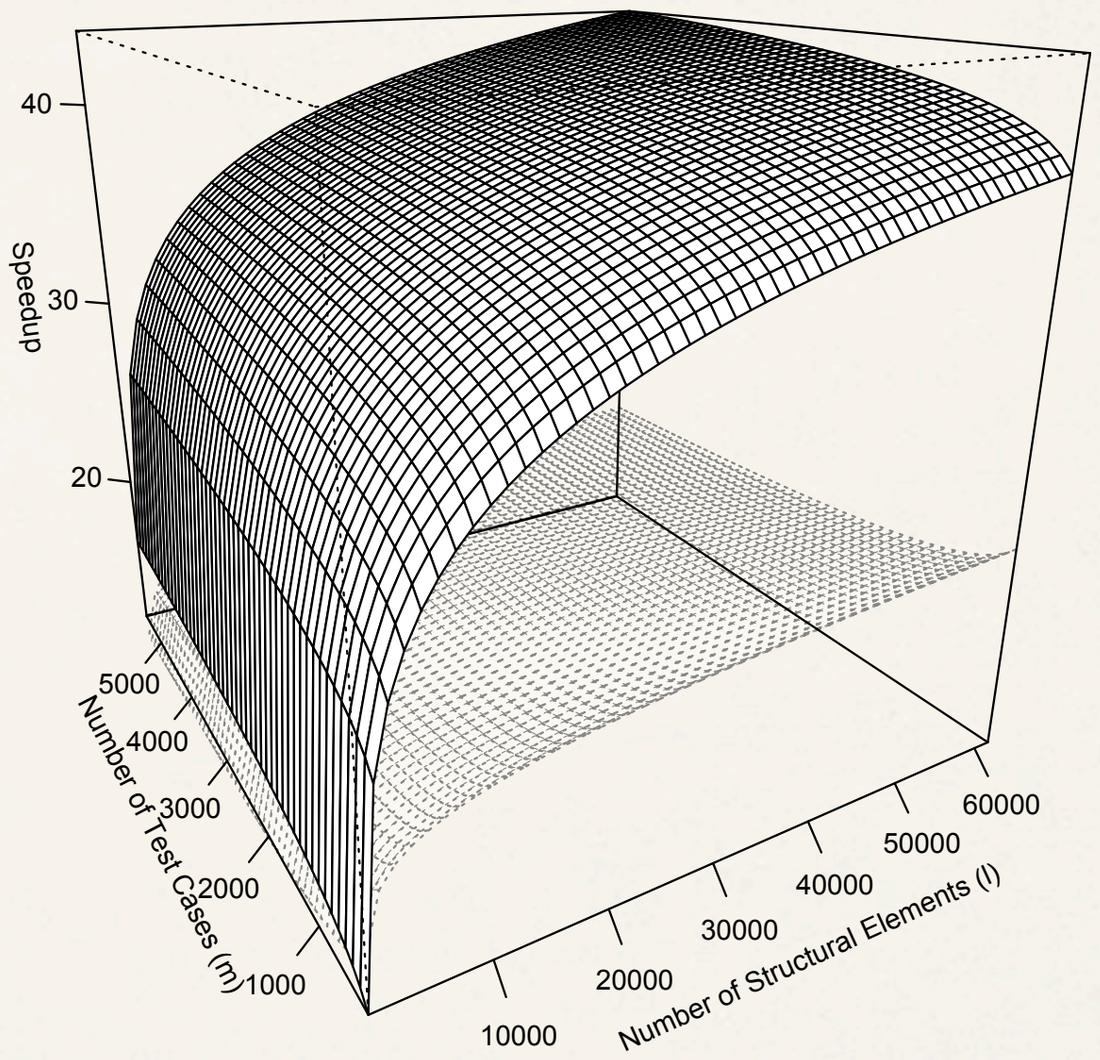
l: program size

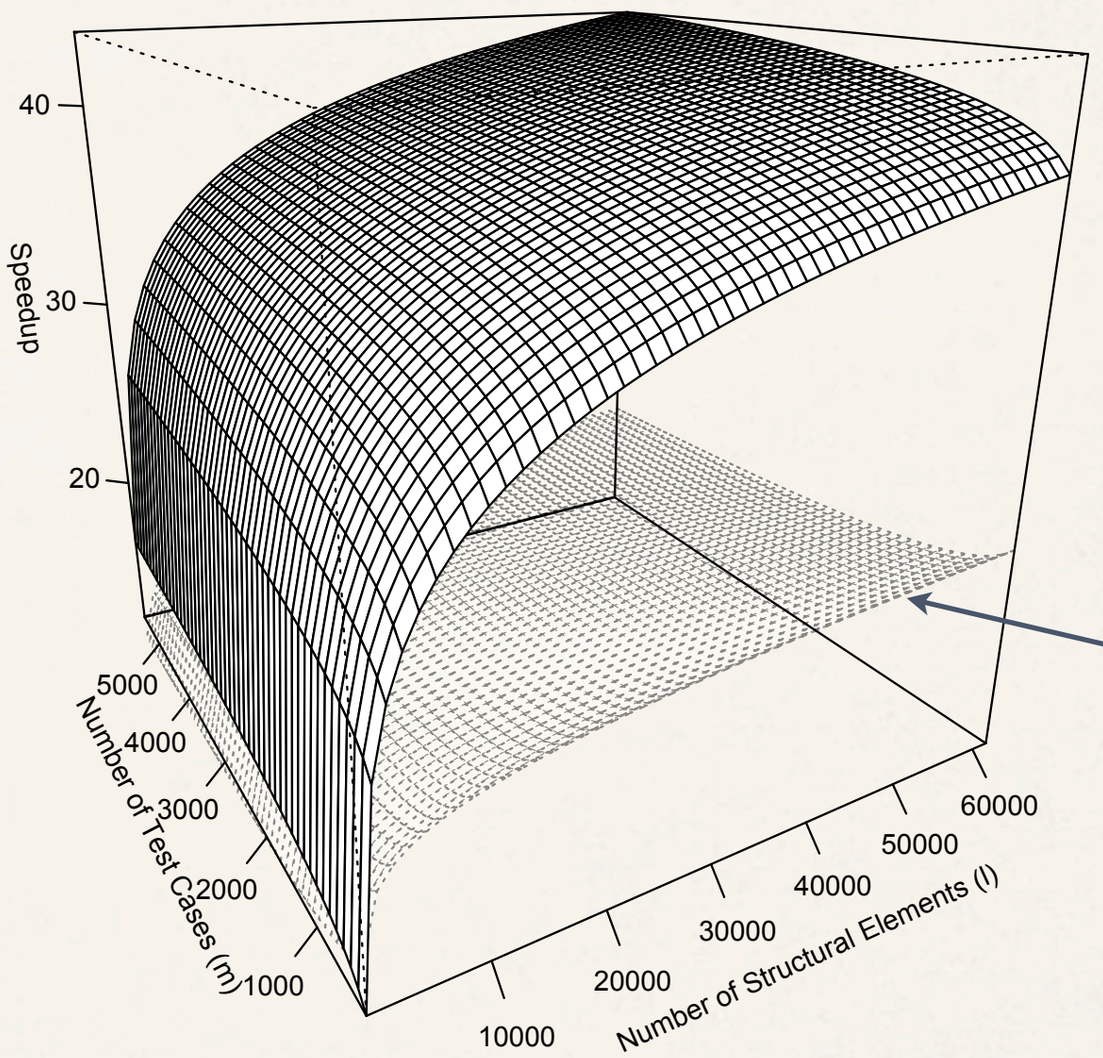
m: test suite size

z: l * m

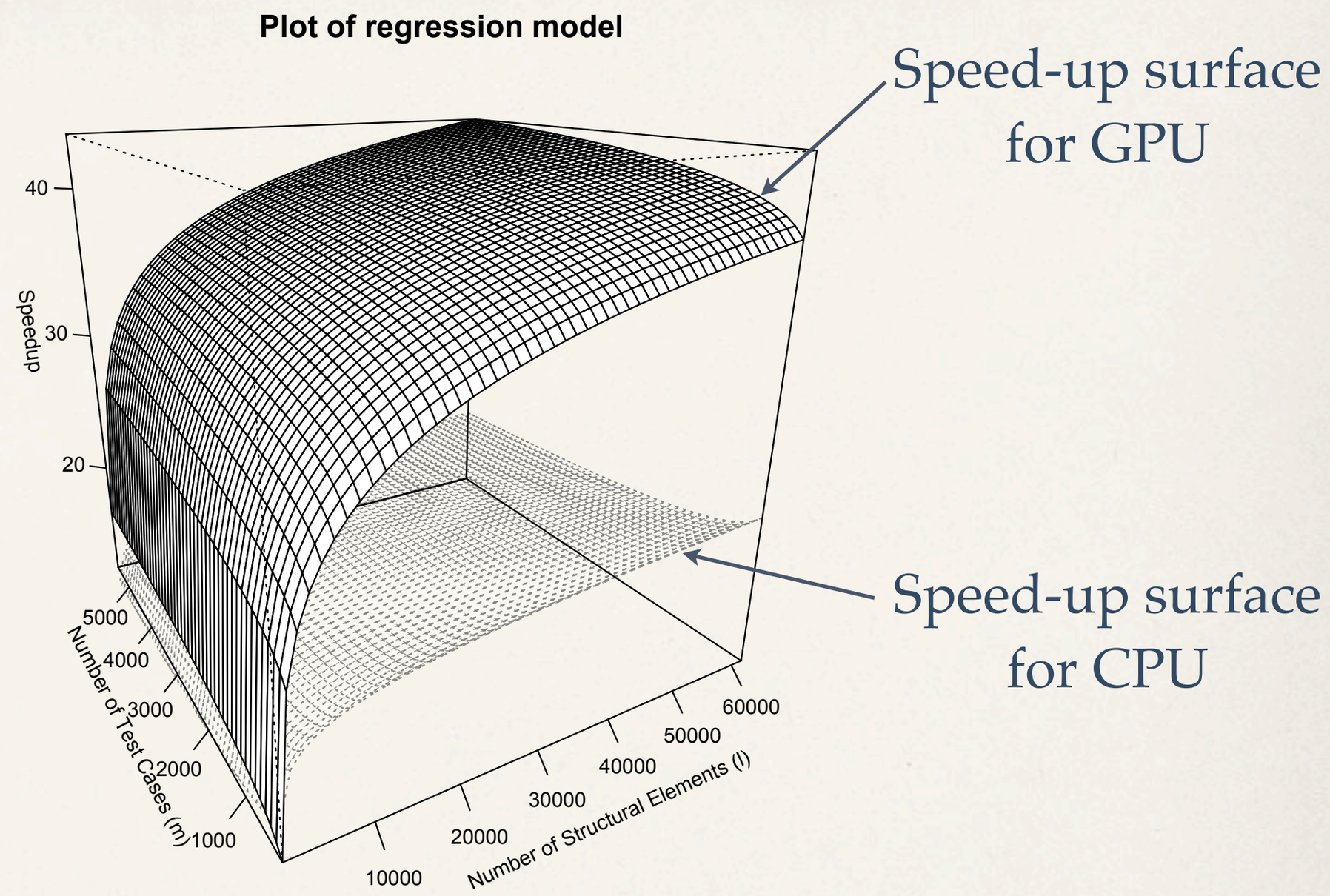
best fit: logarithmic

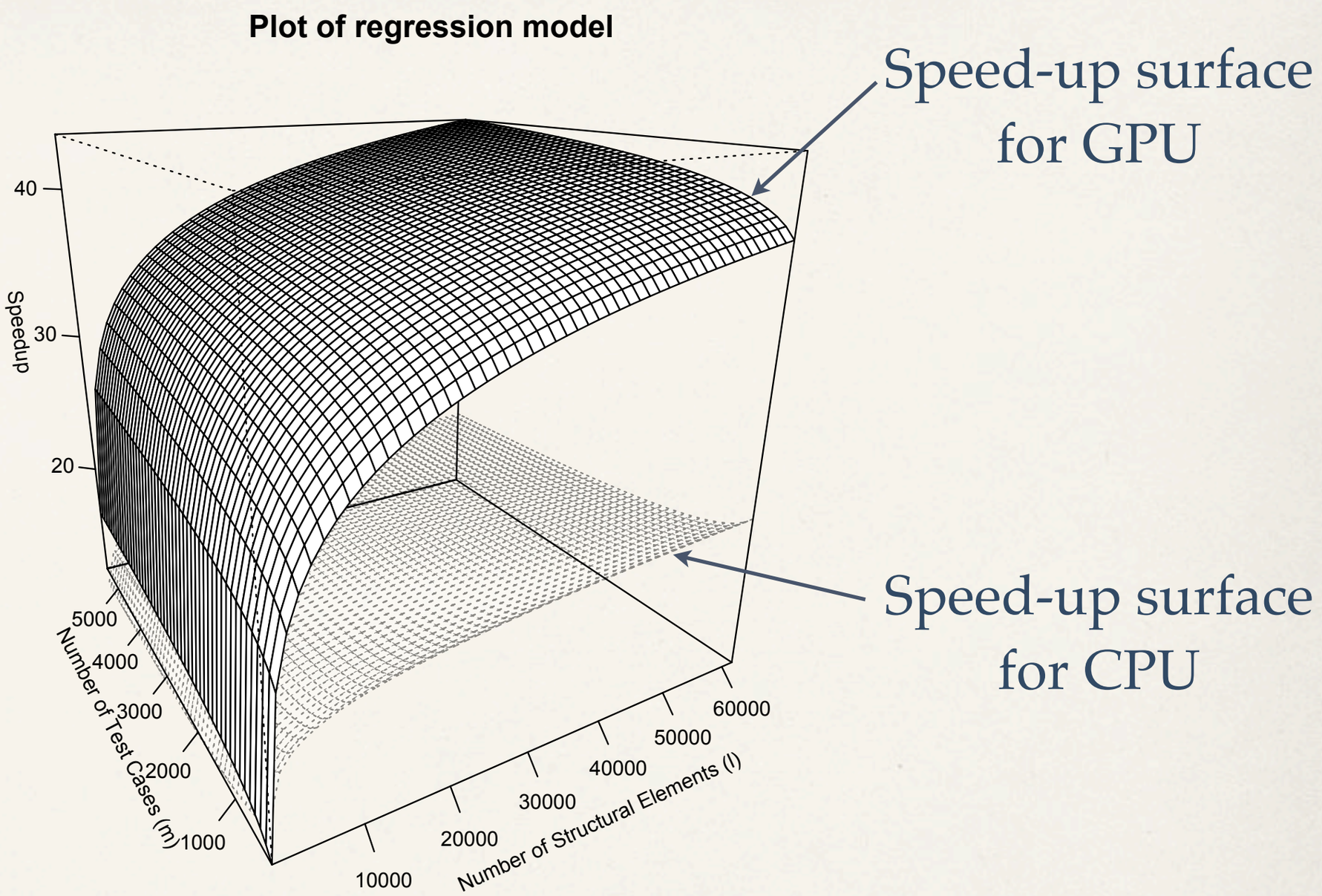
Plot of regression model





Speed-up surface for CPU


Plot of regression model
Speed-up surface for GPU
Speed-up surface for CPU
Speedup
40
30
20
Number of Test Cases (m)
5000
4000
3000
2000
1000
Number of Structural Elements (l)
10000
20000
30000
40000
50000
60000



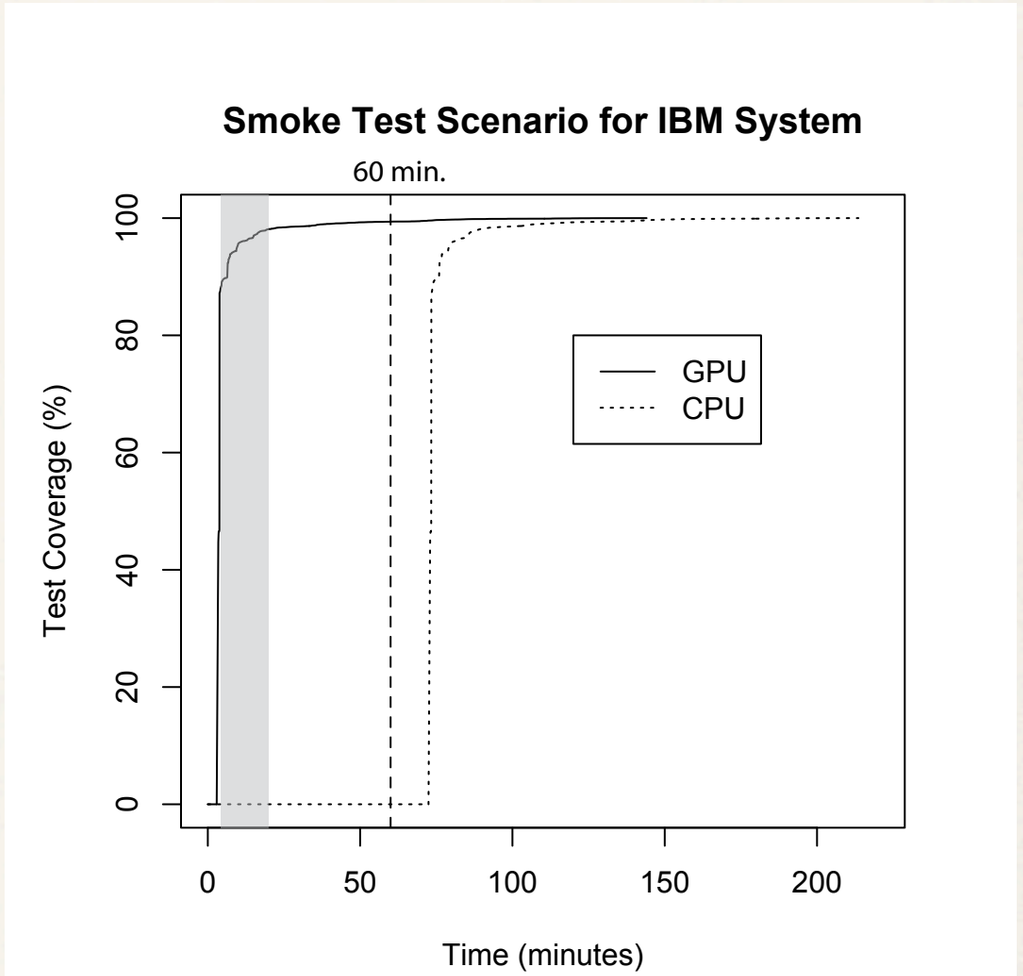
$$Speedup = 3.85 \log(l) + 1.69 \log(m) - 28.2$$

Benefits of Speed-up

- If optimisation itself takes too long, you may not meet the submission deadline

Benefits of Speed-up

- If optimisation itself takes too long, real world usage can be restricted.



Lessons Learned

- It is EASY: no need to switch languages, etc: many excellent wrappers out there.
- It is powerful: the language is fully expressive.
- It is available: most of you already have a machine that can achieve massive parallelism.

Lessons Learned

- You do need to care about a bit of low-level details: for example, endian-ness or hardware characteristics
- You need to be careful with memory
- Debugging can be hard: low observability
  - Maybe an interesting SBST problem?

- GPGPU is Easy
- GPGPU does Work
- GPGPU is Available
- http://www.macresearch.org/opencl