Parallel and in-process compilation of individuals for genetic programming on GPU

Hakan Ayral [Corresp., 1] , **Songül Albayrak** [1]

[1] Department of Computer Engineering, Yildiz Technical University, İstanbul, Turkey

Corresponding Author: Hakan Ayral
Email address: hayral@gmail.com

Three approaches to implement genetic programming on GPU hardware are compilation, interpretation and direct generation of machine code. The compiled approach is known to have a prohibitive overhead compared to other two.

This paper investigates methods to accelerate compilation of individuals for genetic programming on GPU hardware. We apply in-process compilation to minimize the compilation overhead at each generation; and we investigate ways to parallelize in-process compilation. In-process compilation doesn't lend itself to trivial parallelization with threads; we propose a multiprocess parallelization using memory sharing and operating systems interprocess communication primitives. With parallelized compilation we achieve further reductions on compilation overhead. Another contribution of this work is the code framework we built in C# for the experiments. The framework makes it possible to build arbitrary grammatical genetic programming experiments that run on GPU with minimal extra coding effort, and is available as open source.

Parallel and in-process compilation of individuals for genetic programming on GPU

Hakan Ayral[1] **and Songül Albayrak**[1]

[1]**Yildiz Technical University, Computer Engineering Department, Istanbul, TURKEY**

Corresponding author:
Hakan Ayral[1]

Email address: hayral@gmail.com , songul@ce.yildiz.edu.tr

ABSTRACT

Three approaches to implement genetic programming on GPU hardware are compilation, interpretation and direct generation of machine code. The compiled approach is known to have a prohibitive overhead compared to other two.

This paper investigates methods to accelerate compilation of individuals for genetic programming on GPU hardware. We apply in-process compilation to minimize the compilation overhead at each generation; and we investigate ways to parallelize in-process compilation. In-process compilation doesn't lend itself to trivial parallelization with threads; we propose a multiprocess parallelization using memory sharing and operating systems interprocess communication primitives. With parallelized compilation we achieve further reductions on compilation overhead. Another contribution of this work is the code framework we built in C# for the experiments. The framework makes it possible to build arbitrary grammatical genetic programming experiments that run on GPU with minimal extra coding effort, and is available as open source.

INTRODUCTION

Genetic programming is an evolutionary computation technique, 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 as test cases along with expected results. Grammatical genetic programming is a subfield of genetic programming, where the search space is restricted to a language defined as a BNF grammar, thus ensuring all individuals to be syntactically valid.

Processing power provided by graphic processing units (GPUs) make them an attractive platform for evolutionary computation due to the inherently parallelizable nature of the latter. First genetic programming implementations shown to run on GPUs were Chitty (2007) and Harding and Banzhaf (2007).

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 be 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 and cartesian genetic programming are inherently suitable for simple interpreters and circuit simulators implementable on a GPU. On the other hand grammatical genetic programming aims to make higher level constructs and structures representable, using individuals that represent strings of tokens belonging to a language defined by a grammar; unfortunately executing such a representation sooner or later requires some form of compilation or complex interpretation.

In this paper we first present three benchmark problems we implemented to measure compilation times with. We use grammatical genetic programming for the experiments, therefore we define the benchmark

46 problems with their grammars, test cases and fitness functions.

47 Then we set a baseline by measuring the compilation time of individuals for those three problems,
48 using the conventional CUDA compiler Nvcc. Afterwards we measure the speedup obtained by the
49 in-process compilation using the same benchmark problem setups. We proceed by presenting the obstacles
50 encountered on parallelization of in-process compilation. Finally we propose a parallelization scheme for
51 in-process compilation, and measure the extra speedup achieved.

PRIOR WORK

53 Harding and Banzhaf (2009) deals with the compilation overhead of individuals for genetic programming
54 on GPU using CUDA. Article proposes a distributed compilation scheme where a cluster of around 16
55 computers compile different individuals in parallel; and states the need for large number of fitness cases
56 to offset the compilation overhead. It correctly predicts that this mismatch will get worse with increasing
57 number of cores on GPUs, but also states that "a large number of classic benchmark GP problems fit
58 into this category". Based on figure 5 of the article it can be computed that for a population size of 256,
59 authors required *25 ms/individual* in total[1].

60 Langdon and Harman (2010b) presents first use of grammatical genetic programming on the GPU,
61 applied to a string matching problem to improve gzip compression; with a grammar constructed from
62 fragments of an existing string matching CUDA code. Based on figure 11 of the accompanying technical
63 reportLangdon and Harman (2010a) a population of 1000 individuals (10 kernels of 100 individuals
64 each) takes around 50 seconds to compile using nvcc from CUDA v2.3 SDK, which puts the average
65 compilation time to approximately *50 ms/individual*.

66 In Langdon (2011) an overview of genetic programming on GPU hardware is provided, along with a
67 brief presentation and comparison of compiled and interpreted approaches. As part of the comparison
68 it underlines the trade-off between the speed of compiled code versus the overhead of compilation, and
69 states that the command line CUDA compiler was especially slow, hence why interpreted approach is
70 usually preferred.

71 Pospichal et al. (2011) investigate the acceleration of grammatical evolution by use of GPUs, by
72 considering performance impact of different design decisions like thread/block granularity, different types
73 of memory on GPU, host-device memory transactions. As part of the article compilation to PTX form
74 and loading to GPU with JIT compilation on driver level, is compared with directly compiling to cubin
75 object and loading to GPU without further JIT compilation. For a kernel containing 90 individuals takes
76 540ms to compile to CUBIN with sub-millisecond upload time to GPU, vs 450ms for compilation to
77 PTX and 80ms for JIT compilation and upload to GPU using nvcc compiler from CUDA v3.2 SDK. Thus
78 PTX+JIT case which is the faster of the two achieves average compilation time of *5.88 ms/individual*.

79 Lewis and Magoulas (2011) proposes an approach for improving compilation times of individuals for
80 genetic programming on GPU, where common statements on similar locations are aligned as much as
81 possible across individuals. After alignment individuals with overlaps are merged to common kernels
82 such that aligned statements become a single statement, and diverging statements are enclosed with
83 conditionals to make them part of the code path only if the value of individual_ID parameter matches
84 an individual having that divergent statements. Authors state that in exchange for faster compilation
85 times, they get slightly slower GPU runtime with merged kernels as all individuals need to evaluate every
86 condition at the entry of each divergent code block coming from different individuals. In results it is
87 stated that for individuals with 300 instructions, compile time is 347 ms/individual if it's unaligned, and
88 *72 ms/individual* if it's aligned (time for alignment itself not included) with nvcc compiler from CUDA
89 v3.2 SDK.

90 da Silva et al. (2015) provides a comparison of compilation, interpretation and direct generation
91 of machine code methods for genetic programming on GPUs. Five benchmark problems consisting of
92 Mexican Hat and Salutowicz regressions, Mackey-Glass time series forecast, Sobel Filter and 20-bit
93 Multiplexer are used to measure the comparative speed of the three mentioned methods. It is stated that
94 compilation method uses nvcc compiler from CUDA V5.5 SDK. Compilation time breakdown is only
95 provided for Mexican Hat regression benchmark on Table 6, where it is stated that total nvcc compilation
96 time took 135,027 seconds and total JIT compilation took 106,458 seconds. Table 5 states that Mexican

---

[1] This number includes network traffic, XO, mutation and processing time on GPU, in addition to compilation times. In our case the difference between compilation time and total time has constantly been at sub-millisecond level per population on all problems; thus for comparison purposes compile times we present can also be taken as total time with an error margin of $^{1ms}/_{pop.size}$

2/16

97 Hat problem uses 400K generations and a population size of 36. Therefore we can say that an average
98 compilation time of $^{(135,027+106,458)}/_{36 \times 400,000} = $ *16.76 ms/individual* is achieved.

IMPLEMENTED PROBLEMS FOR MEASUREMENT

100 We implemented three problems as benchmark to compare compilation speed. They consist of a general
101 program synthesis problem, Keijzer-6 as a regression problem Keijzer (2003), and 5-bit Multiplier as a
102 multi output boolean problem . The latter two are included in the "Alternatives to blacklisted problems"
103 table on White et al. (2012).
104   We use grammatical genetic programming as our representation and phenotype production method;
105 therefore all problems are defined with a BNF grammar that defines a search space of syntactically valid
106 programs, along with some test cases and a fitness function specific to the problem. For all three problems,
107 a genotype which is a list of (initially random) integers derives to a phenotype which is a valid CUDA C
108 expression, or code block in form of a list of statements. All individuals are appended and prepended with
109 some initialization and finalization code which serves to setup up the input state and write the output to
110 GPU memory afterwards. See Appendix for BNF Grammars and codes used to surround the individuals.

Search Problem

112 Search Problem is designed to evolve a function which can identify whether a search value is present in
113 an integer list, and return its position if present or return -1 otherwise.
114   We first proposed this problem as a general program synthesis benchmark in Ayral and Albayrak (ress).
115 The grammar for the problem is inspired by O'Neill et al. (2014); we designed it to be a subproblem
116 of the more general *integer sort problem* case along with some others. It also bears some similarity to
117 problems presented in Helmuth and Spector (2015) based on, the generality of its usecase, combined with
118 simplicity of its implementation.
119   Test cases consist of unordered lists of random integers in the range $[0, 50]$, and list lengths vary
120 between 3 and 20. Test cases are randomly generated but half of them are ensured to contain the value
121 searched, and others ensured not to contain. We employed a binary fitness function, which returns 1 if the
122 returned result is correct (position of searched value or -1 if it's not present on list) or 0 if it's not correct;
123 hence the fitness of an individual is the sum of its fitnesses over all test cases, which evolutionary engine
124 tries to maximize.

Keijzer-6

126 Keijzer-6 function, introduced in Keijzer (2003), is the function $K_6(x) = \sum_{n=1}^{x} \frac{1}{n}$ which maps a single
127 integer parameter to the partial sum of harmonic series with number of terms indicated by its parameter.
128 Regression of Keijzer-6 function is one of the recommended alternatives to replace simpler symbolic
129 regression problems like quartic polynomial White et al. (2012).
130   For this problem we used a modified version of the grammar given in Nicolau and Fenton (2016),
131 and Fagan et al. (2016), with the only modification of increasing constant and variable token ratio as
132 the expression nesting gets deeper. We used the root mean squared error as fitness function which is the
133 accepted practice for this problem.

5-bit multiplier

135 5-bit multiplier problem consists of finding a boolean relation that takes 10 binary inputs to 10 binary
136 outputs, where two groups of 5 inputs each represent an integer up to $2^5 - 1$ in binary, and the output
137 represents a single integer up to $2^{10} - 1$, such that the output is the multiplication of the two input numbers.
138 This problem is generally attacked as 10 independent binary regression problems, with each bit of the
139 output is separately evolved as a circuit or boolean function.
140   It's easy to show that the number of *n*-bit input *m*-bit output binary relations are $2^{m(2^n)}$, which grows
141 super-exponentially. Multiple output multiplier is the recommended alternative to Multiplexer and Parity
142 problems in White et al. (2012)
143   We transfer input to and output from GPU with bits packed as a single 32bit integer; hence there is a
144 code preamble before first individual to unpack the input bits, and a post-amble after each individual to
145 pack the 10 bits computed by evolved expressions as an integer.
146   The fitness function for 5-bit multiplier computes the number of bits different between the individual's
147 response and correct answer, by computing the pop count of these two XORed.

DEVELOPMENT AND EXPERIMENT SETUP

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 [2] 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*). GPUs are accessed for computation through NVidia CUDA v8 API and libraries, running on top of Windows Server 2012 R2 operating system.

Development Environment

Codes related to grammar generation, parsing, derivation, genetic programming, evolution, fitness computation and GPU access has been implemented in C#, using *managedCuda* [3] for CUDA API bindings and NVRTC interface, along with *CUDAfy.NET* [4] for interfacing to NVCC command line compiler. The grammars for the problems has been prepared such that the languages defined are valid subsets of CUDA C language specialized towards the respective problems.

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:

  (i) *ptx* : Cuda source code to Ptx compilation time per individual

 (ii) *jit* : Ptx to Cubin object compilation time per individual

(iii) *other* : All remaining operations a GP cycle requires (i.e compiled individuals running on GPU, downloading produced results, computing fitness values, evolutionary selection, cross over, mutation, etc.)

The value of *other* is measured 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 $\frac{1ms}{pop.size}$.

Each data point on plots corresponds to the average of one of those measurements for the corresponding (*populationsize*, *measurementtype*, *experiment*) triple. Each average is computed over the measurement values obtained for the first 10 generations of 15 different populations for given size (thus effectively the compile times of 150 generations averaged). The reason for not using 150 generations of a single population directly is that a population gains bias towards to 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.

EXPERIMENT RESULTS

Conventional Compilation as Baseline

NVCC is the default compiler of CUDA platform, it is distributed as a command line application. In addition to compilation of cuda C source codes, it performs tasks such as the separation of source code as

---

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

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

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

(a) Per individual compile time
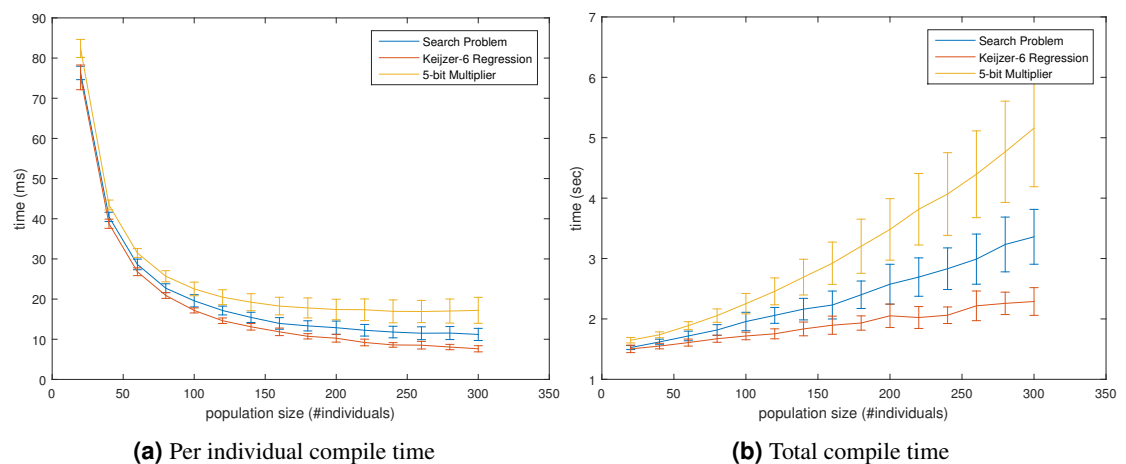
(b) Total compile time

Figure 1. Nvcc compilation times by population size.

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.

Fib1(a) 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 Fig.1(b) 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[5] context, which is the case in genetic programming. Eventhough 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.

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 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 drawback, NVRTC addresses these issues by providing a library interface that eliminates overhead of spawning separate processes, and extra disk I/O.

On figures 2,3 and 4 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 per population), 10.88 ms/individual for 5-bit multiplier problem (at 100 individuals per population[6]), and 6.89 ms/individual for Search Problem (at 280 individuals per population[7]). 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.5).

---

[5]not as in hard real time, but as prolonged, successive and throughput sensitive use

[6]compilation speed at 300 individuals per population is 13.29 ms/individual

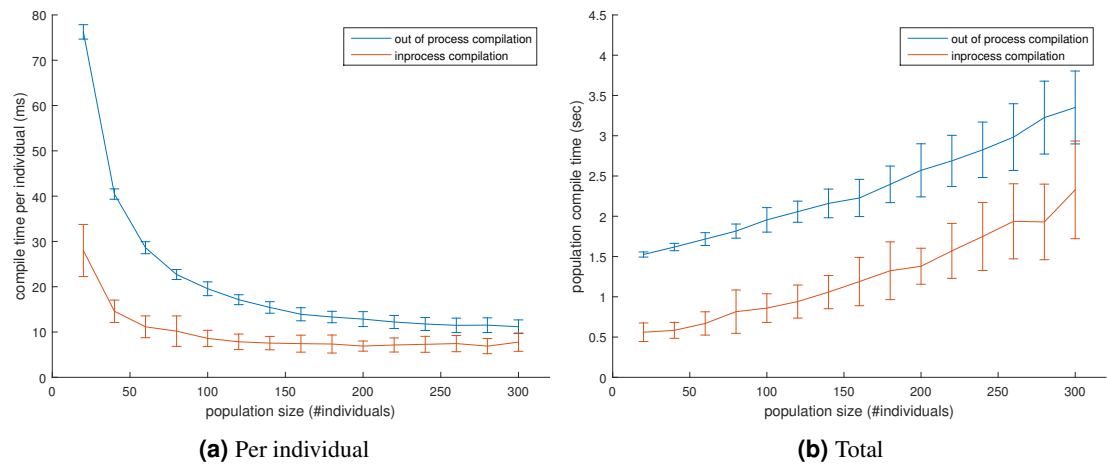[7]compilation speed at 300 individuals per population is 7.76 ms/individual

(a) Per individual

(b) Total

Figure 2. In-process and out of process compilation times by population size, for Search Problem

(a) Per individual

(b) Total
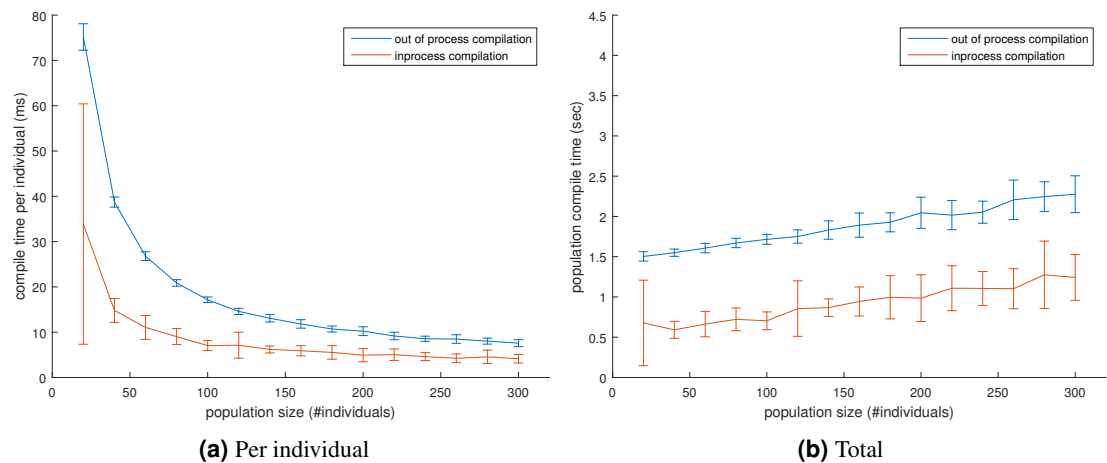
Figure 3. In-process and out of process compilation times by population size, for Keijzer-6 Regression
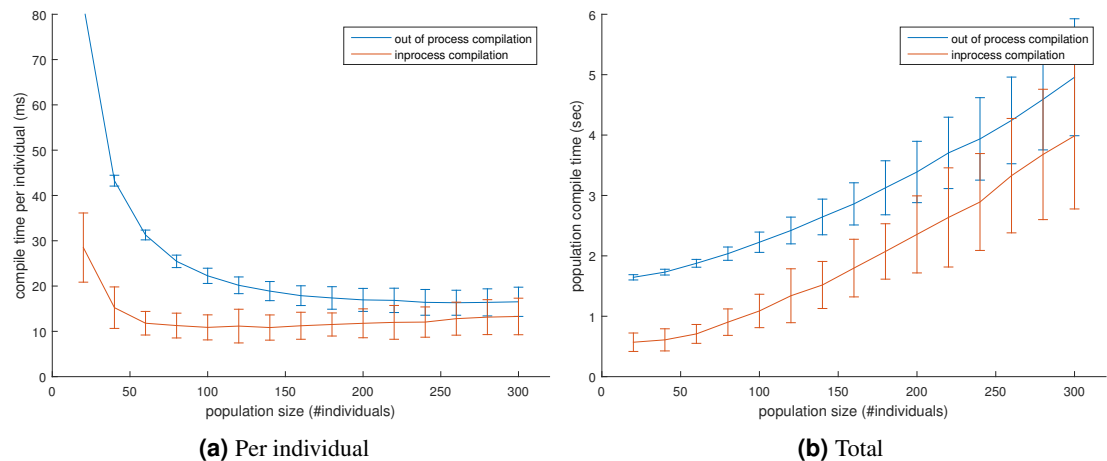

(a) Per individual

(b) Total

Figure 4. In-process and out of process compilation times by population size, for 5-bit Multiplier

Parallelizing In-process Compilation

Infeasibility of parallelization with threads

A first 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
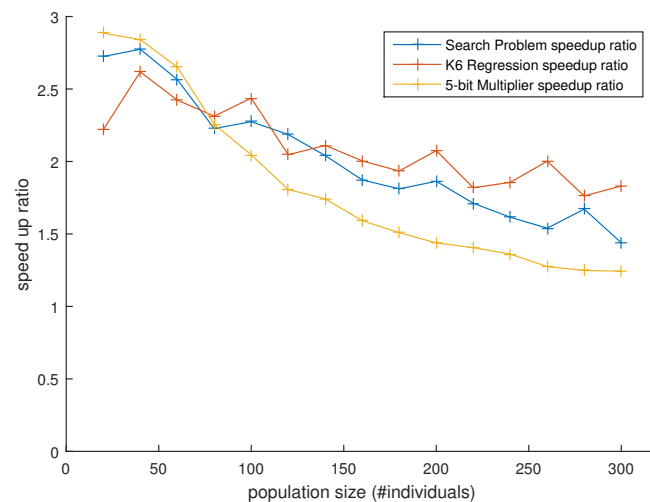
Figure 5. Compile time speedup ratios between conventional and in-process compilation by problem

229 it turns out that NVRTC library is not designed for multi-threaded use; we noticed that when multiple
230 compilation calls are made from different threads at the same time, the execution is automatically
231 serialized.
232 Stack trace in Fig.6 shows *nvrtc64_80.dll* calling OS kernel's *EnterCriticalSection* function to block
233 for exclusive execution of a code block, and gets unblocked by another thread which also runs a block
234 from same library, 853ms later via the release of the related lock. The pattern of green blocks on three
235 threads in addition to main thread in Fig.6 shows that calls are perfectly serialized one after another,
236 despite being called at the same time which is hinted by the red synchronization blocks preceding them.
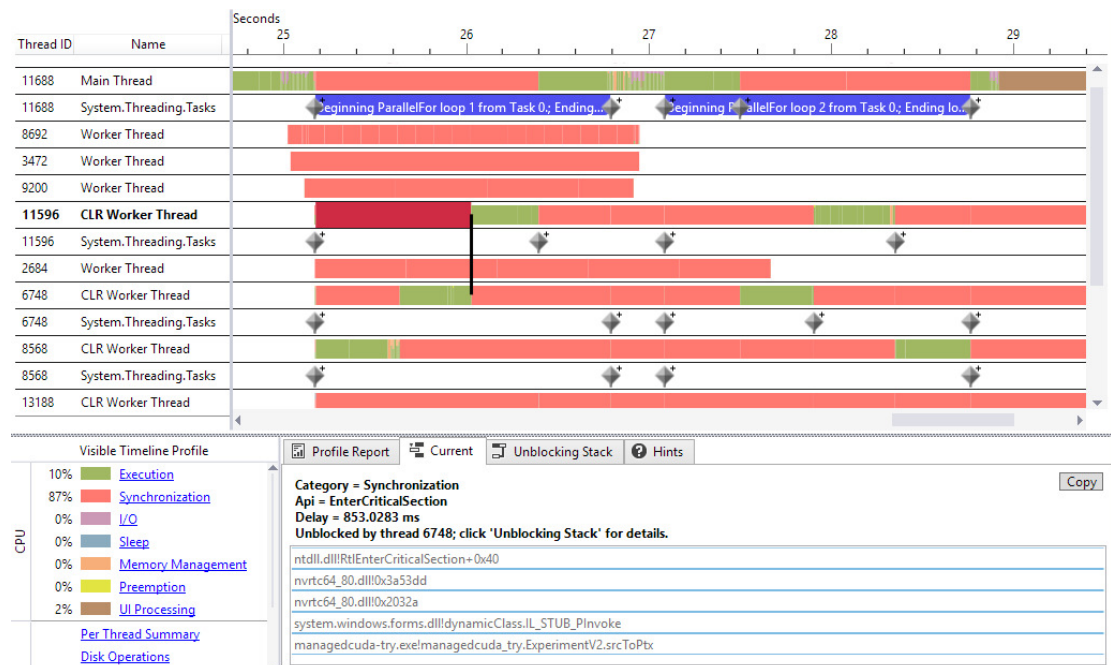

Figure 6. NVRTC library serializes calls from multiple threads

237 Although NVRTC compiles CUDA source to PTX with a single call, the presence of compiler options
238 setup function which affects the following compilation call, and use of critical sections at function entries,
239 show that apparently this is a stateful API. Furthermore, unlike CUDA APIs' design, mentioned state is
240 most likely not stored in thread local storage (TLS), but stored on the private heap of the dynamic loading

7/16

242 moving the kept state to TLS requires source level modifications.

Parallelization with daemon processes

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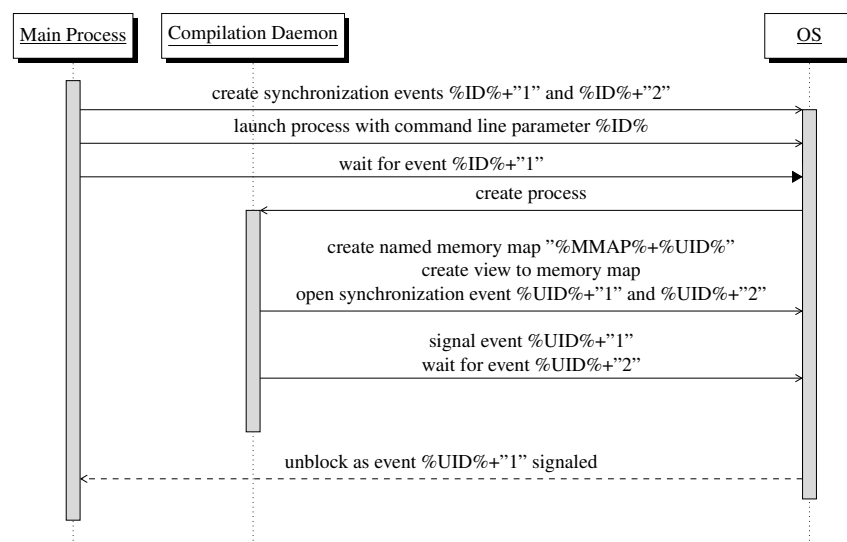254 and (*available* → *processing*) is triggered by the main process.

Figure 7. Sequence Diagram for creation of a compilation daemon process and related interprocess communication primitives

255 The communication between the main process and compilation daemons are handled via shared views
256 to memory maps. Each daemon register a named memory map and create a memory view, onto which
257 main process also creates a view to after the daemon signals state transition from *starting* to *available*.
258 (see Fig.7) CUDA source is passed through this shared memory, and compiled device dependent CUBIN
259 object file is also returned through the same. To signal the state transition (*starting* → *available*) daemon
260 process signals the first event and starts waiting for the second event at the same time. Once a daemon
261 leaves the *starting* state, it never returns back to it.
262 When the main process generate a new population to be compiled it partitions the individuals in a
263 balanced way, such that the difference of number of individuals between any pair of partitions is never
264 more than one. Once the individuals are partitioned, the generated CUDA codes for each partition are
265 passed to the daemon processes. Each daemon waits in the blocked state till main process wakes that
266 specific daemon for a new batch of source to compile by signaling the second named event of that process
267 (see Fig.8). Main process signals all daemons asynchronously to start compiling; then starts waiting for
268 the completion of daemon processes' work. To prevent the UI thread of main process getting blocked too,
269 main process maintains a separate thread for each daemon process it communicates with, therefore while
270 waiting for daemon processes to finish their jobs only those threads of main process are blocked. Main
271 process signaling the second event and daemon process unblocking as a result, corresponds to the state
272 transition (*available* → *processing*).
273 When a daemon process arrives to *processing* state, it reads the CUDA source code from the shared
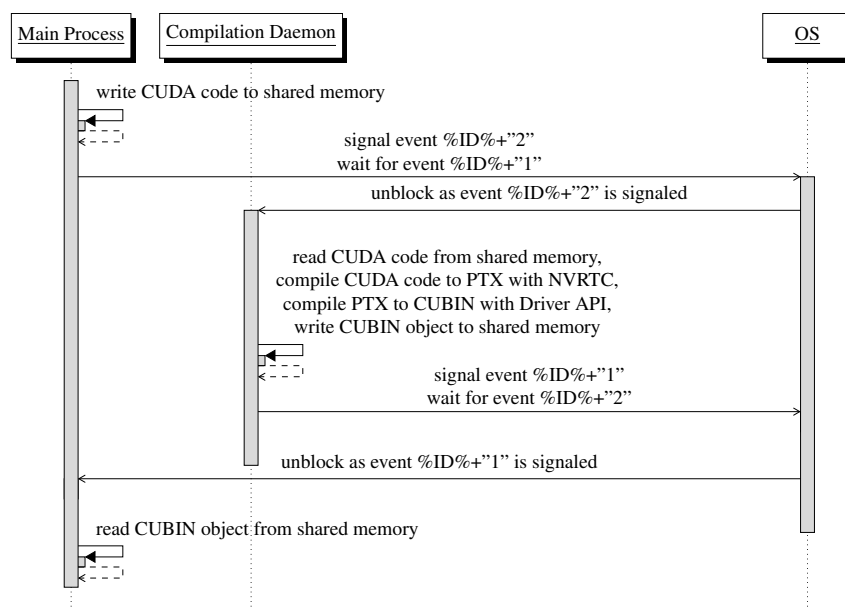
Figure 8. Sequence Diagram for compilation on daemon process and related interprocess communication

274 view of the memory map related to its ID, and compiles the code using NVRTC library.

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

Cost of Parallelization

279 The parallelization approach we propose is virtually overhead free when compared to a hypothetical
280 parallelization scenario using threads. As the daemon processes are already resident and waiting in the
281 memory along with the loaded NVRTC library, the overhead of both parallelization approaches is limited
282 to the time cost of memory moves from/to shared memory and synchronization by named events[8]. The
283 only difference between the two is, in a context switch between threads of same process, processor keeps
284 the Translation Look Aside Buffer (TLB), but in case of a context switch to another process TLB is
285 flushed as processor transitions to a new virtual address space; we conjecture that the impact would be
286 negligible.

287 About the memory cost, all modern operating systems recognize when an executable binary or shared
288 library gets loaded multiple times; OS keeps a single copy of the related memory pages on physical
289 memory, and separately maps those to virtual address spaces of each process using those. This not only
290 saves physical RAM, but also allows better space locality for L2/L3 processor caches. Hence memory
291 consumption by multiple instances of our daemon processes each loading NVRTC library (*nvrtc64_80.dll*
292 is almost 15MB) to their own address space, is almost the same as the consumption of a single instance.

Speedup Achieved with Parallel Compilation

294 At the end of each batch of experiments main application dumps the collected raw measurements to a
295 file. We imported this data to Matlab filtered by experiment and measurement types, and aggregated
296 the experiment values for each population size to produce the Tables 1,2,3, and to create the Figures
297 9,10,11,12,13,14.

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

---

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

Table 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

Table 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 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

302 Parallel compilation brought the per individual compilation time to 2.17 ms/individual for 5-bit
303 Multiplier, to 2.20 ms/individual for Keijzer-6 regression and to 2.13 milliseconds for the Search Problem;
304 these are almost an order of magnitude faster than previous published results. Also we measured
305 a compilation speedup of $\times 3.45$ for regression problem, $\times 5.26$ for search problem, and $\times 7.60$ for
306 multiplication problem, when compared to the latest Nvcc V8 compiler, without requiring any code
307 modification, and without any runtime performance penalty.

308 Notice that our experiment platform consisted of dual Xeon E5-2670 processors running at 2.6Ghz;
309 for compute bound tasks increase on processor frequency almost directly translates to performance
310 improvement at an equal rate[9]. Therefore we can conjecture that to be able to compile a population of
311 300 individuals at sub-millisecond durations, the required processor frequency is around $2.6 \times 2.13 =$
312 $5.54 Ghz$[10] which is currently available.

CONCLUSION

314 In this paper we present a new method to accelerate the compilation of genetic programming individuals,
315 in order to keep the compiled approach as a viable option for genetic programming on gpu.

316 By using an in-process GPU compiler, we replaced disk file based data transfer to/from the compiler
317 with memory accesses, also we mitigated the overhead of repeated launches and tear downs of the
318 command line compiler. Also we investigated ways to parallelize this method of compilation, and
319 identified that in-process compilation function automatically serializes concurrent calls from different
320 threads. We implemented a daemon process that can have multiple running instances and service another
321 application requesting CUDA code compilation. Daemon processes use the same in-line compilation
322 method and communicate through operating system's Inter Process Communication primitives.

323 We measured compilation times just above 2.1 ms/individual for all three benchmark problems; and
324 observed compilation speedups ranging from $\times 3.45$ to $\times 7.60$ based on problem, when compared to
325 repeated command line compilation with latest Nvcc v8 compiler.

326 All data and source code of software presented in this paper is available at https://github.com/hayral/Parallel-
327 and-in-process-compilation-of-individuals-for-genetic-programming-on-GPU

ACKNOWLEDGMENTS

APPENDIX

Search Problem
Grammar Listing

```
334 <expr>         ::= <expr2> <bi-op> <expr2> | <expr2>
335 <expr2>        ::= <int> | <var-read> | <var-indexed>
336
337 <var-read>     ::= tmp | i | OUTPUT | SEARCH
338 <var-indexed>  ::= INPUT[<var-read> % LENINPUT]
339 <var-write>    ::= tmp | OUTPUT
340
341 <bi-op>        ::= + | -
342
343 <int>          ::= 1 | 2 | (-1)
344
345 <statement>    ::= <assignment> | <if> | <loop>
346 <statement2>   ::= <assignment> | <if2>
347 <statement3>   ::= <assignment>
348
349 <loop>         ::= for(i=0;i #lesser# LENINPUT;i++){<c-block2>}
350
351 <if>           ::= if (<cond-expr>) {<c-block2>}
352 <if2>          ::= if (<cond-expr>) {<c-block3>}
```

---

[9]assuming all other things being equal

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

11/16

```
353
354   <cond−expr>   ::= <expr> <comp−op> <expr>
355   <comp−op>     ::= #lesser# | #greater# | == | !=
356   <assignment>  ::= <var−write> = <expr>;
357
358   <c−block>     ::= <statements>
359   <c−block2>    ::= <statements2>
360   <c−block3>    ::= <statements3>
361
362   <statements>  ::= <statement>
363                   | <statement><statement>
364                   | <statement><statement><statement>
365
366   <statements2> ::= <statement2>
367                   | <statement2><statement2>
368                   | <statement2><statement2><statement2>
369
370   <statements3> ::= <statement3>
371                   | <statement3><statement3>
372                   | <statement3><statement3><statement3>
```

Listing 1. Grammar for Search Problem

373 ***Code Preamble for Whole Population***

```
374   __constant__ int _INPUT[NUMTESTCASE][MAX_TESTCASE_LEN];
375   __constant__ int _LENINPUT[NUMTESTCASE];
376   __constant__ int _SEARCH[NUMTESTCASE];
377   __constant__ int CORRECTANSWER[NUMTESTCASE];
378
379
380   __global__ void createdFunc(int *_OUTPUT)
381   {
382       int *INPUT = _INPUT[threadIdx.x];
383       int LENINPUT = _LENINPUT[threadIdx.x];
384       int SEARCH = _SEARCH[threadIdx.x];
385
386       int i;
387       int tmp;
388       int OUTPUT;
```

Listing 2. Code preamble for whole population on Search Problem

389 **Keijzer-6 Regression Problem**
390 ***Grammar Listing***

```
391   <e>   ::= <e2> + <e2> | <e2> − <e2> | <e2> * <e2> | <e2> / <e2>
392         | sqrtf(fabsf(<e2>)) | sinf(<e2>) | tanhf(<e2>)
393         | expf(<e2>) | logf(fabsf(<e2>)+1)
394         | x | x | x | x
395         | <c><c>.<c><c> | <c><c>.<c><c> | <c><c>.<c><c> | <c><c>.<c><c>
396
397   <e2> ::= <e3> + <e3> | <e3> − <e3> | <e3> * <e3> | <e3> / <e3>
398         | sqrtf(fabsf(<e3>)) | sinf(<e3>) | tanhf(<e3>)
399         | expf(<e3>) | logf(fabsf(<e3>)+1)
400         | x | x | x | x | x | x
401         | <c><c>.<c><c> | <c><c>.<c><c> | <c><c>.<c><c>
402         | <c><c>.<c><c> | <c><c>.<c><c> | <c><c>.<c><c>
403
404   <e3> ::= <e3> + <e3> | <e3> − <e3> | <e3> * <e3> | <e3> / <e3>
405         | sqrtf(fabsf(<e3>)) | sinf(<e3>) | tanhf(<e3>)
406         | expf(<e3>) | logf(fabsf(<e3>)+1)
407         | x | x | x | x | x | x | x | x
408         | <c><c>.<c><c> | <c><c>.<c><c> | <c><c>.<c><c>
409         | <c><c>.<c><c> | <c><c>.<c><c> | <c><c>.<c><c>
410         | <c><c>.<c><c> | <c><c>.<c><c>
411
412   <c>   ::= 0|1|2|3|4|5|6|7|8|9
```

Listing 3. Grammar for Keijzer-6 Regression

5-bit Multiplier Problem

Grammar Listing

```
<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 4. Grammar for 5-bit Multiplier Problem

Code Preamble for each Individual

```
__global__ void createdFunc0(int *OUTPUT)
{
int tid = blockIdx.x *blockDim.x + threadIdx.x;;

int a0 = tid & 0x1;
int a1 = (tid & 0x2) >> 1;
int a2 = (tid & 0x4) >> 2;
int a3 = (tid & 0x8) >> 3;
int a4 = (tid & 0x10) >> 4;
int b0 = (tid & 0x20) >> 5;
int b1 = (tid & 0x40) >> 6;
int b2 = (tid & 0x80) >> 7;
int b3 = (tid & 0x100) >> 8;
int b4 = (tid & 0x200) >> 9;

int o0,o1,o2,o3,o4,o5,o6,o7,o8,o9;
```

Listing 5. Code preamble for 5-bit Multiplier Problem
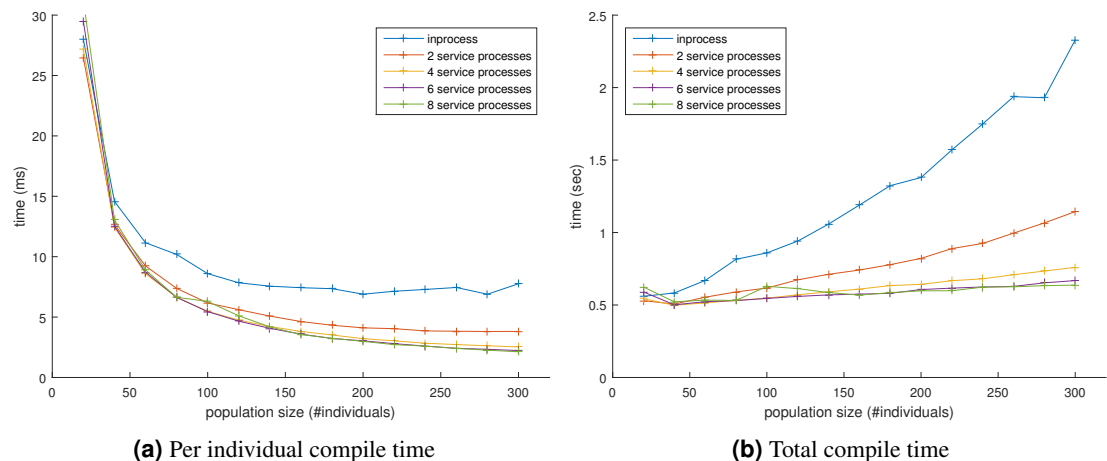
Compilation Time and Speedup Ratio Plots

(a) Per individual compile time

(b) Total compile time

Figure 9. Nvcc compilation times for Search Problem by number of servicing resident processes

(a) Speedup ratio against conventional compilation

(b) Speedup ratio against in-process compilation
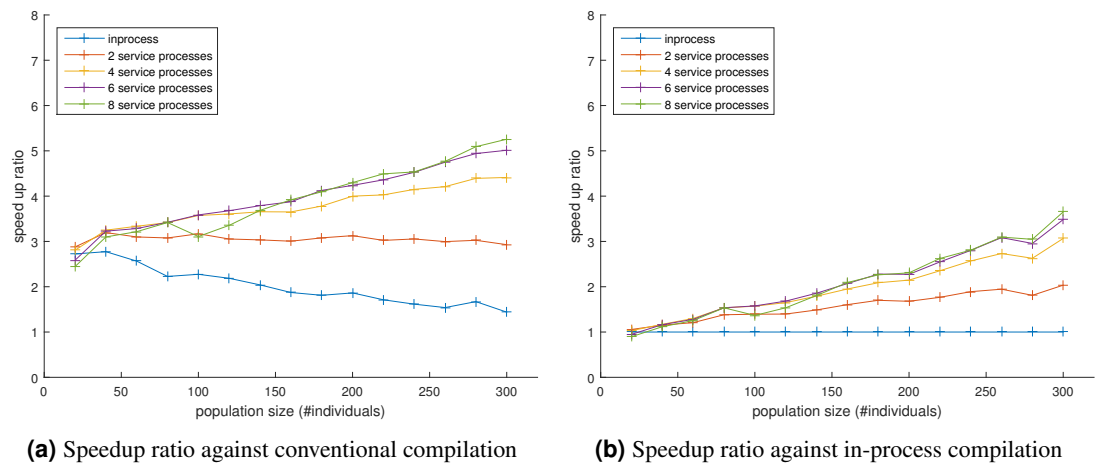
Figure 10. Parallelization speedup on Search problem

(a) Per individual compile time

(b) Total compile time
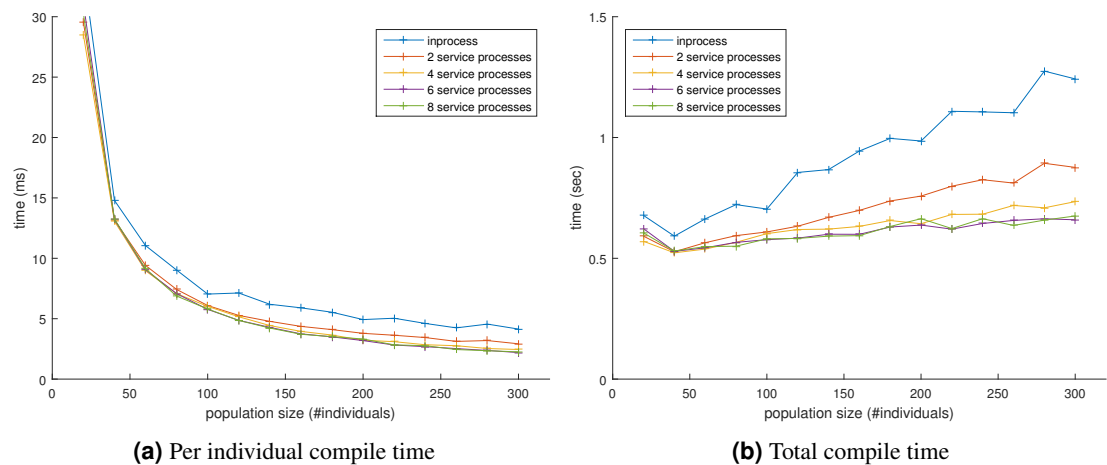
Figure 11. Nvcc compilation times for Keijzer-6 regression by number of servicing resident processes

(a) Speedup ratio against conventional compilation

(b) Speedup ratio against in-process compilation
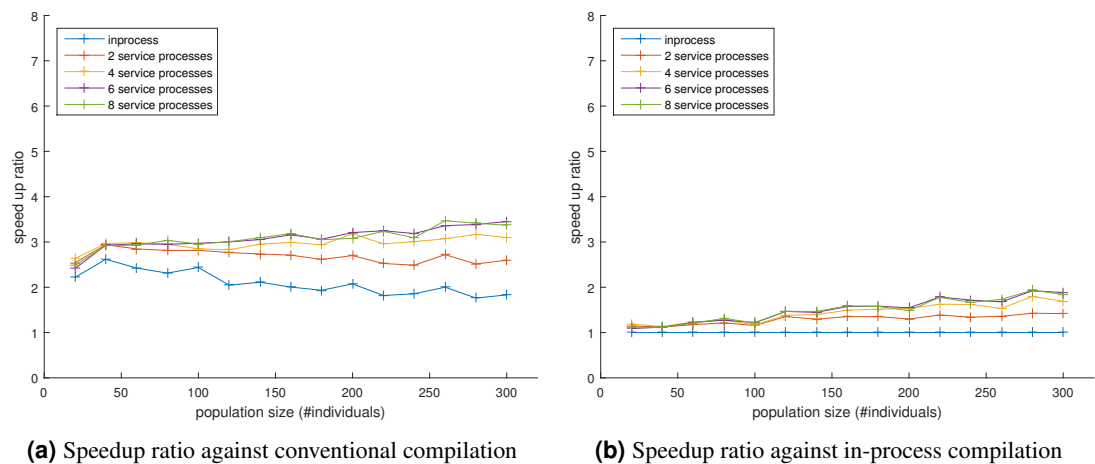
Figure 12. Parallelization speedup on Keijzer-6 regression

(a) Per individual compile time

(b) Total compile time
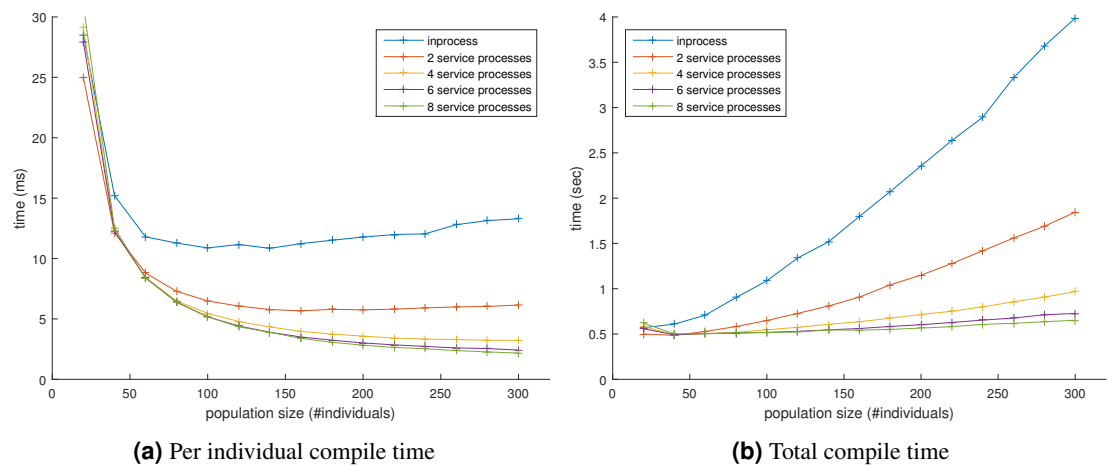
Figure 13. Nvcc compilation times for 5-bit Multiplier by number of servicing resident processes

(a) Speedup ratio against conventional compilation

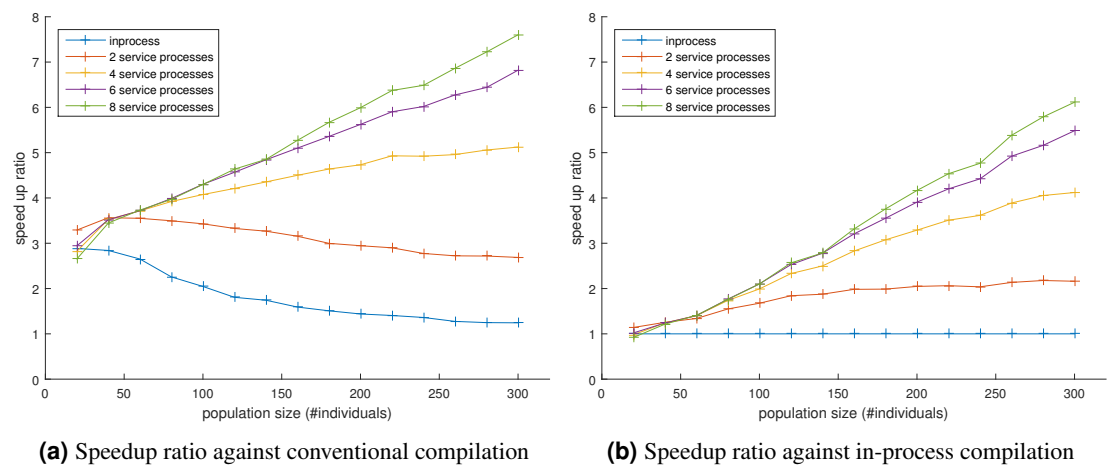**(b)** Speedup ratio against in-process compilation

Figure 14. Parallelization speedup on 5-Bit multiplier

REFERENCES

Ayral, H. and Albayrak, S. (in press). Effects of Population, Generation and Test Case Count on Grammatical Genetic Programming for Integer Lists (in press).

Chitty, D. M. (2007). A data parallel approach to genetic programming using programmable graphics hardware. *Proc. of the Conference on Genetic and Evolutionary Computation ({GECCO})*, 2:1566–1573.

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

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 2016)*, pages 5060–5067.

Harding, S. and Banzhaf, W. (2007). Fast Genetic Programming on GPUs. In *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445, pages 90–101. Springer.

Harding, S. S. L. S. and Banzhaf, W. (2009). Distributed genetic programming on GPUs using CUDA. In *Workshop on Parallel Architectures and Bioinspired Algorithms*, pages 1–10.

Helmuth, T. and Spector, L. (2015). General Program Synthesis Benchmark Suite. In *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference - GECCO '15*, pages 1039–1046, New York, New York, USA. ACM Press.

Keijzer, M. (2003). Improving Symbolic Regression with Interval Arithmetic and Linear Scaling. *Genetic Programming Proceedings of EuroGP2003*, 2610:70–82.

460   Langdon, W. and Harman, M. (2010a). Evolving gzip matches Kernel from an nVidia CUDA Template.
461   Technical Report February.
462   Langdon, W. B. (2011). Graphics processing units and genetic programming: an overview. *Soft Computing*
463   *- A Fusion of Foundations, Methodologies and Applications*, 15(8):1657–1669.
464   Langdon, W. B. and Harman, M. (2010b). Evolving a CUDA kernel from an nVidia template. *IEEE*
465   *Congress on Evolutionary Computation*, pages 1–8.
466   Lewis, T. E. and Magoulas, G. D. (2011). Identifying similarities in TMBL programs with alignment to
467   quicken their compilation for GPUs. *Proceedings of the 13th annual conference companion on Genetic*
468   *and evolutionary computation - GECCO '11*, page 447.
469   Nicolau, M. and Fenton, M. (2016). Managing Repetition in Grammar-Based Genetic Programming.
470   *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference - GECCO '16*, pages
471   765–772.
472   O'Neill, M., Nicolau, M., and Agapitos, A. (2014). Experiments in program synthesis with grammatical
473   evolution: A focus on Integer Sorting. In *Evolutionary Computation (CEC), 2014 IEEE Congress on*,
474   pages 1504–1511.
475   Pospichal, P., Murphy, E., O'Neill, M., Schwarz, J., and Jaros, J. (2011). Acceleration of Grammatical
476   Evolution Using Graphics Processing Units. *Proceedings of the 13th annual conference companion on*
477   *Genetic and evolutionary computation - GECCO '11*, pages 431–438.
478   White, D. R., McDermott, J., Castelli, M., Manzoni, L., Goldman, B. W., Kronberger, G., Jaśkowski, W.,
479   O'Reilly, U.-M., and Luke, S. (2012). Better GP benchmarks: community survey results and proposals.
480   *Genetic Programming and Evolvable Machines*, 14(1):3–29.