REPUBLIC OF TURKEY
YILDIZ TECHNICAL UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

GRAMMATICAL GENETIC PROGRAMMING ON HETEROGENEOUS COMPUTING PLATFORMS

HAKAN AYRAL

Ph.D. THESIS
DEPARTMENT OF COMPUTER ENGINEERING
PROGRAM OF COMPUTER ENGINEERING

ADVISER
ASSOC. PROF. DR. SONGÜL ALBAYRAK

İSTANBUL, 2017

REPUBLIC OF TURKEY

YILDIZ TECHNICAL UNIVERSITY

GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

GRAMMATICAL GENETIC PROGRAMMING ON HETEROGENEOUS COMPUTING PLATFORMS

A thesis submitted by Hakan AYRAL in partial fulfillment of the requirements for the degree of DOCTOR OF PHILOSOPHY is approved by the committee on 15.08.2017 in Department of Computer Engineering, Computer Engineering Program.

Thesis Adviser

Assoc. Prof. Dr. Songül ALBAYRAK

Yıldız Technical University

Approved By the Examining Committee

Assoc. Prof. Dr. Songül ALBAYRAK

Yıldız Technical University _____

Assoc. Prof. Dr. M. Fatih AMASYALI

Yıldız Technical University _____

Asst. Prof. Dr. Murat Can GANİZ

Marmara University _____

Assoc. Prof. Dr. A. Gökhan YAVUZ

Yıldız Technical University _____

Prof. Dr. A. Muhammed ULUDAĞ

Galatasaray University _____

This work was supported in part by Turkcell Academy.

ACKNOWLEDGEMENTS

First and foremost, I want to thank my family for their constant support during the years this work took, and many more before that.

I'm grateful to Assoc. Prof. Songül Albayrak for accepting me as a graduate student at an unexpected time. Her continuous support, guidance and planning kept this work on track.

I'm in debt to late Prof. Ahmet Coşkun Sönmez for encouraging and accepting me to pursue a doctoral degree. Our shared interests from many domains of science have been an inspiration since my undergraduate years.

I'm thankful to Assoc. Prof. M. Fatih Amasyalı and Asst. Prof. Murat Can Ganiz for the support and the time they spared for this thesis over many years.

Last but not least, I want to thank Prof. A. Muhammed Uludağ for including me in his research and collaborations which was invaluable for the realization of this work.

Dedicated to the memory of Professor Ahmet Coşkun Sönmez.

August, 2017

Hakan AYRAL

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

BNF	Backus-Naur Form
CUBIN	CUDA Binary
FPGA	Field Programmable Gate Array
GE	Grammatical Evolution
GP	Genetic Programming
GPU	Graphics Processing Unit
IPC	Inter-process communication
JIT	Just-in-time (as in JIT compilation)
LNCS	Lecture Notes on Computer Science
MIMD	Multiple Instruction Multiple Data
NaN	Not a Number
NVCC	NVidia CUDA Compiler
PBIL	Population Based Incremental Learning
RMSE	Root Mean Squared Error
SIMD	Single Instruction Multiple Data
SoC	System on a Chip
TLB	Translation Look-aside Buffer

LIST OF FIGURES

LIST OF TABLES

Page

ABSTRACT

GRAMMATICAL GENETIC PROGRAMMING ON HETEROGENOUS COMPUTING PLATFORMS

Hakan AYRAL

Department of Computer Engineering

Ph.D. Thesis

Adviser: Assoc. Prof. Dr. Songül ALBAYRAK

Genetic programming is a population based, stochastic global optimization algorithm which aims to find “programs” fulfilling certain behavioral specifications provided as test cases. The programs may be as simple as arithmetic expressions on some variables or complex as a complete function involving state variables, loops and conditionals. Grammatical genetic programming (a.k.a. grammatical evolution) is a grammar based variant of genetic programming where the search space and methodology is modified to limit the search space to members of a language defined by a formal grammar. This approach provides the benefit of having a smaller search space, and of all the candidates having a valid syntactic form in respect to defined grammar. Grammars used are generally customized to the program at hand, but they can be more generic when the form of the solution is not obvious, which usually is the case with general program synthesis, in contrast to categorical problems like regression, classification, path finding and boolean problems.

Conventional computing platforms consist of homogenous processor hardware, like the case of a computer with multi-core multi-processor setup, where each core is identical to others. On the other hand a heterogeneous computing platform simultaneously uses multiple types of processing elements each with their own unique architecture and strengths. The most prevalent example today is the combined use of CPU and GPU, where CPU provides high frequency, high complexity processing cores

with deep pipelining and large local cache per core, while the GPU provides vast numbers (*in the order of thousands*) of simpler cores with lower frequency each consisting of some ALU and register file for state keeping, but missing per core cache or control units, which have to be shared by groups of 32 cores.

The aim of this dissertation is to investigate and propose new methods to accelerate grammatical genetic programming by parallelization on mentioned types of computing platforms.

In this dissertation we first present an overview of a recent shift in benchmarking practices for genetic programming and propose a new benchmark problem targeting general program synthesis. Then we investigate the interaction of some evolutionary parameters which ultimately affect the parallelization design for grammatical genetic programming. Afterwards we present a new technique which first brings the GPU compilation of individuals in-process, then further parallelizes this compilation. The method we present achieves an order of magnitude faster compilation speed over prior work on literature. Lastly we present a GPU based interpreter for grammatical genetic programming with arbitrary grammars in order to target general program synthesis; this is the first work in literature to present grammatical evolution on GPU with a general purpose interpreter to accommodate arbitrary grammars.

Keywords: Genetic programming, grammatical evolution, heterogeneous computing, gpu, parallel computing

ÖZET

HETEROJEN HESAPLAMA PLATFORMLARI İÇİN GRAMER YÖNLENDİRMELİ GENETİK PROGRAMLAMA

Hakan AYRAL

Bilgisayar Mühendisliği Anabilim Dalı

Doktora Tezi

Tez Danışmanı: Doç. Dr. Songül ALBAYRAK

Genetik programlama nüfus tabanlı, stokastik bir küresel optimizasyon algoritması olup, test-vakaları şeklinde temsil edilmiş davranış spesifikasyonlarını gerçekleyen "programlar" bulmayı amaçlamaktadır. Programlar, değişkenler ve sabitler kullanan bir aritmetik ifade kadar basit olabilir, veya durum değişkenleri, döngüler ve koşullar içeren tam bir fonksiyon kadar karmaşık olabilir. Gramer yönlendirmeli genetik programlama, genetik programlamanın gramer temelli bir alt türüdür; burada arama uzayı ve arama metodolojisi, arama uzayını formel bir gramer tarafından tanımlanan dilin üyeleri ile sınırlandıracak şekildedir. Bu yaklaşım, arama uzayının küçültülmesini ve tüm adayların, yazımsal (syntactic) açıdan geçerli bir sözdizimsel biçimine sahip olmasını sağlar. Kullanılan formel gramer cevabı evrimleştirilmek istenen soruya özel olarak tasarlanır, ancak aranan cevabın şeklinin yeterince bilinmediği problemlerde daha genel bir gramer kullanılabilir; bu durumla benzetim, sınıflandırma, yol bulma ve ikilik fonksiyonlar gibi kategorik problemlerin aksine daha çok genel amaçlı program sentezi ile ilgili problemlerde karşılaşılır.

Konvansiyonel hesap platformları, her çekirdeğin diğerleriyle aynı olduğu, çok çekirdekli ve çok işlemcili bilgisayarlar gibi homojen bir hesap donanımına dayanır. Öte

yandan, heterojen bir hesap platformu, her biri kendi mimarisi ve avantajlarıyla birlikte birden çok hesaplama elemanı kullanır. Bunun en yaygın örneği CPU'ların her biri derin bir pipeline mimarisi ve büyük cache'lere sahip yüksek karmaşıklık ve yüksek saat hızlarında çalışan çekirdekleri ile, GPU'nun çok sayıda *(binler mertebesinde)* ama daha düşük saat hızında ve daha basit, sadece sınırlı miktarda ALU ve register içeren, kontrol ünitesi ve önbelleği 32'li gruplar halinde paylaşmak zorunda olduğu çekirdeklerin bir arada kullanılmasıdır.

Bu tezin temel amacı, bahsedilen tür heterojen hesap platformlarında paralelleştirme yoluyla gramer genetik programlamanın hızlandırılması için yeni yöntemler araştırmak ve önermektir.

Bu tez çalışmasında öncelikle genetik programlama için benchmark uygulamalarında yakın zamanda yaşanan bir kaymaya genel bir bakış sunuyoruz, ve genel amaçlı program sentezi sınıfında yeni bir benchmark problemi öneriyoruz. Ardından, gramer genetik programlama için paralelleştirme tasarımını etkileyen bazı evrimsel parametrelerin bağımlılıklarını ve etkileşimlerini değerlendiriyoruz. Daha sonra, genetik programlama bireylerinin GPU için derlenmesini önce işleç içine çeken ve ardından bunu da paralelleştirmeye dayanan yeni bir yöntem sunuyoruz. Sunduğumuz bu yöntemin literatürdeki önceki çalışmalara kıyasla bir mertebe daha yüksek derleme hızları sağladığı görülmektedir. Son olarak önceden sabitlenmemiş bir gramer ile gramer tabanlı genetik programlama gerçekleştirebilmek ve bu sayede genel amaçlı program sentezi problemlerinde GPU'yu daha verimli kullanabilmek için, GPU üstünde çalışan genel amaçlı bir yorumlayıcı sunuyoruz; bu, literatürde GPU üstünde gramer tabanlı evrim için yorumlama yöntemini önceden sabitlenmemiş herhangi bir gramer ile kullanabilen ilk çalışma olma özelliğini taşımaktadır.

Anahtar Kelimeler: Genetik programlama, gramer yönlendirmeli evrim, heterojen hesaplama, gpu, paralel hesaplama

CHAPTER 1

INTRODUCTION

1.1 Genetic Programming on a Broader Context

Genetic programming is an evolutionary computation technique consisting of multiple evolutionary algorithms, where the objective is to find a “program” (i.e. a simple expression, a sequence of statements, or a full-scale function) that satisfy a behavioral specification expressed in terms of test-cases along with expected outputs. To put GP on perspective, let us briefly present evolutionary computation.

1.1.1 Evolutionary Computation and Evolutionary Algorithms

Evolutionary computation is a family of global optimization algorithms based on how biological evolution works. It is considered part of soft computing and artificial intelligence. Algorithms belonging to evolutionary computation are population based and stochastic in nature. Like other population based techniques, evolutionary computation tracks multiple solution candidates simultaneously; these candidates are modified and replaced with each iteration which is called a generation. At each generation, candidate solutions with low fitness are stochastically removed, and new candidates to replace them are produced, using specific methods to combine parts of high fitness candidates along with small random perturbations. Fitness is a problem specific function which is defined to measure the distance to the correct solution, similar to a heuristic.

Most evolutionary algorithms which are part of evolutionary computing are easy to implement but exhibit behaviors complex to analyze. Evolutionary search algorithms are driven towards the solution by the problem's fitness function; it provides guidance to the search in a heuristic way, such that it measures the closeness to the solution but doesn't necessarily provide information on how to get closer.

Evolutionary computation not only deals with the evolution of soft solutions like the parameter sets, configurations, or software code (in the case of GP), but it's also used to evolve hardware in form of digital or analog circuits, which is the sub-field called evolutionary hardware design, also known as evolvable hardware. One of the most popular example of evolutionarily designed hardware are the antennas evolved for NASA's Space Technology 5 mission, which were produced and flown in 2005 (Figure2.1). This is the first evolved hardware ever launched to space; the design was competitive to the human-designed model and outperformed it in metrics such as mass and cost.

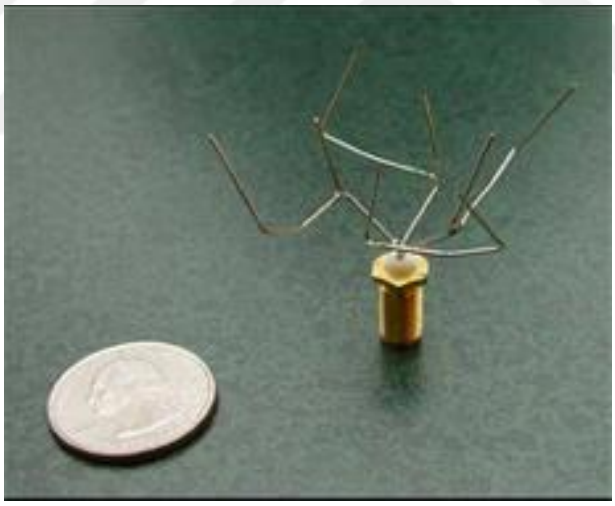


Figure 1.1 Antenna Evolved for NASA Space Flight[1]

Most evolvable hardware projects focus on optimization of digital circuits like digital filters[1], fault tolerant circuits [2]–[4], adders, multipliers and multiplexers [5]–[7]; but it is employed to evolve analog circuits[8] too.

[1] Historical web page: https://www.nasa.gov/centers/ames/research/technology-onepagers/evolvable_systems.html

1.1.2 Genetic Programming

Genetic programming is an evolutionary computation technique; where the objective is to find a "program" that satisfy a behavioral specification expressed as a set of test cases where each member is a (test-case, expected result) pair. Specifically it is a population based evolutionary search algorithm, and the evolved objects may belong to different scopes and scales (i.e. a simple expression, a sequence of statements, a decision tree, or a full-scale function).

1.1.3 Terminology Borrowed From Biology

Genetic programming and evolutionary computing in general borrow heavily from terminology pertaining to biological evolution. Some of these terms along with their meaning in genetic programming context are as follows.

1.1.3.1 Individual & Population

Like most evolutionary algorithms, genetic programming is a population based global optimization technique. Therefore at any point multiple possible solution candidates are pursued. Each candidate is called an individual and the set of all individuals is called the population.

1.1.3.2 Genome/Genotype

Analogous to the genetic material in living organisms, every GP individual posses a finite length binary string, from which all structural and behavioral features of the individual are derived. This binary string is called the genome of the individual; it may be stored in packed form as a list of bytes or integers for ease of use and modification. The terms genome and genotype are used interchangeably in GP context, term "genotype" is preferred in statements making comparison to phenotype. Evolutionary operators mutation and crossover act directly on genotype, effects of these operators on the phenotype are indirect results, carried from genotype by translation.

There are implementation variants with exceptions, where individuals do not have genomes of their own, like *Population Based Incremental Learning* (PBIL). PBIL is not a

genetic programming algorithm; it's an estimation of distribution algorithm, which is an evolutionary algorithm more suitable for multi dimensional parameter optimization. PBIL represents the whole population as a single probability distribution, from which individuals are instantiated by sampling the distribution as needed, and modifications are merged back to a single distribution once again, at the end of each generation.

1.1.3.3 Phenotype

Phenotype is the manifestation of a feature or a property, on the final form of an individual. In the case of genetic programming this is the program represented by an individual. Phenotype is derived from genotype through a specific translation process which depends on type of GP employed. (i.e. for grammatical GP translation is guided by the grammar) Translation process doesn't necessarily use all the genotypic material; some regions of genotype are not used, these regions are called non-coding regions or introns. A modification on genotype may or may not cause a change on phenotype, as different genotypes may (and most of time do) map to same phenotype, or the change on genotype may have occured on a non-coding region.

1.1.3.4 Codon

In biology the smallest unit of genetic material is the nucleotide, but it isn't the smallest meaningful unit; the smallest meaningful units of generic material are constant length blocks of nucleotides. For example, human DNA consists of 4 types of nucleotides (Adenosine, Cytosine, Guanine, Thymine), but only triplets of nucleotides encode amino acids. A triplet of nucleotides is called a codon, and the mapping from nucleotide triplets to amino acids is called the standard codon table. The codon table is almost exactly the same among all living organisms and it is represented using a table with 4^3=64 entries. This 64 entries redundantly encode the 20 amino acids, and the STOP code which is redundantly represented as 3 different entries (UAA, UAG, UGA).

Evolutionary algorithms employ very similar mechanisms; genome of an individual which is a random bit string, is commonly stored and used in constant length blocks of

bits, such as bytes or integers. The byte/integer representation is the genetic programming analog of biologic codon.

1.1.3.5 Generation

For evolutionary algorithms where the evaluation, selection, and other evolutionary operator applications are performed each time over the whole population, one complete cycle of these phases is called a generation. Not all evolutionary algorithms are generation based, there exist algorithms updating only a few individuals every cycle; these are called steady state algorithms. For genetic programming, generational approach is the accepted standard in the literature.

1.1.3.6 Selection

Both in biology and computer science, evolution is based on the survival of the fittest. Selection is the phase of a generation where the individuals which will survive to the next generation are selected. There are many selection methods employed by evolutionary algorithms. Most commonly used selection methods are *Roulette Wheel Selection*, and *Tournament Selection*.

Roulette Wheel Selection method is based on dividing an interval to partitions proportional in size to fitnesses of the associated individuals; then a random number is picked from a uniform distribution on this interval. The selected individual is the one associated to the partition in which the picked random number is included. This way a discrete random variable with probability distribution proportional to fitness values is constructed over a continuous random variable with uniform distribution. This process is repeated to choose each individual of the new population; some individuals (especially those with high fitness) may be selected multiple times, creating multiple copies of the individual, and some (those with low fitness) do not get selected at all.

Tournament Selection on the other hand starts with the selection of a fixed number of individuals randomly; number of individuals selected (generally 2 or 3) is called the tournament size. Individual with highest fitness in a tournament proceeds to the new population. For each individual of the new population, an independent tournament is

performed. Once again, some individuals may be selected multiple times and some may never be selected. There are two advantages of tournament selection over roulette wheel selection. One is, it conserves the diversity of the population by giving low fitness individuals a higher selection chance compared to roulette wheel selection. The other is, it doesn't require to sum and normalize the fitness values of all individuals, or keep track of partition intervals, because the fitness comparisons are local to the tournament participants.

1.1.3.7 Crossover

After the selection of individuals for the population of next generation, evolutionary operators are applied. Crossover is an evolutionary operator where two individuals exchange parts of their genetic material. There are three types of crossover, of which the most common being the *One Point Crossover*. It is applied as follows; first two individuals are picked randomly and a random variable with uniform distribution over unit interval is sampled. If the sampled value is greater than the crossover probability, the individuals are transferred to new population unchanged; if it is lesser than the crossover probability, a random integer up to genome size is picked and the tails of genomes of the individuals are exchanged starting from the position indicated by the random integer.

Another crossover type is *Two Point Crossover*; this time two random integers up to genome size are picked, and the section residing between the two positions indicated by these numbers are exchanged between the individuals; it works the same as one point crossover in all other aspects.

Final type of crossover is *Uniform Crossover*. It is applied at gene level (be it bit, byte or integer), the genes of same position on the two genomes are exchanged with probability equal to crossover probability; this is applied independently for each genome position.

1.1.3.8 Mutation

Mutation is another evolutionary operator; it can be performed before or after the application of crossover. Each gene of each individual is subjected to mutation independently, with a set probability called *mutation rate*. If the genes are bits they are simply flipped when mutated, if the genes are bytes or integers, a new random byte or integer overwrites the old one. The limit behavior of evolutionary search is equivalent to random search as the mutation probability approaches 1.

1.1.3.9 Intron

The size of functional parts of human DNA is assessed to be between 8 to 15%, interspersed through the remaining 85 to 92% consisting of junk DNA which is not coding anything [9]; this non-coding parts are called introns. Similarly in evolutionary algorithms some parts of the genome might left unused (i.e. in grammatical GP every individual use different amount of genotype till they achieve a complete construction) or have no effect on the phenotype even when used (i.e. when a byte is used to make a choice but the number of alternatives to choose from is 1), and sometimes non-coding parts are deliberately incorporated to the encoding of individuals in order to introduce robustness. The same term “intron” is used to describe this non-coding regions in evolutionary algorithms.

1.1.4 Fitness Function in GP

In the case of genetic programming, fitness functions are used as metrics to define distance between pairs of outputs produced by different programs given the same input. It is used to measure the difference of the output of an individual from the expected output for a test case; hence the minimization of the sum of fitness values over all test cases, is equivalent to approaching a complete solution. Fitness function for a single test case associated with the set of all possible output values defines a metric space over possible outputs. The sum of metrics of all test cases of a problem, give us another metric function which is defined over the set of all possible programs for the test cases. Using this new metric, we can construct a metric space over the set of valid programs for a problem defined by its test cases.

The fitness landscape is the co-domain of the fitness function of a such space. For GP and especially grammatical GP fitness landscape is not smooth or derivable, thus analytical methods like gradient ascend/descent are inapplicable for these.

Fitness functions can be designed in different ways; when evolving functions with continuous outputs, the fitness can be defined as the difference between the current output and desired output. Instead of summing or averaging the differences for each test case, total fitness can be defined as the Root Mean Squared Error (RMSE).

1.2 Grammatical Evolution

Grammatical Evolution (a.k.a. Grammatical GP, grammar guided GP) is a form of genetic programming where the search space is restricted to a formal language defined as a BNF grammar, thus ensuring all individuals to be syntactically valid. Such exclusion of syntactically invalid programs dramatically reduces the search space. While still being infinite, it has lower dimensionality. On the other hand a syntactically valid program is not necessarily semantically meaningful too; as a matter of fact most of them are not.

Grammatical evolution uses context free grammars expressed in Backus-Naur Form; but we show on section 1.2.4 that we can hide some context information into the grammar by duplication of rules.

1.2.1 Backus-Naur Form (BNF)

A grammar in BNF form consists of a 4-tuple $\{N,T,P,S\}$ such that N and T are sets of non-terminals and terminals respectively, and $S \in N$ is the designated start symbol. P is the set of production rules, it associates every non-terminal to one or more strings of symbols belonging to $N \cup T$.

A derivation tree for a BNF grammar is a tree having the S as the root node, and each node with a symbol belonging to N has child nodes corresponding to each symbol from the string of symbols for the selected production rule to expand that node. By this construction a derivation tree can either be finite, with all leaf nodes belonging to T, or infinite such that some branches go infinitely deeper by solely using symbols from N.

1.2.2 Genotype-Phenotype Mapping in Grammatical Evolution

The most common method used for genotype-phenotype mapping in grammatical evolution is the Modulo Mapping Rule. It maps a codon to a choice of expansion rule by selecting the expansion option with position current codon value modulo number of expansion options for current non-terminal:

$$selected = codon\_value \ \ mod \ \#\{expansions\ of\ current\ nt\}$$

Even though the modulo operator maps the uniform distribution of codon values homogenously to available expansion options, when the number of options is small and maximum codon value is not divisible to number of options, a selection bias in favor of options at smaller numbered positions occur. This happens due to last codon values being mapped to some of the options from the start without being able to cover till the end.

Alternative mapping functions has been proposed like the Bucket Rule which divides the codon value to product of all numbers of expansion options for all non-terminals of grammar:

$$selected = \frac{codon\_value}{\prod_{i=1}^{\#\{nt\}} \#\{expansions\ of\ nt_i\}} \ mod \ \#\{expansions\ of\ current\ nt\}$$

1.2.3 Unbounded Expansion on General Grammars

A BNF grammar defines a language, which is the set of all strings of terminal symbols where a valid derivation tree exists having those symbols as leaf nodes in same order. In the case of an unconstrained grammar the language defined has infinitely many elements, as you can write infinitely many different programs which are valid in the defined language.

When expanding the derivation tree, we start with the start symbol of the grammar as the root node. At any point, any leaf node with a non-terminal token is expanded by choosing one of the production rules of the non-terminal. Leaf nodes with terminal tokens are not expanded (as production rules are only defined for non-terminals), they represent literal strings that we will concatenate in depth-first traversal order to produce the result of the derivation. If a non-terminal token contains itself in one of its

expansion rules, or a loop can be constructed by cycling through a set of non-terminals referring each other in their expansions, then it means that some infinite expansions are also part of the language. Of course in actual implementation such an infinite expansion would lead to stack overflow, by exhausting the stack as each successive recursive call to rule expansion function allocating a new stack frame.

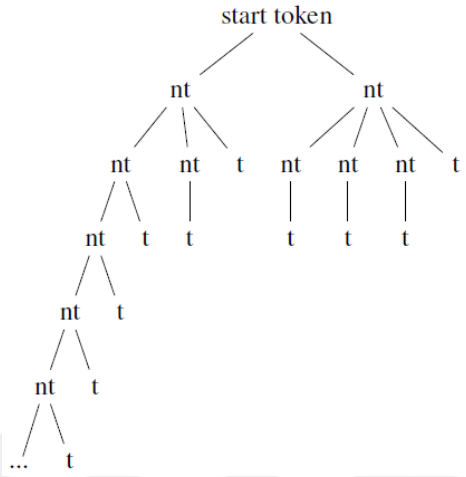


Figure 1.2 An infinite non-terminal branch

Such infinite trees do not represent any program as some branches never reach terminal tokens, but each different subset of non-terminals which can produce such a loop represent an independent orthogonal dimension for the defined language, along which a program can expand infinitely.

A first example comes from the allowed number of statements in a code block; on an unconstrained grammar this can be expressed with a recursive expansion rule, such as

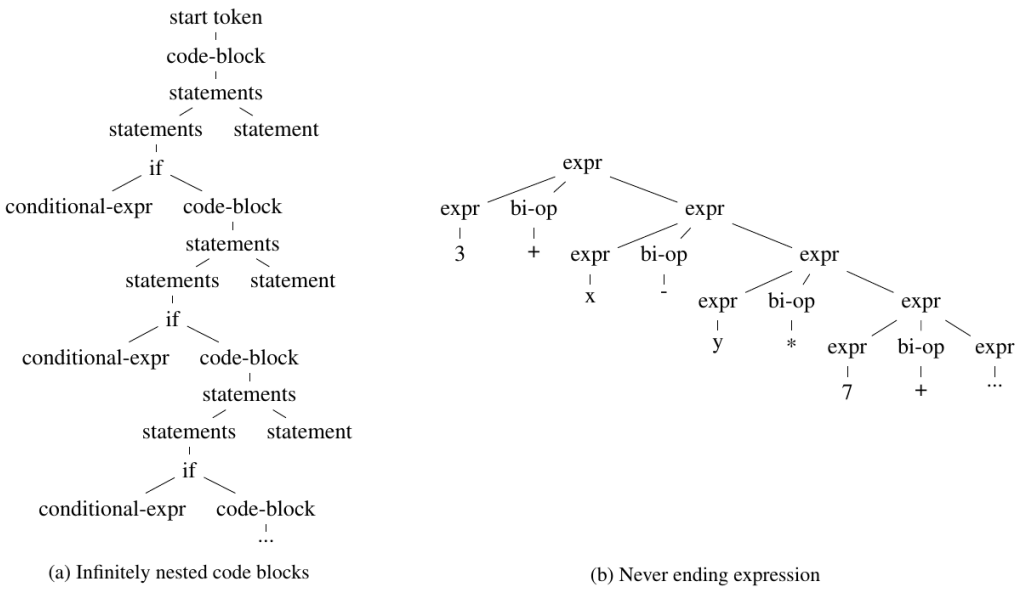
<statements> ::= <statement> | <statement><statements>

The *statements* token above has 50% chance to get expanded to itself followed by a statement token. Even though the expected value of the number of statement tokens generated by this rule is $\sum_{n=1}^{\infty} n\frac{1}{2^n} = 2$, the distribution of it has a long tail; therefore arbitrarily large numbers of consecutive statements can be observed, but with exponentially decaying probabilities.

The real cause of complication is that, the source of randomness used to choose from possible expansion rules is provided by the finite length genome string of individual, which is a random list of integers with length in the order of hundreds. When a request

to obtain next random value exhaust the unused values on the genotype of individual, common practice is to recycle the genotype by using it again from the start. This is the most common solution called wrap around in the literature, but it has the risk of picking up periodic correlations on the stream of values, obtained by the concatenations of genotype with itself. When hundreds of individuals over hundreds of generations are considered, a periodicity due to size of the genome, and some periodicities that may arise as possible cyclic arrangements of non-terminal expansion rules, may coincide on a common period and form an infinite expansion loop without ever reaching a terminal rule. This is a known phenomenon; some alternatives to wrap-around reuse of genotype has been studied and proposed in [10] and [11].

The number of statements growing unbounded is one of the dimensions a general grammar can go expanding indefinitely; two other examples are the number of nested code blocks and expression length, as shown on Fig.1.3. In the ideal case of genotypes with no periodicity (i.e. infinitely long random genotype) expansion trees are expected to stop eventually before diverging to infinity, as the probability of having the same non-terminal expansion sequence decays exponentially at each step.



(a) Infinitely nested code blocks

(b) Never ending expression

Figure 1.3 Two derivation trees with infinite depth expansion branches

1.2.4 Expansion Order

Another implementation decision about applying expansion rules is how to select the next non-terminal to expand when there is more than one. Obviously a depth first selection will always expand the left-most non-terminal node. A breadth first traversal order is possible but tricky, because each expansion also modifies the tree. It can be implemented by tracking all non-terminal child nodes in a FIFO queue to impose an order, with new non-terminals added to the end of the queue, and the next non-terminal to expand is retrieved from the front; when the queue becomes empty it means that all leaf nodes of the tree are terminal nodes.

Independent of how we choose the next non-terminal leaf to expand, it is impossible to limit unbounded expansion, without some extra information about our current depth on derivation tree.

1.2.5 Setting Limits on Expansion

A way to provide depth information at expansion time is to introduce redundant copies of expansion rules to the BNF differentiated by their depth. Let's consider the expansion rule

<expr> ::= <expr> <bi-op> <expr> | <int> | <var>

with possible infinite expansions as illustrated on Fig.1.3(b). Its first production rule is both left-recursive and right-recursive. To exclude infinite expansions of this type from the language, and to limit the maximum depth up to a constant K, we can replace this rule with multiple depth annotated copies of the form:

<expr-N> ::= <expr-N+1> <bi-op> <expr-N+1> | <int> | <var>

for each $N \in \{1,2, \dots, K-1\}$, and a last rule of form:

<expr-K> ::= <int> | <var>

which omits the recursive part. This new set of *K* rules allows expressions consisting of an integer, a variable, or a combination of those with up to *K-1* binary operators.

The same annotated duplication can be extended to bound the expansion depth of cyclical self-references involving sequences of different non-terminals as illustrated on Fig.1.4.

Static bounding by duplication of rules with counters allows embedding depth tracking and limiting on a derivation branch involving multiple types of non-terminals, but it requires a consistent numbering across duplicates of all types of tokens involved.

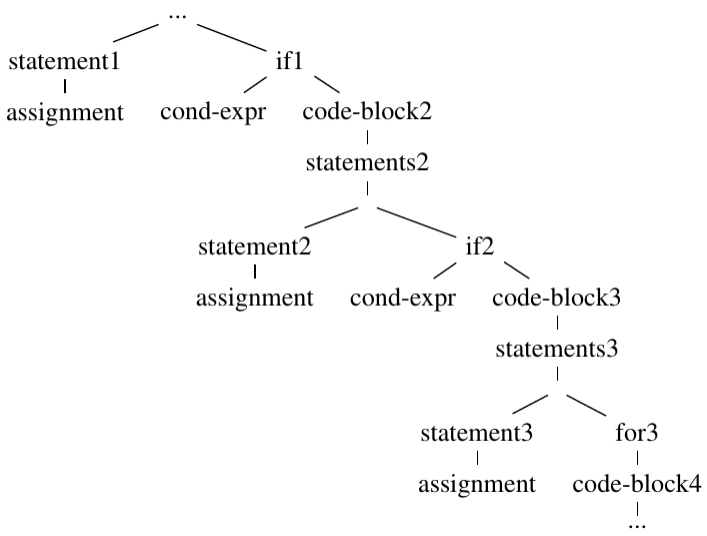


Figure 1.4 Incorporating depth information by duplication of expansion rules

1.3 Heterogeneous Computing

Heterogeneous computing is defined as the use of more than one kind of processors simultaneously on the same hardware platform. It differs from parallel computing such that parallel computing involves multiple identical cores or processors; while the heterogeneous computing employ dissimilar coprocessors each with different capabilities, architecture and processing characteristics.

The most common example is the simultaneous use of CPU and GPU, and it's not limited to desktop or server class personal computers; today even entry level mobile devices (smart phones, tablets, ARM based Raspberry Pi and clones) come with integrated multi-core CPU and multi-core GPU.

Second most common heterogeneous computing platform example is CPU+FPGA platforms. One form of CPU-FPGA cooperation is achieved by interfacing the FPGA as an add-on coprocessor card with onboard RAM through PCI-E bus; the FPGA on the

card is configured at runtime as a coprocessor with desired specialization and characteristics. The acquisition of FPGA manufacturer Altera by Intel in 2015 for $16.7 billion[1] is an indicator that CPU+FPGA cooperation may be more prevalent in near future.

Another form of CPU, FPGA cooperation is observed on FPGA based SoC implementations. A soft-core processor is surrounded with specialized computation blocks, facilitating the integration by keeping the buses inside the FPGA. As this is a popular approach FPGA vendors provide very configurable and scalable soft-core processor generators with their EDA tools; specifically Xilinx provide Microblaze and PicoBlaze, while Altera provide Nios and Nios II soft-processors. Because of the increasing popularity of ARM development and platform support, along with decreased licensing costs, recently FPGA vendors started to incorporate hard ARM cores on the same die as their FPGA. This allows users to have the same SoC approach without sacrificing logic elements from FPGA fabric, and it frees the vendor from maintaining a proprietary development tool chain for their custom soft-processor.

1.4 Genetic Programming on GPU

Processing power provided by graphic processing units make them an attractive platform for evolutionary computation due to the inherently parallelizable nature of the latter.

Just like in the CPU case, genetic programming on GPU requires the code represented by individuals to be rendered to an executable form; this can achieved by compilation to an executable binary object, by conversion to an intermediate representation of a custom interpreter developed to run on GPU, or by directly generating machine-code for the GPU architecture. Compilation of individuals' codes for GPU is known to have a prohibitive overhead that is hard to offset with the gains from the GPU acceleration.

Compiled approach for genetic programming on GPU is especially important for grammatical genetic programming; the representation of individuals for Linear GP and

[1] https://newsroom.intel.com/press-kits/intel-acquisition-of-altera/

Cartesian GP are inherently suitable for simple implementation on a GPU. On the other hand grammatical genetic programming aims to make higher level constructs and structures representable, by using individuals that represent strings of tokens belonging to a language defined by a grammar. Unfortunately execution of a program belonging to a such language is non-trivial, and sooner or later requires some form of compilation or complex interpretation.

1.5 Literature Review

1.5.1 An Overview of Genetic Programming Literature

Although there have been some earlier works dealing with application of evolutionary techniques for code generation purposes, the consensus of the researchers on the field is that the work upon which genetic programming has been separated as a distinct field is a series of books by Koza starting with [12], where he provides a detailed description of what genetic programming is, some introductory examples, and implementations of those in LISP. The use of LISP expressions as default representation of individuals for GP by Koza, made the tree based representation (the natural topology of LISP expressions) up to today even though LISP itself is rarely used anymore. This is the most cited work on the subject with over seventeen thousand citations; the series is continued with three more books [13]–[15], of which the last two was written by Koza together with multiple co-authors.

Another influential early book which provided wide dissemination of the field is [16]; it contributed to the presentation of alternative forms and representations on genetic programming, such as linear and grammatical GP, multi-objective GP, and distributed GP, along with some sample applications.

The “Genetic Programming” series has been part of LNCS since year 2000, where each issue contains proceedings of that year’s EuroGP conference; and the “Genetic Programming Theory and Practice” book series [17]–[29] is being published annually since 2003.

1.5.2 Prior Work on Grammatical Genetic Programming

Grammatical genetic programming is first proposed by C.Ryan, J.Collins, and M.O'Neill in [30]; in that article they present the working principles and genotype to phenotype translation procedure we mention on section 1.2. In [31] O'Neill and Ryan apply grammatical evolution to Santa Fe ant trail problem and successfully evolve a function that guides a virtual ant on a grid to gather food.

In [32] O'Neill, Ryan and Nicolau investigate the effect of incorporating introns to grammatical evolution. They achieve this by introducing some non-terminal expansion rules with label "intron", and which does nothing if selected, leaving the parent non-terminal in its former state. Number of production rules not being a power of 2 introduces a selection bias especially in the common case where the number of rules is small (see Fig.1.5). Padding the production rules with introns up to a number which is a power of 2 allows uniform selection probability among options. Another way to balance the selection probability without using introns, is to use more bits than necessary from the genome at the expense of wasting limited genome string (see Fig.1.5).

Genetic Code	Choice
00	A
01	B
10	C
11	A

Choice	Probability
A	2/4
B	1/4
C	1/4

Genetic Code	Choice
000	A
001	B
010	C
011	A
100	B
101	C
110	A
111	B

Choice	Probability
A	3/8 (.375)
B	3/8 (.375)
C	2/8 (.25)

Figure 1.5 Selection probability distributions on a 3 choice expansion without introns (left) using 2 bits from genome (right) using 3 bits of genome[1]

Keijzer, O'Neill, Ryan and Cattolico present an alternative method to map the genotype to phenotype called the Bucket Rule in [33]. The results presented on the paper show

[1] As presented in [32]

that 8 out of 18 comparisons are statistically insignificant, and others exhibit a marginal improvement for some test cases, while causing regression on others.

In [34] O'Neill, Ryan, Keijzer and Cattolico propose a new crossover operator called Ripple Crossover for grammatical evolution, and compare it to conventional sub-tree crossover operator.

In [35] O'Neill, Brabazon, Nicolau, Garraghy and Keenan propose a position independent genotype-phenotype mapping called π-Grammatical Evolution. When two individuals exchange parts of their genome through crossover, the exchanged codons are used for the mapping of different non-terminals compared to their original use; this completely change the meaning of codon values in their new context, and the same happens to all following codons in chain. π-GE propose to provide some degree of position independence by encoding each codon as a byte pair, where fist byte decides which non-terminal to expand next, and the second byte is used to choose an expansion rule for the non-terminal via modulo rule as usual. This way when a codon is moved to an arbitrary position on a new individual, it doesn't necessarily expand the left-most non-terminal on tree, but it can bring along the decision of which non-terminal to expand next.

A meta-grammatical evolution approach has been proposed by O'Neill and Ryan in [36], where GE is used to evolve some BNF grammars containing some predefined terminals. The evolved grammar is in turn used to evolve individuals which map to concrete implementations stemming from the said terminals.

In [37] Karpuzcu present a grammar describing a small subset of Verilog hardware definition language, and evolve a full adder using open source Icarus Verilog software as simulator.

In [38] O'Neill, Nicolau, and Agapitos propose two grammar one with multiple loops in sequential order and another allowing nested loops, to evolve a general purpose integer sort algorithm for lists of arbitrary length. They showed that it was not possible to evolve a general sort algorithm without adding *swap* functionality as an intrinsic function to grammar. The grammar we propose in section 2.3 is inspired in part by this work.

Byrne, Fenton, Hemberg, McDermott and O'Neill propose a shape grammar for the evolution of complex pylon structure using GE in [39]. They use SLFFEA as an open source finite element analysis software for structural analysis of evolved pylons.

In [40] Fagan, Fenton and O'Neill investigate a position independent initialization method for GE based on derivation tree shape and slope; they compare it to random initialization, ramped half-and-half initialization, anf sensible initialization.

Recently Fenton, McDermott, Fagan, Forstenlechter, O'Neill and Hemberg put [41] on arXiv, where they present the second version of their python based grammatical genetic programming tool PonyGE2. One implementation decision in PonyGE2 which differ from conventional grammatical GP is the issue of codon consumption for unit production rules. When an expansion rule consists of a single option, the accepted implementation is not to waste a codon from genome as there isn't really a choice; but PonyGE2 do consume a codon for unit productions too. The explanation provided is that this helps prevent the ripple caused by mismatch in genome position and tree position for linear tree operations.

1.5.3 Prior Work on Genetic Programming Running on GPU

Acceleration of genetic programming by use of graphic processing hardware has been first proposed in 2007 independently by Harding and Banzhaf [42], and by Chitty [43]. At that time the two main languages providing access to GPU for general purpose programming were either not yet released (OpenCL V1.0 was released in 2009) or just recently released (CUDA V1.0 was released in June 2007). [43] access the GPU for custom computation through non-conventional use fragment shader programs; fragment shaders are small programs running on the GPU cores which compute the final color of a pixel, modern GPU hardware run an independent copy of the related fragment shader for every pixel of every textured surface[1]. [13] too uses fragment shaders, but through a C# library called Accelerator which hides the graphics pipeline

[1] Therefore the number of pixels computed is always much higher than the number of pixels available on screen. Techniques like occlusion culling, stenciling and z-buffer rejection allow the hardware to identify if a pixel will be visible before fragment shading, and discard those which won't be visible; but the fragment shader invocations still remain much higher than pixel count of screen.

abuse needed for the redirection of computation results back to memory instead of painting to screen as intended with shaders.

In [44] Langdon and Banzhaf present a SIMD interpreter for GP that runs on GPU. They use Mackey-Glass regression problem which is a chaotic time series along with RMSE fitness. They employ RapidMind library to use the GPU for general computation, and report a 7 times speedup using GPU compared to CPU.

Wilson and Bazhaf present the first genetic programming implementation running on a game console (Xbox 360 release date 2005) and on a portable media player (Zune 2nd gen) in [45]. While the game console implementation benefit from the GPU acceleration of the device, portable media player implementation uses CPU due to lack of graphics processor. When compared to GPU accelerated GP on a PC, game console is observed to run almost 3 times faster; this can be attributed to the early adoption of next generation GPU hardware by console manufacturers before they become available to consumer use for keeping up on competition between PC gaming vs console gaming.

[46] deals with the compilation overhead of individuals for genetic programming on GPU using CUDA. Article proposes a distributed compilation scheme where a cluster of around 16 computers compile different individuals in parallel; and states the need for large number of fitness cases to offset the compilation overhead. It correctly predicts that this mismatch will get worse with increasing number of cores on GPUs, but also states that "a large number of classic benchmark GP problems fit into this category". Based on figure 5 of the article it can be computed that for a population size of 256, authors required *25 ms/individual* in total (*including network traffic, XO, mutation, processing on GPU, and compilation*).

The first genetic programming implementation for GPU benefiting from the guidance of a grammar was presented in 2010 by Langdon and Harman[47]. It uses GP to evolve better string matching problem solutions, in order to improve GZip compression performance. It employs a grammar constructed from fragments of the string matching code sample distributed as part of CUDA SDK. The evolutionary algorithm used is similar to conventional grammatical genetic programming, but differs in some

aspects, such as the arity of all production rules to be less than 3 and all terminals being a reference to a complete line of a human programmed code. Figure 11 of the accompanying technical report[10] shows that a population of 1000 individuals *(10 kernels of 100 individuals each)* takes around 50 seconds to compile using NVCC from CUDA v2.3 SDK, which puts the average compilation time to approximately *50 ms/individual*.

Dietz and Young present a MIMD interpreter for genetic programming in [48], which is implemented using CUDA to run on GPU. They investigate the causes of overhead of interpretation method on GPU, and point to divergence of execution and divergence of memory access between different individuals.

In [49] an overview of genetic programming on GPU hardware is provided, along with a brief presentation and comparison of compiled and interpreted approaches. As part of the comparison it underlines the trade-off between the speed of compiled code versus the overhead of compilation, and states that the command line CUDA compiler was especially slow, hence why interpreted approach is usually preferred.

the first work which implemented standardized grammatical evolution running on GPU was by Pospichal, Murphy and O'Neill in 2011[50].

[50] investigates the acceleration of grammatical evolution by use of GPUs. It analyzes the performance impact of different design decisions like thread/block granularity, different types of memory on GPU, host-device memory transactions. As part of the article, compilation to PTX form and then JIT compiling the PTX on driver level, is compared with direct compilation to CUBIN object and loading to GPU without further JIT compilation. For a kernel containing 90 individuals, it takes 540ms to compile to CUBIN with sub-millisecond upload time to GPU, vs 450ms for compilation to PTX and 80ms for JIT compilation and upload to GPU by using NVCC compiler from CUDA v3.2 SDK. Thus PTX+JIT case which is the faster of the two, achieves an average compilation time of *5.88 ms/individual*.

[51] proposes an approach for improving compilation times of individuals for genetic programming on GPU, where common statements on similar locations are aligned as much as possible across individuals. After alignment individuals with overlaps are

merged to common kernels such that aligned statements become a single statement, and diverging statements are enclosed with conditionals to make them part of the code path only if the value of individual ID parameter matches an individual having that divergent statements. Authors state that in exchange for faster compilation times, they get slightly slower GPU runtime with merged kernels as all individuals need to evaluate every condition at the entry of each divergent code block coming from different individuals. In results it is stated that for individuals with 300 instructions, compile time is *347 ms/individual* if it's unaligned, and *72 ms/individual* if it's aligned (time for alignment itself not included) with NVCC compiler from CUDA v3.2 SDK.

[52] provides a comparison of compilation, interpretation and direct generation of machine code methods for genetic programming on GPUs. Five benchmark problems consisting of Mexican Hat and Salutowicz regressions, Mackey-Glass time series forecast, Sobel Filter and 20-bit Multiplexer are used to measure the comparative speed of the three mentioned methods. It is stated that compilation method uses NVCC compiler from CUDA V5.5 SDK. Compilation time breakdown is only provided for Mexican Hat regression benchmark on Table 6, where it is stated that total NVCC compilation time took 135,027 seconds and total JIT compilation took 106,458 seconds. Table 5 states that Mexican Hat problem uses 400K generations and a population size of 36. Therefore we can say that an average compilation time of (135,027+106,458)/36×400,000 = *16.76 ms/individual* is achieved.

1.6 Objective of the Thesis

In this work we aim to investigate and improve upon the acceleration of grammar based genetic programming through use of heterogeneous platforms where computing powers of CPU and GPU hardware are used simultaneously according to the advantages of their respective architectures.

In Chapter 2 we review previous works on benchmarking of problem solving ability for genetic programming implementations; then we propose a new benchmark problem of our own, targeting general program synthesis, and designed primarily for being solved with grammatical evolution.

In Chapter 3 we compare the effect of certain evolutionary parameters like population size, maximum number of generations allowed and number of test cases employed. Population and test cases are the two bases to which data-parallelism and instruction-parallelism correspond; and the number of generations corresponds to the time axis. Any tradeoff between these three directly affects the parallelization scheme to be employed and the speed at which solutions are obtained.

In Chapter 4 we present an improvement for the compiled approach to grammatical genetic programming. First, we propose an in-process compilation method to reclaim the time wasted on spawning a new compilation process at each generation and on disk based inter process communication. In-process compilation method we present first compile to an intermediate representation (PTX) in memory using a shared library, the compile to final machine code using the driver API of the GPU. Then we present a non-trivial parallelization scheme, which allows multiple instances of in-process compilation to work in parallel on multiple-cores of CPU, using memory-mapped data transfer between processes and OS level IPC primitives.

In Chapter 5 we present a new interpreter for genetic programming on GPU. It is designed primarily for grammatical GP; it has a grammatical evolution engine, a translator to turn the grammatically produced individual to the intermediate representation of our interpreter, and the interpreter running on GPU implemented with CUDA. It makes use of some new hardware extensions of recent GPUs, and provides a general purpose grammatical GP platform running on GPU without relying on compilation.

Chapter 6 consists of concluding remarks about the results presented and their significance on how to improve grammatical genetic programming performance for especially on a general program synthesis context.

1.7 Hypothesis

The dissertation presented here is an analytical and argumentative one. We analyze the factors contributing to the speed and accuracy of grammatical genetic programming technique; especially when CPU and GPU are used together for tasks

suitable to their respective architecture. Then we hypothesize and argument that modern GPU hardware can be used in more efficient ways, by which they yield higher computation power for grammatical genetic programming, especially when applied to general program synthesis tasks.

CHAPTER 2

BENCHMARKING GRAMMATICAL GENETIC PROGRAMMING

In all scientific endeavors, it is necessary to have some benchmark to compare against, in order to talk of comparative improvement in quality or quantity. This is especially true if the claimed improvement is related to methodology and not to a particular application, as the new method must produce the same improvement on different applications, which necessitates a benchmark suite consisting of multiple benchmarks for applications to diverse domains.

Genetic Programming is no exception, and the importance of the case has been brought to attention in [53], where it is argued that some of the benchmarks problems widely employed in GP articles are popular only because of historical reasons, and that even though articles on applications of genetic programming focus on domain-specific non-trivial problems, the articles on comparison and analysis of methodology uses benchmark problems that are often too simple.

2.1 Subtleties of Benchmarking Genetic Programming

[53] which is published as part of proceedings of GECCO 2012 conference, underlines the repeated use of ancient benchmarks providing low quality comparisons, and makes a call to GP community to establish better benchmarks. A survey on this subject has been conducted with conference participants subsequently, and the results are published later same year [54].

Table 2.1 Problem domains used on papers published in EuroGP and GECCO GP conferences between 2009-2012[1]

Category	Number of papers	Percentage
Symbolic regression	77	36.2
Quartic polynomial	15	7.0
Classification	45	21.1
UCI database examples	23	10.8
Predictive modeling	30	14.1
Boolean	37	17.4
Parity	31	14.6
Multiplexer	21	9.9
Path finding and planning	44	20.7
Artificial ant	24	11.3
Control systems	5	2.4
Game playing	5	2.4
Dynamic optimisation	7	3.3
Traditional programming	8	3.8
Constructed benchmarks	12	5.6
Other	10	4.7
Max problem	5	2.4

[54] contains the results of the mentioned survey, along with some statistics on classes of benchmark problems employed on genetic programming articles published in the last four years (see Table3.1). It provides an in-depth analysis of grouping of subjects observed on these papers, followed by some remarks and propositions like which benchmark problems to avoid and which problems to employ in the future. It is stated that, of the papers published in EuroGP and GECCO GP conferences between years 2009-2012, 36.2 percent employed symbolic regression problems, 21.1 percent employed classification problems, 20.7 percent employed path finding problems and 17.4 percent employed Boolean function (i.e. parity, multiplexers) problems.

2.2 Grammatical Genetic Programming and General Program Synthesis

While the programs represented by individuals in Linear Genetic Programming resemble most to the linear machine code, those represented by Cartesian Genetic Programming mostly resemble to digital circuits or neural networks, and the ones represented by tree based genetic programming resemble (non-surprisingly) to trees.

[1] as presented on survey [54]. Notice that some articles present problems from more than one category.

On the other hand the programs represented in Grammatical GP, not only resemble but are literally pieces of structured source code.

Therefore even though grammatical GP can be used to evolve simple arithmetic or boolean expressions, which can also be represented with tree based GP, linear GP and cartesian GP. General program synthesis problems (described as “True Programming” in [54]) which involve arbitrary conditionals, loops, state variables and nesting can only be represented with grammatical GP in a natural and human readable way. General program synthesis can be achieved with linear GP too, but the evolved solutions will be black box solutions to some degree, instead of structured source code which is more easily understandable and modifiable by humans.

In fact the individuals evolved with grammatical GP can be compiled to an equivalent linear GP representation (*which we present in Chapter 5*). Still, the set of programs that a grammar defines as a language is a strict subset of set of programs belonging to a minimal linear representation covering that language, as long as they are compared under the same linearised size bound.

The proof for this assertion is trivial; the grammatical GP is designed to exclude programs from a set when they are syntactically invalid according to a grammar, but it does not add new members to the set. Therefore you can always find an individual which is a valid linear program, but isn’t a member of the language defined by the grammar. The same mismatch is observed with real world compilers all the time, where some raw assembly code (or byte code for JITed languages like Python, Java, C#) is inserted in the middle of a high level source code to perform some computation faster (or in small space) which is impossible with the sole use of high level grammar.

2.3 A New Benchmark Problem

Here we propose a new benchmark problem for genetic programming, which consists of identifying whether a given search value is present in an integer list. We propose two variants of this problem, first variant requires the evolved function to return true if the searched value is present in the list, or false if it isn’t. This is a question with a

binary answer; from GP point of view it is interesting for its use of both integer and boolean values.

The second variant requires the function to return the position of the searched value in the list, or -1 if the value is not present. We first proposed this problem as a general program synthesis benchmark in [55]. The grammar for the problem is inspired by [38], and it bears some similarity to problems presented in [56] based on the generality of its use case and simplicity of its implementation.

Even though Search Problem is parametric, such that the minimum/maximum length of integer lists and the range from which the list elements are chosen can be adjusted, different values of these parameters do not affect the difficulty of the problem, they only determine the generality of test cases. In out experiments we used test cases with lists of random integers from the range (0, 50), and list lengths varied between 3 and 20. Test cases are randomly generated but half of them are ensured to contain the value searched, and others ensured not to contain.

We employed a binary fitness function, which returns 1 if the returned result is correct or 0 if it's not correct; hence the fitness of an individual is the sum of its fitnesses over all test cases, which evolutionary engine tries to maximize.

```
<expr>        ::= <expr> <bi-op> <expr> | <const>
                  | <var-read> | <var-indexed>
<var-read>    ::= tmp | i | OUTPUT | SEARCH | LENINPUT
<var-indexed> ::= INPUT[<var-read> % LENINPUT]
<var-write>   ::= tmp | OUTPUT
<bi-op>       ::= + | -
<const>       ::= 1 | 2 | (-1)
<statement>   ::= <assignment> | <if> | <loop>
<loop>        ::= for(i=0;i “<” <var-read>;i++){<statements>}
<if>          ::= if ( <cond-expr> ) { <statements> }
<cond-expr>   ::= <expr> <comp-op> <expr>
<comp-op>     ::= “>” | “<” | == | !=
<assignement> ::= <var-write> = <expr>;
<statements>  ::= <statement> | <statement><statements>
<start>       ::= <preamble> <statements> <post-amble>
```

Listing 2.1 – BNF grammar for Search Problem

The general BNF grammar for the problem is shown on Listing 2.1. where terminal nodes are C keywords and expressions, thus the grammar define a valid subset of C programming language. The same grammar bounded for expression length, number of consecutive statements and number of nested code blocks is shown on Listing 2.2. The start rule of both grammars is "<start>".

The generated code is surrounded by a preamble and post-amble to setup the local variables and return the value of OUTPUT which is the same for all individuals. The pre/postamble can be made part of the grammar by adding them as terminals before and after the expansion of start token; or they can be added outside of evolutionary code generation when grammar expansion for the individual is finished. The preamble and post-amble can be seen on Listing 2.3 as the non-italic lines at the beginning and at the end.

```
<expr>        ::= <expr2> <bi-op> <expr2> | <expr2>
<expr2>       ::= <int> | <var-read> | <var-indexed>
<var-read>    ::= tmp | i | OUTPUT | SEARCH | LENINPUT
<var-indexed> ::= INPUT[<var-read> % LENINPUT]
<var-write>   ::= tmp | OUTPUT
<bi-op>       ::= + | -
<int>         ::= 1 | 2 | (-1)
<statement>   ::= <assignment> | <if> | <loop>
<statement2>  ::= <assignment> | <if2>
<statement3>  ::= <assignment>
<loop>        ::= for(i=0;i "<" <var-read>;i++){<statements2>}
<if>          ::= if ( <cond-expr> ) { <statements2> }
<if2>         ::= if ( <cond-expr> ) { <statements3> }
<cond-expr>   ::= <expr> <comp-op> <expr>
<comp-op>     ::= ">" | "<" | == | !=
<assignement> ::= <var-write> = <expr>;
<statements>  ::= <statement> | <statement><statement>
                  | <statement><statement><statement>
<statements2> ::= <statement2> | <statement2><statement2>
                  | <statement2><statement2><statement2>
<statements3> ::= <statement3> | <statement3><statement3>
                  | <statement3><statement3><statement3>
<start>       ::= <preamble> <statements> <post-amble>
```

Listing 2.2 – A depth bounded version of the grammar for Search Problem

We will be referring to this problem as "*Search Problem*" in the following chapters. Simplest correct solution evolvable with this grammar is presented in Listing 2.3; the

code generated by the non-static part which corresponds to the expansion of <statements> token are the lines printed in italic typeface.

```
void func0(int* INPUT,int SEARCH,int LENINPUT){
  int tmp=0;
  int i=0;
  int OUTPUT=0;
  OUTPUT = -1;
  for(i=0;I < LENINPUT;i++) {
    if (INPUT[i % LENINPUT] == SEARCH) {
      OUTPUT = I;
    }
  }
  return OUTPUT;
}
```

Listing 2.3 – Simplest correct solution evolvable with grammar for Search Problem

The solutions obtained by GP almost always contain some junk code with no effect. An actual solution obtained by evolution is presented in Listing 2.4, notice the mentioned junk code with no final effect; only three lines of evolved code contribute to desired output. By definition junk code does not affect the behavior of a solution, but to minimize compilation overhead and for clarity purposes, evolved solutions with junk code caused by introns can be “cleaned up” by dead code removal techniques.

```
void func0(int* INPUT,int SEARCH,int LENINPUT){
  int tmp=0;
  int i=0;
  int OUTPUT=0;
  if ((-1) + SEARCH != tmp-i) {
    if (1 > i) {
      i = 1+i;
      tmp = 2+OUTPUT;
    }
  }
  if (SEARCH == v - OUTPUT) {
    if (OUTPUT+SEARCH == 1) {
      i = SEARCH;
      OUTPUT = 2;
    }
    OUTPUT = SEARCH-OUTPUT;
  }
  for(i=0;i < LENINPUT;i++){
    if (SEARCH == INPUT[i % LENINPUT]) {
```

```
        OUTPUT = SEARCH;
        OUTPUT = i;
      }
    }
    return OUTPUT;
}
```

Listing 2.4 – An actual correct solution evolved for Search Problem containing Junk code

2.4 Other Benchmark Problems

In addition to the presented Search Problem, we also use on the following chapters some other benchmark problems recommended as replacements to older ones in [54].

2.4.1 Keijzer-6 Regression

Keijzer-6 function proposed as a regression benchmark and introduced in [57], is the function $K_6(x) = \sum_{n=1}^{x} \frac{1}{n}$ which maps a single integer parameter to the partial sum of harmonic series with number of terms indicated by the parameter. Regression of Keijzer-6 function is one of the recommended alternatives in [54] to replace simpler symbolic regression problems like quartic polynomial regression.

For this problem we used the root mean squared error as fitness function which is the accepted practice for this problem, and a modified version of the grammar given in [58] and [40]. The only two modification of ours is the increase of token ratio of constants and variables, to promote terminal tokens as the expression nesting gets deeper (the bold expansion options on listing); and replacement of the C terminals with CUDA C floating point math functions (see Listing 2.5). Duplication of some expansion options is a common method in grammar generic programming to assign different selection probabilities to different expansion options; in our modification, x is a terminal token and <c> always expands to a terminal in one expansion, therefore their duplication increase the probability of expansion to a terminal node instead of a non-terminal.

```
<e>  ::= <e2> + <e2> | <e2> - <e2> | <e2> * <e2> | <e2> / <e2>
         | sqrtf(fabsf(<e2>)) | sinf(<e2>) | tanhf(<e2>)
         | expf(<e2>) | logf(fabsf(<e2>)+1)
         | x | x | x | x
```

```
          | <c><c>.<c><c> | <c><c>.<c><c> | <c><c>.<c><c>
          | <c><c>.<c><c>

<e2> ::= <e3> + <e3> | <e3> - <e3> | <e3> * <e3> | <e3> / <e3>
          | sqrtf(fabsf(<e3>)) | sinf(<e3>) | tanhf(<e3>)
          | expf(<e3>) | logf(fabsf(<e3>)+1)
          | x | x | x | x | x | x
          | <c><c>.<c><c> | <c><c>.<c><c> | <c><c>.<c><c>
          | <c><c>.<c><c> | <c><c>.<c><c> | <c><c>.<c><c>

<e3> ::= <e3> + <e3> | <e3> - <e3> | <e3> * <e3> | <e3> / <e3>
          | sqrtf(fabsf(<e3>)) | sinf(<e3>) | tanhf(<e3>)
          | expf(<e3>) | logf(fabsf(<e3>)+1)
          | x | x | x | x | x | x | x | x
          | <c><c>.<c><c> | <c><c>.<c><c> | <c><c>.<c><c>
          | <c><c>.<c><c> | <c><c>.<c><c> | <c><c>.<c><c>
          | <c><c>.<c><c> | <c><c>.<c><c>

<c>  ::= 0|1|2|3|4|5|6|7|8|9
```

Listing 2.5 – Modified BNF grammar for Keijzer-6 regression problem

2.4.2 5-Bit Multiplier

5-bit multiplier problem consists of finding a boolean relation that takes 10 binary inputs to 10 binary outputs. Two groups of 5 inputs each represent an integer up to $2^5 - 1$ in binary, and the output represents a single integer up to 2^{10}–1, such that the output is the multiplication of the two input numbers. This problem can be attacked as 10 independent binary regression problems, with each bit of the output is separately evolved as a circuit or boolean function; this means not bothering to exploit any possible correlation or shared structure there may be between different bits of the output, for the sake of implementation simplicity.

It's easy to show that the number of n-bit input m-bit output binary relations is $2^{m(2^n)}$, which grows exponentially on m and super-exponentially on n. Multi-output multiplier is the recommended alternative to Multiplexer and Parity problems in [54].

We transfer input to and output from GPU with bits packed as a 32bit integer; hence there is a code preamble before first individual to unpack the input bits, and a post-amble after each individual to pack the 10 bits computed by evolved expressions as an integer.

The fitness function we used for 5-bit multiplier is the number of different bits between the individual's response and the correct answer, computed as the pop count of individual's response exclusive-or correct answer.

```
<start> ::= o0=<expr>;o1=<expr>;o2=<expr>;o3=<expr>;o4=<expr>;
            o5=<expr>;o6=<expr>;o7=<expr>;o8=<expr>;o9=<expr>;
<expr>  ::= (<expr2> <bi-op> <expr2>) | <var> | (~ <var>)
<expr2> ::= (<expr2> <bi-op> <expr2>) | <var> | (~ <var>)
            | <var> | (~ <var>)
<var>   ::= a0 | a1 | a2 | a3 | a4 | b0 | b1 | b2 | b3 | b4
<bi-op> ::= & | #or#
```

Listing 2.6 – BNF grammar for 5-bit Multiplier problem

CHAPTER 3

EFFECTS OF POPULATION SIZE, NUMBER OF GENERATIONS AND NUMBER OF TEST CASES

In this chapter we investigate the effects of modifying three evolutionary parameters, namely (i) the maximum number of generations allowed, (ii) size of population in terms of individuals, and (iii) the number of test cases used, on the number and time distribution of solutions found in the context of Search Problem; on the other hand the results presented in this chapter can be generalized to other grammatical genetic programming problems targeting general program synthesis.

3.1 Experiments

We performed three experiments; first one serves as a baseline for the others, and the second trades off number of generations for a larger population while maintaining same computation time, in order to compare the distribution of fitness traces against baseline. The last employs only half of the test cases, to investigate the effects of trading off generalization accuracy for decreased computation time.

3.1.1 Baseline Experiment

As a baseline measurement, we evolved 50 independent populations using *Search Problem* grammar. We used a population size of 100 individuals per population. There were 40 test cases with only half of them containing the searched integer; lists had length varying from 3 to 7 integers. Same test cases have been used for all populations; crossover and mutation rates were both set to 0.7. We employed elitism and two way tournament selection on all experiments.

It can be seen on Fig.3.1 that 12 out of 50 populations managed to evolve a successful individual that return the position of the searched value or -1 in case it isn't in the list. On this experiment we expressed "fitness" as error (*number of test cases failed*) where lower error on plots meaning higher fitness. Fig.3.1 consists of two plots with fitness traces for each population; plot below shows the trace of fitness of best individuals for successful populations (those who managed to evolve an individual with zero error before 500 generation limit), and plot above show the same for failed populations. This separation is purely to better portray the difference of shape in distributions of fitness traces, between successful and unsuccessful populations. Each trace line corresponds to a population, and values attained by a line are the fitness values of the best individual of that population for the indicated generation. Therefore vertical jumps correspond to the evolution of a new best individual having lower error than previous best for that population; and horizontal segments correspond to generations where the population couldn't evolve any better individuals, so the best from previous generation has been preserved by elitism.

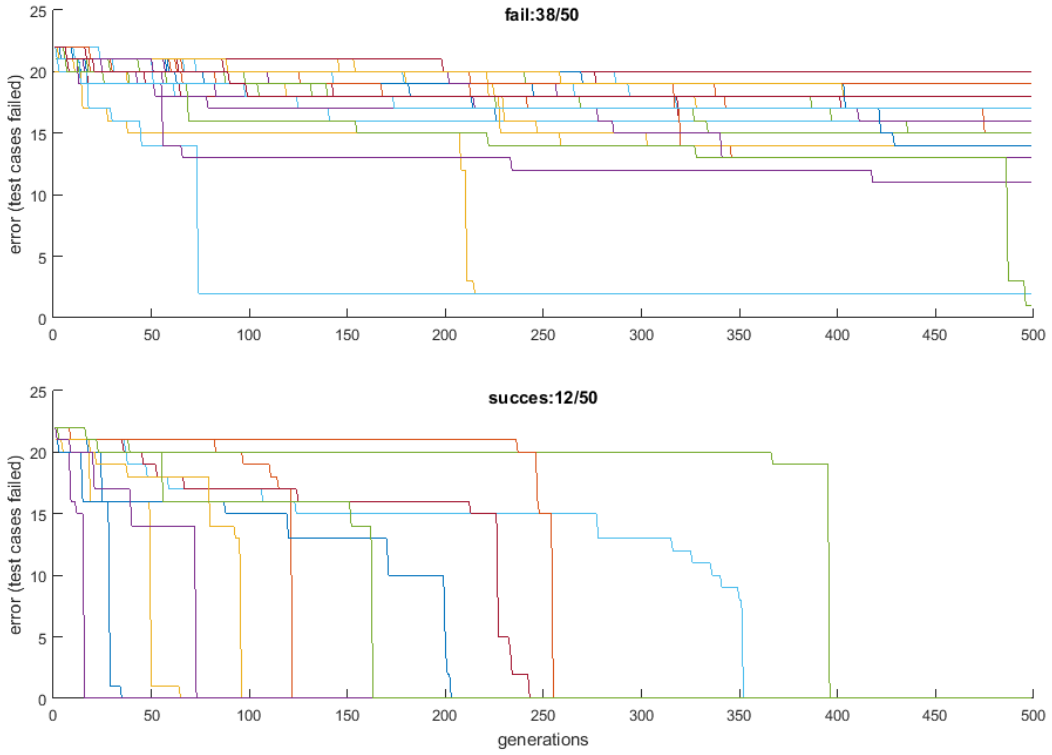


Figure 3.1 Fitness traces of best individuals of 50 populations for 500 generations

The average fitness of best individuals is only meaningful for the failed populations; successful populations leave the process when a zero error individual is evolved, causing the number of populations to average over to be different on different parts of the horizontal axis. This makes the same plot of average for successful populations hard to interpret. Fig.3.2 shows the mentioned average fitness of best individuals of "failed" populations; jagged traces can be seen to converge to a smoother curve upon averaging. We use the term "failed" loosely here, because any of the remaining individuals might still converge to a successful solution if the evolution was allowed to continue with more generations; this is further indicated by the decreasing average error by generations.

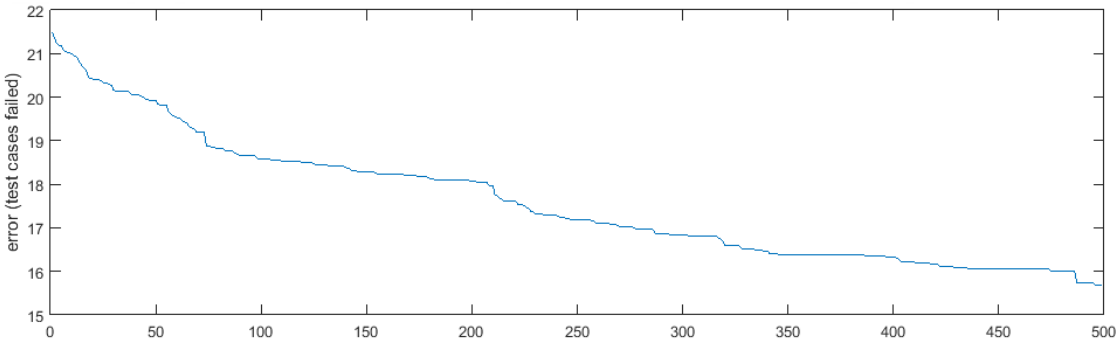


Figure 3.2 Fitness average of failed populations' best individuals

3.1.2 More Populations, Less Generations, Same Wall-Clock Time

The distribution of the points where traces of successful populations intersect the horizontal axis on Fig.3.1 hints that conditional probability of a population to succeed at a given generation decreases with each generation, under the condition that it will eventually succeed. Thus, we repeated the experiment with the same settings, except limiting the maximum number of generations five-fold (from 500 to 100). To maintain an equal amount of computation (hence equal computation time), we also increased the number of populations five-fold (from 50 to 250). Fig.3.3 shows the fitness traces of the experiment, where 32 out of 250 populations managed to produce an error free individual. Although the success rate drops to 0.128 (32/250) from previous 0.24 (12/50), trying more independent populations for shorter times yielded 2.66 (32/12) times more error free individuals for the same amount of computation. Notice that the points where populations reach an error free individual are distributed more

homogeneously across the first 100 generations, in contrast to previous experiment. Yet, the points at which error free individuals emerge don't get more frequent at left edge of horizontal axis neither; to the contrary on the first 15 generations no error free individual is observed, in stark contrast to the generation range 15-100.

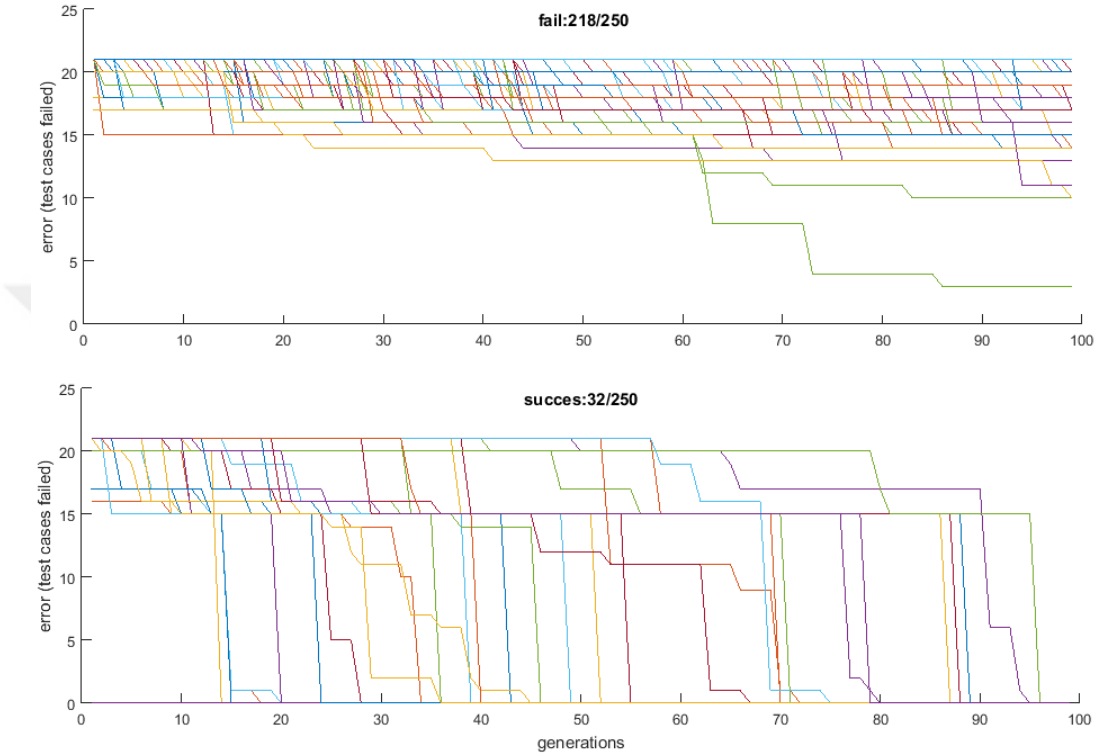


Figure 3.3 Fitness trace of best individuals for 250 population over 50 generations

Cumulative density of the truncated probability distribution of having an error free individual in terms of number of generations is the mathematical counterpart of this intuition. It can be used to minimize the computation time needed to obtain a solution, by optimizing the number of generations allowed per population. As the number of generations allowed per population change, the expected value of solutions obtained per generation change too. It can be estimated with regression fitting, but this requires computing large numbers of fitness traces to smooth the empirical cumulative density from which the probability distribution can be extracted after curve fitting. The extreme case of allowing only one generation is equivalent to random search, as the individuals don't get a chance to cooperate through crossover across generations.

3.1.3 Less Test Cases

We conducted a third set of experiments, with the same settings as the first one, but this time decreasing the number of test cases from 40 to 20, which halves the computation time in exchange for an increased risk of false positives labeled as successful. Figures Fig.3.4 and Fig.3.5 show the traces of best individuals of populations, and the average of best individuals from unsuccessful populations. The success rate became 0.26 (13/50) with an increase of 2% success rate, which is statistically insignificant to attribute to false positives which may stem from a lack of test cases.

The traces for successful populations on figure Fig.3.4 shows that 38% (5/13) of solutions are concentrated in the narrow range of first 25 generations. It can be conjectured that the decreased number of test cases contributed to early success of evolution in contrast to previous experiment setups. (*notice that on Fig3.4 and Fig3.5 the vertical axis for failed populations do not start from 0*)

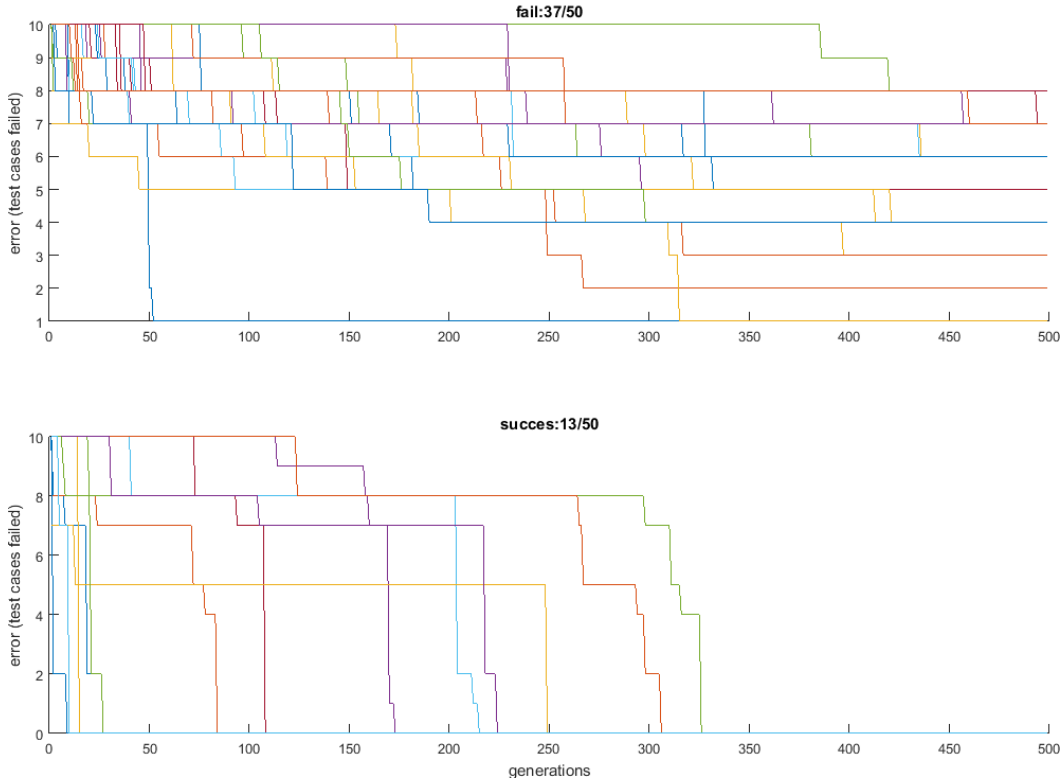


Figure 3.4 Fitness trace of best individuals for 50 populations using only half of the test cases

The ratio of number of test cases to false positives is a parameter that can be used to tune the performance of evolution. As the number of test cases affect the computation time linearly (*i.e. doubling the test cases doubles the computation time*), it may be beneficial to use minimum indicative number of test cases; we expect this number to be very dependent on the problem. Total computation time can be minimized by allowing a higher false positive rate and thoroughly checking the individual solutions later, instead of thoroughly checking the whole population earlier.

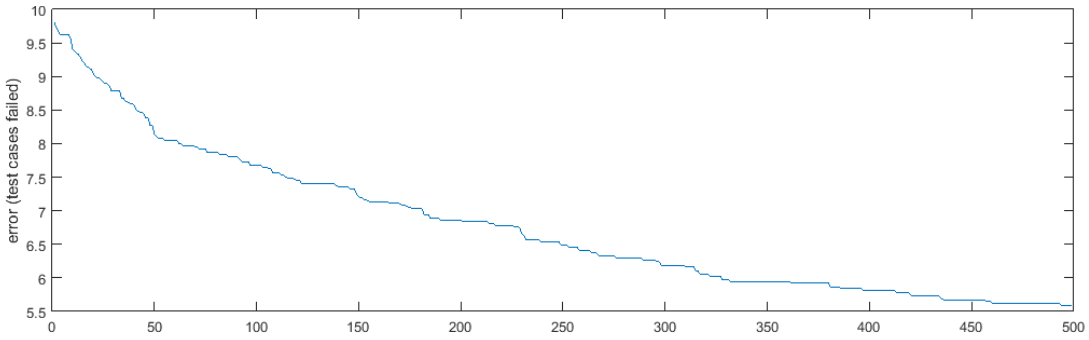


Figure 3.5 Fitness average of failed populations' best individuals

3.2 Motivation for Parallelization

Population size (number of individuals), number of generations evolution is allowed to run for, and number of test cases employed, are three important evolutionary parameters which affect the probability and frequency of obtaining successful individuals. Increase in any of those three parameters, increase the probability of obtaining a successful individual, and decrease the expected value of number of generations till the next successful individual; but it also increase the computation time linearly. For the tradeoffs between these parameters that preserve the amount of computation, we expect there to be a Pareto front of optimality where multiple configurations of parameters can yield the same optimal behavior in terms of expectancy of computation amount required to obtain a solution.

On a serialized computation model, each of these three parameters contribute to total time complexity as a multiplicative factor; hence a unit increase in any one of those increase the time cost by an amount equal to multiplication of other two. On the other hand only generations have a time based data dependency between them;

individuals and test cases are neither temporally dependent, nor inter-dependent. Therefore all *(individual,testcase)* pairs can be evaluated in parallel, only to be synchronized between generations for the consolidation of fitness data and creation of next population.

CHAPTER 4

COMPILATION OF INDIVIDUALS FOR GENETIC PROGRAMMING ON GPU

In this chapter we investigate methods to accelerate compilation of individuals for genetic programming on GPU hardware. We compare the conventional out-of-process compilation with an in-process compilation strategy that we propose to minimize the compilation overhead at each generation. Then we investigate ways to parallelize in-process compilation; in-process compilation doesn't lend itself to trivial parallelization with threads, we propose a multi-process parallelization using memory sharing and operating system's inter-process communication primitives. With parallelized compilation we achieve further reductions on compilation overhead.

4.1 Development and Experiment Setup

4.1.1 Hardware Platform

All experiments have been conducted on a dual Xeon E5-2670 (8 physical 16 logical cores per CPU, 32 cores in total) platform running at 2.6Ghz equipped with 60GB RAM, along with dual SSD storage and four NVidia GRID K520 GPUs. Each GPU itself consists of 1536 cores spread through 8 multiprocessors running at 800Mhz, along with 4GB GDDR5 RAM and is able to sustain 2 teraflops of single precision operations (in total 6144 cores and 16GB GDDR5 VRAM which can theoretically sustain 8 teraflops single precision computation assuming no other bottlenecks)[1]. GPUs are accessed for

[1] see validation of hardware used for experiment: http://www.techpowerup.com/gpuz/details/7u5xd/

computation through NVidia CUDA v8 API and libraries, running on top of Windows Server 2012 R2 operating system.

4.1.2 Development Environment

Codes related to grammar generation, parsing, derivation, genetic programming, evolution, fitness computation and GPU access has been implemented in C#, using managedCuda[1] for CUDA API bindings and NVRTC interface, along with CUDAfy.NET[2] for interfacing to NVCC command line compiler. The grammars for the problems have been prepared such that, the languages they define are valid subsets of CUDA C language with specialization towards the respective problems.

4.1.3 Experiment Parameters

We ran each experiment with population sizes starting from 20 individual per population going up to 300, with increments of 20. As the subject of interest is compilation times and not fitness, we measured the following three parameters to evaluate compilation speed:

- *ptx* : CUDA source code to PTX compilation time per individual
- *jit* : PTX to Cubin object compilation time per individual
- *other* : All remaining operations a GP cycle requires (i.e. uploading compiled individuals to run on GPU, downloading produced results, computing fitness values, evolutionary selection, cross over, mutation, etc.)

We have measured the value of *other* to be always at sub-millisecond level, in all experiments, all problems and for all population sizes; therefore it does not appear on plots. For all practical purposes *ptx + jit* can be considered as the total time cost of a complete cycle for a generation, with an error margin of 1ms/pop.size.

Each data point on plots correspond to the average of one of those three types of measurements for a given (population size; measurement type; experiment) triple.

[1] https://kunzmi.github.io/managedCuda/

[2] https://cudafy.codeplex.com/

Each average is computed over the measurement values obtained for the first 10 generations of 15 different populations with given size (thus effectively the compile times of 150 generations averaged). The reason for not directly using 150 generations of a single population is that a population gains bias towards a certain type of individuals after certain number of generations, and stops representing the inherent unbiased distribution of grammar.

The number of test cases used is dependent to the nature of problem; on the other hand as each test case is run as a GPU thread, it is desirable that the number of test cases are a multiple of 32 on any problem, as finest granularity for task scheduling on modern GPUs is a group of 32 threads which is called a *Warp*. For non multiple of 32 test cases, GPU transparently rounds up the number to nearest multiple of 32 and allocate cores accordingly, with some threads from the last warp work on cores with output disabled. The number of test cases we used during experiments were 32 for Search Problem, 64 for regression of Keijzer-6 function and 1024 (= $2^{(5+5)}$) for 5-bit Binary Multiplier Problem. For all experiments both mutation and crossover rate was set to 0.7; these rates do not affect the compilation times.

4.2 Conventional Compilation as Baseline

NVCC is the default compiler of CUDA platform, it is available as a command line application distributed with CUDA SDK. In addition to compilation of CUDA C source code, it performs tasks such as the separation of source code as host code and device code, calling the underlying host compiler (GCC or Visual C compiler) for host part of source code, and linking compiled host and device object files.

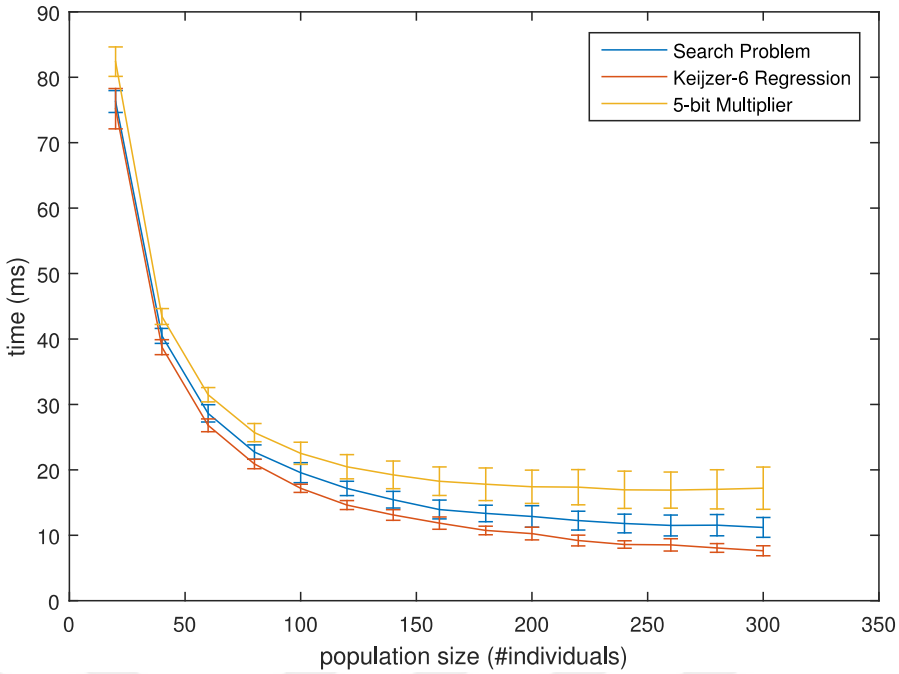


Figure 4.1 Nvcc compile times by population size (per individual time)

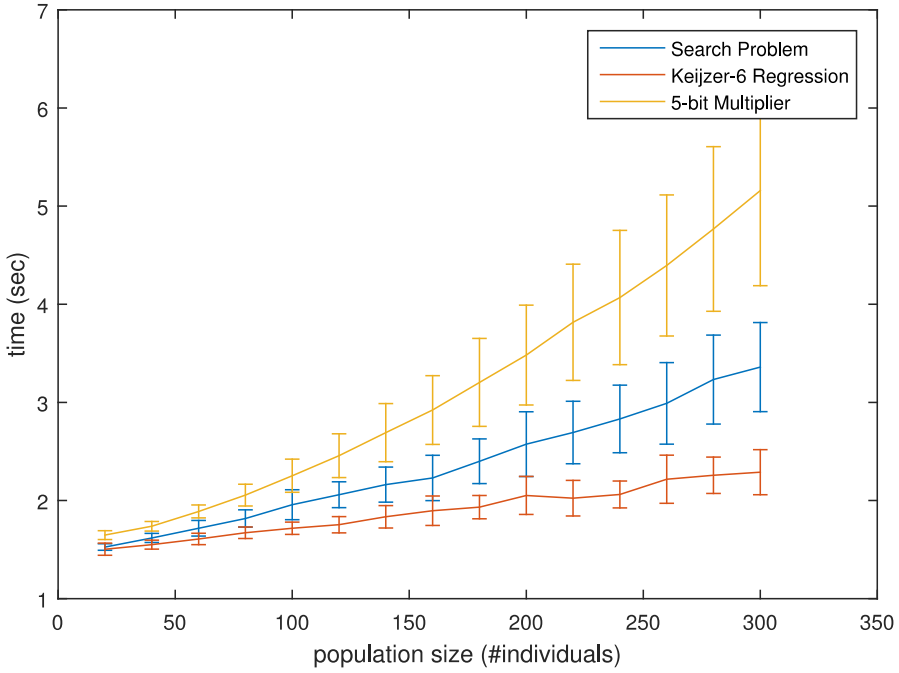


Figure 4.2 Nvcc compile times by population size (total time)

Figure 4.1 shows that compilation times level out at 11.2 ms/individual for Search Problem, at 7.62 ms/individual for Keijzer-6 regression, and at 17.2 ms/individual for 5-bit multiplier problem. It can be seen on Figure 4.2 that, even though not obvious, the total compilation time does not increase linearly, which is most observable on trace of

5-bit multiplier problem. As NVCC is a separate process, it isn't possible to measure the distribution of compilation time between source to PTX, PTX to CUBIN, and all other setup work (i.e. process launch overhead, disk I/O); therefore it is not possible to pinpoint the source of nonlinearity on total compilation time.

The need for successive invocations of NVCC application, and all data transfers being handled over disk files are the main drawbacks of NVCC use in a real time context, which is the case in genetic programming. Even though the repeated creation and teardown of NVCC process most probably guarantees that the application stays on disk cache, this still prevents it to stay cached on processor L1/L2 caches.

4.3 In-process Compilation

NVRTC is a runtime compilation library for CUDA C, it was first released as part of v7 of CUDA platform in 2015. NVRTC accepts CUDA source code and compiles it to PTX in-memory. The PTX string generated by NVRTC can be further compiled to device dependent CUBIN object file and loaded to GPU with CUDA Driver API, still without persisting it to a disk file. This provides optimizations and performance not possible in off-line static compilation.

Without NVRTC, for each compilation a separate process needs to be spawned to execute NVCC at runtime. This has significant overhead, NVRTC addresses this by providing a library interface that eliminates overhead of spawning separate processes, and extra disk I/O. On the other hand NVRTC performs only the first part of the compilation which is converting the CUDA source to PTX form; in order to use compiled PTX code a second round of compilation must be performed either with NVCC (which would defeat the purpose) or with the CUDA Driver API which incorporates a light JIT compiler that can compile PTX to CUBIN binary form executable on GPU.

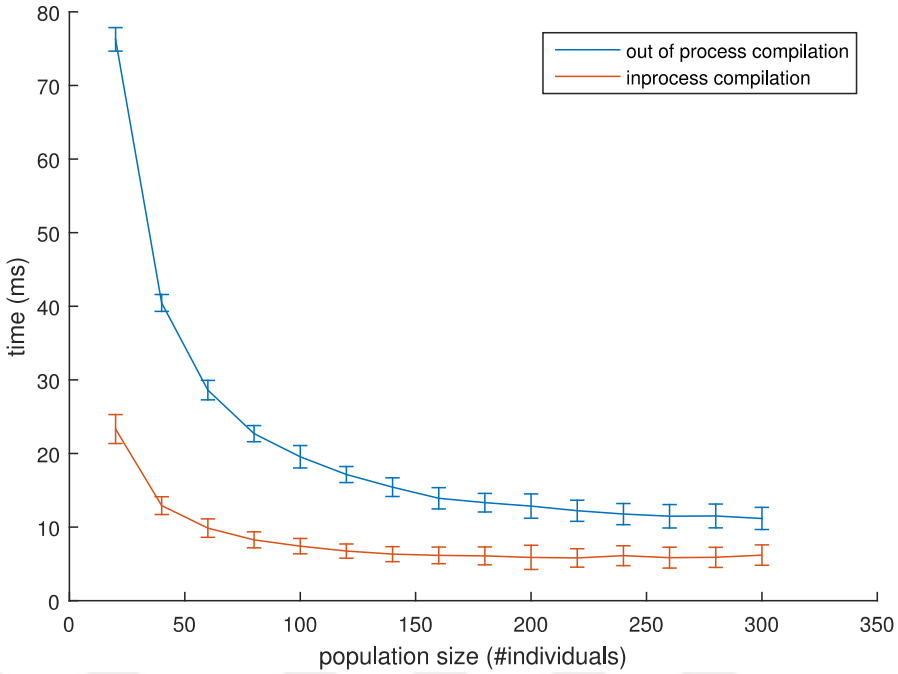


Figure 4.3 In-process and out of process compilation times by population size, for Search Problem (per individual)

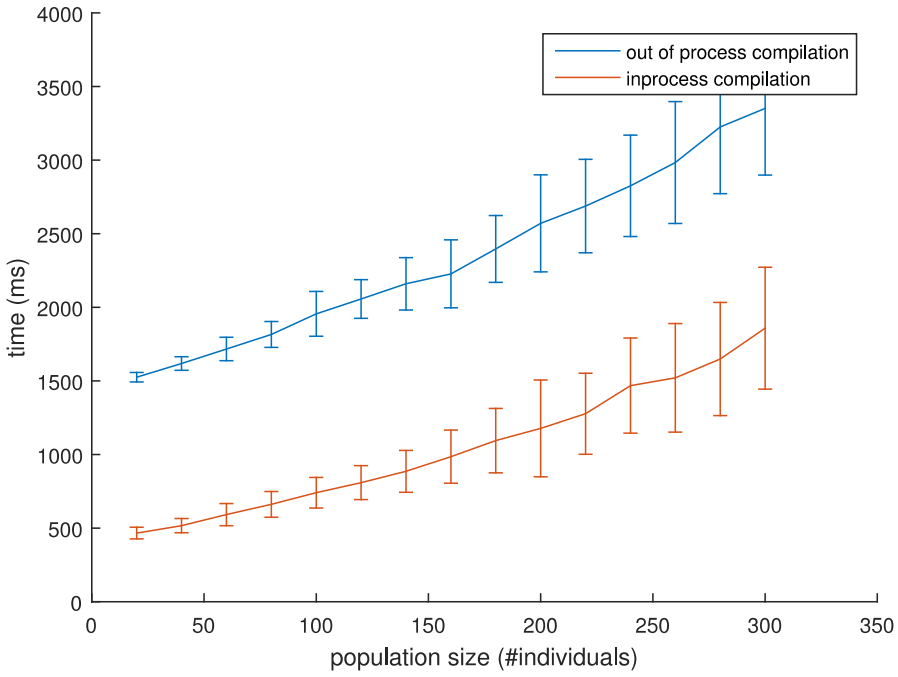


Figure 4.4 In-process and out of process compilation times by population size, for Search Problem (total)

On figures 4.3-4.8 it can be seen that in-process compilation of individuals not only provides reduced compilation times for all problems on all population sizes, it also allows to reach asymptotically optimal per individual compilation time with much

smaller populations. The fastest compilation times achieved with in-process compilation is 4.14 ms/individual for Keijzer-6 regression (at 300 individuals/population), 10.88 ms/individual for 5-bit multiplier problem (at 100 individuals/population[1]), and 6.89 ms/individual for Search Problem (at 280 individuals/population[2]). The total compilation time speed ups are measured to be in the order of 261% to 176% for the K6 regression problem, 288% to 124% for the 5-bit multiplier problem, and 272% to 143% for the Search Problem, depending on population size (see Fig.4.9).

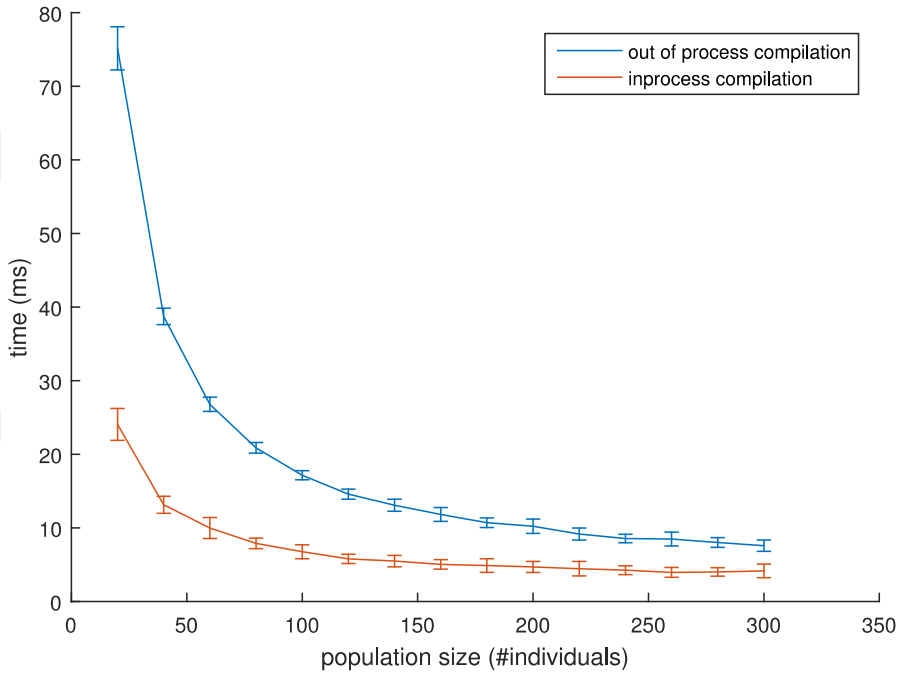


Figure 4.5 In-process and out of process compilation times by population size, for Keijzer-6 Regression(per individual)

[1] compilation speed at 300 individuals/population is 13.29 ms/individual for 5-bit Multiplier

[2] compilation speed at 300 individuals/population is 7.76 ms/individual for Search Problem

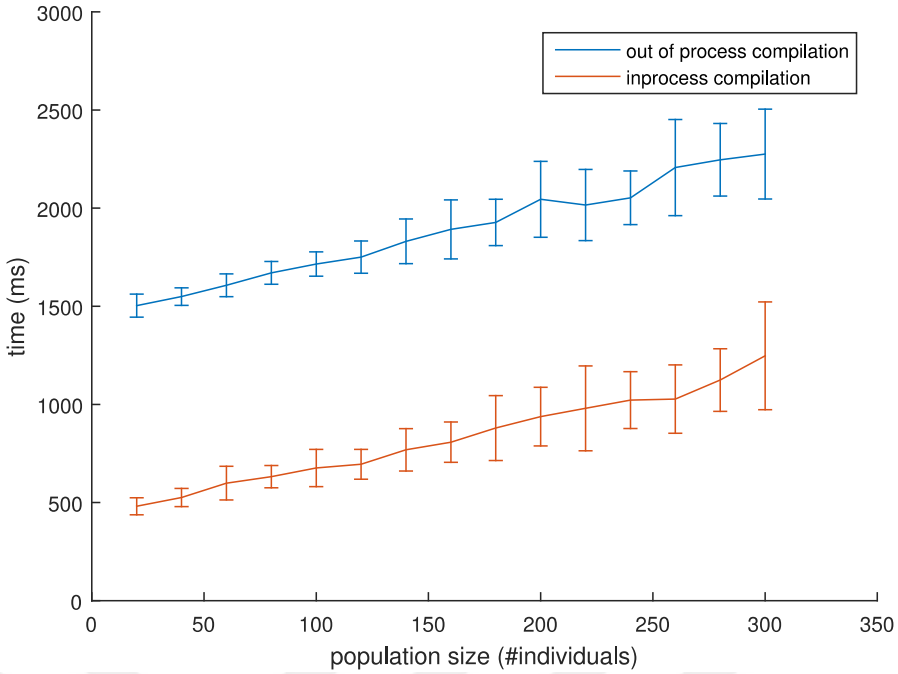


Figure 4.6 In-process and out of process compilation times by population size, for Keijzer-6 Regression (total)

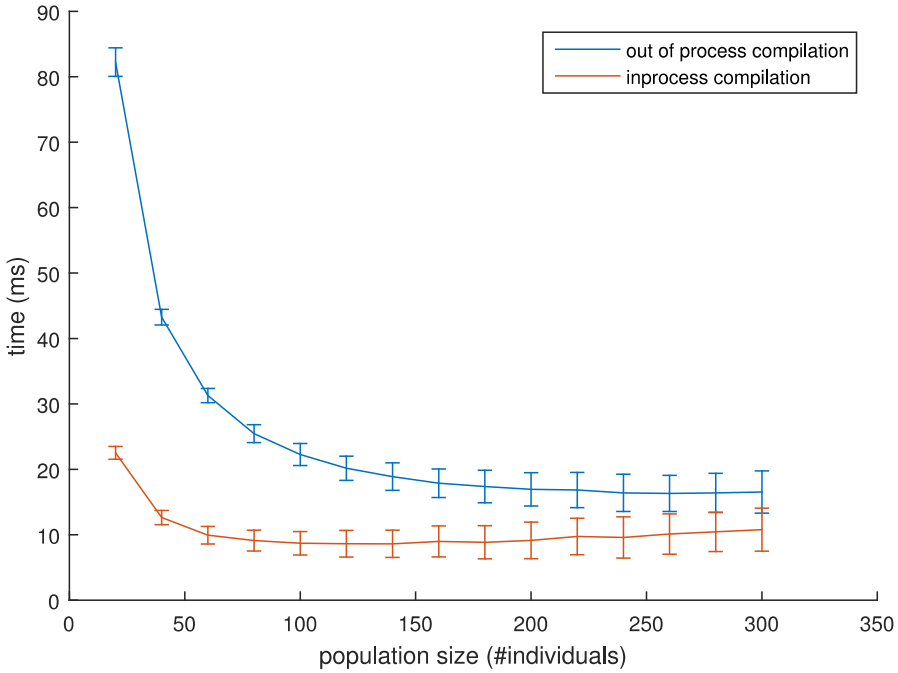


Figure 4.7 In-process and out of process compilation times by population size, for 5-bit Multiplier (per individual)

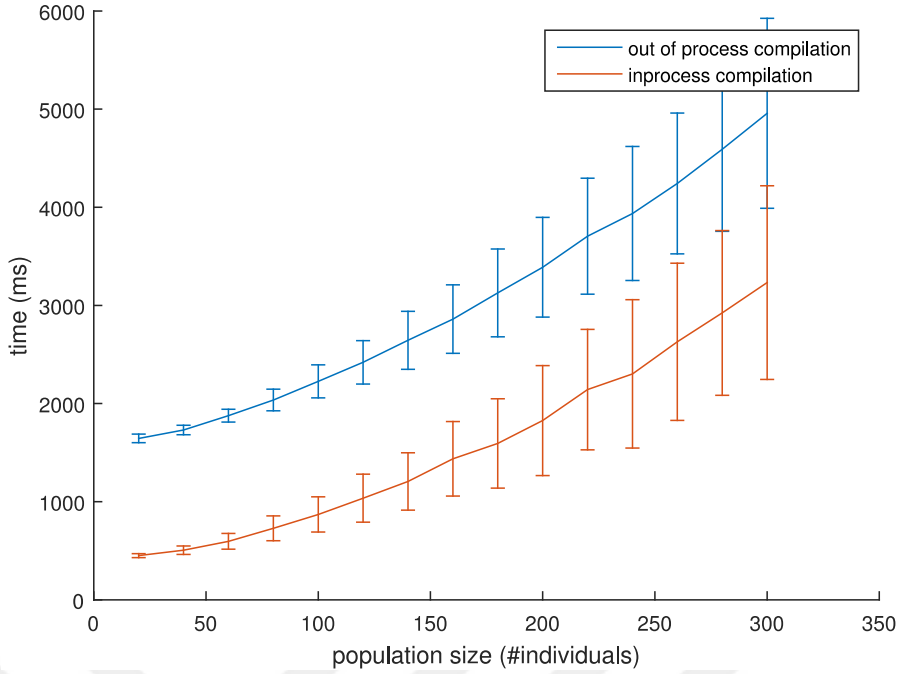


Figure 4.8 In-process and out of process compilation times by population size, for 5-bit Multiplier (total)

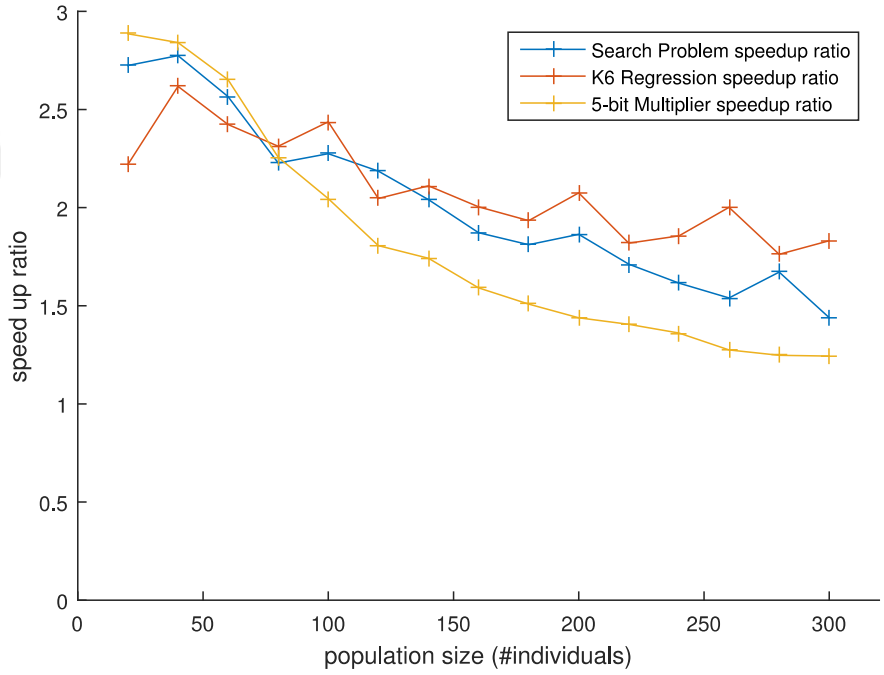


Figure 4.9 Compile time speedup ratios between conventional and in-process compilation by problem

4.4 Parallelizing In-process Compilation

4.4.1 Infeasibility of parallelization with threads

The natural approach to parallelize in-process compilation comes to mind as, to partition the individuals and spawn multiple threads that will compile each partition in parallel through NVRTC library. Unfortunately it turns out that NVRTC library is not designed for multi-threaded use; we noticed that when multiple compilation calls are made from different threads at the same time, the execution is automatically serialized.

Stack trace in Fig.4.10 shows *nvrtc64_80.dll* calling OS kernel's *EnterCriticalSection* function to block for exclusive execution of a code block, and gets unblocked by another thread which also runs a block from same library, 853ms later via the release of the related lock. The pattern of green blocks on three threads in addition to main thread in Fig.4.10 shows that calls are perfectly serialized one after another, despite being called at the same time which is hinted by the red synchronization blocks preceding them.

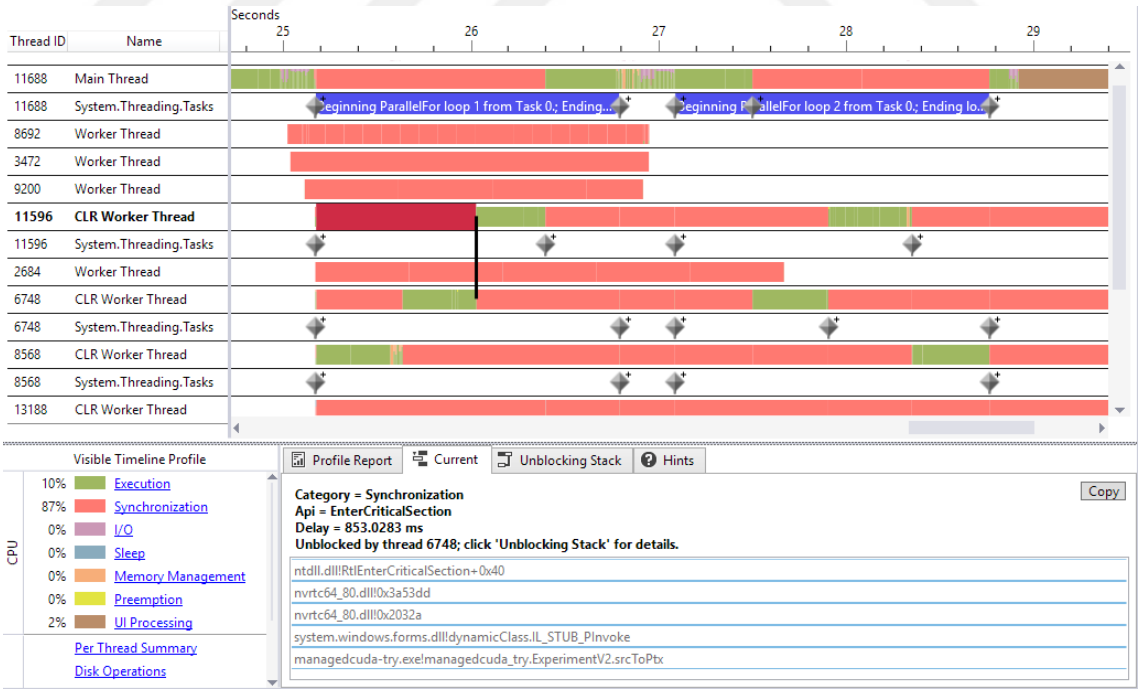


Figure 4.10 NVRTC library serializing calls from multiple threads

Although NVRTC compiles CUDA source to PTX with a single call, the presence of compiler options setup function which affects the following compilation call, and use of critical sections at function entries, show that apparently this is a stateful API. Furthermore, unlike CUDA APIs' design, mentioned state is most likely not stored in thread local storage (TLS), but stored on the private heap of the dynamic loading library, making it impossible for us to trivially parallelize this closed source library using threads, as moving the kept state to TLS requires source level modifications.

4.4.2 Parallelization with Daemon Processes

Therefore as a second approach we implemented a daemon process which stays resident. It is launched from command line with a unique ID as command line parameter to allow multiple instances. Instances of daemon are launched as many times as the wanted level of parallelism and each instance identifies itself with the ID received as parameter. Each launched process register two named synchronization events with the operating system, for signaling the state transitions of a simple state machine consisting of *{starting, available, processing}* states which represent the state of that instance. Main process also has copies of same state machines for each instance to track the states of daemons. Thus both processes (main and daemon) keep a consistent view of the mirrored state machine by monitoring the named events which allows state transitions to be performed in lock step. State transition can be initiated by both processes, specifically *(starting → available)* and *(processing → available)* is triggered by the daemon, and *(available → processing)* is triggered by the main process.

The communication between the main process and compilation daemons are handled via shared views to memory maps. Each daemon register a named memory map and create a memory view, onto which main process also creates a view to after the daemon signals state transition from starting to available. (see Fig.4.11) CUDA source is passed through this shared memory, and compiled device dependent CUBIN object file is also returned through the same. To signal the state transition *(starting → available)* daemon process signals the first event and starts waiting for the second

event at the same time. Once a daemon leaves the starting state, it never returns back to it.

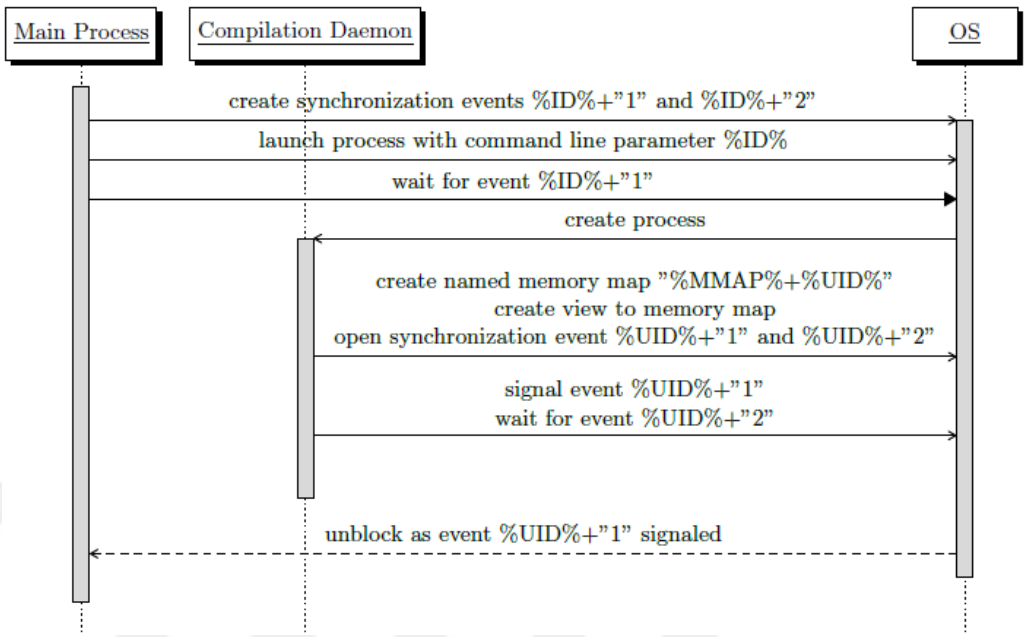


Figure 4.11 Sequence Diagram for creation of a compilation daemon process and related inter-process communication primitives

When the main process generates a new population to be compiled it partitions the individuals in a balanced way, such that the difference of number of individuals between any pair of partitions is never more than one. Once the individuals are partitioned, the generated CUDA codes for each partition are passed to the daemon processes. Each daemon waits in the blocked state till main process wakes that specific daemon for a new batch of source to compile by signaling the second named event of that process (see Fig.4.12). Main process signals all daemons asynchronously to start compiling; then starts waiting for the completion of daemon processes' work. To prevent the UI thread of main process getting blocked too, main process maintains a separate thread for each daemon process it communicates with, therefore while waiting for daemon processes to finish their jobs only those threads of main process are blocked. Main process signaling the second event and daemon process unblocking as a result, corresponds to the state transition *(available → processing)*.

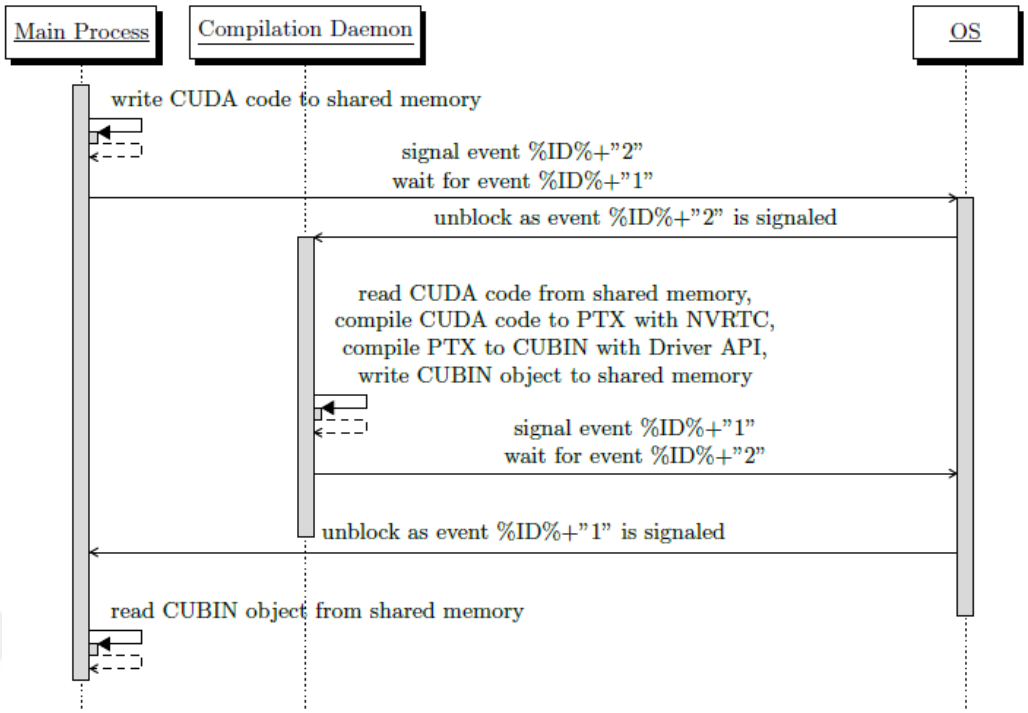


Figure 4.12 Sequence Diagram for compilation on daemon process and related inter-process communication

When a daemon process arrives to processing state, it reads the CUDA source code from the shared view of the memory map related to its ID, and compiles the code using NVRTC library.

Once a daemon finishes compiling and writes the Cubin object to shared memory, it signals the first event to unblock the related thread in main process and starts to wait for the second event once again. This signaling, blocking pair corresponds to the state transition *(processing → available)*.

4.4.3 Cost of Parallelization

The parallelization approach we propose is virtually overhead free when compared to a hypothetical parallelization scenario using threads. As the daemon processes are already resident and waiting in the memory along with the loaded NVRTC library, the overhead of both parallelization approaches is limited to the time cost of memory

moves from/to shared memory and synchronization by named events[1]. The only difference between the two is, in a context switch between threads of same process, processor keeps the Translation Look Aside Buffer (TLB), but in case of a context switch to another process TLB is flushed as processor transitions to a new virtual address space; we conjecture that the impact would be negligible.

About the memory cost, all modern operating systems recognize when an executable binary or shared library gets loaded multiple times; OS keeps a single copy of the related memory pages on physical memory, and separately maps those to virtual address spaces of each process using those. This not only saves physical RAM, but also allows better space locality for L2/L3 processor caches. Hence the memory consumption of multiple instances of our daemon processes, each loading NVRTC library (*nvrtc64 80.dll is almost 15MB*) to their own address space, is almost the same as the consumption of a single instance.

4.4.4 Speedup Achieved with Parallel Compilation

At the end of each batch of experiments main application dumps the collected raw measurements to a file. We imported this data to Matlab filtered by experiment and measurement types, and aggregated the experiment values for each population size to produce the Tables 4.1-4.3, and to create the Figures 4.13-4.15 which illustrate the parallelized compilation times and speed-up ratios achieved.

It can be seen that parallelized in-process compilation of genetic programming individuals is faster for all problems and population sizes when compared to in-process compilation without parallelization; furthermore in-process compilation without parallelization itself was shown to be faster than regular command line nvcc compilation on previous section.

Parallel compilation brought the per individual compilation time to 2.17 ms/individual for 5-bit Multiplier, to 2.20 ms/individual for Keijzer-6 regression and to 2.13 ms/individual for the Search Problem; these are almost an order of magnitude faster

[1] on Windows operating system named events is the fastest IPC primitive, upon which all others (i.e. mutex, semaphore) are implemented

than previous published results. Also we measured a compilation speedup of ×3.45 for regression problem, ×5.26 for search problem, and ×7.60 for multiplication problem, when compared to the latest Nvcc V8 compiler, without requiring any code modification, and without any runtime performance penalty.

Notice that our experiment platform consisted of dual Xeon E5-2670 processors running at 2.6Ghz; for compute bound tasks increase on processor frequency almost directly translates to performance improvement at an equal rate[1]. Therefore we can conjecture that to be able to compile a population of 300 individuals at sub-millisecond durations, the required processor frequency is around 2.6 × 2.13 = *5.54Ghz*[2] which is currently available.

Table 4.1 Compilation Times by Compilation Methods for Search Problem with 300 individuals

Compilation Method	Compilation Time		Speedup Ratio	
	Per Individual	Total	In-process compilation	NVCC compilation
NVCC	11.20 ms	3.36 sec	-	1.00
In-process	7.76 ms	2.33 sec	1.00	1.44
2 daemons	3.81 ms	1.14 sec	2.04	2.93
4 daemons	2.53 ms	0.76 sec	3.07	4.41
6 daemons	2.23 ms	0.67 sec	3.48	5.01
8 daemons	2.13 ms	0.64 sec	3.65	5.26

[1] Assuming all other things being equal

[2] once again, under assumption of all other things being equal. 2.13 is the compilation time of Search Problem with 8 daemons

Table 4.2 Compilation Times by Compilation Methods for Keijzer-6 Regression with 300 individuals

Compilation Method	Compilation Time		Speedup Ratio	
	Per Individual	Total	In-process compilation	NVCC compilation
NVCC	7.63 ms	2.29 sec	-	1.00
In-process	4.14 ms	1.24 sec	1.00	1.83
2 daemons	2.92 ms	0.88 sec	1.42	2.60
4 daemons	2.45 ms	0.73 sec	1.69	3.10
6 daemons	2.20 ms	0.66 sec	1.88	3.45
8 daemons	2.25 ms	0.67 sec	1.84	3.37

Table 4.3 Compilation Times by Compilation Methods for 5-bit Multiplier Problem with 300 individuals

Compilation Method	Compilation Time		Speedup Ratio	
	Per Individual	Total	In-process compilation	NVCC compilation
NVCC	17.20 ms	5.16 sec	-	1.00
In-process	13.29 ms	3.99 sec	1.00	1.24
2 daemons	6.15 ms	1.85 sec	2.16	2.69
4 daemons	3.23 ms	0.97 sec	4.12	5.12
6 daemons	2.42 ms	0.73 sec	5.49	6.82
8 daemons	2.17 ms	0.65 sec	6.11	7.60

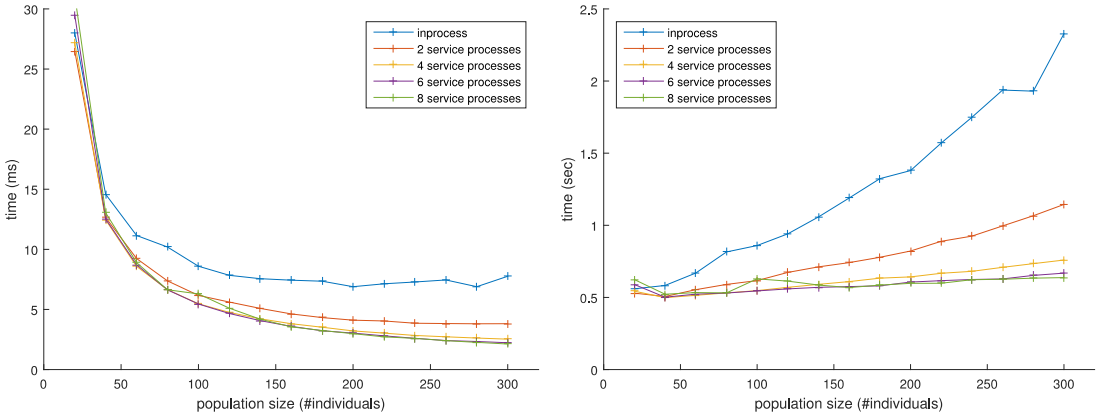


Figure 4.13 Nvcc compilation times for Search Problem by number of servicing resident processes. (left) per individual (right) total

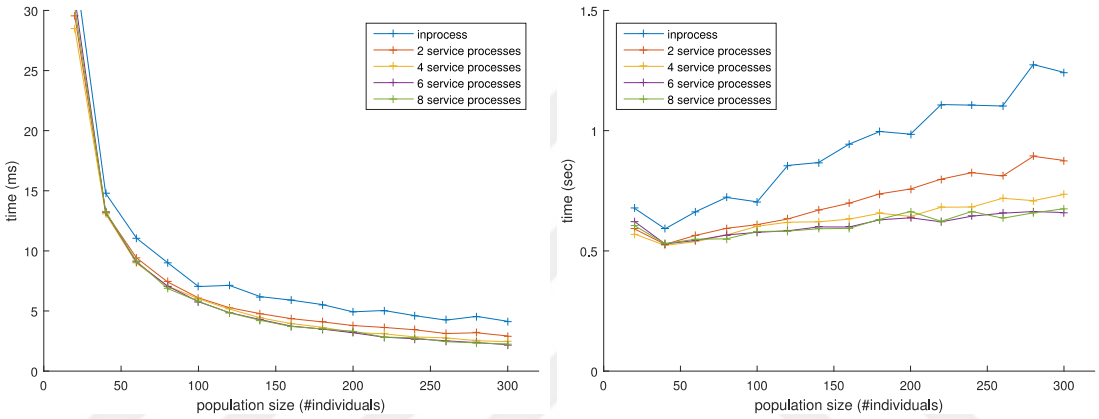


Figure 4.14 Nvcc compilation times for Keijzer-6 regression by number of servicing resident processes. (left) per individual (right) total

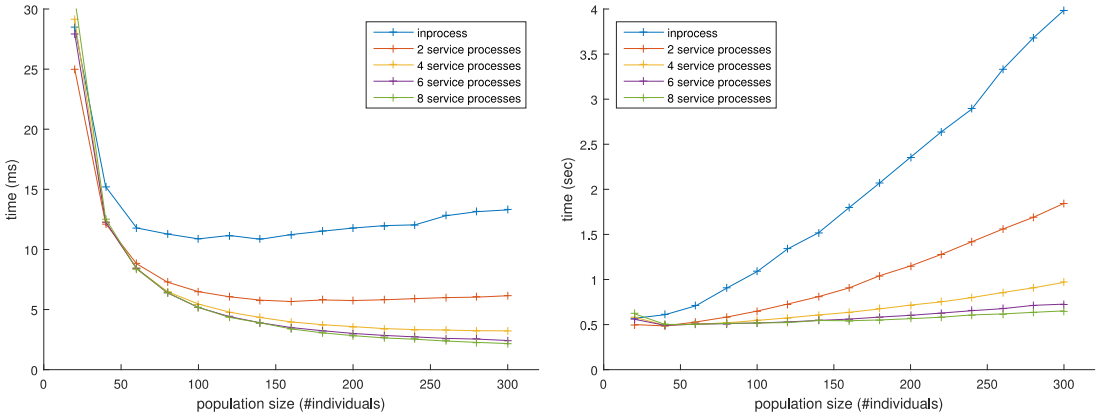


Figure 4.15 Nvcc compilation times for 5-Bit Multiplier by number of servicing resident processes. (left) per individual (right) total

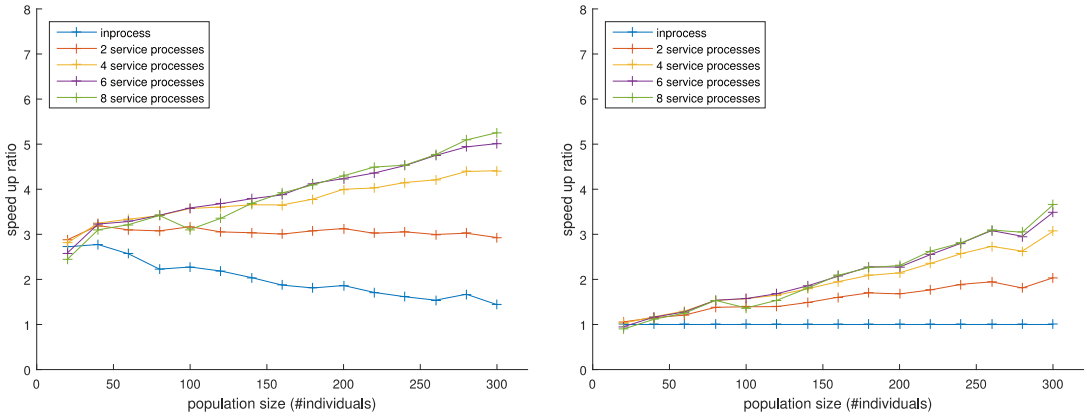


Figure 4.16 Parallelization speedup ratios on Search problem (left) vs conventional compilation (right) vs in-process compilation

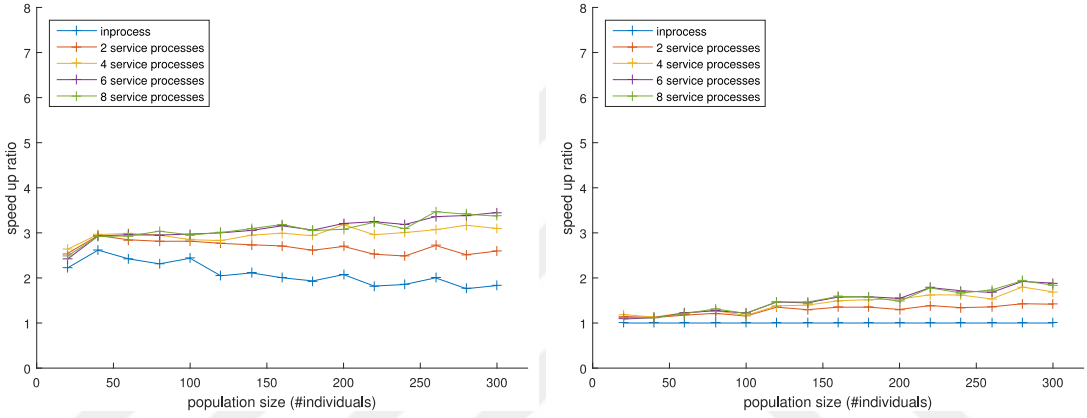


Figure 4.17 Parallelization speedup ratios on Keijzer-6 regression (left) vs conventional compilation (right) vs in-process compilation

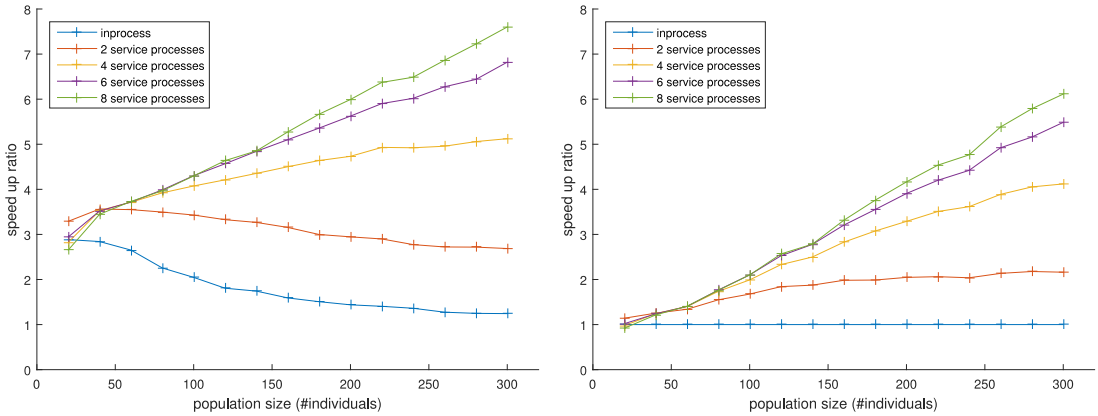


Figure 4.18 Parallelization speedup ratios on 5-Bit multiplier (left) vs conventional compilation (right) vs in-process compilation

CHAPTER 5

INTERPRETER FOR GENERAL PROGRAM SYNTHESIS WITH GENETIC PROGRAMMING ON GPU

In this chapter we present a new general purpose interpreter for grammatical genetic programming designed to support arbitrary grammars. We implemented it both for GPU using CUDA and for CPU using C#. We further implemented a byte-code generator for the opcodes exposed by the interpreter, and integrated it to our grammatical evolution engine as terminal tokens. Any grammar with terminal tokens consisting of these opcodes, produce derivations which are valid sequences of opcodes by gathering the terminals in depth first order and replacing each one with the respective opcode; such opcode sequences can be directly executed on our interpreter,

5.1 Implementation Options for Genetic Programming on GPU

There exists three implementation approaches for genetic programming on GPU; these are compilation, interpretation and direct generation of machine code.

5.1.1 Compiled Approach

In the compiled approach, individuals from population are compiled on the CPU to specific GPU machine code, and the resulting binary is uploaded to GPU where they run in parallel. The problem with compiled approach is that, it is known to have a prohibitive overhead compared to other two, which is the subject addressed on previous chapter.

5.1.2 Direct Generation of Assembly or Machine Code

The direct generation of machine code approach has two variants; first one involves directly generating PTX level assembly for individuals, which still requires a second level compilation to be able to run on GPU, and the second one involves direct generation of final machine code. Both variants involve implementing a mechanism which takes the behavior defined by the genome of individual and translate it to one of the vendor specific executable formats; these are pretty much equivalent to implementing a stripped down compiler on its own.

5.1.2.1 Direct Generation of PTX Followed by JIT Compilation

PTX is an intermediate representation used by NVidia which is architecture agnostic across their different lines of GPU hardware; it acts as a backward/forward compatibility layer, through which old compiled codes run on new hardware, and newly compiled codes run on older hardware. Naturally the cost for this abstraction is the need for a second level of compilation to the actual architecture specific machine code of GPU. This second layer compiles PTX code to a CUBIN object. It is available as part of the NVCC command line compiler, but a lighter JIT version of it resides as part of graphics driver and can be accessed through CUDA Driver API.

Direct generation of PTX involves two things; first implementing a code generator that translates individuals to PTX (which is a non-trivial implementation equivalent to implementing a crude PTX compiler), and second compiling the generated PTX with JIT compiler on driver API. JIT compilation has an overhead of its own, but it is negligible compared to full compilation with command line NVCC compiler.

5.1.2.2 Direct Generation of CUBIN

The technique for direct generation of CUBIN objects is called GMGP (GPU Machine Code Genetic Programming) and it is proposed in [52]. CUBIN objects are in an undocumented, architecture specific, machine code binary format. GMGP generate CUBIN objects with information obtained by byte level comparative analysis of many

small compiled binaries, and combining these fragments in a specific way. Authors present how they extract the machine code information from CUBIN fragments in a semi-automatic way, as the format is a little bit different for every generation of graphics hardware and prone to further changes even by new driver versions.

5.1.3 Interpretation Approach

A compromise between compilation and machine code generation is the interpretation approach; it consists of implementing an application on the target computing platform, that accepts a sequence of opcodes in a specific encoding, and performs the corresponding computation as a proxy. During an interpreter session the only code that is executed on the processor is that of interpreter itself; the memory region containing the opcodes is read as data and never reach the hardware processor directly.

In our case the target platform is the GPU, therefore we implemented a CUDA application which takes a device side buffer as the code memory, and decodes the byte sequences as variable size opcodes.

5.2 Implementation

Our implementation consists of two interpreters (one on GPU and other on CPU) with identical behavior, a set of opcodes that map the functionality exposed by the interpreter to terminal tokens of the grammar, and a base grammar that sequences these tokens as expected by the interpreter and expose the sequences as non-terminals.

5.2.1 Interpreter

We created two identical implementations of our interpreter; one in CUDA targeting GPU, and other in C# targeting CPU. A byte sequence produce the same result when run on any of the implementation, as the opcode side effects (i.e. change in state variables) for each opcode is exactly the same.

This twin implementation even includes replication of unexpected behaviors due to platform differences; one example that was hard to track down is the behavioral difference between CPU and GPU when an arithmetic operation producing NaN or Inf is executed (i.e. division by zero). In such case normally a CPU raise an exception while a GPU will not, but other than that both platforms signal the situation in conformance to the IEEE 754 spec, which is not very interesting. What's interesting is what happens when a NaN (or Inf) gets typecast to an int. Basically what should happen is defined as "undefined behavior" by IEEE 754 spec, thus all hardware and compilers are free to resolve the situation in any way they see fit. We encountered this particular mismatch of behavior, when the looplimit assignment on line 16 of Listing 5.1 receive a NaN or Inf from *interpret_expression()* function, which gets cast to different int values on GPU and CPU, resulting in different numbers of loop executions. We eliminated this mismatch by adding a finiteness check before that assignment, and assigning 0 to looplimit for non-finite values.[1]

Our interpreter implementations consist of three functions only. The entry point is the *interpret_statements()* function which runs a sequence of statements sequentially (see Listing 5.1).

```
1     while ¬(opcode₁ = <eos>)
2         IP ← IP+1
3         opcode ← code[IP]
4         switch opcode₁
5             case <assign-var>
6                   vars[opcode₂] ← interpret_expression()
7
8             case <assign-output>
9                   outputs[opcode₂] ← interpret_expression()
10
11            case <if>
12                  if interpret_bool_expression()
13                      interpret_statements()
14
15            case <for>
16                  looplimit ← interpret_expression()
17                  codeBlockStart ← IP
```

[1] The reason turns out to be DirectX compatibility; ints being devoid of Inf signaling capabilities like floats, CUDA returns a pattern of all bits set when you divide an int by zero or try to cast an Inf to int: https://devtalk.nvidia.com/default/topic/822614/cuda-6-0-const-int-warning-division-by-zero/?offset=5

```
18                  loopdepth ← loopdepth + 1
19                  for loopvar[loopdepth] ∈ (0,looplimit)
20                       IP = codeBlockStart
21                       interpret_statements()
22                  loopdepth ← loopdepth - 1
23
24              case <exit>
25                  halt()
```

Listing 5.1 – Pseudo-code of interpret_statements() function

On Listing 5.1 the sub-indices on $opcode_1$ and $opcode_2$ refer to first and second bytes of the integer encoding the opcode, where first byte is the least significant byte. To emulate nested loops, an array of loop indices is kept along all other state of an individual.

The complete state kept by the interpreter for an individual running on a test case consists of a base pointer to the beginning of memory region containing the opcodes of current individual, an instruction pointer used as increments over base, a flag to signal halted state, a pointer to the memory region containing the input data for the specific test case this instance of individual is working on, a pointer to memory region reserved for the storage of the outputs by this individual, an array to hold local variables, an array to hold loop variables and an integer to indicate the level of loop nesting at current IP. Sizes of mentioned arrays in state can be adjusted at initialization of interpreter.

```
1   switch opcode₁
2       case <lt>
3           return interpret_expression() < interpret_expression()
4
5       case <gt>
6           return interpret_expression() > interpret_expression()
7
8       case <eq>
9           return interpret_expression() = interpret_expression()
10
11      case <not>
12          return ¬interpret_bool_expression()
13
14      case <and>
15          l ← interpret_bool_expression()
16          r ← interpret_bool_expression()
17          return l ∧ r
18
19      case <or>
```

```
20          l ← interpret_bool_expression()
21          r ← interpret_bool_expression()
22          return l ∨ r
```

Listing 5.2 – Pseudo-code of interpret_bool_expression() function

5.2.2 Base Grammar

In its universal form the BNF grammar for interpreter consists of a mixture of terminal tokens corresponding to an opcode with a concrete implementation on the interpreter (tokens indicated with bold on Listing 5.3), and non-terminal tokens with the purpose of creating valid sequences of opcodes.

```
<expr>       ::= <bi-op><expr><expr2> | <const> | <var> | <input>
                 | <loop-index> | <input-length> | <output-length>
                 | <output>
<bool-expr>  ::= <bi-comp-op><expr><expr> | <not><bool-expr>
                 | <bi-bool-op><bool-expr><bool-expr>
<bi-op>      ::= <add> | <sub> | <mul> | <div>
<bi-comp-op> ::= <lt> | <gt> | <eq>
<bi-bool-op> ::= <and> | <or>
<statement>  ::= <assign-var><expr> | <assign-output><expr>
                 | <if><bool-expr><statements>
                 | <for><expr><statements>  | <exit>
<statements> ::= <statement><eos>
                 | <statement><statement><eos>
                 | <statement><statement><statement><eos>
```

Listing 5.3 – Universal Grammar for Interpreter

This is the most general form of the grammar where all interpreter functionality is exposed and no restriction is imposed (other than grouping token sequences as expected by interpreter). Starting from this base new grammars biased or specialized towards specific problems can be constructed by introducing additional rules defining constructs with complex structure, or imposing further restrictions.

5.3 Performance

To measure the performance of our GPU interpreter, we used the Keijzer-6 regression using the previously listed grammar. For all experiments we used a Maxwell class GPU

running at 1.1Ghz with 2GB RAM. For a population of 100 individuals, we measured that GPU interpretation takes 190.7 µsecs/generation on average; and for 200 individuals we measured 423.8 µsecs/generation mean interpretation time. Figure 5.1 and 5.2 show the distribution of interpreter times over 100 generations.

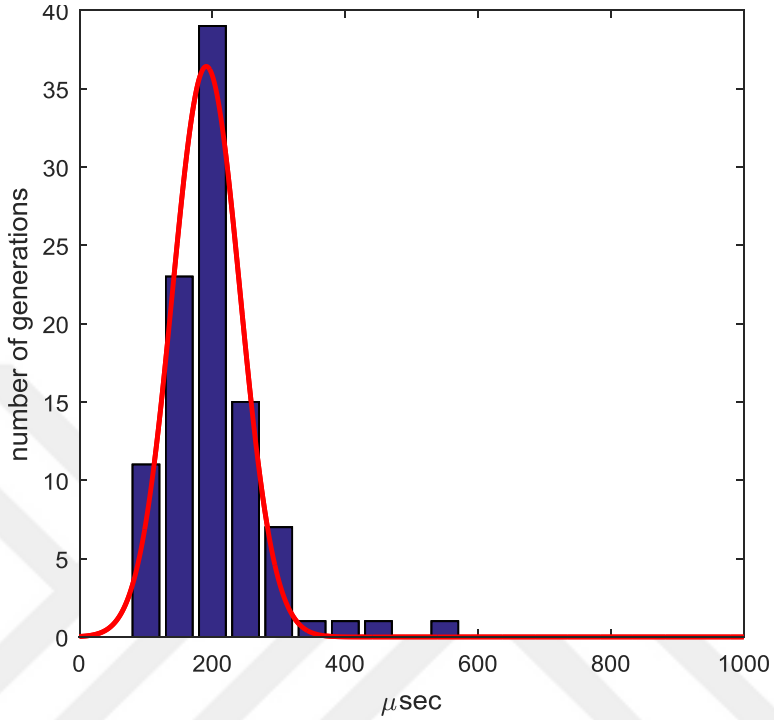


Figure 5.1 GPU interpretation time per generation for 100 individual population

As the GPU process all (individual,test case) pairs in parallel, the reported measurements are for the evaluation of a whole population for a single generation. Of course the parallel processing capability of the GPU is limited by the number of cores available, but a GPU with 1000+ cores can process 100 individuals on 10 test cases simultaneously, by launching an independent interpreter instance for each. On the other hand, evaluation of a single individual with a single test case would take a very similar time on the GPU, because a single core would be processing the pair at the same speed as before, while all remaining cores just idle.

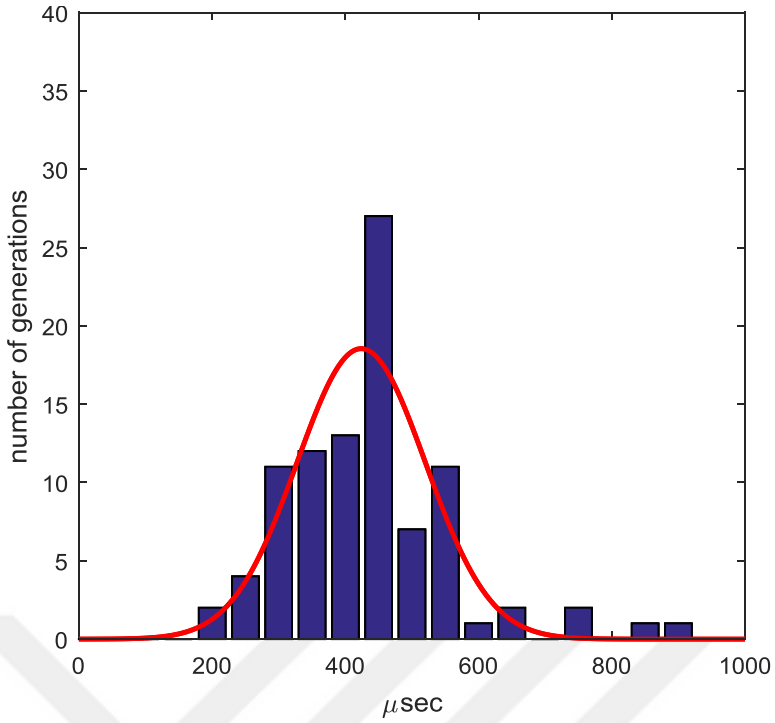


Figure 5.2 GPU interpretation time per generation for 200 individual population

These measured timings only consist of the interpreter activity on the GPU which correspond to the evaluation phase of GP; they do not include selection, genetic operators, phenotype mapping with grammar which happen on CPU, or data upload/download between CPU and GPU over PCI-E bus.

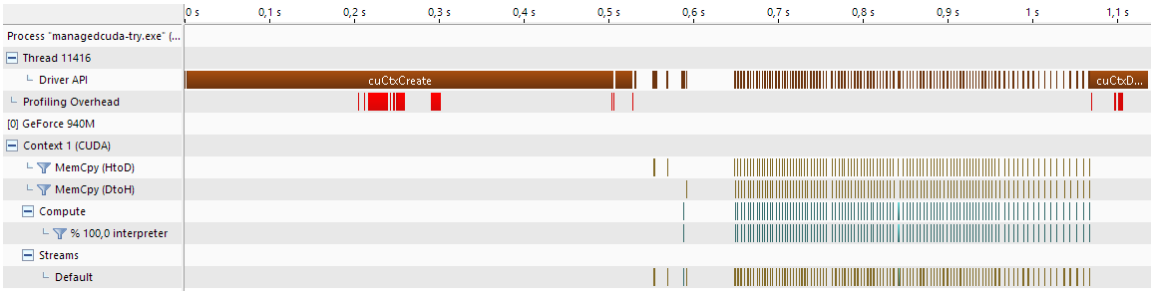


Figure 5.3 Performance profiling result of 100 invocation of GPU interpreter each corresponding to a generation

As it can be seen on Figure 5.3, 100 generation of evolution with GPU interpreter takes almost a second (after removing the profiling overhead indicated by red part) for a population size of 100 individuals; furthermore half of this is time spent on CUDA context setup to initialize the GPU. As a result it is shown that 100 generations with

100 individuals take only 0.5 second, which corresponds to 500ms/100gen = 5ms per generation.

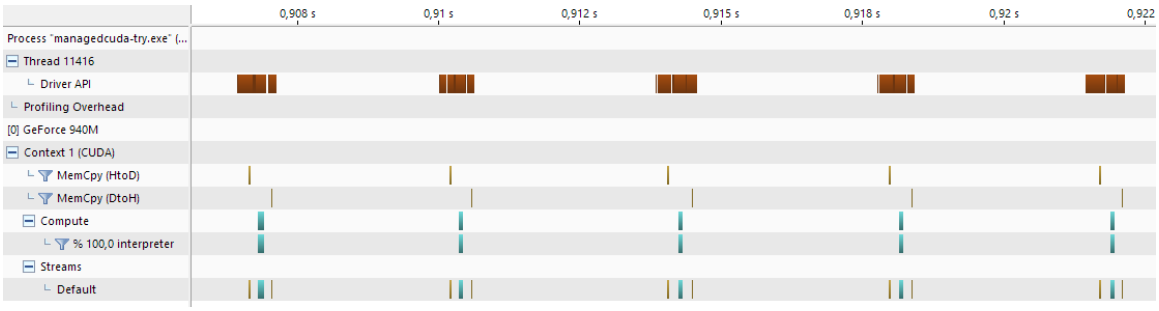


Figure 5.4 Performance profile close-up of five generations

On the other hand upon closer inspection (see Fig 5.4) it can be seen that the actual time spent on GPU is very sparse; this confirms the previous two measurements which states that every generation takes approximately 5ms but time spent on GPU interpretation is 190.7 µsecs/generation on average; hence we can say that only $\frac{190\mu sec}{5000\mu\text{sec}} = 3.8\%$ of processing time is used by the GPU, while 96.2% is used by CPU or spent waiting a memory/IO operation to complete.

CHAPTER 6

RESULTS AND DISCUSSION

In this dissertation we analyzed the grammatical genetic programming in a performance context and investigated the acceleration options available through the use of heterogeneous computing and parallelization. *Evaluation* phase of genetic programming is known to benefit from parallelization especially on GPU, but we show that *compilation* of individuals can also be parallelized on CPU simultaneously to parallel evaluation. Prior work on genetic programming on GPU identified, and we confirmed that, compilation itself is the single most time consuming phase with compilation times ranging from 15 to 70 ms per individual.

Our main contribution is a new parallel and in-process compilation method which consistently achieves compilation times around 2 ms/individual, measured with three different problems on a dual 2.6Ghz Xeon processor based platform. As an increase in processor frequency translates to performance almost linearly for compute bound tasks independent of parallelizability, we conjecture that the method we propose can easily achieve sub-millisecond compilation times per individual at 3.5+ Ghz clock speeds, assuming all other things being equal with no new bottlenecks introduced.

Another contribution we present is a new benchmark problem for grammar genetic programming, with an emphasis on general program synthesis domain. It proves to be easier than the integer sorting problem *(as there is no published work evolving an integer sort implementation without invoking high level structures like swap)* but obviously harder than the Minimum/Maximum Problem. For this benchmark we investigated how the convergence to a solution is affected by parameters like population size, maximum number of generations explored and number of test cases

employed. We used this problem along with other community recommended benchmarks on our subsequent experiments.

Our final contribution is a general purpose interpreter running on GPU. It provides a small set of universal computation primitives that can be mapped to arbitrary grammars. The ability to target arbitrary grammars is paramount for general program synthesis problems. It is the first implementation of grammatical evolution on GPU with an interpreter.

In our compiler and interpreter experiments we always observed the CPU to be the bottleneck, even when working with very capable hardware setups. Therefore a promising line of future investigation is moving all phases (i.e. evaluation, selection, crossover, mutation, genotype-phenotype mapping with grammar) of grammatical genetic programming to GPU side eliminating most, if not all, roundtrips to CPU. Another approach which may be worthy of investigation is implementation of a very small soft-core processor on FPGA, which implements the computation primitives of the interpreter we propose in hardware. Multiple instances of this core running in parallel on the same FPGA fabric, combined with another general purpose processor to handle the evolutionary operations and scheduling of individuals to mentioned GP cores.

REFERENCES

[1] Miller, J. F., (1999). “Digital Filter Design at Gate-level using Evolutionary Algorithms”, Proceedings of The Genetic and Evolutionary Computation Conference-GECCO, 13-17 July 1999, Orlando, Florida.

[2] Frode, M. H., Hartmann, M., Eskelund, F., Haddow, P. C., and Miller J. F., (2002). “Evolving Fault Tolerance on an Unreliable Technology Platform”, Proceedings of The Genetic and Evolutionary Computation Conference-GECCO, 9-13 July 2002, New York, USA, 171–177.

[3] Canham, R. and Tyrrell, A., (2002). “Evolved fault tolerance in evolvable hardware”, Proceedings of the Congress on Evolutionary Computation CEC’02, 2002, (2):1267–1271.

[4] Schnier, T. and Yao, X., (2003). “Using negative correlation to evolve fault-tolerant circuits”, Evolvable Systems From Biology to Hardware, 35–46.

[5] Nassar, K., (2002). “Automatic Creation of Digital Fast Adder Circuits by Means of Genetic Programming”, Genetic Algorithms and Genetic Programming at Stanford, 2002, 187–194.

[6] Aguirre, A. H., and Coello, C. A. C., (2004). “Using Genetic Programming and Multiplexers for the Synthesis of Logic Circuits”, Engineering Optimization, (36):491–511.

[7] Helmuth, T. and Spector, L., (2013). “Evolving a digital multiplier with the pushgp genetic programming system”, Proceedings of The Genetic and Evolutionary Computation Conference-GECCO, 6-10 July 2013, Amsterdam, Netherlands, 1627.

[8] Sapargaliyev, Y. A. and Kalganova, T. G. (2012). “Open-ended evolution to discover analogue circuits for beyond conventional applications”, Genet. Program. Evolvable Mach., 4(13):411–443.

[9] Ponting, C. P. and Hardison, R. C., (2011). “What fraction of the human genome is functional?”, Genome Res., 11(21):1769–1776.

[10] Hugosson, J., Hemberg, E., Brabazon, A. and O’Neill, M., (2010). “Genotype representations in grammatical evolution,” Appl. Soft Comput., 1(10):36–43.

[11] Nicolau, M, O’Neill, M. and Brabazon, A., (2012). “Termination in Grammatical Evolution: grammar design, wrapping, and tails”, 2012 IEEE Congress on Evolutionary Computation, 2012, 1–8.

[12] Koza, J. R., (1993). Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems). MIT Press.

[13] Koza, J. R., (1994). Genetic Programming II: Automatic discovery of reusable programs. MIT Press.

[14] Koza, J. R., Andre, D., Bennett, F. H. and Keane, M. A., (1999). Genetic Programming III: Darwinian Invention and Problem Solving. Morgan Kaufmann Publishing.

[15] Koza, J. R., Keane, M. A., Streeter, M. J., Mydlowec, W., Yu, J. and Lanza, G., (2003). Genetic programming IV: Routine human-competitive machine intelligence, (5). Kluwer Academic Publishers.

[16] Poli, R. and Langdon, W. B., (2008). A Field Guide to Genetic Programming, (1). Lulu Press.

[17] Worzel, W. P., Riolo, R., Kotanchek, M., and Kordon, A., (2016). Genetic Programming Theory and Practice XIII. Springer International Publishing, Cham.

[18] Riolo, R., Worzel, W. P. and Kotanchek, M., (2014). Workshop on Genetic Programming, Genetic Programming Theory and Practice XII. Springer International Publishing, Cham.

[19] Moore, J. H., Riolo, R., and Kotanchek, M., (2014). Genetic Programming Theory and Practice XI. Springer Publishing, New York, NY.

[20] Riolo, R., (2013). Genetic Programming Theory and Practice X. Springer Publishing, New York, NY.

[21] Riolo, R., Vladislavleva, E. and Moore, J. H. (9th : 2011 : A. A. Workshop on Genetic Programming, Genetic programming theory and practice IX. Springer Publishing, New York, NY.

[22] Riolo, R., McConaghy, T. and Vladislavleva, E., (2011). Genetic programming theory and practice VIII. Springer Publishing, New York, NY.

[23] O'Reilly, U.M., Riolo, R. and McConaghy, T., (2010). Genetic Programming Theory and Practice VII. Springer Publishing US, Boston, MA.

[24] Riolo, R., Soule, T. and Worzel, B., (2009). Genetic Programming Theory and Practice VI. Springer Publishing, New York, NY.

[25] Soule, T., Worzel, B. and Riolo, R., (2008). Genetic Programming Theory and Practice V. Springer Publishing.

[26] Riolo, R., Soule, T. and Worzel, B., (2007). Genetic Programming Theory and Practice IV. Springer Publishing US, Boston, MA.

[27] Yu, T., Riolo, R. and Worzel, B., (2006). Genetic programming theory and practice III. Springer Publishing.

[28] O'Reilly, U. M., (2005). Genetic programming theory and practice II. Springer Science+Business Media.

[29] Riolo, R. and Worzel, B., (2003). Genetic programming theory and practice. Kluwer Academic.

[30] Ryan, C., Collins, J. and O'Neill, M., (1998). “Grammatical evolution: Evolving programs for an arbitrary language”, Lecture Notes in Computer Science, (1391):83–96.

[31] O'Neill, M. and Ryan, C., (2000). “Grammar based function definition in Grammatical Evolution”, Proceedings of Genetic and Evolutionary Computing Conference, 3:485–490.

[32] O'Neill, M., Ryan, C., and Nicolau, M., (2001). “Grammar Defined Introns: An Investigation Into Grammars, Introns, and Bias in Grammatical Evolution”, Proceedings of The Genetic and Evolutionary Computation Conference-GECCO, 2001, San Fr. California, USA, 7-11, July, 2001, 97–103.

[33] Keijzer, M., O'Neill, M., Ryan, C., and Cattolico, M., (2002). “Grammatical Evolution Rules: The mod and the Bucket Rule”, Proceedings of 5th European Conference on Genetic Programming EuroGP, 3-5 April 2002, Kinsale, Ireland, 123–130.

[34] O'Neill, M., Ryan, C., Keijzer, M., and Cattolico, M., (2003). “Crossover in Grammatical Evolution”, Genetic Programming and Evolvable Machines, 1(4):67–93.

[35] O'Neill, M., Brabazon, A., Nicolau, M., Garraghy, S. M. and Keenan, P., (2004). “πGrammatical Evolution”, Proceedings of The Genetic and Evolutionary Computation Conference-GECCO, 26-30 June 2004, Seattle, Washington, 617–629.

[36] O'Neill, M. and Ryan, C., (2004). “Grammatical evolution by grammatical evolution: The evolution of grammar and genetic code”, Proceedings of the Seventh European Conference on Genetic Programming EuroGP, 2004, 138–149.

[37] Karpuzcu, U. R., (2005). “Automatic verilog code generation through grammatical evolution”, Proceedings of The Genetic and Evolutionary Computation Conference-GECCO, 25-29 June 2005, Washington D.C., USA, 394.

[38] O'Neill, M., Nicolau, M., and Agapitos, A., (2014). “Experiments in program synthesis with grammatical evolution: A focus on Integer Sorting”, Proceedings of IEEE Congress on Evolutionary Computation CEC, 6-11 Jul 2014, Beijing, China, 1504–1511.

[39] Byrne, J., Fenton, M., Hemberg, E., McDermott, J., and O'Neill, M., (2015). “Optimising Complex Pylon Structures with Grammatical Evolution”, Information Sciences, (316):582–597.

[40] Fagan, D., Fenton, M., and O'Neill, M., (2016). “Exploring Position Independent Initialisation in Grammatical Evolution”, Proceedings of 2016 IEEE Congress on Evolutionary Computation CEC, 24-29 July 2016, Vancouver, Canada, 5060–5067.

[41] Fenton, M., McDermott, J., Fagan, D., Forstenlechner, S., M. O’Neill, and Hemberg, E., (2017). “PonyGE2 : Grammatical Evolution in Python”, arXiv Preprint

[42] Harding, S. and Banzhaf, W., (2007). “Fast Genetic Programming on GPUs”, in

Proceedings of the 10th European Conference on Genetic Programming EuroGP, 11-13 April 2007, Valencia, Spain, (4445):90–101.

[43] Chitty, D.M., (2007). “A data parallel approach to genetic programming using programmable graphics hardware”, Proceedings of Conference on Genetic and Evolutionary Computing, 2007, 1566–1573.

[44] Langdon, W.B. and Banzhaf, W., (2008). “A SIMD Interpreter for Genetic Programming on GPU Graphics Cards”, Genetic Programming, 73–85.

[45] Wilson, G. and Banzhaf, W., (2009). “Deployment of CPU and GPU-based genetic programming on heterogeneous devices”, Proceedings of The Genetic and Evolutionary Computation Conference-GECCO, 8-12 July 2009, Montréal, Canada, 2531.

[46] Harding, S.L. and Banzhaf, W., (2009). “Distributed genetic programming on GPUs using CUDA”, Workshop on Parallel Architectures and Bioinspired Algorithms, 2009, 1–10.

[47] Langdon, W.B. and Harman, M., (2010). “Evolving a CUDA kernel from an nVidia template,”, IEEE Congress on Evolutionary Computation, 2010, 1–8.

[48] Dietz, H. and Young, B., (2010). “MIMD Interpretation on a GPU” Lang. Compil. Parallel Computing, 65–79.

[49] Langdon, W.B., (2011). “Graphics processing units and genetic programming: an overview”, Soft Computing - A Fusion of Foundation Methodology and Application, 8(15):1657–1669.

[50] Pospichal, P., Murphy, E., O’Neill, M., Schwarz, J. and Jaros, J., (2011). “Acceleration of Grammatical Evolution Using Graphics Processing Units”, Proceedings of The Genetic and Evolutionary Computation Conference-GECCO, 12-16 July 2011, Dublin, Ireland, 431–438.

[51] Lewis, T.E. and Magoulas, G.D., (2011). “Identifying similarities in TMBL programs with alignment to quicken their compilation for GPUs”, Proceedings of The Genetic and Evolutionary Computation Conference-GECCO, 12-16 July 2011, Dublin, Ireland, 447.

[52] da Silva, C.P., Dias, D.M., Bentes, C., Pacheco, M.A.C. and Cupertino, L.F., (2015). “Evolving GPU machine code”, Journal of Machine Learning Research, 1(16):673–712.

[53] McDermott, J., De Jong, K., O’Reilly, U.M., White, D.R., Luke, S., Manzoni, L., Castelli, M., Vanneschi, L., Jaskowski, W., Krawiec, K. and Harper, R., (2012). “Genetic programming needs better benchmarks”, Proceedings of The Genetic and Evolutionary Computation Conference-GECCO, 07-11 July 2012, Philedelphia, USA, 791.

[54] White, D.R., McDermott, J., Castelli, M., Manzoni, L., Goldman, B.W., Kronberger, G., Jaśkowski, W., O’Reilly, U.M., and Luke, S., (2012). “Better GP benchmarks: community survey results and proposals”, Genetic Programming and Evolvable Machines, 1(14):3–29.

[55] Ayral, H. and Albayrak, S. (2017). “Effects of Population, Generation and Test Case Count on Grammatical Genetic Programming for Integer Lists”, Journal of Software, 12(5):483-492.

[56] Helmuth, T. and Spector, L., (2015). “General Program Synthesis Benchmark Suite”, Proceedings of The Genetic and Evolutionary Computation Conference-GECCO, 2015, 1039–1046.

[57] Keijzer, M., (2003). “Improving Symbolic Regression with Interval Arithmetic and Linear Scaling”, Proceedings of EuroGP, 2003, 70–82.

[58] Nicolau, M. and Fenton, M., (2016). “Managing Repetition in Grammar-Based Genetic Programming”, Proceedings of The Genetic and Evolutionary Computation Conference-GECCO, 2016, 765–772.

CURRICULUM VITAE

PERSONAL INFORMATION

Name Surname	: Hakan AYRAL
Date of birth and place	: 1980, İstanbul
Foreign Languages	: French, English
E-mail	: hayral@gmail.com

EDUCATION

Degree	**Department**	**University**	**Date of Graduation**
MSc.	Computer Eng.	Galatasaray University	2008
BSc.	Computer Eng.	Kadir Has University	2005
High School		Lycée de Galatasaray	1999

WORK EXPERIENCE

Year	**Corporation/Institute**	**Enrollment**
2008-ongoing	Galatasaray University Mathematics Department	Guest Lecturer
2005-2008	Moorestephens Turkey	CIO

PUBLICATIONS

Papers

1. Ayral, H. and Albayrak S., (2017). “Effects of Population, Generation and Test Case Count on Grammatical Genetic Programming for Integer Lists”, Journal of Software, 12(5):483-492.
2. Ayral, H. and Albayrak S., (2017). “Parallel and in-process compilation of individuals for genetic programming on GPU”, (inreview).

Preprints

1. Uludağ, A. M. and Ayral, H., (2017). “Dynamics of a family of continued fraction maps,” , arXiv preprint.
2. Zeytin, A. and Ayral, H. and Uludağ, A. M., (2017). “InfoMod: A visual and computational approach to Gauss’ binary quadratic forms,”, arXiv preprint.
3. Uludağ, A. M. and Ayral, H., (2016). “A subtle symmetry of Lebesgue’s measure,” , arXiv preprint.
4. Uludağ, A. M. and Ayral, H., (2016). “On the involution of the real line induced by Dyer’s outer automorphism of PGL(2,Z),” , arXiv preprint.

Conference Papers

1. Ayral, H. and Yavuz, S., (2011). “An automated domain specific stop word generation method for natural language text classification”, International Symposium on Innovations in Intelligent Systems and Applications, 15-18 June 2011, İstanbul.

Book Chapters

1. Uludağ, A.M.,(appendix by Ayral, H.) (2015). “Actions of the Modular Group”, appeared in: Handbook of Group Actions, Athanase Papadopoulos, Lizhen Ji and S.-T. Yau (editors) International Press, 2015.

Projects

1. Participated to Tubitak 3501 project “Infographics of the modular group, class number problems and carks: InfoMod” Project No 113R017 .
2. Participated to Tubitak 1001 project “Hypergeometric Galois Actions (GAL-ACT)” Project No 110T690.

CERTIFICATES

Microsoft Certified System Engineer (MCSE)

MEMBERSHIPS

Hakan AYRAL is a member of American Mathematical Society (AMS).