Noname manuscript No.
(will be inserted by the editor)

GPGPU Test Suite Minimisation: Search Based Software Engineering Performance Improvement Using Graphics Cards

Shin Yoo · Mark Harman · Shmuel Ur

the date of receipt and acceptance should be inserted later

Abstract It has often been claimed that SBSE uses so-called 'embarrassingly parallel' algorithms that will imbue SBSE applications with easy routes to dramatic performance improvements. However, despite recent advances in multicore computation, this claim remains largely theoretical; there are few reports of performance improvements using multicore SBSE. This paper shows how inexpensive General Purpose computing on Graphical Processing Units (GPGPU) can be used to massively parallelise suitably adapted SBSE algorithms, thereby making progress towards cheap, easy and useful SBSE parallelism. The paper presents results for three different algorithms: NSGA2, SPEA2, and the Two Archive Evolutionary Algorithm, all three of which are adapted for multi-objective regression test selection and minimization. The results show that all three algorithms achieved performance improvements up to 25 times, using widely available standard GPUs. We also found that the speed-up was observed to be statistically strongly correlated to the size of the problem instance; as the problem gets harder the performance improvements also get better.

1 Introduction

Search Based Software Engineering (SBSE) is a promising sub-area within Software Engineering that reformulates Software Engineering problems as search-based optimisation problems [12, 22, 24]. There has been much recent interest in SBSE, with over 800 papers on the topic and several recent surveys [2, 3, 26, 42].

One weakness shared by many SBSE techniques is their significant execution time. It is an inherent drawback because many meta-heuristic optimisation algorithms are designed in such a way that they evaluate a large number of potential candidates to arrive at the set of proposed solutions. This process is often computationally demanding and can render an SBSE technique infeasible in practice, because it would simply require too much time to optimise for complex real-world SE problems. Lack

Shin Yoo
University College London E-mail: shin.yoo@ucl.ac.uk

Mark Harman
University College London E-mail: Mark.Harman@ucl.ac.uk

Shmuel Ur
University of Bristol E-mail: shmuel.ur@gmail.com

of scalability has been shown to be an important barrier to wider uptake of Software Engineering research [9,13,41].

Fortunately, many of the algorithms used in SBSE, such as the most widely used evolutionary algorithms [26], are classified as 'embarrassingly parallel' due to their inherent potential for parallelism. The high computational requirement created by the necessity to examine many solutions does not necessitate long elapsed time as the examinations can be done in parallel. The computation process (fitness evaluation) for each candidate solution is identical, thereby making the overall process well-suited to Single Instruction Multiple Data (SIMD) parallelism.

The use of multicore computing is rapidly becoming commonplace, with very widely available and inexpensive platforms that offer several hundreds of processing elements that implement SIMD parallelism. Furthermore, it is likely that we shall see significant advances in such platforms in the near future, with thousands and perhaps tens of thousands of simple processing elements becoming available within the reach of 'standard' desktop computation.

Many SBSE fitness functions are ideally-suited to such simple processing elements. However, there has been little work on multicore SBSE. The first authors to suggest the use of multicore SBSE were Mitchell et al. [36] who used a distributed architecture to parallelise modularisation through the application of search-based clustering. Subsequently, Mahdavi et al. [34] used a cluster of standard PCs to implement a parallel hill climbing algorithm. More recently, Asadi et al. [4] used a distributed architecture to parallelise a genetic algorithm for the concept location problem. However, hitherto, no authors[1] have used General Purpose computing on Graphical Processing Units (GPGPU) for SBSE.

In this paper we propose GPGPU SBSE; the use of GPGPU devices to achieve multicore execution of SBSE, using simple fitness computations mapped across the multiple processing elements of standard GPGPU architectures. We report results for the application of GPGPU SBSE to the multi-objective regression test selection problem (also known as test case minimization).

In order to apply GPGPU to Software Engineering optimisation problems, the specialized graphics manipulation hardware needs to be harnessed for a purpose for which it was not originally designed. Fortunately, the recent trend towards 'general purpose' graphics processing units lends itself well to this task. The underlying firmware typically implements certain matrix manipulation operations very efficiently. The key to unlocking the potential of GPGPU therefore lies in the ability to reformulate the optimisation problem in terms of these specific matrix manipulation operations.

This paper focusses on the problem of Search Based Regression Testing, which is one problem in the general area of Search Based Software Testing. Regression Testing is concerned with the process of re–testing software after change. After each change to the system, the pool of available test data needs to be re-executed in order to check whether change has introduced new faults. Regression Testing therefore seeks to answer the question 'has the software regressed?'. There have been several survey papers on Regression Testing applications and techniques that provide a more detailed treatment [19, 27,55].

In search based regression testing, the goal is to use search based optimisation algorithms to find optimal sets of test cases (regression test suite minimisation [54]) or to order test cases for regression testing (regression test prioritisation [33, 49]). This paper concentrates upon the former problem of regression test minimisation. Recent results have shown that this is a promising area of SBSE application; the results obtained from the SBSE algorithms have been shown to be human competitive [47].

Fast regression test minimisation is an important problem for practical software testers, particularly where large volumes of testing are required on a tight build schedule. For instance, the IBM middleware product used as one of the systems in the empirical study in this paper is a case in point.

[1] This paper is an extended version of our SSBSE 2011 paper [58], which was the first to propose GPGPU SBSE.

While it takes over four hours to execute the entire test suite for this system, the typical smoke test scenario performed after each code submit is assigned only an hour or less of testing time, forcing the tester to select a subset of tests from the available pool. A multi-objective approach to test suite minimisation [54] provides an ideal solution as it can recommend subsets of tests that can be executed within different time budgets. However, as the selection of tests in the smoke tests is not static and depends on the code submitted, the given time budget should account for both the computation involved in test suite minimisation and for running the tests. Therefore it is important that test suite optimization will be done in a small fraction of the time, thereby allowing sophisticated minimisation to be used on standard machines.

The paper modifies three widely used evolutionary algorithms (SPEA2, NSGA2 and the Two Archive Algorithm) for the multi-objective regression test minimisation problem. The algorithms are modified to support implementation on a GPU by transforming the fitness evaluation of the population of individual solutions into a matrix-multiplication problem, which is inherently parallel and renders itself very favourably to the GPGPU approach. This transformation to matrix-multiplication is entirely straightforward and may well be applicable to other SBSE problems, allowing them to benefit from similar scale-ups to those reported in this paper.

The modified algorithms have been implemented using `OpenCL` technology, a framework for GPGPU. The paper reports the results of the application of the parallelised GPGPU algorithms on 13 real-world programs, including widely studied, but relatively small examples from the Siemens' suite [29], through larger more realistic real-world examples from the Software-Infrastructure Repository (SIR) for testing [14], and on a very large IBM middleware regression testing problem.

The primary contributions of the paper are as follows:

1. The paper presents results for real-world instances of the multi-objective test suite minimisation problem. The results indicate that dramatic speed–up is achievable. For the systems used in the empirical study, speed–ups over 25`x` were observed. The empirical evidence suggests that, for larger problems where the scale-up is the most needed, the degree of speed–up is the most dramatic; a problem that takes over an hour using conventional techniques, can be solved in minutes using the GPGPU approach. This has important practical ramifications because regression testing cycles are often compressed: overnight build cycles are not uncommon.
2. The paper studies three different multi-objective evolutionary algorithms based on both GPU- and CPU-based parallelisation methods to provide robust empirical evidence for the scalability conferred by the use of GPGPU. The GPGPU parallelisation technique maintained the same level of speed–up across all algorithms studied. The empirical evidence highlights the limitations of CPU-based parallelisation: with smaller problems, multi threading overheads erode the speed–up, whereas with larger problems it fails to scale as well as GPU-based parallelisation.
3. The paper explores the factors that influence the degree of speed–up achieved, revealing that both program size and test suite size are closely correlated to the degree of speed–up achieved. The data have a good fit to a model for which increases in the degree of scale-up achieved are logarithmic in both program and test suite size.

The rest of the paper is organised as follows. Section 2 presents background material on test suite minimisation and GPGPU-based evolutionary computation. Section 3 describes how the test suite minimisation problem is re-formulated for a parallel algorithm, which is described in detail in Section 4. Section 5 describes the details of the empirical study, the results of which are analysed in Section 6. Section 7 discusses threats to validity and Section 8 presents the related work. Section 9 concludes.

2 Background

Multi-Objective Test Suite Minimisation: The need for test suite minimisation arises when the regression test suite of an existing software system grows to such an extent that it may no longer be feasible to execute the entire test suite [44]. In order to reduce the size of the test suite, any *redundant* test cases in the test suite need to be identified and removed. One widely accepted criterion for redundancy is defined in relation to the coverage achieved by test cases [5, 45]. If the test coverage achieved by test case t_1 is a subset of the test coverage achieved by test case t_2, it can be said that the execution of t_1 is redundant as long as t_2 is also executed. The aim of test suite minimisation is to obtain the smallest subset of test cases that are not redundant with respect to a set of test requirements. More formally, test suite minimisation problem can be defined as follows [55]:

Test Suite Minimisation Problem

Given: A test suite of m tests, T, a set of l test requirements $\{r_1, \ldots, r_l\}$, that must be satisfied to provide the desired 'adequate' testing of the program, and subsets of T, T_is, one associated with each of the r_is such that any one of the test cases t_j belonging to T_i can be used to achieve requirement r_i.

Problem: Find a minimal representative set[2], T', of test cases from T that satisfies all r_is.

The testing criterion is satisfied when every test-case requirement in $\{r_1, \ldots, r_l\}$ is satisfied. A test-case requirement, r_i, is satisfied by any test case, t_j, that belongs to T_i, a subset of T. If we represent test cases as vertices of a bipartite graph on the left side, and the requirements on the right side, and the satisfiability relationship as edges between two sides, the minimal representative set of test cases is the hitting set of T_is (i.e. the subset of vertices on the left, the union of whose connected right side vertices equals the set of all requirements). Furthermore, in order to maximise the effect of minimisation, T' should be the minimal hitting set of T_is. The minimal representative set problem is an NP-complete problem as is the dual problem of the minimal set cover problem [20].

The NP-hardness of the problem encouraged the use of heuristics and meta-heuristics. The greedy approach [38] as well as other heuristics for minimal hitting set and set cover problem [10, 28] have been applied to test suite minimisation but these approaches were not cost-cognisant and only dealt with a single objective (test coverage). With the single-objective problem formulation, the solution to the test suite minimisation problem is one subset of test cases that maximises the test coverage with minimum redundancy.

Later, the problem was reformulated as a multi-objective optimisation problem [54]. With the multi-objective problem formulation, the solution to the test suite minimisation problem is not just a single solution but a set of non-dominated, Pareto-efficient solutions. This set of solutions reveals the trade-off between test coverage and the cost of testing that is specific to the test suite in consideration. For example, with the solution to the multi-objective test suite minimisation problem, it is possible not only to know what the minimal subset that achieves the maximum test coverage is, but also to know how much test coverage is possible for any given testing budget.

Since the greedy algorithm may not always produce Pareto optimal solutions for multi-objective test suite minimisation problems, Multi-Objective Evolutionary Algorithms (MOEAs) have been applied [35, 54]. While this paper studies three selected MOEAs, the principle of parallelising fitness evaluation of multiple solutions in the population of an MOEA applies universally to any MOEA.

[2] Given a universe (in our context, all test requirements), a representative set is the set of subsets of universe (in our context, subsets of test requirements achieved by different tests) that whose union is equal to the universe.

GPGPU and Evolutionary Algorithms: Graphics cards have become a compelling platform for intensive computation, with a set of resource-hungry graphic manipulation problems that have driven the rapid advances in their performance and programmability [39]. As a result, consumer-level graphics cards boast tremendous memory bandwidth and computational power. For example, ATI Radeon HD4850 (the graphics card used in the empirical study in the paper), costing about $150 as of April 2010, provides 1000GFlops processing rate and 63.6GB/s memory bandwidth. Graphics cards are also becoming faster more quickly compared to CPUs. In general, it has been reported that the computational capabilities of graphics cards, measured by metrics of graphics performance, have compounded at the average yearly rate of 1.7x (rendered pixels/s) to 2.3x (rendered vertices/s) [39]. This significantly outperforms the growth in traditional microprocessors; using the SPEC benchmark, the yearly rate of growth for CPU performance has been measured at 1.4x by a recent survey [17].

The disparity between the two platforms is caused by the different architecture. CPUs are optimised for executing sequential code, whereas GPUs are optimised for executing the same instruction (the graphics shader) with data parallelism (different objects on the screen). This Single Instruction Multiple Data (SIMD) architecture facilitates hardware-controlled massive data parallelism, which results in the higher performance for certain types of problems in which a large dataset has to be submitted to the same operations.

It is precisely this massive data-parallelism of General-Purpose computing on Graphics Processing Units (GPGPU) that makes GPGPU as an ideal platform for parallel evolutionary algorithms. Many of these algorithms require the calculation of fitness (single instruction) for multiple individual solutions in the population pool (multiple data). Early work has exploited this potential for parallelism with both single- and multi-objective evolutionary algorithms [48,50,51]. However, most existing evaluation has been performed on benchmark problems rather than practical applications.

3 Parallel Formulation of MOEA for Test Suite Minimisation

Parallel Fitness Evaluation: The paper considers, for parallelisation, a multi-objective test suite minimisation problem from existing work [54]. In order to parallelise test suite minimisation, the fitness evaluation of a generation of individual solutions for the test suite minimisation problem is reformulated as a matrix multiplication problem. Instead of computing the two objectives (i.e. coverage of test requirements and execution cost) for each individual solution, the solutions in the entire population are represented as a matrix, which in turn is multiplied by another matrix that represents the trace data of the entire test suite. The result is a matrix that contains information for both test goal coverage and execution cost. While the paper considers structural coverage as the test goal, the proposed approach is equally applicable to any other testing criteria, either coverage generated such as data-flow coverage and functional coverage or even those generated manually, provided that there is a clear mapping between tests and the test requirements they achieve.

More formally, let matrix A contain the trace data that capture the test requirements achieved by each test; the number of rows of A equals the number of test requirements to be covered, l, and the number of columns of A equals the number of test cases in the test suite, m. Entry $a_{i,j}$ of A stores 1 if the test goal f_i was executed (i.e. covered) by test case t_j, 0 otherwise.

$$A = \begin{pmatrix} a_{1,1} \cdots a_{1,m} \\ a_{2,1} \cdots a_{2,m} \\ \cdots \\ a_{l,1} \cdots a_{l,m} \end{pmatrix}$$

The multiplier matrix, B, is a representation of the current population of individual solutions that are being considered by a given MOEA. Let B be an m-by-n matrix, where n is the size of population for the given MOEA. Entry $b_{j,k}$ of B stores 1 if test case t_j is selected by the individual p_k, 0 otherwise. In other words, each column in matrix B corresponds to a vector of decision variables that denote the selected test cases.

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

The fitness evaluation of the entire generation is performed by the matrix multiplication of $C = A \times B$. Matrix C is a l-by-n matrix; entry $c_{i,k}$ of C denotes the number of times test goal f_i was covered by different test cases that had been selected by the individual p_k.

Cost and Coverage In order to incorporate the execution cost as an additional objective to the MOEA, the basic reformulation is extended with an extra row in matrix A. The new matrix, A', is an $l+1$ by m matrix that contains the cost of each individual test case in the last row. The extra row in A' results in an additional row in C' which equals to $A' \times B$ as follows:

$$A' = \begin{pmatrix} a_{1,1} & \dots & a_{1,m} \\ a_{2,1} & \dots & a_{2,m} \\ & \dots & \\ a_{l,1} & \dots & a_{l,m} \\ cost(t_1) & \dots & cost(t_m) \end{pmatrix} \qquad C' = \begin{pmatrix} c_{1,1} & \dots & c_{1,n} \\ c_{2,1} & \dots & c_{2,n} \\ & \dots & \\ c_{l,1} & \dots & c_{l,n} \\ cost(p_1) & \dots & cost(p_n) \end{pmatrix}$$

By definition, an entry $c_{l+1,k}$ in the last row in C' is defined as $c_{l+1,k} = \sum_{j=1}^{m} a_{l+1,j} \cdot b_{j,k} = \sum_{j=1}^{m} cost(t_j) \cdot b_{j,k}$. That is, $c_{l+1,k}$ equals the sum of costs of all test cases *selected* ($b_{j,k}$ equals 1) by the k-th individual solution p_k, i.e. $cost(p_k)$. Similarly, after the multiplication, the k-th column of matrix C' contains the coverage of test requirements achieved by individual solution p_k. However, this information needs to be summarised into a percentage coverage, using a step function f as follows: $coverage(p_k) = \frac{\sum_{i=1}^{m} f(c_{i,k})}{l}$, $f(x) = 1$ $(x > 0)$ or 0 (otherwise). The role of the step function is to translate the linear sum of how many times a test goal has been covered into boolean coverage of whether it was covered or not.

The cost objective can be calculated as a part of the matrix multiplication because it can be linearly computed from the decision variable vectors (columns of matrix B). Other objectives that share the linearity may also be computed using matrix multiplication. However, the coverage of test requirements requires a separate step to be performed. Each column of C' contains the number of times individual testing requirements were *covered* by the corresponding solution; in order to calculate the coverage metric for a solution, it is required to iterate over the corresponding column of C'. The coverage calculation is highly parallel in nature because each column can be independently iterated over and, therefore, can take the advantage of GPGPU architecture by running multiple threads.

4 Algorithms

This section presents the parallel fitness evaluation components for CPU and GPU and introduces the MOEAs that are used in the paper.

Parallel Matrix Multiplication Algorithm: Matrix multiplication is inherently parallelisable as the calculation for an individual entry of the product matrix does not depend on the calculation of any

Algorithm 1: Parallel Matrix Multiplication
Input: The thread id, *tid*; arrays containing $l+1$ by m and m by n matrices, A and B; the width of matrix A' and B, $w_{A'}$ and w_B
Output: An array of length $(l+1)n$ to store matrix C'

```
MatMult($tid$, $A'$, $B$, $w_{A'}$, $w_B$)
(1)  $x \leftarrow tid \mod w_B$
(2)  $y \leftarrow \lfloor tid/w_B \rfloor$
(3)  $v \leftarrow 0$
(4)  for $k = 0$ to $w_{A'} - 1$
(5)      $v \leftarrow v + A'[y \cdot w_{A'} + k] \cdot B[k \cdot w_B + x]$
(6)  $C'[y * w_B + x] \leftarrow v$
```

Algorithm 2: Parallel Coverage Collection
Input: The thread id, *tid*; an array containing the result of matrix-multiplication, C'; the width of matrix A', $w_{A'}$ and the height of matrix A', $h_{A'}$
Output: An array containing the coverage achieved by each individual solution, *coverage*

```
CollectCoverage($tid$, $C'$, $w_{A'}$, $h_{A'}$)
(1)  $e \leftarrow 0$
(2)  for $k = 0$ to $h_{A'} - 1$
(3)      if $C'[k \cdot w_{B'} + tid] > 0$ then $e \leftarrow e + 1$
(4)  $coverage[tid] \leftarrow e/(h_{A'} - 1)$
```

other entry. Algorithm 1 shows the pseudo-code of the parallel matrix multiplication algorithm using the matrix notation in Section 3.

Algorithm 1 uses one thread per element of matrix C', resulting in a total of $(l+1) \cdot n$ threads. Each thread is identified with unique thread id, *tid*. Given a thread id, Algorithm 1 calculates the corresponding element of the resulting matrix, $C'_{y,x}$ given the width of matrix A, w_A, i.e. $y = \frac{tid}{w_B}$ and $x = tid \mod w_B$.

Coverage Collection Algorithm: After matrix-multiplication using Algorithm 1, coverage information is collected using a separate algorithm, pseudo-code of which is shown in Algorithm 2. Unlike Algorithm 1, the coverage collection algorithm only requires n threads, i.e. one thread per column in matrix C'.

The loop in Line (2) and (3) counts the number of structural elements that have been executed by the individual solution p_{tid}. The coverage is calculated by dividing this number by the total number of structural elements that need to be covered.

While coverage information requires a separate collection phase, the sum of costs for each individual solution has been calculated by Algorithm 1 as a part of the matrix multiplication following the extension in Section 3.

5 Experimental Setup

5.1 Research Questions

This section presents the research questions studied in the paper. **RQ1** and **RQ2** concern the scalability achieved by the speed-up through the use of GPGPU, whereas **RQ3** concerns the practical implications of the speed-up and the consequent scalability to the practitioners.

RQ1. Speed–up: what is the speed–up factor of GPU- and CPU-based parallel versions of MOEAs over the untreated CPU-based version of the same algorithms for multi-objective test suite minimisation problem?

RQ2. Correlation: what are the factors of the problem instances that have the highest correlation to the speed–up achieved, and what is the correlation between these factors and the resulting speed–up?

RQ3. Insight: what are the realistic benefits of the scalability that is achieved by the GPGPU approach to software engineers?

RQ1 is answered by observing the dynamic execution time of the parallel versions of the studied algorithms as well as the untreated single-threaded algorithms. For **RQ2**, two factors constitute the size of test suite minimisation problem: the number of test cases in the test suite and the number of test requirements in System Under Test (SUT) that need to be covered. The speed–up values measured for **RQ1** are statistically analysed to investigate the correlation between the speed–up and these two size factors. **RQ3** is answered by analysing the result of test suite minimisation obtained for a real-world testing problem.

5.2 Subjects

Table 1 shows the subject programs for the empirical study. 12 of the programs and test suites are from the Software Infrastructure Repository (SIR) [14]. In order to obtain test suites with varying sizes ranging from a few hundred to a few thousand test cases, the study includes multiple test suites for some subject programs. For `printtokens` and `schedule`, smaller test suites are coverage-adequate test suites, whereas larger test suites include all the available test cases. To avoid selection bias, four smaller test suites were randomly selected from the pool of available tests for each program. In the case of `space`, SIR contains multiple coverage-adequate test suites of similar sizes; four test suites were selected randomly.

The subjects also include a large system-level test suite from IBM. For this subject, the coverage information was maintained at the function level. The test suite contains only 181 test cases, but these test cases are used to cover 61,770 functions in the system.

Each test suite has an associated execution cost dataset. For the subject programs from SIR, the execution costs were measured by observing the number of instructions required by the execution of tests. This was performed using a well-known profiling tool, `valgrind` [37], which executes the given program on a virtual processor. For `ibm`, physical wall-clock time data, measured in seconds, were provided by IBM. The entire test suite for `ibm` takes more than 4 hours to execute.

5.3 Implementation & Hardware

Implementation: The paper uses the open source Java MOEA library, `jMetal` [15, 16] as a library of untreated versions of MOEAs: NSGA-II and SPEA2 are included in the `jMetal` library; The Two Archive algorithm has been implemented using the infrastructure provided by the library. The untreated versions of MOEAs evaluate the fitness of individual solutions in the population one at a time, which incurs method invocations regarding the retrieval of coverage and cost information.

The GPGPU-based parallel versions of these three algorithms are implemented in the OpenCL GPGPU framework using a Java wrapper called `JavaCL` [8]. The CPU-based parallel versions of the three algorithms use a parallel programming library for Java called `JOMP` [7]. `JOMP` allows parameterised configuration of the number of threads to use. The parallelisation is only applied to the fitness evaluation step because it is not clear whether certain steps in the studied algorithms, such as sorting, may yield sufficient efficiency when performed in parallel.

Table 1 Subject programs used for the empirical study.

Subject	Description	Program Size	Test Suite Size
`printtokens`	Lexical analyser	188	315-319[2]
			4,130
`schedule`	Priority scheduler	142	224-227[2]
			2,650
`printtokens2`	Lexical analyser	199	4,115
`schedule2`	Priority scheduler	142	2,710
`tcas`	Aircraft collision avoidance system	65	1,608
`totinfo`	Statistics computation utility	124	1,052
`flex`	Lexical analyser	3,965	103
`gzip`	Compression utility	2,007	213
`sed`	Stream text editor	1,789	370
`space`	Array Definition Language (ADL) interpreter	3,268	154-160[3]
`replace`	Pattern matching & substitution tool	242	5,545
`bash`	Unix shell	6,167	1,061
`ibm`	An IBM middleware system	61,770[1]	181

[1] For the IBM middleware system, the program size represents the number of functions that need to be covered. The coverage objective for the IBM system also denotes function coverage. For all other subject, the size and the coverage objective are measured and calculated using LOC.
[2] For `schedule` and `printtokens`, four coverage-adequate test suites were randomly selected from those provided by SIR, as well as the complete test suite.
[3] For `space`, four randomly selected, coverage-adequate test suites were used.

All three algorithms are configured with population size of 256 following the standard recommendation to set the number of threads to multiples of 32 or 64 [1]. The archive size for SPEA2 and The Two Archive algorithm is also set to 256. The stopping criterion for all three algorithms is to reach the maximum number of fitness evaluations, which is set to 64,000, allowing 250 generations to be evaluated.

All three algorithms solve the test suite minimisation problem by selecting Pareto-optimal subsets of test cases, represented by binary strings that form columns in matrix B in Section 3. The initial population is generated by randomly setting the individual bits of these binary strings so that the initial solutions are randomly distributed in the phenotype space.

NSGA-II and SPEA2 use the binary tournament selection operator. The Two Archive algorithm uses the uniform selection operator as described in the original paper [40]: the selection operator first selects one of the two archives with equal probability and then selects one solution from the chosen archive with uniform probability distribution. All three algorithms use the single-point crossover operator with probability of crossover set to 0.9 and the single bit-flip mutation operator with the mutation rate of $\frac{1}{n}$ where n is the length of the bit-string (i.e. the number of test requirements).

Hardware: All algorithms have been evaluated on a machine with a quad-core Intel Core i7 CPU (2.8GHz clock speed) and 4GB memory, running Mac OS X 10.6.5 with Darwin Kernel 10.6.0 for `x86_64` architecture. The Java Virtual Machine used to execute the algorithms is Java SE Runtime with version 1.6.0_22. The GPGPU-based versions of MOEAs have been evaluated on an ATI Radeon HD4850 graphics card with 800 stream processors running at 625MHz clock speed and 512MB GDDR3 onboard memory.

5.4 Evaluation

The paper compares three MOEAs, each with five different configurations: the untreated configuration (hereafter refered to `CPU`), the GPGPU configuration (`GPU`) and the `JOMP`-based parallel configurations with 1, 2, and 4 threads (`JOMP1/2/4`). The configuration with one thread (`JOMP1`) is included to observe the speed-up achieved by evaluating the fitness of the entire population using matrix multiplication, instead of evaluating the solutions one by one as in the untreated versions of MOEA. Any speed–up achieved by `JOMP1` over `CPU` is, therefore, primarily achieved by the code-level optimisation that removes the method invocation overheads. On the other hand, `JOMP1` does incur an additional thread management overhead.

In total, there are 15 different configurations (three algorithms with five configurations each). For each subject test suite, the 15 configurations were executed 30 times in order to cater for the inherent randomness in dynamic execution time, measured using the system clock. The speed-up is calculated by dividing the amount of the time that the `CPU` configuration required with the amount of the time that parallel configurations required. While we discuss the speed-up values only using the total execution time, we also provide observations of the three parts break-down of the total execution time ($Time_{total}$) for each algorithm as below:

– Initialisation ($Time_{init}$): the time it takes for the algorithm to initialise the test suite data in a usable form; for example, `GPU` configurations of MOEAs need to transfer the test suite data onto the graphics card.
– Fitness Evaluation ($Time_{fitness}$): the time it takes for the algorithm to evaluate the fitness values of different generations during its runtime.
– Remaining ($Time_{remaining}$): the remaining parts of the execution time, most of which is used for archive management, genetic operations, etc.

6 Results

This section presents the speed-up measurements of the single-threaded, CPU-based multi-threaded, and GPGPU-based multi-threaded approaches and analyses the correlation between the speed-up and problem size.

6.1 Speed–up

Figure 1 presents the mean paired speed–up results of all configurations. The mean paired speed–up values were calculated by dividing the execution time of `CPU` with the corresponding execution time of the parallel configurations for each of the 30 observations. Tables 2, 3 and 4 contain the speed–up data in more detail, whereas the statistical analysis of the raw information can be obtained from Tables 12, 13 and 14 in the appendix.

Overall, the observed paired mean speed–up ranges from 0.47`x` to 25.09`x`. While the different archive management strategies used by each MOEAs make it difficult to compare the execution time results directly (because the different amount of heap used by each may affect JVM's performance differently), it is possible to observe the general trend that the speed–up tends to increase as the problem size grows. The speed–up values below 1.0 show that the overhead of thread management and the additional communication can be detrimental for the problems of sufficiently small size. However, as the problem size grows, `JOMP1` becomes faster than `CPU` with all algorithms, indicating that the

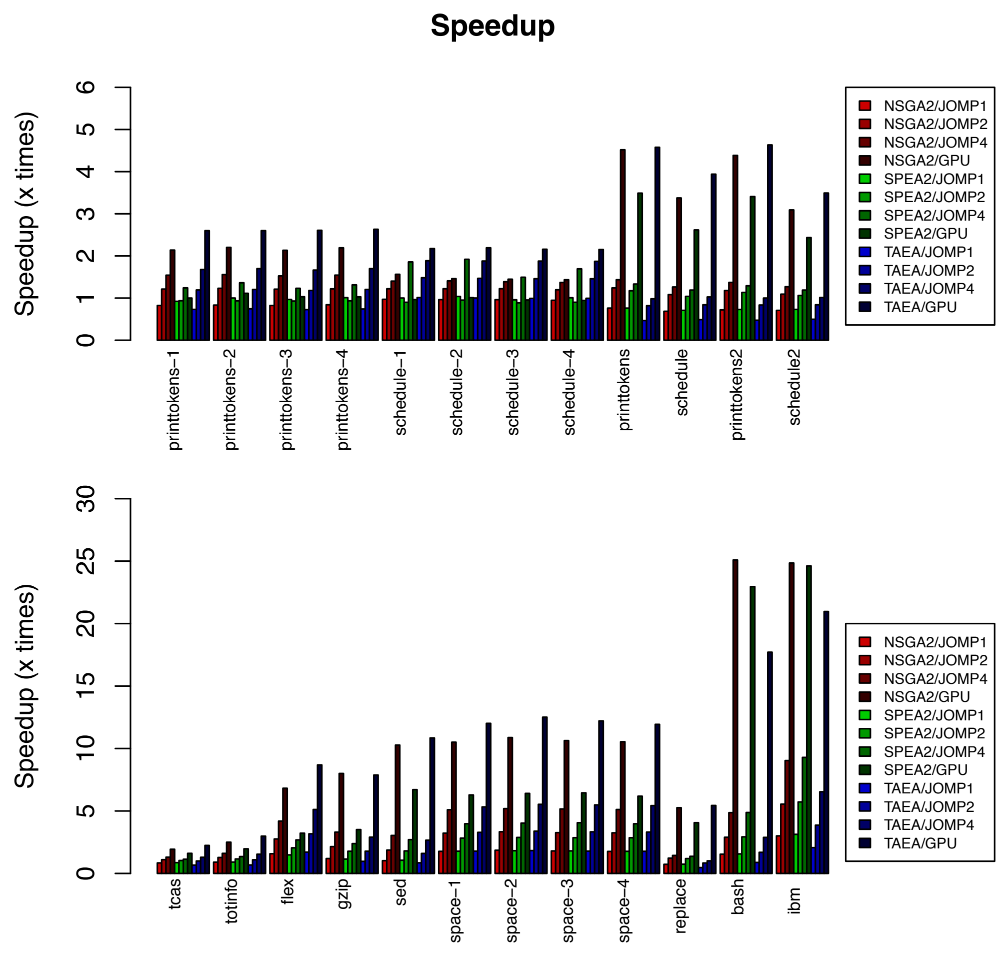

Fig. 1 Mean paired speed-ups achieved by different algorithms and parallel configurations.

amount of reduced method call overhead eventually becomes greater that the thread management overhead.

With the largest dataset, `ibm`, the `GPU` configuration of NSGA-II reduces the average execution time of `CPU`, 4,347 seconds (1 hour 12 minutes and 27 seconds), to the average of 174 seconds (2 minutes and 54 seconds). The speed–up remains consistently above 3.0`x` for all three algorithms if the problem size is larger than that of `flex`, i.e. about 400,000 (103 tests $\times$ 3,965 test requirements).

To provide inferential statistical analysis of the observed execution time data have been compared using the Mann-Whitney 'U' test. The Mann-Whitney 'U' test is a non-parametric statistical hypothesis test, i.e. it allows the comparison of two samples with unknown distribution. The execution time

Table 2 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
ibm	3.01	5.55	9.04	24.85

Table 3 Speed–up results for SPEA2

Subject	S_{JOMP1}	S_{JOMP2}	S_{JOMP4}	S_{GPU}
printtokens-1	0.92	0.94	1.24	1.00
printtokens-2	1.00	0.93	1.36	1.11
printtokens-3	0.97	0.93	1.23	1.03
printtokens-4	1.01	0.94	1.31	1.03
schedule-1	1.00	0.90	1.86	0.97
schedule-2	1.04	0.95	1.92	1.01
schedule-3	0.96	0.89	1.49	0.95
schedule-4	1.01	0.90	1.69	0.94
printtokens	0.76	1.17	1.33	3.49
schedule	0.71	1.04	1.19	2.62
printtokens2	0.73	1.13	1.29	3.41
schedule2	0.73	1.06	1.19	2.44
tcas	0.86	1.03	1.14	1.61
totinfo	0.91	1.16	1.35	1.97
flex	1.48	2.05	2.69	3.22
gzip	1.15	1.78	2.39	3.51
sed	1.05	1.80	2.70	6.71
space-1	1.78	2.83	3.98	6.28
space-2	1.82	2.88	4.03	6.41
space-3	1.80	2.86	4.06	6.45
space-4	1.77	2.86	3.98	6.18
replace	0.74	1.19	1.37	4.06
bash	1.56	2.93	4.88	22.96
ibm	3.13	5.72	9.29	24.62

data observed with JOMP1/2/4 and GPU configurations were compared to those from CPU configuration. The null hypothesis is that there is no difference between the parallel configurations and CPU configuration; the alternative hypothesis is that the execution time of the parallel configurations is smaller than that of CPU configuration.

Tables 5, 6 and 7 present the resulting *p*-values. With JOMP1 and JOMP2 configurations, the alternative hypothesis is rejected for 39 and 12 cases at the confidence level of 95%, respectively, out of of 42 cases with subjects with problem sizes smaller than that of flex, providing evidence that the parallel configurations required more time than the untreated configuration (CPU). With JOMP4 and GPU configurations, the null hypothesis is universally rejected for all subjects with problem sizes larger than that of flex, providing strong evidence that the parallel configurations required less time than the untreated configuration (CPU). The particular results are naturally dependent on the choice of the graphics card that has been used for the experiment. However, these results, taken together, provide strong evidence that, for test suite minimisation problems of realistic sizes, the GPGPU approach can provide a speed–up of at least 3.0x. This finding answers **RQ1**.

6.2 Correlation

Regarding **RQ2**, one important factor that contributes to the level of speed–up is the speed of each individual computational unit in the graphics card. The HD4850 graphics card used in the experiment contains 800 stream processor units that are normally used for the computation of geometric shading. Each of these processors executes a single thread of Algorithm 1, of which there exist more than 800. Therefore, if the individual stream processor is as powerful as a single core of the CPU, the absolute upper bound on speed–up would be 800. In practice, the individual processors run with a clock speed of 625MHz, which makes them much slower and, therefore, less powerful than a CPU core. This results in speed–up values lower than 800.

Table 4 Speed–up results for TAEA

Subject	S_{JOMP1}	S_{JOMP2}	S_{JOMP4}	S_{GPU}
printtokens-1	0.73	1.19	1.68	2.60
printtokens-2	0.75	1.21	1.70	2.60
printtokens-3	0.73	1.18	1.66	2.61
printtokens-4	0.74	1.21	1.70	2.63
schedule-1	1.01	1.48	1.89	2.17
schedule-2	1.00	1.47	1.88	2.19
schedule-3	0.99	1.46	1.88	2.16
schedule-4	0.99	1.46	1.87	2.15
printtokens	0.47	0.82	0.98	4.58
schedule	0.49	0.84	1.03	3.94
printtokens2	0.47	0.83	1.00	4.63
schedule2	0.50	0.84	1.01	3.49
tcas	0.67	1.00	1.29	2.24
totinfo	0.68	1.09	1.54	2.99
flex	1.71	3.17	5.12	8.69
gzip	0.97	1.78	2.91	7.88
sed	0.85	1.60	2.66	10.85
space-1	1.79	3.29	5.33	12.01
space-2	1.83	3.39	5.53	12.51
space-3	1.79	3.33	5.49	12.21
space-4	1.77	3.31	5.43	11.93
replace	0.47	0.84	1.01	5.44
bash	0.88	1.69	2.89	17.71
ibm	2.06	3.87	6.54	20.97

Table 5 Mann-Whitney U test for NSGA-II

Subject	p_{JOMP1}	p_{JOMP2}	p_{JOMP4}	p_{GPU}
printtokens-1	1.00e+00	1.51e-11	8.46e-18	1.51e-11
printtokens-2	1.00e+00	1.51e-11	8.46e-18	1.51e-11
printtokens-3	1.00e+00	1.51e-11	8.46e-18	8.46e-18
printtokens-4	1.00e+00	1.51e-11	1.51e-11	1.51e-11
schedule-1	1.00e+00	1.51e-11	1.51e-11	1.51e-11
schedule-2	1.00e+00	1.51e-11	8.46e-18	1.51e-11
schedule-3	1.00e+00	1.51e-11	1.51e-11	1.51e-11
schedule-4	1.00e+00	1.51e-11	1.51e-11	1.51e-11
printtokens	1.00e+00	8.46e-18	8.46e-18	8.46e-18
schedule	1.00e+00	1.51e-11	1.51e-11	8.46e-18
printtokens2	1.00e+00	1.51e-11	8.46e-18	1.51e-11
schedule2	1.00e+00	1.51e-11	8.46e-18	8.46e-18
tcas	1.00e+00	8.46e-18	8.46e-18	8.46e-18
totinfo	1.00e+00	1.51e-11	8.46e-18	8.46e-18
flex	8.46e-18	8.46e-18	1.51e-11	1.51e-11
gzip	1.51e-11	1.51e-11	1.51e-11	1.51e-11
sed	2.56e-07	8.46e-18	8.46e-18	1.51e-11
space-1	8.46e-18	8.46e-18	1.51e-11	1.51e-11
space-2	8.46e-18	8.46e-18	1.51e-11	1.51e-11
space-3	8.46e-18	8.46e-18	8.46e-18	1.51e-11
space-4	8.46e-18	8.46e-18	8.46e-18	1.51e-11
replace	1.00e+00	8.46e-18	1.51e-11	8.46e-18
bash	8.46e-18	8.46e-18	8.46e-18	8.46e-18
ibm	1.51e-11	8.46e-18	8.46e-18	1.51e-11

Table 6 Mann-Whitney U test for SPEA2

Subject	p_{JOMP1}	p_{JOMP2}	p_{JOMP4}	p_{GPU}
printtokens-1	9.99e-01	9.91e-01	8.71e-01	7.19e-01
printtokens-2	7.84e-01	9.81e-01	2.23e-01	3.06e-02
printtokens-3	9.39e-01	9.91e-01	9.85e-01	4.21e-01
printtokens-4	5.32e-01	9.85e-01	2.06e-01	3.71e-01
schedule-1	6.01e-01	1.00e+00	9.80e-01	9.01e-01
schedule-2	3.99e-02	9.99e-01	9.71e-01	3.27e-01
schedule-3	9.37e-01	1.00e+00	9.42e-01	9.59e-01
schedule-4	4.74e-01	1.00e+00	9.78e-01	9.94e-01
printtokens	1.00e+00	8.46e-18	1.50e-11	8.46e-18
schedule	1.00e+00	1.51e-11	1.50e-11	1.51e-11
printtokens2	1.00e+00	8.46e-18	1.50e-11	8.46e-18
schedule2	1.00e+00	1.51e-11	1.50e-11	1.51e-11
tcas	1.00e+00	1.51e-11	5.51e-07	1.51e-11
totinfo	1.00e+00	1.51e-11	1.50e-11	8.46e-18
flex	8.46e-18	8.46e-18	1.50e-11	8.46e-18
gzip	8.46e-18	8.46e-18	1.50e-11	8.46e-18
sed	8.46e-18	8.46e-18	1.50e-11	8.46e-18
space-1	8.46e-18	1.51e-11	1.50e-11	8.46e-18
space-2	8.46e-18	8.46e-18	1.50e-11	1.51e-11
space-3	8.46e-18	1.51e-11	1.50e-11	8.46e-18
space-4	8.46e-18	1.50e-11	1.50e-11	8.46e-18
replace	1.00e+00	1.50e-11	1.50e-11	8.46e-18
bash	8.46e-18	1.50e-11	1.50e-11	8.46e-18
ibm	8.46e-18	1.50e-11	1.50e-11	8.46e-18

Table 7 Mann-Whitney U test for TAEA

Subject	p_{JOMP1}	p_{JOMP2}	p_{JOMP4}	p_{GPU}
printtokens-1	1.00e+00	1.48e-11	1.50e-11	1.51e-11
printtokens-2	1.00e+00	1.50e-11	1.49e-11	1.51e-11
printtokens-3	1.00e+00	1.50e-11	1.50e-11	1.51e-11
printtokens-4	1.00e+00	1.49e-11	1.49e-11	1.51e-11
schedule-1	3.86e-02	1.48e-11	1.49e-11	1.51e-11
schedule-2	9.96e-01	1.49e-11	1.48e-11	1.50e-11
schedule-3	9.99e-01	1.50e-11	1.50e-11	1.51e-11
schedule-4	7.63e-01	1.50e-11	1.50e-11	1.50e-11
printtokens	1.00e+00	1.00e+00	1.00e+00	1.51e-11
schedule	1.00e+00	1.00e+00	3.66e-10	1.51e-11
printtokens2	1.00e+00	1.00e+00	5.85e-01	1.51e-11
schedule2	1.00e+00	1.00e+00	2.54e-06	1.51e-11
tcas	1.00e+00	4.50e-01	1.50e-11	1.51e-11
totinfo	1.00e+00	1.50e-11	1.50e-11	1.51e-11
flex	1.50e-11	1.50e-11	1.50e-11	1.50e-11
gzip	1.00e+00	1.50e-11	1.49e-11	1.51e-11
sed	1.00e+00	1.50e-11	1.50e-11	1.51e-11
space-1	1.50e-11	1.50e-11	1.50e-11	1.51e-11
space-2	1.50e-11	1.50e-11	1.50e-11	1.51e-11
space-3	1.50e-11	1.50e-11	1.50e-11	1.51e-11
space-4	1.50e-11	1.50e-11	1.50e-11	1.51e-11
replace	1.00e+00	1.00e+00	2.10e-02	1.51e-11
bash	1.00e+00	1.50e-11	1.50e-11	1.51e-11
ibm	1.50e-11	1.50e-11	1.50e-11	1.51e-11

Table 8 Spearman's rank correlation coefficients between three size factors and speed–ups

Algorithm	Config	ρ_z	ρ_l	ρ_m
NSGA-II	`JOMP1`	0.2257	0.6399	-0.8338
	`JOMP2`	0.4908	0.7800	-0.6423
	`JOMP4`	0.4788	0.8227	-0.6378
	`GPGPU`	0.8760	0.8617	-0.2299
SPEA2	`JOMP1`	0.2327	0.6646	-0.7827
	`JOMP2`	0.7897	0.7977	-0.3375
	`JOMP4`	0.6852	0.7201	-0.3286
	`GPGPU`	0.9022	0.7618	-0.1286
TAEA	`JOMP1`	-0.0084	0.5225	-0.9302
	`JOMP2`	0.1527	0.6580	-0.8867
	`JOMP4`	0.1671	0.6686	-0.8760
	`GPGPU`	0.8723	0.8729	-0.2536

In order to answer **RQ2**, statistical regression analysis was performed on the correlation between the observed speed–up and the factors that constitute the size of problems.

Three size factors have been analysed for the statistical regression: size of test goal set, size of test suite and their product. The number of test requirements and the number of test cases are denoted by l and m respectively, following the matrix notation in Section 3: l is proportional to the number of threads the GPGPU-version of the algorithm has to execute (as the size of the matrix C' is l-by-n and n is fixed); m denotes the amount of computation that needs to be performed by a single thread (as each matrix-multiplication kernel computes a loop with m iterations). In addition to these measurement, another size factor $z = l \cdot m$ is considered to represent the *perceived* size of the minimisation problem. Table 8 shows the results of Spearman's rank correlation analysis between size factors and observed speed–ups.

Spearman's rank correlation is a non-parametric measure of how well the relationship between two variables can be described using a monotonic function. As one variable increases, the other variable will tend to increase monotonically if the coefficient is close to 1, whereas it would decrease monotonically if the coefficient is close to -1.

In all algorithms and configurations, the size factor l shows the strongest positive correlation with speed–ups. The correlation coefficients for z are weaker than those for l or, in the case of `JOMP1` for the Two Archive algorithm, negative. The correlation for m remains negative for all algorithms and configurations.

To gain further insights into the correlation between size and the speed–up observed, a regression analysis was performed. Factor z is considered in isolation, whereas l and m are considered together; each variable has been considered in its linear form (z, l and m) and logarithmic form ($\log z, \log l$ and $\log m$). This results in 6 different combinations of regression models. Tables 9, 10 and 11 in the appendix present the detailed results of regression analysis for the three algorithms respectively.

With a few exceptions of very small margins (NSGA-II with `JOMP4` and SPEA2 with `JOMP1`, `JOMP4`, and `GPU`), the model with the highest r^2 correlation for all algorithms and configurations is $S_p = \alpha \log l + \beta \log m + \gamma$. Figure 2 shows the 3D plot of this model for the GPU and `JOMP4` configuration of the Two Archive algorithm.

The observed trend is that the inclusion of $\log l$ results in higher correlation, whereas models that use l in its linear form tend to result in lower correlation. This supports the results of Spearman's rank correlation analysis in Table 8. The coefficients for the best-fit regression model for `GPU`, $S_p = \alpha \log l + \beta \log m + \gamma$, can explain why the speed–up results for `space` test suites are higher than those for test suites with z values such as `tcas`, `gzip` and `replace`. Apart from `bash` and `ibm`, `space` has the

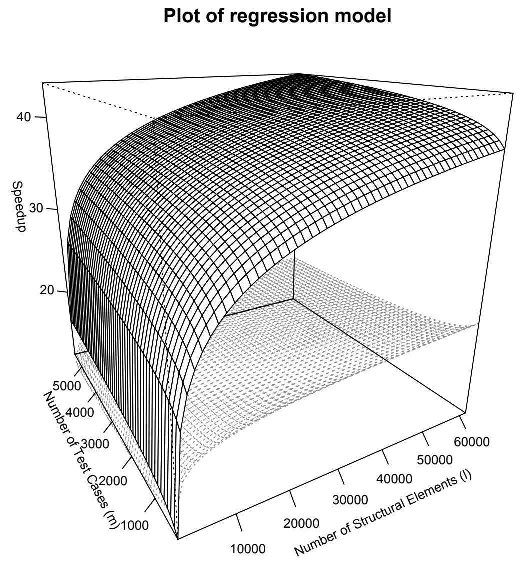

Fig. 2 3D-plot of regression model $S_p = \alpha \log l + \beta \log m + \gamma$ for GPU(solid line) and JOMP4(dotted line) configurations for Two Archive algorithm

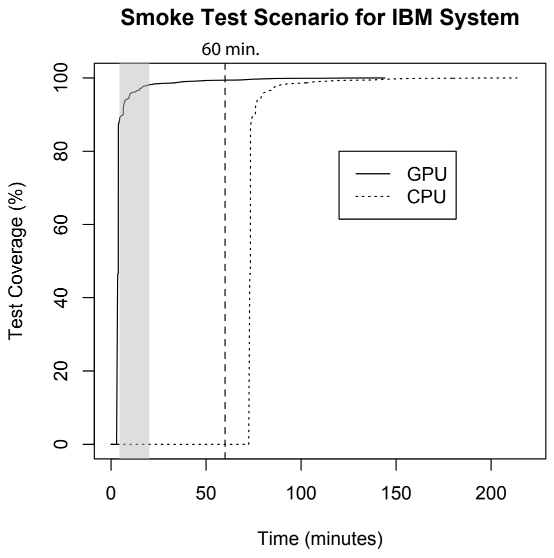

Fig. 3 Comparison of smoke test scenarios for IBM System (`ibm`). The solid line shows the trade-offs between time and test coverage when `GPU` configuration of NSGA-II is used, whereas the dotted line shows that of `CPU`. The grey area shows the interesting trade-off that the `CPU` configuration fails to exploit within 60 minutes.

largest number of test requirements to cover, i.e. l. Since α is more than twice larger than β, a higher value of l has more impact to S_p than m.

Based on the analysis, **RQ2** is answered as follows: the observed speed–up shows a strong linear correlation to the log of the number of test requirements to cover and the log of the number of test cases in the test suite. The positive correlation provides evidence that GPU-based parallelisation scales up.

Furthermore, within the observed data, the speed–up continues to increase as the problem size grows, which suggests that the graphics card did not reach its full computational capacity. It may be that, for larger problems, the speed–up would be even greater than those observed in this paper. The finding that the scalability factor increases with overall problem size is a very encouraging finding; as the problem gets harder, the solution gets better.

6.3 Insights

This section discusses a possible real-world scenario in which the parallelisation of multi-objective test suite minimisation can have a high impact. A smoke test is a testing activity that is usually performed in a very short window of time to detect the most obvious faults, such as system crashes. IBM's smoke test practice is to allow from 30 to 60 minutes of time to execute a subset of tests from a large test suite that would require more than 4 hours to execute in its entirety.

Using static smoke test suite is problematic as running the same tests at every regression greatly reduces the likelihood of finding bugs. Therefore it is important to recalculate the most relevant smoke test suite given the changes to the code. It is for this reason that the cost of computation, especially the actual time it takes, becomes very important.

Figure 3 shows two possible smoke test scenarios based on the results of `CPU` and `GPGPU` configurations of NSGA-II. It is a plot of how much test requirement coverage can be achieved during the given time, including the time needed for multi-objective test suite minimisation. The solid line represents the scenario based on the `GPGPU` configuration of the algorithm, whereas the dotted line represents the scenario based on the `CPU` configuration. The beginning flat segment at the bottom shows the time each configuration spends on the optimisation process; the curved segment shows the trade-off between time and test coverage achieved by the optimised test suite. Since the `CPU` configuration of NSGA-II takes longer than 60 minutes to terminate, it cannot contribute to any smoke test scenario that must be completed within 60 minutes. On the other hand, the `GPGPU` configuration allows the tester to consider a subset of tests that can be executed within 30 minutes. If the grey region was wider than Figure 3, the difference between two configurations would have been even more dramatic.

This answers **RQ3** as follows: a faster execution of optimisation algorithms enables the tester not only to use the algorithms but also to exploit their results more effectively. This real-world smoke test example from IBM demonstrates that scale–ups accrued from the use of `GPGPU` are not only sources of efficiency improvement, they can also make possible test activities that are simply impossible without this scalability.

The ability to execute a sophisticated optimisation algorithm within a relatively short time also allows the tester to consider state-of-the-art regression testing techniques with greater flexibility. The greater flexibility is obtained because the cost of the optimisation does not have to be amortised across multiple iterations. Many state-of-the-art regression testing techniques require the use of continuously changing sets of testing data, such as recent fault history [54] or the last time a specific test case has been executed [18,30]. In addition to the use of dynamic testing data, the previous work also showed that repeatedly using the same subset of a large test suite may impair the fault detection capability of the regression testing [57].

7 Threats to Validity

Threats to internal validity concern the factors that could have affected the experiments in the paper. While GPGPU architecture has been researched for some time, the commercially available GPGPU frameworks such as CUDA and OpenCL are still in their early stages and, therefore, may contain faults in the implementation.

The GPGPU matrix-multiplication routine has been manually tested and validated with additional data apart from the test suites chosen for the empirical study. Regarding the precision of the GPGPU-based calculation, the aim of the paper is to investigate the potential speed–up that can be gained by using GPGPU, rather than to consider the effectiveness of the actual test suite minimisation in the context of regression testing. Therefore, the precision issue does not constitute a major issue for the aim of this study.

Threats to external validity concern any factor that might prevent the generalisation of the result presented by the paper. Since the performance of GPGPU computing is inherently hardware specific, the results reported in the paper may not be reproducible in their exact form using other combinations of hardware components. However, with the advances in graphics card architecture, it is more likely that experiments with the same approach with newer graphics card will only improve the speed–up results reported in the paper.

It should be also noted that optimising test suite minimisation using evolutionary computation is an inherently ideal candidate for GPGPU computation as the reformulated problem, matrix-multiplication, is highly parallel in nature. Other problems in search-based software engineering may not render themselves as easily as the test suite minimisation problem. However, this issue is inherent in any attempts to parallelise a software engineering technique and not specific to GPGPU approach.

Threats to construct validity arise when measurements used in the experiments do not capture the concepts they are supposed to represent. The speed–up calculation was based on the measurements of execution time for both algorithms using system clock, which was chosen because it represents the *speed* of a technique to the end-user. Regarding the measurements of problem size used for the regression analysis, there may exist more sophisticated measurements of test suites and program source code that correlates better with the speed–up. However, both the number of test requirements and the number of test cases are probably the most readily available measurements about source code and test suites and provide a reasonable starting point for this type of analysis.

8 Related Work

Recent developments in graphics hardware provide an affordable means of parallelism: not only is the hardware more affordable than that of multiple PCs but also the management cost is much smaller than that required for a cluster of PCs, because it depends on a single hardware component. GPGPU has been successfully applied to various scientific computations [6, 21], while Langdon [32] recently used GPGPU for performance improvements in optimization problems. However, these techniques have not been applied to Search Based Software Engineering problems, motivating the work of this paper, i,e., the use of GPGPU to achieve performance improvements in SBSE.

As a first instance of GPGPU SBSE, we study the SBSE application domain of regression testing. Regression test selection (also known as test suite minimisation) aims to reduce the number of tests to be executed by calculating the minimum set of tests that are required to satisfy the given test requirements. The problem has been formulated as the minimal hitting set problem [28], which is NP-hard [20].

Various heuristics for the minimal representative set problem, or the minimal set cover problem (the dual of the former), have been suggested for test suite minimisation [11, 38]. However, empirical evaluations of these techniques have reported conflicting views on the impact on fault detection capability: some reported no impact [52, 53] while others reported compromised fault detection capability [43, 44].

One potential reason why test suite minimisation has a negative impact on the fault detection capability is the fact that the criterion for minimisation is structural coverage; achieving coverage alone may not be sufficient for revealing faults. This paper uses the multi-objective approach based on Multi-Objective Evolutionary Algorithm (MOEA) introduced by Yoo and Harman [54]; the paper also presents the first attempt to parallelise test suite minimisation with sophisticated criteria for scalability. Multi-objective forms of regression testing problems are increasingly popular in SBSE work, since most real-world regression testing scenarios need to satisfy multiple objectives [23]. Our SBSE approach to regression testing has also been used at Google [59].

Population-based evolutionary algorithms are ideal candidates for GPGPU parallelisation [39] and existing work has shown successful implementations for classical problems. Tsutsui and Fujimoto implemented a single-objective parallel Genetic Algorithm (GA) using GPU for the Quadratic Assignment Problem (QAP) [48]. Wilson and Banzaf implemented a linear Genetic Programming (GP) algorithm on XBox360 game consoles [50]. Langdon and Banzaf implemented GP for GPU using an SIMD interpreter for fitness evaluation [32]. Wong implemented an MOEA on GPU and evaluated the implementation using a suite of benchmark problems [51]. Wong's implementation parallelised not only the fitness evaluation step but also the parent selection, crossover & mutation operator as well as the dominance checking.

Despite the highly parallelisable nature of many techniques used in SBSE, few parallel algorithms have been used. Of 763 papers on SBSE [60] only three present results for parallel execution of SBSE. Mitchell et al. used a distributed architecture for their clustering tool `Bunch` [36]. Mahdavi et al. [34] used a cluster of standard PCs to implement a parallel hill climbing algorithm. Asadi et al. used a distributed Server-Client architecture for Concept Location problem [4]. All three of these previous approaches use a distributed architecture that requires multiple machines. The present paper is the first work on SBSE that presents results for the highly affordable parallelism based on GPGPU.

There is existing work that re-implements meta-heuristic algorithms on GPGPU for non-SBSE applications. This previous work ports the meta-heuristic algorithms in their entirety [51] to GPGPU, whereas the present paper concerns the practical improvement in scalability of SBSE, rather than the technical feasibility of GPGPU-based meta-heuristic implementation. The present paper thus re-implements only the most parallelisable module in Multi-Objective Evolutionary Algorithms (MOEAs) and performs an extensive empirical evaluation of the impact on scalability.

We believe that this approach may also prove to be applicable to many other SBSE problems. To achieve this, it will be necessary to develop new ways to port the fitness computation to the GPGPU device. For some applications, such as test data generation and re-generation problems [3, 56], this may not be possible, because the fitness function requires execution of the program under test; we cannot be sure that GPGPU devices will ever develop to the point that execution of arbitrary code will become possible.

However, for other SBSE applications, such as requirements optimisation [46,61], prediction system feature selection [31] and requirements sensitivity analysis [25], fitness computation remains an oft-repeated and, thereby, SIMD-friendly requirement. Also, in these application areas, the underlying Software Engineering problem may be characterised using a table (albeit a very large one). In such cases, where the SBSE problem can be formulated in terms of the optimisation of choices, based on a spreadsheet of known values, this may prove to port well onto the SIMD architecture offered by GPGPU devices.

Furthermore, for requirements optimisation problems there is known to be a close similarity between the problem representation for search based requirements and search based regression testing [26]. As a result, the techniques used here may also prove to be readily applicable to these problems with little need for significant modification of our approach.

9 Conclusion

This paper presented the first results on GPGPU SBSE; the use of GPGPU-based massive parallelism for improving scalability of regression testing, based on Search-Based Software Engineering (SBSE). The advances in GPGPU architecture and the consequent availability of parallelism provide an ideal platform for improving SBSE scalability through SIMD parallelism.

The paper presents an evaluation of the GPGPU-based test suite minimisation for real-world examples that include an industry-scale test suite. This approach to GPGPU SBSE was evaluated on three popular multi-objective evolutionary algorithms. The results show that the GPGPU-based optimisation can achieve a speed–up of up to 25.09`x` compared to a single-threaded version of the same algorithm executed on a CPU. The highest speed–up achieved by the CPU-based parallel optimisation was 9.29`x`. Statistical analysis shows that the speed–up correlates to the logarithmic of the problem size, i.e. the size of the program under test and the size of the test suite. This finding indicates that as the problem becomes larger, the scalability of the proposed approach increases; a very attractive finding. Future work will include an empirical study of a wider range of test suites, as well as seeking insights into why MOEAs benefit differently from parallelisation.

References

1. ATI Stream Computing: OpenCL Programming Guide Rev. 1.05. AMD Corporation (2010)
2. Afzal, W., Torkar, R., Feldt, R.: A systematic review of search-based testing for non-functional system properties. Information and Software Technology **51**(6), 957–976 (2009)
3. Ali, S., Briand, L.C., Hemmati, H., Panesar-Walawege, R.K.: A systematic review of the application and empirical investigation of search-based test-case generation. IEEE Transactions on Software Engineering **36**(6), 742–762 (2010)
4. Asadi, F., Antoniol, G., Guéhéneuc, Y.: Concept locations with genetic algorithms: A comparison of four distributed architectures. In: Proceedings of 2^{nd} International Symposium on Search based Software Engineering (SSBSE 2010), pp. 153–162. IEEE Computer Society Press, Benevento, Italy (2010)
5. Black, J., Melachrinoudis, E., Kaeli, D.: Bi-criteria models for all-uses test suite reduction. In: Proceedings of the 26th International Conference on Software Engineering, pp. 106–115 (2004)
6. Boyer, M., Tarjan, D., Acton, S.T., Skadron, K.: Accelerating leukocyte tracking using cuda: A case study in leveraging manycore coprocessors. In: Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS) (2009)
7. Bull, J.M., Westhead, M.D., Kambites, M.E., Obrzalek, J.: Towards OpenMP for java. In: Proceedings of the 2nd European Workshop on OpenMP, pp. 98–105 (2000)
8. Chafik, O.: JavaCL: opensource Java wrapper for OpenCL library (2009). http://code.google.com/p/javacl/. Accessed June 6^{th} 2010
9. Chau, P.Y.K., Tam, K.Y.: Factors affecting the adoption of open systems: An exploratory study. MIS Quarterly **21**(1) (1997)
10. Chen, T., Lau, M.: Heuristics towards the optimization of the size of a test suite. In: Proceedings of the 3rd International Conference on Software Quality Management, vol. 2, pp. 415–424 (1995)
11. Chen, T.Y., Lau, M.F.: Dividing strategies for the optimization of a test suite. Information Processing Letters **60**(3), 135–141 (1996)
12. Clark, J., Dolado, J.J., Harman, M., Hierons, R.M., Jones, B., Lumkin, M., Mitchell, B., Mancoridis, S., Rees, K., Roper, M., Shepperd, M.: Reformulating software engineering as a search problem. IEE Proceedings — Software **150**(3), 161–175 (2003)
13. Cordy, J.R.: Comprehending reality - practical barriers to industrial adoption of software maintenance automation. In: IEEE International Workshop on Program Comprehension (IWPC '03), pp. 196–206. IEEE Computer Society (2003)
14. Do, H., Elbaum, S.G., Rothermel, G.: Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering **10**(4), 405–435 (2005)
15. Durillo, J., Nebro, A., Alba, E.: The jmetal framework for multi-objective optimization: Design and architecture. In: Proceedings of Congress on Evolutionary Computation 2010, pp. 4138–4325. Barcelona, Spain (2010)
16. Durillo, J.J., Nebro, A.J., Luna, F., Dorronsoro, B., Alba, E.: jMetal: A Java Framework for Developing Multi-Objective Optimization Metaheuristics. Tech. Rep. ITI-2006-10, Departamento de Lenguajes y Ciencias de la Computación, University of Málaga, E.T.S.I. Informática, Campus de Teatinos (2006)
17. Ekman, M., Warg, F., Nilsson, J.: An in-depth look at computer performance growth. SIGARCH Computer Architecture News **33**(1), 144–147 (2005)
18. Engström, E., Runeson, P., Wikstrand, G.: An empirical evaluation of regression testing based on fix-cache recommendations. In: Proceedings of the 3rd International Conference on Software Testing Verification and Validation (ICST 2010), pp. 75–78. IEEE Computer Society Press (2010)
19. Engström, E., Runeson, P.P., Skoglund, M.: A systematic review on regression test selection techniques. Information and Software Technology **52**(1), 14–30 (2009)
20. Garey, M.R., Johnson, D.S.: Computers and Intractability: A guide to the theory of NP-Completeness. W. H. Freeman and Company, New York, NY (1979)
21. Govindaraju, N.K., Gray, J., Kumar, R., Manocha, D.: Gputerasort: High performance graphics coprocessor sorting for large database management. In: ACM SIGMOD (2006)
22. Harman, M.: The current state and future of search based software engineering. In: FOSE '07: 2007 Future of Software Engineering, pp. 342–357. IEEE Computer Society, Washington, DC, USA (2007)
23. Harman, M.: Making the case for MORTO: Multi objective regression test optimization. In: 1^{st} International Workshop on Regression Testing (Regression 2011). Berlin, Germany (2011)

24. Harman, M., Jones, B.F.: Search based software engineering. Information and Software Technology **43**(14), 833–839 (2001)
25. Harman, M., Krinke, J., Ren, J., Yoo, S.: Search based data sensitivity analysis applied to requirement engineering. In: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation (GECCO '09), pp. 1681–1688. ACM, Montreal, Canada (2009)
26. Harman, M., Mansouri, A., Zhang, Y.: Search based software engineering trends, techniques and applications. ACM Computing Surveys (2012). To appear
27. Harrold, M., Orso, A.: Retesting software during development and maintenance. In: Frontiers of Software Maintenance (FoSM 2008), pp. 99–108. IEEE Computer Society Press (2008)
28. Harrold, M.J., Gupta, R., Soffa, M.L.: A methodology for controlling the size of a test suite. ACM Transactions on Software Engineering and Methodology **2**(3), 270–285 (1993)
29. Hutchins, M., Foster, H., Goradia, T., Ostrand, T.: Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In: Proceedings of the 16th International Conference on Software Engineering (ICSE 1994), pp. 191–200. IEEE Computer Society Press (1994)
30. Kim, J.M., Porter, A.: A history-based test prioritization technique for regression testing in resource constrained environments. In: Proceedings of the 24th International Conference on Software Engineering, pp. 119–129. ACM Press (2002)
31. Kirsopp, C., Shepperd, M., Hart, J.: Search heuristics, case-based reasoning and software project effort prediction. In: GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1367–1374. Morgan Kaufmann Publishers, San Francisco, CA 94104, USA (2002)
32. Langdon, W.B., Banzhaf, W.: A SIMD interpreter for genetic programming on GPU graphics cards. In: M. O'Neill, L. Vanneschi, S. Gustafson, A.I. Esparcia Alcazar, I. De Falco, A. Della Cioppa, E. Tarantino (eds.) Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008, *Lecture Notes in Computer Science*, vol. 4971, pp. 73–85. Springer (2008)
33. Li, Z., Harman, M., Hierons, R.M.: Search Algorithms for Regression Test Case Prioritization. IEEE Transactions on Software Engineering **33**(4), 225–237 (2007)
34. Mahdavi, K., Harman, M., Hierons, R.M.: A multiple hill climbing approach to software module clustering. In: IEEE International Conference on Software Maintenance, pp. 315–324. IEEE Computer Society Press, Los Alamitos, California, USA (2003)
35. Maia, C.L.B., do Carmo, R.A.F., de Freitas, F.G., de Campos, G.A.L., de Souza, J.T.: A multi-objective approach for the regression test case selection problem. In: Proceedings of Anais do XLI Simpòsio Brasileiro de Pesquisa Operacional (SBPO 2009), pp. 1824–1835 (2009)
36. Mitchell, B.S., Traverso, M., Mancoridis, S.: An architecture for distributing the computation of software clustering algorithms. In: IEEE/IFIP Proceedings of the Working Conference on Software Architecture (WICSA '01), pp. 181–190. IEEE Computer Society, Amsterdam, Netherlands (2001)
37. Nethercote, N., Seward, J.: Valgrind: A program supervision framework. In: Proceedings of ACM Conference on Programming Language Design and Implementation, pp. 89–100. ACM Press (2007)
38. Offutt, J., Pan, J., Voas, J.: Procedures for reducing the size of coverage-based test sets. In: Proceedings of the 12th International Conference on Testing Computer Software, pp. 111–123. ACM Press (1995)
39. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. Computer Graphics Forum **26**(1), 80–113 (2007)
40. Praditwong, K., Yao, X.: A new multi-objective evolutionary optimisation algorithm: The two-archive algorithm. In: Proceedings of Computational Intelligence and Security, International Conference, *Lecture Notes in Computer Science*, vol. 4456, pp. 95–104 (2006)
41. Premkumar, G., Potter, M.: Adoption of computer aided software engineering (CASE) technology: An innovation adoption perspective. Database **26**(2&3), 105–124 (1995)
42. Räihä, O.: A survey on search-based software design. Tech. Rep. D-2009-1, Department of Computer Science, University of Tampere (2009)
43. Rothermel, G., Elbaum, S., Malishevsky, A., Kallakuri, P., Davia, B.: The impact of test suite granularity on the cost-effectiveness of regression testing. In: Proceedings of the 24th International Conference on Software Engineering (ICSE 2002), pp. 130–140. ACM Press (2002)
44. Rothermel, G., Harrold, M., Ronne, J., Hong, C.: Empirical studies of test suite reduction. Software Testing, Verification, and Reliability **4**(2), 219–249 (2002)
45. Rothermel, G., Harrold, M.J., Ostrin, J., Hong, C.: An empirical study of the effects of minimization on the fault detection capabilities of test suites. In: Proceedings of International Conference on Software Maintenance (ICSM 1998), pp. 34–43. IEEE Computer Society Press, Bethesda, Maryland, USA (1998)
46. Saliu, M.O., Ruhe, G.: Bi-objective release planning for evolving software systems. In: Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC-FSE 2007), pp. 105–114. ACM Press, New York, NY, USA (2007)

47. de Souza, J.T., Maia, C.L., de Freitas, F.G., Coutinho, D.P.: The human competitiveness of search based software engineering. In: Proceedings of 2^{nd} International Symposium on Search based Software Engineering (SSBSE 2010), pp. 143–152. IEEE Computer Society Press, Benevento, Italy (2010)
48. Tsutsui, S., Fujimoto, N.: Solving quadratic assignment problems by genetic algorithms with GPU computation: a case study. In: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference (GECCO 2009), pp. 2523–2530. ACM Press (2009)
49. Walcott, K.R., Soffa, M.L., Kapfhammer, G.M., Roos, R.S.: Time aware test suite prioritization. In: Proceedings of the International Symposium on Software Testing and Analysis, pp. 1–12 (2006)
50. Wilson, G., Banzhaf, W.: Deployment of CPU and GPU-based genetic programming on heterogeneous devices. In: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference (GECCO 2009), pp. 2531–2538. ACM Press, New York, NY, USA (2009)
51. Wong, M.L.: Parallel multi-objective evolutionary algorithms on graphics processing units. In: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference (GECCO 2009), pp. 2515–2522. ACM Press, New York, NY, USA (2009)
52. Wong, W.E., Horgan, J.R., London, S., Mathur, A.P.: Effect of test set minimization on fault detection effectiveness. Software Practice and Experience **28**(4), 347–369 (1998)
53. Wong, W.E., Horgan, J.R., Mathur, A.P., Pasquini, A.: Test set size minimization and fault detection effectiveness: A case study in a space application. The Journal of Systems and Software **48**(2), 79–89 (1999)
54. Yoo, S., Harman, M.: Pareto efficient multi-objective test case selection. In: Proceedings of International Symposium on Software Testing and Analysis, pp. 140–150. ACM Press (2007)
55. Yoo, S., Harman, M.: Regression testing minimisation, selection and prioritisation: A survey. Software Testing, Verification, and Reliability **22**(2), 67–120 (2012)
56. Yoo, S., Harman, M.: Test data regeneration: Generating new test data from existing test data. Journal of Software Testing, Verification and Reliability **22**(3), 171–201 (2012)
57. Yoo, S., Harman, M., Ur, S.: Measuring and improving latency to avoid test suite wear out. In: Proceedings of the Interntional Conference on Software Testing, Verification and Validation Workshop (ICSTW 2009), pp. 101–110. IEEE Computer Society Press (2009)
58. Yoo, S., Harman, M., Ur, S.: Highly scalable multi-objective test suite minimisation using graphics card. In: LNCS: Proceedings of the 3rd International Symposium on Search-Based Software Engineering, *SSBSE*, vol. 6956, pp. 219–236 (2011)
59. Yoo, S., Nilsson, R., Harman, M.: Faster fault finding at Google using multi objective regression test optimisation. In: 8^{th} European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '11). Szeged, Hungary (2011). Industry Track
60. Zhang, Y.: SBSE repository (2011). www.sebase.org/sbse/publications/repository.html. Accessed February 14^{th} 2011
61. Zhang, Y., Harman, M., Finkelstein, A., Mansouri, A.: Comparing the performance of metaheuristics for the analysis of multi-stakeholder tradeoffs in requirements optimisation. Journal of Information and Software Technology **53**(6), 761–773 (2011)

Appendix

Tables 9, 10 and 11 present the results of regression analysis for the three algorithm respectively. Tables 12, 13 and 14 contain the mean and standard deviation of $Time_{total}, Time_{init}, Time_{fitness}$ and $Time_{remaining}$ for NSGA-II, SPEA2 and Two Archive algorithm respectively.

Table 9 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

Table 10 Regression Analysis for SPEA2

Config	Model	α	β	γ	R^2
JOMP1	$S_p \sim z$	1.60e-07	-	1.03e+00	0.5085
	$S_p \sim \log z$	1.89e-01	-	-1.16e+00	0.2988
	$S_p \sim l + m$	3.37e-05	-1.20e-04	1.21e+00	0.7443
	$S_p \sim \log l + m$	2.58e-01	-6.08e-05	-3.57e-01	0.7987
	$S_p \sim l + \log m$	3.23e-05	-1.79e-01	2.19e+00	0.7883
	$S_p \sim \log l + \log m$	2.50e-01	-7.97e-02	1.17e-01	0.7982
JOMP2	$S_p \sim z$	3.67e-07	-	1.31e+00	0.6289
	$S_p \sim \log z$	5.31e-01	-	-4.94e+00	0.5567
	$S_p \sim l + m$	7.41e-05	-1.02e-04	1.53e+00	0.6867
	$S_p \sim \log l + m$	6.14e-01	4.59e-05	-2.22e+00	0.8656
	$S_p \sim l + \log m$	7.24e-05	-1.78e-01	2.52e+00	0.7031
	$S_p \sim \log l + \log m$	6.30e-01	9.28e-02	-2.85e+00	0.8700
JOMP4	$S_p \sim z$	6.26e-07	-	1.78e+00	0.5504
	$S_p \sim \log z$	7.86e-01	-	-7.37e+00	0.3657
	$S_p \sim l + m$	1.23e-04	-2.40e-04	2.25e+00	0.5965
	$S_p \sim \log l + m$	9.38e-01	-2.73e-05	-3.44e+00	0.6443
	$S_p \sim l + \log m$	1.20e-04	-3.56e-01	4.19e+00	0.6081
	$S_p \sim \log l + \log m$	9.56e-01	3.15e-02	-3.78e+00	0.6442
GPU	$S_p \sim z$	2.32e-06	-	2.25e+00	0.8777
	$S_p \sim \log z$	3.12e+00	-	-3.42e+01	0.6666
	$S_p \sim l + m$	3.82e-04	1.98e-04	3.06e+00	0.5713
	$S_p \sim \log l + m$	3.01e+00	8.99e-04	-1.52e+01	0.6657
	$S_p \sim l + \log m$	3.90e-04	5.17e-01	4.89e-02	0.5791
	$S_p \sim \log l + \log m$	3.38e+00	1.96e+00	-2.88e+01	0.7417

Table 11 Regression Analysis for Two Archive

Config	Model	α	β	γ	R^2
JOMP1	$S_p \sim z$	7.34e-08	-	9.35e-01	0.1280
	$S_p \sim \log z$	9.65e-02	-	-1.92e-01	0.0931
	$S_p \sim l + m$	1.78e-05	-1.74e-04	1.14e+00	0.5412
	$S_p \sim \log l + m$	1.94e-01	-1.20e-04	-7.59e-02	0.7637
	$S_p \sim l + \log m$	1.54e-05	-2.79e-01	2.68e+00	0.7108
	$S_p \sim \log l + \log m$	1.64e-01	-2.01e-01	1.22e+00	0.8350
JOMP2	$S_p \sim z$	1.60e-07	-	1.59e+00	0.1587
	$S_p \sim \log z$	2.57e-01	-	-1.45e+00	0.1731
	$S_p \sim l + m$	3.72e-05	-2.98e-04	1.95e+00	0.4942
	$S_p \sim \log l + m$	4.31e-01	-1.73e-04	-7.67e-01	0.8095
	$S_p \sim l + \log m$	3.27e-05	-4.94e-01	4.69e+00	0.6461
	$S_p \sim \log l + \log m$	3.84e-01	-3.04e-01	1.22e+00	0.8571
JOMP4	$S_p \sim z$	3.12e-07	-	2.33e+00	0.1865
	$S_p \sim \log z$	5.21e-01	-	-3.84e+00	0.2196
	$S_p \sim l + m$	6.95e-05	-5.20e-04	2.97e+00	0.4990
	$S_p \sim \log l + m$	8.17e-01	-2.82e-04	-2.18e+00	0.8556
	$S_p \sim l + \log m$	6.17e-05	-8.50e-01	7.69e+00	0.6322
	$S_p \sim \log l + \log m$	7.46e-01	-4.77e-01	9.01e-01	0.8880
GPU	$S_p \sim z$	1.64e-06	-	4.96e+00	0.5728
	$S_p \sim \log z$	2.79e+00	-	-2.82e+01	0.7056
	$S_p \sim l + m$	2.83e-04	-3.54e-04	6.02e+00	0.4516
	$S_p \sim \log l + m$	3.05e+00	5.02e-04	-1.31e+01	0.9417
	$S_p \sim l + \log m$	2.76e-04	-6.36e-01	9.59e+00	0.4620
	$S_p \sim \log l + \log m$	3.21e+00	9.47e-01	-1.94e+01	0.9603

Table 12: Execution time of NSGA-II algotirhm

Subject	Config	$\bar{T}_{total}$	$\sigma_{T_{total}}$	$\bar{T}_{init}$	$\sigma_{T_{init}}$	$\bar{T}_{fitness}$	$\sigma_{T_{fitness}}$	$\bar{T}_{remaining}$	$\sigma_{T_{remaining}}$
printtokens-1	CPU	12265.23	133.47	0.00	0.00	8565.77	63.68	3699.47	104.78
printtokens-1	JOMP1	14869.77	379.38	5.83	0.45	11084.23	310.38	3779.70	119.98
printtokens-1	JOMP2	10112.50	146.97	5.70	0.46	5905.90	87.01	4200.90	107.54
printtokens-1	JOMP4	7950.77	165.02	5.67	0.47	3633.93	63.73	4311.17	127.42
printtokens-1	GPGPU	5739.60	145.89	469.33	3.64	1934.33	124.20	3335.93	100.65
printtokens-2	CPU	12518.40	146.89	0.00	0.00	8756.17	68.64	3762.23	109.13
printtokens-2	JOMP1	15029.73	383.27	5.77	0.62	11220.00	297.28	3803.97	120.04
printtokens-2	JOMP2	10162.13	140.35	5.73	0.44	5954.07	88.72	4202.33	102.93
printtokens-2	JOMP4	8036.67	131.23	5.80	0.40	3692.87	68.79	4338.00	103.05
printtokens-2	GPGPU	5685.50	146.80	468.90	2.20	1867.03	109.29	3349.57	95.57
printtokens-3	CPU	12335.80	131.26	0.00	0.00	8626.63	60.42	3709.17	102.36
printtokens-3	JOMP1	14965.80	416.84	5.80	0.48	11179.60	297.77	3780.40	156.59
printtokens-3	JOMP2	10185.57	128.75	5.53	0.50	5930.50	73.85	4249.53	102.33
printtokens-3	JOMP4	8086.13	167.18	5.77	0.42	3688.27	84.40	4392.10	112.38
printtokens-3	GPGPU	5784.47	157.36	468.33	4.78	1952.60	127.56	3363.53	82.57
printtokens-4	CPU	12360.47	115.52	0.00	0.00	8670.77	62.58	3689.70	81.87
printtokens-4	JOMP1	14690.43	337.86	5.73	0.44	11004.50	264.70	3680.20	141.57
printtokens-4	JOMP2	10140.37	115.35	5.70	0.46	5944.17	85.72	4190.50	104.89
printtokens-4	JOMP4	8010.47	199.55	5.63	0.48	3673.30	129.08	4331.53	129.24
printtokens-4	GPGPU	5642.50	157.42	467.80	2.91	1883.70	139.35	3291.00	90.03
schedule-1	CPU	7638.67	117.48	0.00	0.00	4439.20	70.69	3199.47	65.11
schedule-1	JOMP1	7871.10	107.36	3.40	0.49	4679.13	86.89	3188.57	82.29
schedule-1	JOMP2	6257.03	90.83	3.47	0.50	2630.67	54.54	3622.90	105.58
schedule-1	JOMP4	5450.87	97.13	3.57	0.50	1740.70	43.13	3706.60	84.68
schedule-1	GPGPU	4893.73	219.03	475.37	2.50	1631.47	213.80	2786.90	82.99
schedule-2	CPU	7745.30	104.94	0.00	0.00	4499.87	39.01	3245.43	80.30
schedule-2	JOMP1	8032.80	113.87	3.53	0.50	4793.07	75.86	3236.20	90.57
schedule-2	JOMP2	6341.13	111.05	3.60	0.49	2676.43	49.23	3661.10	97.28
schedule-2	JOMP4	5504.40	141.96	3.47	0.50	1760.17	57.80	3740.77	104.00
schedule-2	GPGPU	5304.50	112.85	474.90	2.17	2028.83	21.46	2800.77	93.53
schedule-3	CPU	7646.40	124.66	0.00	0.00	4461.60	53.81	3184.80	89.71
schedule-3	JOMP1	7941.90	129.85	3.47	0.50	4715.47	99.74	3222.97	92.20
schedule-3	JOMP2	6251.20	95.49	3.47	0.50	2632.60	39.48	3615.13	96.39
schedule-3	JOMP4	5509.93	125.49	3.60	0.49	1750.40	50.08	3755.93	92.32
schedule-3	GPGPU	5285.13	120.56	474.57	1.56	2026.90	19.19	2783.67	104.82
schedule-4	CPU	7611.70	92.16	0.00	0.00	4430.17	41.45	3181.53	69.16
schedule-4	JOMP1	8033.37	122.39	3.47	0.50	4792.00	92.38	3237.90	96.45
schedule-4	JOMP2	6359.90	85.07	3.63	0.48	2693.93	45.06	3662.33	84.43
schedule-4	JOMP4	5553.03	100.72	3.53	0.50	1771.70	38.32	3777.80	88.11
schedule-4	GPGPU	5307.77	112.28	474.83	1.85	2037.33	20.37	2795.60	96.75
printtokens	CPU	201468.50	1017.39	0.00	0.00	168824.77	933.12	32643.73	217.17
printtokens	JOMP1	264294.97	730.51	12.20	0.40	231541.57	668.03	32741.20	268.03
printtokens	JOMP2	162367.67	368.62	12.47	0.50	124352.20	351.64	38003.00	298.90
printtokens	JOMP4	140384.07	319.11	12.23	0.42	102300.67	184.97	38071.17	242.94
printtokens	GPGPU	44592.67	234.10	470.10	1.35	12097.70	26.65	32024.87	234.80
schedule	CPU	95693.77	607.90	0.00	0.00	74140.63	504.25	21553.13	175.79
schedule	JOMP1	139348.20	609.53	16.73	0.51	117751.40	547.86	21580.07	310.49
schedule	JOMP2	88385.17	383.10	16.53	0.50	63433.77	271.22	24934.87	304.31
schedule	JOMP4	75779.67	584.89	16.63	0.55	50686.53	377.03	25076.50	480.73
schedule	GPGPU	28351.07	324.36	464.73	1.59	6899.33	20.27	20987.00	328.10
printtokens2	CPU	200409.53	1007.63	0.00	0.00	167983.10	860.30	32426.43	256.84
printtokens2	JOMP1	278160.67	788.57	12.57	0.50	245605.30	794.70	32542.80	217.64
printtokens2	JOMP2	169781.93	604.65	12.33	0.54	132011.97	481.70	37757.63	340.18
printtokens2	JOMP4	146077.10	460.93	12.43	0.50	108003.00	325.04	38061.67	335.96
printtokens2	GPGPU	45705.40	221.84	470.67	1.30	13294.90	27.15	31939.83	219.62
schedule2	CPU	88307.47	907.51	0.00	0.00	66683.77	728.58	21623.70	409.08
schedule2	JOMP1	124601.87	585.05	16.33	0.60	102931.23	557.67	21654.30	409.55
schedule2	JOMP2	80791.20	587.41	16.70	0.97	55709.73	231.85	25064.77	491.77
schedule2	JOMP4	69575.73	536.93	16.50	1.06	44214.20	299.20	25345.03	424.66
schedule2	GPGPU	28571.07	359.22	462.53	1.80	6794.00	21.28	21314.53	363.45
tcas	CPU	33098.07	403.64	0.00	0.00	19479.47	335.46	13618.60	126.26
tcas	JOMP1	39282.17	423.11	14.07	1.31	25542.43	408.53	13725.67	178.59
tcas	JOMP2	30021.70	308.38	14.20	1.30	14239.20	215.67	15768.30	189.47
tcas	JOMP4	25391.43	229.51	14.10	1.47	9460.17	189.81	15917.17	242.64
tcas	GPGPU	17099.93	166.20	466.40	2.32	3476.97	22.49	13156.57	163.79
totinfo	CPU	33547.10	414.49	0.00	0.00	23190.27	212.64	10356.83	236.96
totinfo	JOMP1	37089.93	305.02	13.23	1.33	26853.87	285.81	10222.83	146.45
totinfo	JOMP2	26280.27	277.43	13.43	1.17	14567.70	155.58	11699.13	196.05
totinfo	JOMP4	20867.60	208.74	13.70	1.59	8988.33	76.84	11865.57	189.78
totinfo	GPGPU	13409.37	131.48	465.00	3.04	3065.73	28.02	9878.63	132.19
flex	CPU	68898.23	562.97	0.00	0.00	66074.40	554.00	2823.83	60.97
flex	JOMP1	43761.27	1667.94	14.07	0.73	40925.87	1678.87	2821.33	74.69
flex	JOMP2	24966.77	513.57	13.63	0.48	21769.90	509.27	3183.23	60.48
flex	JOMP4	16441.23	232.20	13.90	0.54	13223.03	231.11	3204.30	71.58
flex	GPGPU	10103.20	65.62	465.27	2.31	7225.33	10.47	2412.60	66.89
gzip	CPU	73950.87	959.62	0.00	0.00	70627.90	946.41	3322.97	58.53
gzip	JOMP1	62003.70	1154.83	12.77	0.76	58680.57	1157.14	3310.37	65.29
gzip	JOMP2	34440.27	591.68	12.40	0.55	30655.77	582.96	3772.10	87.57
gzip	JOMP4	22367.40	539.74	12.57	0.80	18535.70	545.24	3819.13	82.22
gzip	GPGPU	9240.03	69.80	463.90	1.70	5831.23	9.23	2944.90	69.46
sed	CPU	124817.33	1976.92	0.00	0.00	120265.57	1930.36	4551.77	72.83
sed	JOMP1	122040.30	1435.61	11.73	0.57	117454.93	1441.36	4573.63	68.62
sed	JOMP2	66612.53	649.39	11.53	0.56	61453.23	645.79	5147.77	82.37
sed	JOMP4	41056.63	385.78	11.47	0.56	35821.10	398.33	5224.07	81.48

sed	GPGPU	12147.77	75.00	467.27	2.14	7498.07	11.22	4182.43	77.26
space-1	CPU	128911.03	3000.98	0.00	0.00	125323.07	2979.24	3587.97	62.34
space-1	JOMP1	72884.27	1545.06	13.27	0.81	69262.87	1538.28	3608.13	74.12
space-1	JOMP2	39989.00	878.97	13.13	0.43	35861.10	859.12	4114.77	89.34
space-1	JOMP4	25293.07	395.23	13.20	0.40	21164.57	392.79	4115.30	86.67
space-1	GPGPU	12270.57	61.69	466.80	1.80	8622.20	10.40	3181.57	64.01
space-2	CPU	126462.67	2652.95	0.00	0.00	122919.20	2592.98	3543.47	87.69
space-2	JOMP1	68066.23	1147.03	13.07	0.44	64527.43	1164.34	3525.73	78.28
space-2	JOMP2	37911.63	594.27	13.17	0.37	33840.57	583.31	4057.90	71.06
space-2	JOMP4	24380.70	555.05	13.00	0.26	20286.67	553.22	4081.03	69.44
space-2	GPGPU	11625.40	68.16	465.10	1.62	8021.33	9.84	3138.97	68.86
space-3	CPU	130576.67	2677.40	0.00	0.00	126974.30	2640.58	3602.37	73.72
space-3	JOMP1	72470.93	1543.13	13.03	0.31	68864.00	1531.11	3593.90	75.07
space-3	JOMP2	39988.90	784.99	13.10	0.54	35870.73	777.28	4105.07	78.43
space-3	JOMP4	25302.80	447.97	13.20	0.40	21153.63	433.21	4135.97	74.92
space-3	GPGPU	12279.10	84.11	466.67	1.94	8622.53	8.05	3189.90	86.12
space-4	CPU	128981.73	3442.49	0.00	0.00	125395.00	3394.39	3586.73	78.57
space-4	JOMP1	73208.10	2310.12	13.10	0.30	69642.43	2325.78	3552.57	61.09
space-4	JOMP2	39689.37	800.83	13.13	0.34	35634.33	818.95	4041.90	91.29
space-4	JOMP4	25216.80	351.18	13.07	0.36	21115.67	332.19	4088.07	82.16
space-4	GPGPU	12233.17	81.10	466.30	1.73	8622.07	10.74	3144.80	80.37
replace	CPU	325246.37	1698.49	0.00	0.00	281927.93	1405.19	43318.43	848.20
replace	JOMP1	445375.07	1524.35	13.30	0.46	402127.67	1236.82	43234.10	835.37
replace	JOMP2	265138.93	1078.85	13.20	0.48	214949.63	672.27	50176.10	848.70
replace	JOMP4	225739.00	892.89	13.20	0.48	175134.70	253.04	50591.10	888.86
replace	GPGPU	61807.93	519.29	472.07	3.92	18291.03	31.20	43044.83	523.68
bash	CPU	2071836.07	29845.30	0.00	0.00	2051591.57	29674.30	20244.50	206.06
bash	JOMP1	1346585.83	16962.75	53.30	1.39	1326396.90	16966.54	20135.63	159.67
bash	JOMP2	715605.03	11951.22	53.37	1.25	693778.67	11979.47	21773.00	150.21
bash	JOMP4	425783.60	5673.48	54.07	1.73	403970.63	5677.03	21758.90	209.60
bash	GPGPU	82574.53	194.61	517.07	2.35	62371.57	13.89	19685.90	189.82
ibm	CPU	4347517.13	462072.40	0.00	0.00	4178547.93	831883.88	168969.20	709728.80
ibm	JOMP1	1445294.57	38625.23	136.07	3.59	1406525.60	39448.65	38632.90	5916.45
ibm	JOMP2	783762.87	20494.64	136.13	3.48	745728.40	20522.99	37898.33	3591.78
ibm	JOMP4	481433.27	11823.74	135.63	3.77	444301.60	11749.65	36996.03	784.43
ibm	GPGPU	174990.80	5095.10	613.67	61.27	136661.40	849.63	37715.73	4931.15

Table 13: Execution time of SPEA2 algotirhm

Subject	Config	$\bar{T}_{total}$	$\sigma_{T_{total}}$	$\bar{T}_{init}$	$\sigma_{T_{init}}$	$\bar{T}_{fitness}$	$\sigma_{T_{fitness}}$	$\bar{T}_{remaining}$	$\sigma_{T_{remaining}}$
printtokens-1	CPU	54737.30	7183.20	0.00	0.00	8562.97	75.10	46174.33	7167.90
printtokens-1	JOMP1	60409.20	7729.38	5.73	0.44	10825.50	121.39	49577.97	7709.26
printtokens-1	JOMP2	59056.57	7209.10	5.73	0.44	5840.13	69.33	53210.70	7212.29
printtokens-1	JOMP4	52838.37	15410.19	5.77	0.42	3571.33	39.62	49261.27	15434.46
printtokens-1	GPGPU	55810.83	7876.04	471.83	14.16	3535.63	35.46	51803.37	7864.52
printtokens-2	CPU	61021.80	9959.34	0.00	0.00	8769.17	68.85	52252.63	9922.10
printtokens-2	JOMP1	61762.40	7379.62	5.80	0.40	11007.90	168.96	50748.70	7391.38
printtokens-2	JOMP2	67094.07	12558.61	5.77	0.42	5971.47	76.16	61116.83	12560.31
printtokens-2	JOMP4	54581.23	15862.40	5.90	0.30	3634.47	68.27	50940.87	15904.60
printtokens-2	GPGPU	56347.47	8010.39	468.90	2.36	3521.87	30.39	52356.70	8007.62
printtokens-3	CPU	55246.60	8305.32	0.00	0.00	8658.43	66.02	46588.17	8299.09
printtokens-3	JOMP1	57619.10	6555.12	5.83	0.37	10936.97	133.77	46676.30	6526.33
printtokens-3	JOMP2	60952.13	11448.43	5.63	0.48	5931.23	59.59	55015.27	11439.49
printtokens-3	JOMP4	57878.43	19310.31	5.67	0.47	3601.83	44.26	54270.93	19337.38
printtokens-3	GPGPU	54563.63	7163.49	467.97	1.83	3530.53	22.73	50565.13	7159.91
printtokens-4	CPU	59433.97	7999.19	0.00	0.00	8692.00	70.85	50741.97	7986.66
printtokens-4	JOMP1	59541.43	6497.08	5.67	0.47	10950.07	156.36	48585.70	6514.42
printtokens-4	JOMP2	64436.50	8461.38	5.77	0.42	5915.53	64.97	58515.20	8458.26
printtokens-4	JOMP4	56524.23	20522.42	5.67	0.47	3590.87	51.32	52927.70	20535.99
printtokens-4	GPGPU	58235.23	5732.37	467.17	2.38	3101.50	570.40	54666.57	5759.73
schedule-1	CPU	103525.27	10751.91	0.00	0.00	4445.07	31.51	99080.20	10752.81
schedule-1	JOMP1	104730.07	13222.87	3.70	0.46	4700.47	109.48	100025.90	13203.26
schedule-1	JOMP2	115522.53	9903.85	3.63	0.48	2618.30	49.96	112900.60	9911.08
schedule-1	JOMP4	100832.93	35751.09	3.77	0.42	1669.10	31.57	99160.07	35770.11
schedule-1	GPGPU	108538.97	13846.59	475.30	1.66	2570.70	165.46	105492.97	13856.20
schedule-2	CPU	111070.57	8120.00	0.00	0.00	4522.30	32.82	106548.27	8118.41
schedule-2	JOMP1	107589.33	8283.42	3.70	0.46	4799.63	61.40	102786.00	8297.07
schedule-2	JOMP2	117569.23	9742.41	3.57	0.50	2684.60	48.63	114881.07	9751.08
schedule-2	JOMP4	106101.20	37473.79	3.40	0.49	1694.33	39.42	104403.47	37495.24
schedule-2	GPGPU	110804.83	9195.29	476.23	2.68	2616.37	10.64	107712.23	9196.71
schedule-3	CPU	67744.33	8141.45	0.00	0.00	4479.80	34.50	63264.53	8158.15
schedule-3	JOMP1	71534.80	8905.75	3.67	0.47	4676.60	68.23	66854.53	8912.32
schedule-3	JOMP2	77546.20	9676.54	3.47	0.50	2624.80	59.67	74917.93	9667.77
schedule-3	JOMP4	67597.97	24248.38	3.60	0.49	1665.37	35.18	65929.00	24263.71
schedule-3	GPGPU	71874.93	8807.73	474.87	1.48	2621.33	8.62	68778.73	8808.51
schedule-4	CPU	83107.30	9619.47	0.00	0.00	4449.13	40.74	78658.17	9599.56
schedule-4	JOMP1	83825.20	11900.34	3.67	0.47	4780.03	83.15	79041.50	11903.98
schedule-4	JOMP2	93997.57	13035.56	3.57	0.50	2658.10	56.89	91335.90	13040.25
schedule-4	JOMP4	83453.57	30179.48	3.73	0.44	1682.83	32.68	81767.00	30193.44
schedule-4	GPGPU	88935.70	9241.76	474.93	2.06	2622.30	10.36	85838.47	9240.44
printtokens	CPU	218854.33	1726.93	0.00	0.00	168196.60	1514.85	50657.73	327.32
printtokens	JOMP1	287307.47	1127.93	12.37	0.55	236604.07	929.05	50691.03	355.26
printtokens	JOMP2	186358.63	576.28	12.47	0.85	127034.93	346.14	59311.23	386.97
printtokens	JOMP4	164231.80	2976.55	12.20	0.40	103544.77	472.16	60674.83	2558.07
printtokens	GPGPU	62718.07	617.76	470.47	1.50	12305.87	27.83	49941.73	617.07
schedule	CPU	108266.97	707.22	0.00	0.00	73895.10	578.85	34371.87	213.91
schedule	JOMP1	152762.60	743.22	16.60	0.49	118391.37	599.61	34354.63	549.03
schedule	JOMP2	104064.37	734.29	16.43	0.50	63909.73	307.38	40138.20	679.72
schedule	JOMP4	91078.00	2702.94	16.57	0.50	50229.77	412.43	40831.67	2467.27
schedule	GPGPU	41389.13	616.10	464.77	1.12	7065.23	24.31	33859.13	621.14
printtokens2	CPU	218065.40	885.64	0.00	0.00	167678.67	670.63	50386.73	385.17
printtokens2	JOMP1	298648.77	940.76	12.43	0.62	248103.87	788.46	50532.47	367.08
printtokens2	JOMP2	192170.00	478.57	12.47	0.50	132948.23	405.78	59209.30	293.85
printtokens2	JOMP4	168897.87	2981.31	12.17	0.37	108599.40	488.40	60286.30	2579.16
printtokens2	GPGPU	63975.20	273.85	485.47	79.59	13622.70	27.21	49867.03	313.69
schedule2	CPU	101383.53	1208.80	0.00	0.00	67082.43	1167.92	34301.10	624.25
schedule2	JOMP1	138615.47	2661.56	16.73	1.09	103845.77	1939.76	34752.97	918.69
schedule2	JOMP2	95390.57	928.84	16.67	1.11	55567.10	200.16	39806.80	881.44
schedule2	JOMP4	85278.97	2484.97	16.37	0.60	44539.50	439.27	40723.10	2316.45
schedule2	GPGPU	41635.23	648.52	462.50	1.41	6981.63	23.46	34191.10	641.56
tcas	CPU	41985.83	338.61	0.00	0.00	19457.23	195.99	22528.60	207.74
tcas	JOMP1	48657.30	796.67	14.10	1.37	25970.57	431.74	22672.63	412.30
tcas	JOMP2	40586.10	295.08	13.70	1.16	14255.30	214.12	26317.10	227.65
tcas	JOMP4	37024.63	2397.64	14.30	1.00	9625.57	222.57	27384.77	2238.37
tcas	GPGPU	26093.17	226.88	465.57	2.73	3544.30	63.69	22083.30	188.19
totinfo	CPU	40370.83	424.62	0.00	0.00	23053.97	205.96	17316.87	250.47
totinfo	JOMP1	44349.27	366.13	13.47	1.33	27013.60	233.96	17322.20	259.02
totinfo	JOMP2	34824.07	391.81	13.10	1.11	14644.00	189.26	20166.97	285.14
totinfo	JOMP4	29982.03	1869.13	13.63	1.76	8985.83	155.63	20982.57	1745.97
totinfo	GPGPU	20475.40	242.92	464.03	1.96	3071.80	56.35	16939.57	222.17
flex	CPU	85954.07	1862.67	0.00	0.00	67851.50	638.68	18102.57	1641.89
flex	JOMP1	57963.13	1777.55	13.83	0.64	40302.90	1188.90	17646.40	1186.56
flex	JOMP2	41961.70	1604.81	13.90	0.83	21351.73	522.14	20596.07	1538.22
flex	JOMP4	32708.03	4129.65	14.30	0.59	12716.27	304.51	19977.47	4062.74
flex	GPGPU	26750.37	1510.63	465.13	1.96	7324.43	28.65	18960.80	1490.76
gzip	CPU	90650.70	1740.26	0.00	0.00	73402.77	908.02	17247.93	1519.11
gzip	JOMP1	78942.40	2740.94	13.37	2.85	60347.33	1754.09	18581.70	1995.90
gzip	JOMP2	51022.47	1655.06	12.67	0.54	30961.97	595.97	20047.83	1539.96
gzip	JOMP4	38451.73	4173.62	12.40	0.49	18194.50	454.46	20244.83	4111.11
gzip	GPGPU	25936.73	1699.42	464.23	2.03	5960.30	22.46	19512.20	1683.50
sed	CPU	136519.00	1517.32	0.00	0.00	123618.43	1653.37	12900.57	1057.79
sed	JOMP1	129451.17	1686.10	11.53	0.62	116950.97	1566.68	12488.67	902.30
sed	JOMP2	75734.37	1315.20	11.73	0.44	61552.53	469.44	14170.10	1074.68
sed	JOMP4	50505.77	1092.84	11.73	0.44	35754.93	437.79	14739.10	1097.59
sed	GPGPU	20424.73	1333.13	467.80	2.20	7508.57	16.12	12448.37	1322.58

space-1	CPU	144209.27	2733.97	0.00	0.00	130875.73	2677.66	13333.53	327.63
space-1	JOMP1	80985.53	1312.87	13.13	0.34	67619.57	1327.43	13352.83	244.88
space-1	JOMP2	51000.47	761.16	13.07	0.25	35512.43	712.74	15474.97	274.02
space-1	JOMP4	36331.47	1763.56	13.17	0.37	21285.83	526.62	15032.47	1454.21
space-1	GPGPU	22959.10	424.20	467.00	4.86	8677.07	8.74	13815.03	424.17
space-2	CPU	140650.20	2961.17	0.00	0.00	127750.53	2992.23	12899.67	275.49
space-2	JOMP1	77375.50	1296.40	12.97	0.18	64315.27	1264.78	13047.27	253.93
space-2	JOMP2	48832.40	578.12	13.13	0.34	33724.50	528.94	15094.77	294.24
space-2	JOMP4	34990.77	1657.58	13.00	0.58	20272.27	444.72	14705.50	1387.08
space-2	GPGPU	21956.33	388.68	466.50	1.86	8069.80	11.64	13420.03	390.20
space-3	CPU	146322.57	2154.38	0.00	0.00	133057.60	2094.56	13264.97	232.56
space-3	JOMP1	81090.77	1440.19	13.13	0.43	67842.80	1410.05	13234.83	252.39
space-3	JOMP2	51179.13	956.43	13.10	0.40	35734.40	895.88	15431.63	274.58
space-3	JOMP4	36106.43	1604.97	13.13	0.50	21230.63	421.11	14862.67	1428.65
space-3	GPGPU	22679.07	279.30	466.53	2.01	8672.27	8.97	13540.27	277.75
space-4	CPU	143502.33	3316.32	0.00	0.00	129979.30	3294.07	13523.03	307.61
space-4	JOMP1	81009.73	1379.42	13.07	0.25	67473.00	1373.53	13523.67	273.02
space-4	JOMP2	50280.77	1640.55	13.13	0.34	35278.87	550.71	14988.77	1489.76
space-4	JOMP4	36162.23	1593.24	12.93	0.25	20978.10	366.36	15171.20	1510.98
space-4	GPGPU	23215.27	480.33	465.63	1.52	8675.47	8.98	14074.17	479.95
replace	CPU	348104.93	2187.83	0.00	0.00	281450.50	1536.71	66654.43	1342.59
replace	JOMP1	469796.70	1506.10	13.40	0.55	403195.30	980.27	66588.00	1178.14
replace	JOMP2	292941.60	3840.15	13.17	0.37	215710.27	774.62	77218.17	3352.11
replace	JOMP4	254205.57	3788.56	13.13	0.34	175560.03	745.43	78632.40	3260.21
replace	GPGPU	85664.63	735.20	471.57	2.19	18829.70	31.29	66363.37	736.89
bash	CPU	2130058.33	26423.09	0.00	0.00	2099565.87	26226.12	30492.47	296.06
bash	JOMP1	1363130.33	25211.50	53.93	1.57	1332820.57	25252.03	30255.83	282.71
bash	JOMP2	727041.77	9231.70	53.60	1.54	692904.27	9237.17	34083.90	1158.28
bash	JOMP4	436764.23	6370.28	54.33	1.19	402476.50	6614.83	34233.40	1205.63
bash	GPGPU	92768.33	298.38	516.80	1.74	62381.57	12.13	29869.97	294.00
ibm	CPU	4605875.50	460148.49	0.00	0.00	4558144.60	456561.49	47730.90	3707.93
ibm	JOMP1	1472382.37	43731.58	137.50	3.70	1423957.80	43151.02	48287.07	4840.65
ibm	JOMP2	805397.90	19822.93	136.53	3.79	755478.63	19194.49	49782.73	5430.14
ibm	JOMP4	496257.47	15594.50	135.97	2.96	447501.90	14863.69	48619.60	1572.93
ibm	GPGPU	187426.13	6736.59	645.33	157.36	136904.93	1187.38	49875.87	6015.35

Table 14: Execution time of Two Archive algotirhm

Subject	Config	$\bar{T}_{total}$	$\sigma_{T_{total}}$	$\bar{T}_{init}$	$\sigma_{T_{init}}$	$\bar{T}_{fitness}$	$\sigma_{T_{fitness}}$	$\bar{T}_{remaining}$	$\sigma_{T_{remaining}}$
printtokens-1	CPU	9126.60	126.85	0.00	0.00	7742.80	102.28	1383.80	45.34
printtokens-1	JOMP1	12467.17	88.56	5.67	0.47	11116.57	184.10	1344.93	132.62
printtokens-1	JOMP2	7658.47	132.18	5.43	0.50	6122.23	50.27	1530.80	155.90
printtokens-1	JOMP4	5439.40	158.63	5.83	0.37	3862.43	36.76	1571.13	163.76
printtokens-1	GPGPU	3514.80	115.19	471.27	16.17	2121.33	101.65	922.20	33.19
printtokens-2	CPU	9397.17	116.54	0.00	0.00	7966.17	97.71	1431.00	31.85
printtokens-2	JOMP1	12594.73	166.41	5.60	0.49	11217.67	103.84	1371.47	131.01
printtokens-2	JOMP2	7799.87	155.56	5.70	0.46	6231.20	46.99	1562.97	157.90
printtokens-2	JOMP4	5536.53	142.51	5.80	0.40	3914.27	36.16	1616.47	156.45
printtokens-2	GPGPU	3619.00	132.69	468.73	2.61	2173.53	112.02	976.73	32.15
printtokens-3	CPU	9070.03	107.27	0.00	0.00	7711.27	100.75	1358.77	19.02
printtokens-3	JOMP1	12474.47	147.29	5.70	0.46	11166.27	108.31	1302.50	126.92
printtokens-3	JOMP2	7678.93	128.32	5.67	0.60	6174.80	36.77	1498.47	138.76
printtokens-3	JOMP4	5454.77	158.54	5.33	0.47	3895.80	31.81	1553.63	147.48
printtokens-3	GPGPU	3480.23	91.11	469.60	6.18	2110.93	78.05	899.70	22.23
printtokens-4	CPU	9266.03	103.05	0.00	0.00	7881.93	89.80	1384.10	29.26
printtokens-4	JOMP1	12518.37	159.55	5.70	0.46	11195.47	83.36	1317.20	147.05
printtokens-4	JOMP2	7689.80	146.34	5.53	0.50	6165.00	48.71	1519.27	170.63
printtokens-4	JOMP4	5458.00	161.15	5.57	0.50	3874.77	35.22	1577.67	162.46
printtokens-4	GPGPU	3523.97	131.34	470.67	8.37	2123.60	119.18	929.70	30.61
schedule-1	CPU	6075.77	84.26	0.00	0.00	4843.37	63.16	1232.40	42.59
schedule-1	JOMP1	5998.63	174.12	3.50	0.50	4818.23	56.72	1176.90	162.28
schedule-1	JOMP2	4110.47	228.08	3.27	0.44	2776.67	54.51	1330.53	186.69
schedule-1	JOMP4	3240.50	239.13	3.53	0.50	1865.17	49.68	1371.80	196.83
schedule-1	GPGPU	2794.13	38.75	475.10	2.68	1574.63	14.94	744.40	37.45
schedule-2	CPU	6127.60	64.40	0.00	0.00	4890.77	53.02	1236.83	21.30
schedule-2	JOMP1	6126.73	167.85	3.37	0.48	4944.30	46.59	1179.07	163.06
schedule-2	JOMP2	4184.70	212.05	3.53	0.50	2824.23	37.85	1356.93	195.68
schedule-2	JOMP4	3282.10	239.99	3.57	0.62	1883.30	50.56	1395.23	195.09
schedule-2	GPGPU	2794.80	17.58	475.23	4.42	1569.53	13.10	750.03	15.16
schedule-3	CPU	5937.73	86.90	0.00	0.00	4756.67	61.36	1181.07	32.17
schedule-3	JOMP1	5985.37	180.29	3.33	0.47	4842.53	67.39	1139.50	150.55
schedule-3	JOMP2	4080.90	210.65	3.40	0.49	2781.77	42.53	1295.73	177.25
schedule-3	JOMP4	3182.80	231.91	3.40	0.49	1859.47	57.61	1319.93	179.05
schedule-3	GPGPU	2750.83	34.19	475.00	4.06	1587.83	16.11	688.00	38.95
schedule-4	CPU	6041.80	170.24	0.00	0.00	4818.93	130.01	1222.87	45.36
schedule-4	JOMP1	6078.70	177.02	3.50	0.50	4912.13	59.95	1163.07	164.25
schedule-4	JOMP2	4156.27	222.57	3.57	0.50	2801.57	59.08	1351.13	197.12
schedule-4	JOMP4	3247.27	232.23	3.20	0.40	1881.07	47.90	1363.00	192.78
schedule-4	GPGPU	2807.13	37.09	474.17	1.37	1590.10	23.66	742.87	49.50
printtokens	CPU	116826.10	624.55	0.00	0.00	106223.67	599.05	10602.43	71.58
printtokens	JOMP1	249776.57	1050.07	12.03	0.18	239145.43	1066.37	10619.10	95.66
printtokens	JOMP2	142503.97	679.32	12.13	0.34	130157.10	739.65	12334.73	193.63
printtokens	JOMP4	118927.97	554.48	12.07	0.25	106493.77	563.52	12422.13	160.67
printtokens	GPGPU	25521.53	142.94	471.53	2.93	14880.73	113.84	10169.27	60.71
schedule	CPU	62042.50	865.19	0.00	0.00	55003.57	828.49	7038.93	47.23
schedule	JOMP1	126570.23	554.57	14.63	0.80	119589.93	564.16	6965.67	120.73
schedule	JOMP2	73743.40	367.67	14.73	1.00	65575.97	380.06	8152.70	142.48
schedule	JOMP4	60463.40	441.32	14.67	0.65	52194.73	499.49	8254.00	210.47
schedule	GPGPU	15748.17	124.66	466.57	6.87	8648.73	81.42	6632.87	68.09
printtokens2	CPU	123929.63	1103.11	0.00	0.00	113296.40	1078.22	10633.23	35.64
printtokens2	JOMP1	261326.87	910.11	12.13	0.34	250733.73	981.87	10581.00	198.33
printtokens2	JOMP2	148466.10	470.87	12.00	0.00	136103.07	525.71	12351.03	168.36
printtokens2	JOMP4	123841.30	433.05	12.07	0.25	111424.20	436.40	12405.03	241.67
printtokens2	GPGPU	26748.17	133.66	471.20	2.01	16101.30	109.75	10175.67	53.47
schedule2	CPU	55375.07	707.72	0.00	0.00	48138.83	681.86	7236.23	67.16
schedule2	JOMP1	111285.10	737.20	14.33	0.47	104218.50	346.99	7052.27	618.16
schedule2	JOMP2	65829.53	287.60	14.90	1.11	57524.27	354.33	8290.37	201.67
schedule2	JOMP4	54595.20	419.74	14.87	1.45	46196.63	466.64	8383.70	246.43
schedule2	GPGPU	15855.27	70.13	464.43	2.63	8644.97	73.30	6745.87	41.31
tcas	CPU	20871.43	90.69	0.00	0.00	16171.60	71.28	4699.83	38.20
tcas	JOMP1	31284.30	245.13	13.93	1.48	26661.40	308.60	4608.97	213.03
tcas	JOMP2	20876.13	183.96	13.37	1.25	15493.03	253.82	5369.73	223.50
tcas	JOMP4	16128.80	224.15	13.27	0.81	10689.37	139.32	5426.17	289.61
tcas	GPGPU	9308.23	166.32	472.33	25.73	4563.17	142.94	4272.73	90.77
totinfo	CPU	20730.80	484.32	0.00	0.00	17584.63	476.84	3146.17	28.83
totinfo	JOMP1	30592.67	208.11	12.93	0.85	27496.30	182.47	3083.43	129.72
totinfo	JOMP2	18946.33	155.43	13.40	1.69	15360.80	170.58	3572.13	170.04
totinfo	JOMP4	13462.13	275.21	12.67	0.54	9813.60	203.27	3635.87	171.23
totinfo	GPGPU	6940.73	38.39	465.30	2.15	3762.67	33.53	2712.77	28.69
flex	CPU	71001.13	549.11	0.00	0.00	70153.80	541.84	847.33	30.53
flex	JOMP1	41637.27	1145.73	13.30	0.46	40827.47	1149.14	796.50	133.08
flex	JOMP2	22405.10	745.27	13.20	0.40	21483.07	739.98	908.83	158.59
flex	JOMP4	13882.63	412.73	13.30	0.46	12958.20	344.10	911.13	159.68
flex	GPGPU	8171.10	24.95	464.70	1.62	7274.80	8.56	431.60	25.19
gzip	CPU	57868.43	684.16	0.00	0.00	56558.87	678.22	1309.57	29.27
gzip	JOMP1	59787.40	858.20	12.97	0.75	58544.00	859.46	1230.43	151.80
gzip	JOMP2	32495.43	697.43	12.83	0.37	31049.13	669.21	1433.47	160.91
gzip	JOMP4	19922.97	283.05	12.70	0.46	18453.80	232.66	1456.47	172.53
gzip	GPGPU	7342.70	23.51	465.40	1.99	6017.30	16.96	860.00	20.26
sed	CPU	101820.83	2056.00	0.00	0.00	100210.43	2055.16	1610.40	34.95
sed	JOMP1	119835.67	1005.12	11.27	0.44	118254.57	1027.73	1569.83	121.53
sed	JOMP2	63544.80	665.11	11.23	0.42	61734.17	690.43	1799.40	115.92
sed	JOMP4	38235.60	431.45	11.17	0.37	36391.20	376.47	1833.23	124.44

sed	GPGPU	9386.03	28.80	467.40	1.17	7753.57	11.73	1165.07	26.66
space-1	CPU	124296.60	1476.27	0.00	0.00	122756.60	1459.08	1540.00	24.09
space-1	JOMP1	69564.13	1057.82	13.30	0.46	68097.53	989.21	1453.30	166.96
space-1	JOMP2	37794.93	631.99	13.70	0.46	36065.37	571.77	1715.87	133.71
space-1	JOMP4	23342.97	498.70	13.50	0.50	21573.17	465.26	1756.30	121.08
space-1	GPGPU	10346.67	35.57	467.60	3.10	8763.73	10.90	1115.33	39.34
space-2	CPU	121329.57	1983.92	0.00	0.00	119826.43	1967.27	1503.13	25.03
space-2	JOMP1	66364.53	1008.83	13.70	0.46	64902.90	923.76	1447.93	160.68
space-2	JOMP2	35846.80	494.02	13.40	0.49	34136.60	466.86	1696.80	154.29
space-2	JOMP4	21940.43	360.38	13.37	0.48	20221.23	257.62	1705.83	199.13
space-2	GPGPU	9700.07	25.03	466.83	1.55	8147.17	9.38	1086.07	27.75
space-3	CPU	126114.77	2653.15	0.00	0.00	124586.27	2628.86	1528.50	34.71
space-3	JOMP1	70485.43	912.41	13.63	0.48	69016.53	918.78	1455.27	153.50
space-3	JOMP2	37861.03	440.74	13.47	0.50	36182.00	490.41	1665.57	185.65
space-3	JOMP4	22984.63	462.90	13.63	0.48	21253.93	419.50	1717.07	208.97
space-3	GPGPU	10327.40	29.02	467.47	1.89	8763.63	13.79	1096.30	29.10
space-4	CPU	123416.00	1596.55	0.00	0.00	121890.70	1580.69	1525.30	25.12
space-4	JOMP1	69634.33	1280.05	13.50	0.50	68162.20	1222.59	1458.63	124.06
space-4	JOMP2	37268.80	670.57	13.67	0.47	35554.63	620.05	1700.50	143.71
space-4	JOMP4	22738.10	273.34	13.40	0.49	21004.40	227.08	1720.30	183.28
space-4	GPGPU	10341.97	31.83	467.63	3.31	8765.67	9.73	1108.67	33.49
replace	CPU	197579.20	4653.83	0.00	0.00	183648.37	4512.88	13930.83	175.74
replace	JOMP1	420619.97	906.13	13.27	0.44	406724.80	864.00	13881.90	276.66
replace	JOMP2	236038.03	814.39	13.40	0.49	219521.77	557.41	16502.87	632.13
replace	JOMP4	195827.97	468.77	13.33	0.60	179560.27	376.03	16254.37	300.60
replace	GPGPU	36315.70	322.90	473.40	2.50	22125.53	179.07	13716.77	256.86
bash	CPU	1193310.07	18851.34	0.00	0.00	1188921.17	18790.88	4388.90	69.23
bash	JOMP1	1354828.80	20224.41	53.00	0.97	1350286.23	20221.98	4489.57	414.13
bash	JOMP2	705998.83	7061.49	53.57	0.92	700673.77	7103.10	5271.50	741.65
bash	JOMP4	413619.80	6377.16	54.10	1.64	408339.13	6412.11	5226.57	471.50
bash	GPGPU	67379.67	81.06	518.43	9.40	63013.40	26.44	3847.83	71.51
ibm	CPU	2916323.63	46962.47	0.00	0.00	2914980.77	46967.53	1342.87	35.61
ibm	JOMP1	1414298.60	50912.92	136.37	3.06	1413007.20	50902.83	1155.03	156.13
ibm	JOMP2	754894.63	20118.08	135.70	2.90	753402.47	20084.07	1356.47	147.91
ibm	JOMP4	446434.20	13787.74	133.87	3.02	444931.47	13775.29	1368.87	171.03
ibm	GPGPU	139075.90	84.86	622.10	65.31	137737.47	84.69	716.33	74.27