Towards Highly Optimized Cartesian Genetic Programming: From Sequential via SIMD and Thread to Massive Parallel Implementation

Radek Hrbacek
Brno University of Technology, Czech republic
Faculty of Information Technology
ihrbacek@fit.vutbr.cz

Lukas Sekanina
Brno University of Technology, Czech republic
Faculty of Information Technology
sekanina@fit.vutbr.cz

ABSTRACT

Most implementations of Cartesian genetic programming (CGP) which can be found in the literature are sequential. However, solving complex design problems by means of genetic programming requires parallel implementations of search methods and fitness functions. This paper deals with the design of highly optimized implementations of CGP and their detailed evaluation in the task of evolutionary circuit design. Several sequential implementations of CGP have been analyzed and the effect of various additional optimizations has been investigated. Furthermore, the parallelism at the instruction, data, thread and process level has been applied in order to take advantage of modern processor architectures and computer clusters. Combinational adders and multipliers have been chosen to give a performance comparison with state of the art methods.

Categories and Subject Descriptors

B.6.0 [**Hardware**]: Logic Design—*General*; I.2.8 [**Computing methodologies**]: Artificial intelligence—*Problem Solving, Control Methods, and Search*

Keywords

Cartesian Genetic Programming, Parallel Computing, SIMD, AVX, Cluster, Combinational Circuit Design

1. INTRODUCTION

The evolutionary design conducted by means of genetic programming (GP) is a very computationally demanding design method. In order to reduce the design time, various accelerators of genetic programming have been proposed. The accelerators are typically developed to speed up the main components of the method – the search algorithm and the fitness evaluation procedure. While the former case is usually investigated in the field of parallel and distributed evolutionary algorithms, the latter one typically involves accelerating an application specific simulator in which every candidate phenotype has to be evaluated. The most famous parallel approaches to GP are represented by Koza's Beowulf-style parallel cluster [10] and recently, an approach developed for the cloud environment [17].

As CGP has been accelerated on GPUs [6] and FPGAs [20], in this contribution, the parallelism at the instruction, data, thread and process level has been applied in order to take advantage of modern processor architectures and computer clusters. The goal of this paper is to provide a set of parallel CGP implementations that can be used on these widely accessible parallel computers. The proposed implementations will be compared and evaluated in the task of adder and multiplier evolutionary design. The reason for choosing these two problems is that the literature includes several case studies that can be used for comparative purposes.

Another important assumption of this study is that CGP starts with a randomly generated initial population and the objective is to find a fully functional solution, i.e. an n_i-input/n_o-output circuit which provides $n_\mathrm{o} \cdot 2^{n_\mathrm{i}}$ correct output bits when all possible input combinations are evaluated. In other words, the goal is not in minimizing the number of gates, delay or other criteria. Note that for the evolutionary circuit optimization, in which CGP starts with a fully functional solution and the goal is to minimize the number of gates, efficient and fast fitness calculation methods based on formal functional equivalence checking algorithms have already been proposed [21]. While such methods are capable of optimizing circuits having hundreds of inputs and thousands of gates, only relatively small circuits (with ten to fifteen inputs and less than 100 gates) have been evolved so far in the proposed scenario. Finally, this work does not take into account advanced methods such as divide and conquer [18, 16] or self-modifying CGP [7] which allow for reducing the problem complexity and consequently applying the standard CGP on sub-problems. Therefore, all techniques reported in this paper operate on the whole circuit.

Parallel CGP implementations are usually focused on an efficient phenotype evaluation, which is the most time critical operation of CGP due to the fact that the circuit evaluation time grows exponentially with the number of circuit inputs. In order to accelerate a candidate circuit evaluation, one can apply a parallel evaluation of multiple training vectors by means of bit-level instructions [14], circuit precompilation techniques or streaming SIMD extensions (SSE) of modern processors [22].

On the other hand, the search algorithm used in the standard CGP is a simple (1+λ) evolution strategy. A natural approach to accelerating the search is evaluating λ offspring on λ processors in parallel. A few attempts were made to introduce more advanced operators into this search method, but only a small improvement was reported in [3]. However, a noticeable improvement can be obtained when the standard CGP is replaced by parallel coevolutionary CGP [9].

The rest of the paper is organized as follows. Section 2 introduces CGP and its usage as combinational circuit design method. The implementation of several sequential solutions is discussed in Section 3. Section 4 deals with the CGP parallelisation. Section 5 is dedicated to experiments and the achieved results. Final conclusions can be found in Section 6.

2. CARTESIAN GENETIC PROGRAMMING

Cartesian genetic programming has been introduced by Miller [12] as a branch of genetic programming. Unlike GP which uses tree representation, an individual in CGP is represented by a directed acyclic graph which enables the candidate solution to have multiple outputs and automatically reuse intermediate results. This makes CGP very suitable for design of various kinds of digital circuits, such as arithmetic and logic circuits, digital filters, etc. [13].

CGP uses a cartesian grid of $n_\mathrm{r} \times n_\mathrm{c}$ programmable elements (nodes) interconnected by a feed-forward network (Figure 1). Each node's input (usually each node has a fixed number of inputs $n_\mathrm{ni} = 2$) can be connected either to one of n_i primary inputs or to a node output in the preceding l columns. By setting the l-back parameter and the grid size, one can control the area and delay of the circuit. Each node can be programmed to perform one of n_ni-input functions defined in the set Γ (let $n_\mathrm{f} = |\Gamma|$). The n_o primary circuit outputs are connected either to the primary inputs or nodes. The output connectivity can be optionally restricted by the o-back parameter.

Since all the CGP parameters are fixed, each chromosome is encoded using a fixed-sized array of $n_\mathrm{r} \cdot n_\mathrm{c} \cdot (n_\mathrm{ni} + 1) + n_\mathrm{o}$ integers. Each primary input is assigned a number from $\{0, ..., n_\mathrm{i} - 1\}$ and the nodes are assigned numbers from $\{n_\mathrm{i}, ..., n_\mathrm{i} + n_\mathrm{r} \cdot n_\mathrm{c} - 1\}$. The genotype is of fixed length, whereas the phenotype is of variable length depending on the number of inactive nodes, i.e. nodes whose output is not used by any other node or primary output. Hence, the genotype-phenotype mapping is not injective. The existence of genotypes with the same fitness is usually referred to as neutrality. The role of neutrality has been intensively studied [24] and it was shown that for certain problems the neutrality significantly reduces the computational effort and helps to find more innovative solutions [11].

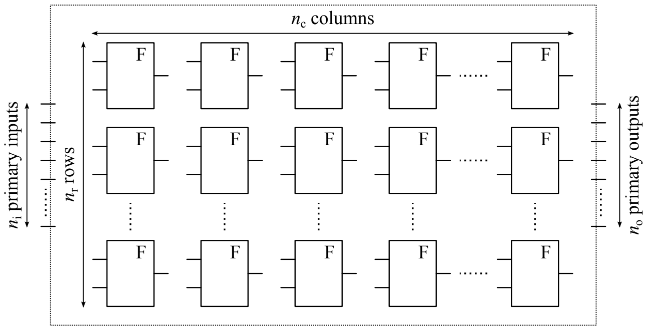

Figure 1: Cartesian genetic programming scheme.

In CGP, a simple mutation based $(1 + \lambda)$ evolutionary strategy is used as a search mechanism. The population size $1 + \lambda$ is usually very small, typically, λ is between 1 and 15. The initial population is constructed either randomly (evolutionary design) or by mapping of a known solution to the CGP chromosome (evolutionary optimization) [21]. In each generation, a randomly selected individual with the best fitness value is passed to the next generation unmodified and its λ offspring individuals are created by means of point mutation operator which modifies m randomly selected genes of the chromosome. The mutation rate m is usually set to modify up to 5 % of the genes. Despite numerous attempts, no useful crossover operator has been introduced. For some problem classes (e.g. symbolic regression problem), special crossover operators have been investigated [3], however, in the case of digital circuits design, none of them has been confirmed as useful.

In the case of combinational circuit evolution, the fitness function corresponds to the quality of the candidate circuit measured as the number of correct output bits compared to a specified truth table. In order to obtain a fully working circuit, all combinations of input values have to be evaluated. For a circuit with n_i inputs and n_o outputs, 2^{n_i} test vectors need to be fetched to the primary inputs and $n_\mathrm{o} \cdot 2^{n_\mathrm{i}}$ output bits have to be verified so as to compute the fitness value.

3. SEQUENTIAL IMPLEMENTATION

Every CGP implementation must process all the 2^{n_i} test vectors on the whole phenotype for the entire population of individuals and compare all the n_o outputs to the desired ones. This requirement directly implies the presence of 3 independent nested loops – the test vector loop, the loop over all nodes and the population loop. The order of these 3 loops is crucial for the performance and the optimal choice varies among different implementations. In a very naive implementation, one can process each test vector separately on all nodes no matter if they are active or not. However, in order to take advantage of modern superscalar out-of-order processors, the parallelism at various levels has to be employed and special attention to memory access policy has to be paid.

The most fundamental optimization we can apply is the bit-level parallelism. Instead of separate test vector processing, up to 64 test vectors can be processed in parallel on 64-bit processors thanks to bitwise operations. Furthermore, by introducing the data-level parallelism using SIMD instructions, 128 or even 256 test vectors can fit into the SSE or AVX registers respectively.

One of the most commonly used optimization in CGP is the detection of inactive nodes. Before processing each individual, the genotype is traversed in the reversed order and the nodes whose output is never used are marked as inactive. While processing, all inactive nodes are skipped and thus only the phenotype is treated.

3.1 Interpreted implementation

The *interpreted* implementation is very simple, yet for smaller circuits very efficient. The principle is shown in Algorithm 1. At the beginning, an initial population is created randomly just like in any other implementation. Then, in

```
randomly create and evaluate initial population;
while termination condition is false do
    for i in 1 to P do
        if i = best_ind then
            continue;
        end
        copy ind[best_ind] chromosome to ind[i];
        mutate ind[i];
        analyze ind[i];
        foreach node do
            foreach test vector do
                compute node output value;
            end
        end
        foreach primary output do
            ind[i].fit += number of wrong bits;
        end
        if ind[i].fit > ind[best_ind].fit then
            best_ind := i;
        end
    end
end
```

Algorithm 1: Interpreted implementation

every generation, the evaluation of the population is going on as follows: For each individual not being the best individual from the previous generation, the chromosome of the best individual is copied and mutated. After that, the chromosome is analyzed in order to find the inactive nodes. For each active node, all test vectors are processed according to the node function. The test vector loop is put inside the node loop because of the overhead of the `switch` statement the node function is based on. Besides the overhead, by putting the test vector loop inside, the compiler is able to optimize this loop by unrolling. After computing each node's output value, the primary outputs are checked against desired values and the number of wrong bits is accumulated. This can be done very efficiently just by `XOR`ing the actual output value and the expected value and counting the number of ones. Since SSE 4.2, a special instruction `POPCNT` exists which allows to count the number of ones with the latency of 3 clock cycles (on the Intel Sandy Bridge microarchitecture) [4].

Since all of the intermediate results for all test vectors have to be kept in memory during the evaluation, the memory usage of the interpreted implementation is not optimal (up to $n_r \cdot n_c \cdot 2^{n_i}/8$ bytes), but it is still efficient for small circuits, until the required memory size exceeds the cache size.

3.2 Native implementation

By introducing the *native* implementation, both the memory requirements and the `switch` statement overhead can be significantly reduced. The principle can be seen from Algorithm 2. Just like in the interpreted implementation, each individual except the previous best one is copied, mutated and analyzed. The difference lies in the evaluation process

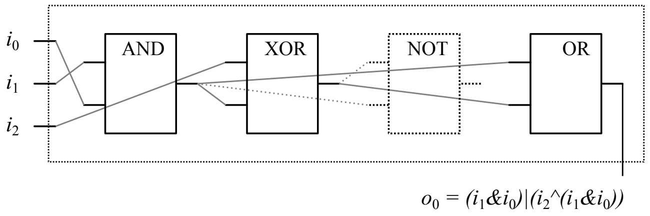

Figure 2: CGP individual example.

```
randomly create and evaluate initial population;
while termination condition is false do
    for i in 1 to P do
        if i = best_ind then
            continue;
        end
        copy ind[best_ind] chromosome to ind[i];
        mutate ind[i];
        analyze ind[i];
        compile ind[i];
    end
    foreach test vector do
        for i in 1 to P do
            if i = best_ind then
                continue;
            end
            ind[i].fit += run compiled code;
        end
    end
    for i in 1 to P do
        if ind[i].fit > ind[best_ind].fit then
            best_ind := i;
        end
    end
end
```

Algorithm 2: Native implementation.

– instead of traversing the chromosome and computing the node outputs directly, the chromosome is compiled at first. The compiled program is then executed on each test vector for each individual in the population. This technique was first introduced in [22] and the implementation presented in this paper further improves this principle by introducing subroutine parameters, native fitness calculation and by utilizing AVX instructions. Moreover, the native implementation uses slightly less memory since there is no need to keep the intermediate results for all test vectors.

Figure 2 depicts a very simple circuit with $n_i = 3$ primary inputs, $n_o = 1$ primary output and $n_c = 4$ nodes. Listing 1 shows the compiled chromosome from Figure 2 in 64-bit version. The compiled subroutine has 4 parameters passed on in the 64-bit registers `RDI`, `RSI`, `RDX` and `RCX` [15] – the pointers to the primary inputs filled by corresponding test vectors, node outputs, desired primary outputs and spe-

Listing 1: Native 64b implementation example.

```
push    %rbx                ; store RBX

mov     0x08(%rsi),%rbx     ; node n0
and     0x00(%rsi),%rbx     ; AND
mov     %rbx,0x18(%rdi)     ; n0 := i1 AND i0

mov     0x10(%rsi),%rbx     ; node n1
xor     0x18(%rdi),%rbx     ; XOR
mov     %rbx,0x20(%rdi)     ; n1 := i2 XOR n0

mov     0x18(%rdi),%rbx     ; node n3
or      0x28(%rdi),%rbx     ; OR
mov     %rbx,0x30(%rdi)     ; n3 := n0 OR n2

xor     %rax,%rax           ; RAX := 0

mov     0x30(%rsi),%rbx     ; output O0
xor     0x0(%rdx),%rbx
and     0x0(%rcx),%rbx      ; mask m0
popcnt  %rbx,%bdx           ; error count
add     %rbx,%rax           ; accumulate

pop     %rbx                ; restore RBX
retq                        ; return RAX
```

cial masks.[1] For each node, the first input value is loaded from the memory to the RBX register and the desired operation is performed (optionally, the second input value is loaded).[2] The node output is then stored back to the memory. After processing all active nodes, the number of wrong output bits is accumulated in the RAX register. For each primary output, the corresponding node output is loaded from the memory and XORed with the desired value. After applying the mask, the number of incorrect bits is computed using the POPCNT instruction and accumulated in the RAX register. The register RAX is used for integer return values [15], thus the subroutine returns the number of wrong bits for a given test vector.

The same example compiled in the AVX version can be seen in Listing 2. Here, the calling convention is the same and the register RAX has the same purpose as in the 64-bit version. The intermediate results are computed in the YMM0 register. The register YMM1 contains just ones and serves for computing the NOT operation using XOR, since there is not an AVX instruction for this purpose. Compared to the 64-bit version, the error computation is more complicated as there is only a 32-bit and 64-bit POPCNT instruction available.

[1]After XORing the actual and desired output value, the result is ANDed with this mask, which enables to specify which output bits are not considered (we don't care about their values).

[2]There is no need to avoid the output dependency thanks to hardware register renaming.

Listing 2: Native AVX implementation example.

```
push      %rbx                        ; store RBX

mov       0xXXXXXXXX,%rax             ; RAX := &avx_ones
vmovdqa   0x0(%rax),%ymm1             ; YMM1 := 111..111

vmovdqa   0x20(%rsi),%ymm0            ; node n0
vandps    0x0(%rdi),%ymm0,%ymm0       ; AND
vmovdqa   %ymm0,0x60(%rdi)            ; n0 := i1 AND i0

vmovdqa   0x40(%rsi),%ymm0            ; node n1
vxorps    0x60(%rdi),%ymm0,%ymm0      ; XOR
vmovdqa   %ymm0,0x80(%rdi)            ; n1 := i2 XOR n0

vmovdqa   0x60(%rdi),%ymm0            ; node n3
vorps     0xa0(%rdi),%ymm0            ; OR
vmovdqa   %ymm0,0xc0(%rdi)            ; n3 := n0 OR n2

xor       %rax,%rax                   ; RAX := 0

mov       0xc0(%rsi),%rbx             ; output o0[0]
xor       0x0(%rdx),%rbx
and       0x0(%rcx),%rbx              ; mask m0[0]
popcnt    %rbx,%rbx                   ; error count
add       %rbx,%rax
mov       0xc8(%rsi),%rbx             ; output o0[1]
xor       0x8(%rdx),%rbx
and       0x8(%rcx),%rbx              ; mask m0[1]
popcnt    %rbx,%rbx                   ; error count
add       %rbx,%rax
mov       0xd0(%rsi),%rbx             ; output o0[2]
xor       0x10(%rdx),%rbx
and       0x10(%rcx),%rbx             ; mask m0[2]
popcnt    %rbx,%rbx                   ; error count
add       %rbx,%rax
mov       0xd8(%rsi),%rbx             ; output o0[3]
xor       0x18(%rdx),%rbx
and       0x18(%rcx),%rbx             ; mask m0[3]
popcnt    %rbx,%rbx                   ; error count
add       %rbx,%rax

pop       %rbx                        ; restore RBX
retq                                  ; return RAX
```

The native implementation enables to introduce more optimizations than the interpreted implementation. The fitness computation can be stopped after exceeding the number of wrong bits matching the best individual. Both native and interpreted implementations can detect neutral mutations and skip recomputing fitness values for individuals affected only by neutral mutations [5].

The efficiency of the native implementation lies in exploiting the instruction-level parallelism offered by modern superscalar out-of-order processors by reducing branch mispredictions and cache misses and increasing the arithmetic intesity.

4. PARALLEL IMPLEMENTATION

Until the beginning of the 21st century, the aim of the processor architects was to increase the single threaded performance by means of extracting more instruction-level parallelism (ILP) and utilizing superscalar out-of-order execution, sophisticated branch predictors, multi-level cache hierarchy, etc. However, growing power consumption and limited ILP extractable from common sequential code together with increasing transistor density led to the introduction of multiprocessors [8]. Since then, special attention has to be paid to parallel computing in order to make use of modern processor architectures.

4.1 Thread parallelism

The purpose of the thread-level parallelism (TLP) in CGP is to speed up the whole evolutionary process – both the fitness calculation and the genetic operators. Both interpreted and native parallel implementations are discussed in this subsection; the OpenMP library has been used for managing the threads.

Algorithm 3 shows the scheme of the interpreted parallel implementation, very similar to the corresponding sequential variant. The outer population loop has to be scheduled dynamically, because if it were scheduled statically, the thread responsible for the previous best individual would have less work than the others resulting in poor load balancing.

The parallelization of the native implementation is some-

```
randomly create and evaluate initial population;
while termination condition is false do
    #pragma omp for schedule(dynamic)
    for i in 1 to pop_size do
        if i = best_ind then
            continue;
        end
        copy ind[best_ind] chromosome to ind[i];
        mutate ind[i];
        analyze ind[i];
        foreach node do
            foreach test vector do
                compute node output value;
            end
        end
        foreach primary output do
            ind[i].fit += number of wrong bits;
        end
        #pragma omp critical
        if ind[i].fit > ind[best_ind].fit then
            best_ind := i;
        end
    end
end
```

Algorithm 3: Parallel interpreted implementation.

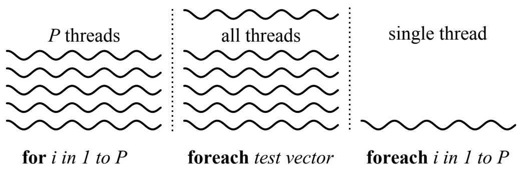

Figure 3: Parallel native implementation threading.

```
randomly create and evaluate initial population;
while termination condition is false do
    #pragma omp for
    for i in 1 to P do
        if i = best_ind then
            continue;
        end
        copy ind[best_ind] chromosome to ind[i];
        mutate ind[i];
        analyze ind[i];
        compile ind[i];
    end
    foreach i in 1 to P do
        fit[i] = 0;
    end
    #pragma omp barrier
    #pragma omp for nowait
    foreach test vector do
        foreach i = 1 to P do
            if i = best_ind then
                continue;
            end
            fit[i] += run compiled code;
        end
    end
    foreach i in 1 to P do
        #pragma omp atomic
        ind[i].fit += fit[i];
    end
    #pragma omp barrier
    #pragma omp single
    for i in 1 to P do
        if ind[i].fit > ind[best_ind].fit then
            best_ind := i;
        end
    end
end
```

Algorithm 4: Parallel native implementation.

what more complicated, since it consists of three separate loops, each having a different number of iterations. Figure 3 depicts the threading scheme. At first, a new population has to be created using up to P threads, then the evaluation can utilize all the threads (if there are enough test vectors) and at the end, the new best individual has to be found, unfortunately by a single thread.

Algorithm 4 shows the overall parallel implementation principle. A special attention had to be paid for the fitness accumulation across test vectors treated by different threads. Each thread has its own partial fitness values which are finally atomically accumulated to the total fitness values doing a manual reduction over the fitness values (OpenMP offers only reduction over scalar variables).

4.2 Process parallelism (inter-population)

Spatially structured evolutionary algorithms have been intensively studied in the past and a variety of approaches differing in the used evolutionary algorithm or communication topology has emerged [19, 2]. By introducing multiple populations evolving in parallel, one can increase the population

```
seed := randomly generated individual;
while termination condition is false do
    run evolutionary design starting with seed;
    exchange best individuals among islands;
    if global best fitness is higher than local then
        seed := global best individual;
    end
end
```

Algorithm 5: Isolated islands model.

diversity and thus make the EA more explorative leading to a higher probability of finding the global optimum for particular problems.

The combinational circuit design is a very complex problem, the search space is generally rugged containing lots of local optima and thus the potential of exploiting parallel EA is high. Unfortunately, the absence of a crossover operator in CGP is a very limiting factor since most parallel models take advantage of combining genotypes from different isolated populations. Nevertheless, the model of isolated islands with migration of the best individuals in each population can be applied to this problem.

Algorithm 5 describes the parallel evolutionary process. At the beginning, each population starts with a randomly generated initial population. Until a perfectly working circuit is found, the evolutionary process is executed on each island and after specified number of generations, the best individual from each island is broadcasted to the other islands. If the global best individual has higher fitness value than the local best individual, the island is seeded by the global best one.

After migration, each isolated population is evolving independently and can explore different areas in the search space. This makes the search algorithm more effective and speeds up the evolutionary process.

The implementation is based on the Open MPI library and can be executed on computer clusters of arbitrary size as well as on a single multiprocessor giving a great scalability to the evolutionary design process.

5. EXPERIMENTAL RESULTS

In this section, experiments regarding the implementation performance are presented and the scalability of the implementation is demonstrated on selected combinational circuit design problems. All experiments were performed on a computer cluster of 112 nodes with the following hardware configuration: 2× 8-core Intel E5-2670, 128 GB RAM, 2× 600 GB 15 k scratch hard disks, connected by gigabit Ethernet and Infiniband links.

The implementations have been examined by means of the common metrics: *speedup*, defined as the ratio of the sequential implementation execution time to the parallel execution time, and *efficiency*, the ratio between the achieved speedup and the number of threads.

5.1 Sequential implementation efficiency

The performance of the sequential implementations has been measured in the task of a combinational adder design. Table 1 and Figure 4 summarize the mean evolution times obtained from 100 independent runs for individual sequential implementations. The CGP parameters were set as follows: population of 5 individuals, $n_c = 100$ nodes, mutation rate 5 %, $\Gamma = \{\texttt{BUF}, \texttt{NOT}, \texttt{AND}, \texttt{OR}, \texttt{XOR}, \texttt{NAND}, \texttt{NOR}, \texttt{XNOR}\}$. The goal was not to find a fully functional solution, the evolu-

Table 1: **Sequential implementation performance (combinational adders, 10 000 generations).**

width	evolution time [s]			
	interpreted		native	
	64b	256b	64b	256b
1 × 1	0.00382	-	0.00477	-
2 × 2	0.00908	-	0.02168	-
3 × 3	0.01994	-	0.04954	-
4 × 4	0.02442	0.02398	0.05497	0.07092
5 × 5	0.04681	0.03215	0.07076	0.08550
6 × 6	0.11488	0.08280	0.11168	0.11109
7 × 7	0.97894	0.40800	0.28149	0.21899
8 × 8	6.27716	2.14536	0.88332	0.55520
9 × 9	32.64657	9.84838	3.63436	2.21870
10 × 10	154.38932	47.59685	14.99801	8.75244

Table 2: Sequential implementation speedup (combinational adders, 10 000 generations).

width	speedup [-]		
	interpreted	native	
	256b	64b	256b
1 × 1	-	0.80084	-
2 × 2	-	0.41882	-
3 × 3	-	0.40250	-
4 × 4	1.01835	0.44424	0.34433
5 × 5	1.45599	0.66153	0.54749
6 × 6	1.38744	1.02865	1.03412
7 × 7	2.39936	3.47771	4.47025
8 × 8	2.92592	7.10633	11.30612
9 × 9	3.31492	8.98276	14.71428
10 × 10	3.24369	10.29399	17.63957

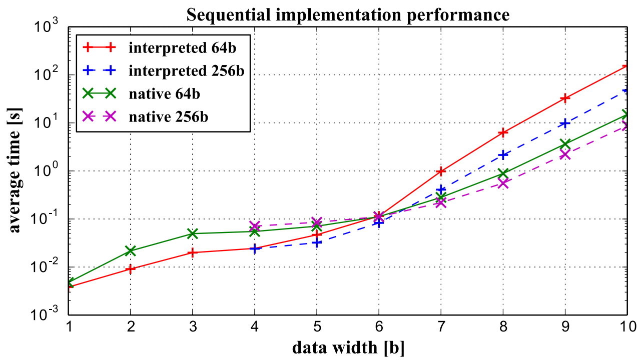

Figure 4: Sequential implementation performance (combinational adders, 10 000 generations).

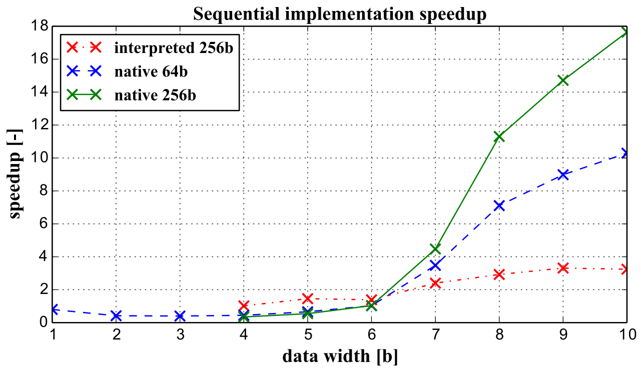

Figure 5: Sequential implementation speedup (combinational adders, 10 000 generations).

tion has been stopped after reaching 10 000 generations. The achieved speedup relative to the 64b interpreted implementation can be seen in Table 2 and Figure 5. The experimental results indicate that for small circuits, the best implementation is the 64b interpreted implementation. Starting with 8 primary inputs, the number of test vectors goes over the limit of 256 test vectors and the AVX implementation comes to the foreground. The native AVX implementation needs even larger circuits to overcome the compilation overhead and to be sufficiently efficient, however, the achieved speedup is significant.

5.2 Parallel implementation efficiency

The sequential implementation performance is not substantially dependent on the number of nodes, which is not the case of the parallel implementation efficiency. Therefore, in order to evaluate the parallel speedup and efficiency, a more complex circuit has been chosen for the experiment, namely the combinational multiplier with $n_c = 800$ nodes. The population size was 5 individuals, the evolutionary process was stopped after 100 000 generations in the case of data widths 4–6 bits, 10 000 otherwise. Table 3 summarizes

Table 3: Parallel implementation efficiency.

width	threads	time [s]	speedup [-]	efficiency [%]
4 × 4	1	1.092	-	-
	2	0.705	1.550	77.484
	3	0.683	1.598	53.277
	4	0.490	2.227	55.677
5 × 5	1	1.409	-	-
	2	0.857	1.644	82.213
	3	0.837	1.683	56.094
	4	0.567	2.486	62.149
6 × 6	1	3.616	-	-
	2	2.048	1.766	88.295
	3	1.950	1.855	61.827
	4	1.226	2.950	73.751
7 × 7	1	0.584	-	-
	2	0.372	1.571	78.554
	3	0.295	1.983	66.103
	4	0.247	2.361	59.033
	5	0.212	2.755	55.101
	6	0.198	2.947	49.119
	7	0.202	2.899	41.410
	8	0.196	2.985	37.312
8 × 8	1	1.663	-	-
	2	0.925	1.797	89.872
	3	0.678	2.452	81.745
	4	0.539	3.082	77.044
	5	0.448	3.710	74.202
	6	0.399	4.163	69.385
	7	0.380	4.370	62.424
	8	0.347	4.786	59.831
9 × 9	1	6.108	-	-
	2	3.251	1.879	93.945
	3	2.202	2.774	92.466
	4	1.717	3.556	88.908
	5	1.422	4.294	85.879
	6	1.230	4.964	82.730
	7	1.105	5.529	78.990
	8	0.985	6.199	77.482
10 × 10	1	25.825	-	-
	2	13.455	1.919	95.972
	3	9.070	2.847	94.912
	4	7.077	3.649	91.231
	5	5.840	4.422	88.440
	6	4.905	5.265	87.758
	7	4.363	5.919	84.553
	8	3.842	6.722	84.029

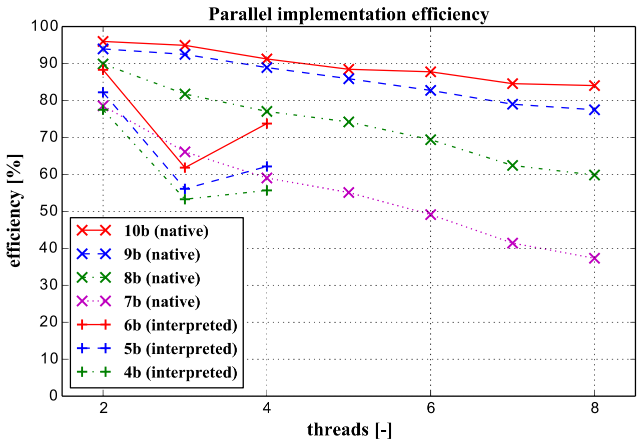

Figure 6: Parallel implementation efficiency.

the results. For each input data width, 100 independent runs were performed and the mean evolution time has been calculated.

The interpreted parallel implementation is limited to λ threads since each thread treats its own individual. Besides that, the number of threads should divide λ for the best load balancing. This is not the case of the native parallel implementation, however, sufficiently many test vectors need to be evaluated to fully utilize all eight cores of the E5-2670 processor (Figure 6).

The parallel efficiency is affected by the dynamic frequency scaling which is present in Intel's processors. In real deployment, when all processor cores are fully loaded, the sequential implementation evinces slightly worse performance due to lower frequency, hence the parallel efficiency is better. Especially on multi-socket systems, one must mind the affinity of the threads, preferably by binding individual threads to specific processor cores, to reduce cache misses and keep the memory access time as low as possible.

5.3 Test problems

Three different configurations of the evolutionary algorithm have been examined – standard single-population CGP optionally parallelized, multi-population CGP with few isolated islands and massively parallel CGP exploiting tens of islands. The performance of these approaches has been evaluated in the task of a combinational adder and multiplier design, in the literature generally considered as very difficult tasks [23, 18]. The evolutionary design was stopped after finding a fully functional solution.

Table 4: Combinational adder design performance.

width	nodes	hosts/	time [s]		
	$n_r \times n_c$	threads	mean	median	std
6 × 6	1 × 100	1/6	39.4644	31.713	27.87117
		6/1	12.3851	10.620	7.966
		60/1	3.259	2.871	1.608
7 × 7	1 × 150	1/6	142.419	103.722	139.862
		6/1	53.479	44.286	37.883
		60/1	17.569	16.410	7.864
8 × 8	1 × 200	1/6	367.915	307.545	254.455
		6/1	129.546	106.111	76.887
		60/1	57.961	48.986	44.527
9 × 9	1 × 250	1/6	3085.891	2802.061	1548.292
		6/1	1607.212	1413.261	795.004
		60/1	525.032	462.648	250.616

Table 5: Combinational multiplier design performance.

width	nodes	hosts/	time [s]		
	$n_r \times n_c$	threads	mean	median	std
2 × 2	1 × 50	1/1	0.00451	0.00333	0.00385
3 × 2	1 × 100	1/1	0.0451	0.0338	0.0319
3 × 3	1 × 200	1/1	1.897	1.469	1.605
		6/1	0.530	0.417	0.403
4 × 3	1 × 400	1/4	10.365	8.427	8.726
		6/1	5.106	3.979	3.991
		60/1	2.457	2.210	1.140
4 × 4	1 × 800	1/4	817.689	874.075	148.275
		6/1	538.058	458.494	310.345
		60/1	191.175	154.922	141.206
5 × 4	1 × 1200	60/1	761.327	700.151	303.906
5 × 5	1 × 1600	60/2	16452.75[3]		

Table 4 shows the statistics for combinational adders of data widths 6–9 bits calculated on a set of 100 independent runs for each experimental setup, namely the mean evolution time, median and standard deviation. In the case of the multi-population approaches, the migration of the best individuals occurred every 100 000 generations. It can be observed that the multi-population approach even with few isolated islands significantly reduces the time requirements on the design process compared to the single-population CGP using the same computational capacity (6 threads vs. 6 processes). By increasing the number of islands, the evolution time decreases and according to the standard deviation, the convergence becomes more stable. The stalling effect in the fitness function, commonly observed when using other approaches [18, 1], is mitigated as a consequence of a more explorative search.

The evolutionary design of combinational multipliers is an even more complex task. No satisfactory results related to techniques operating on the whole circuit without decomposition have been published so far. While paper [23] reports only a single complete run for the 4-bit multiplier, with the aid of the proposed highly optimized CGP implementation, we can routinely design 4-bit multipliers and moreover, 5-bit multipliers are feasible as well (Table 5).

6. CONCLUSIONS

In this paper, highly optimized CGP implementations have been presented. Starting with several sequential versions, the paper thoroughly examines miscellaneous implementation aspects and gives detailed performance comparisons of the proposed approaches. Parallelism at various levels has been applied in order to speed up the evolutionary design process. The native implementation based on compilation of the genotype into machine code exploits the instruction-level parallelism by reducing program branching and increasing the arithmetic intensity. A large amount of test vectors can be evaluated in parallel thanks to the use of AVX instructions. Besides a thread-parallel version, a process-parallel implementation based on the isolated islands model has been proposed.

The performance and scalability has been demonstrated on the task of combinational adders and multipliers design which is believed to be a very complex task. No additional knowledge has been introduced into the design process. All

[3]Due to the very high computational effort, only a single experiment has been executed for the 5-bit multiplier.

experiments started from a randomly generated initial population. In comparison with the previously published results regarding similar evolutionary design approaches, much more complex circuits are feasible to be designed with the proposed CGP implementation.

Note that the absence of a crossover operator in CGP is a potential limiting factor and by inventing a suitable one, more efficient parallel evolutionary approaches could be applied. In our future research, we will focus on investigating more sophisticated spatially structured evolutionary algorithms with the aim of designing even more complex combinational circuits on computer clusters.

7. ACKNOWLEDGMENTS

This work was supported by the Czech Science Foundation project 14-04197S. The access to the CERIT-SC computing and storage facilities provided under the programme Center CERIT Scientific Cloud, part of the Operational Program Research and Development for Innovations, reg. no. CZ. 1.05/3.2.00/08.0144 is highly appreciated.

8. REFERENCES

[1] T. Aoki, N. Homma, and T. Higuchi. Evolutionary synthesis of arithmetic circuit structures. *Artif. Intell. Rev.*, 20(3-4):199–232, Dec. 2003.

[2] E. Cantu-Paz. *Efficient and Accurate Parallel Genetic Algorithms.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[3] J. Clegg, J. A. Walker, and J. F. Miller. A new crossover technique for cartesian genetic programming. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1580–1587, London, 7-11 July 2007. ACM Press.

[4] A. Fox. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. `http://agner.org/optimize/instruction_tables.pdf`, 2013.

[5] B. W. Goldman and W. F. Punch. Reducing wasted evaluations in cartesian genetic programming. In *Proceedings of the 16th European Conference on Genetic Programming*, EuroGP'13, pages 61–72, Berlin, Heidelberg, 2013. Springer-Verlag.

[6] S. Harding and W. Banzhaf. Implementing cartesian genetic programming classifiers on graphics processing units using gpu.net. In *13th Annual Genetic and Evolutionary Computation Conference, GECCO 2011, Companion Material Proceedings, Dublin, Ireland, July 12-16, 2011*, pages 463–470. ACM, 2011.

[7] S. Harding, J. F. Miller, and W. Banzhaf. Self modifying cartesian genetic programming: Parity. In *2009 IEEE Congress on Evolutionary Computation*, pages 285–292. IEEE Press, 2009.

[8] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2011.

[9] R. Hrbacek and M. Sikulova. Coevolutionary cartesian genetic programming in fpga. In *Advances in Artificial Life, ECAL 2013, Proceedings of the Twelfth European Conference on the Synthesis and Simulation of Living Systems*, pages 431–438. MIT Press, 2013.

[10] J. R. Koza, M. A. Keane, M. J. Streeter, W. Mydlowec, J. Yu, and G. Lanza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence.* Kluwer Academic Publishers, 2003.

[11] J. Miller and S. Smith. Redundancy and computational efficiency in cartesian genetic programming. *Evolutionary Computation, IEEE Transactions on*, 10(2):167–174, 2006.

[12] J. Miller and P. Thomson. Cartesian genetic programming. In *Genetic Programming*, volume 1802 of *Lecture Notes in Computer Science*, pages 121–132. Springer Berlin Heidelberg, 2000.

[13] J. F. Miller, editor. *Cartesian Genetic Programming.* Natural Computing Series. Springer Verlag, 2011.

[14] R. Poli and J. Page. Solving high-order boolean parity problems with smooth uniform crossover, sub-machine code gp and demes. *Genetic Programming and Evolvable Machines*, 1(1-2):37–56, Apr. 2000.

[15] R. Seyfarth. *Introduction to 64 Bit Intel Assembly Language Programming for Linux.* CreateSpace Independent Publishing Platform, 2012.

[16] A. P. Shanthi and R. Parthasarathi. Practical and scalable evolution of digital circuits. *Appl. Soft Comput.*, 9(2):618–624, Mar. 2009.

[17] D. Sherry, K. Veeramachaneni, J. McDermott, and U.-M. O'Reilly. Flex-gp: Genetic programming on the cloud. In *Applications of Evolutionary Computation - EvoApplications 2012*, volume 7248 of *LNCS*, pages 477–486. Springer, 2012.

[18] E. Stomeo, T. Kalganova, and C. Lambert. Generalized disjunction decomposition for evolvable hardware. *Trans. Sys. Man Cyber. Part B*, 36(5):1024–1043, Oct. 2006.

[19] M. Tomassini. *Spatially Structured Evolutionary Algorithms: Artificial Evolution in Space and Time (Natural Computing Series).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[20] Z. Vasicek and L. Sekanina. Hardware accelerator of cartesian genetic programming with multiple fitness units. *Computing and Informatics*, 29(6):1359–1371, 2010.

[21] Z. Vasicek and L. Sekanina. Formal verification of candidate solutions for post-synthesis evolutionary optimization in evolvable hardware. *Genetic Programming and Evolvable Machines*, 12(3):305–327, 2011.

[22] Z. Vasicek and K. Slany. Efficient phenotype evaluation in cartesian genetic programming. In *Proc. of the 15th European Conference on Genetic Programming*, LNCS 7244, pages 266–278. Springer Verlag, 2012.

[23] V. K. Vassilev, D. Job, and J. F. Miller. Towards the automatic design of more efficient digital circuits. *Evolvable Hardware, NASA/DoD Conference on*, 0:151, 2000.

[24] T. Yu and J. Miller. Finding needles in haystacks is not hard with neutrality. In *Genetic Programming*, volume 2278 of *Lecture Notes in Computer Science*, pages 13–25. Springer Berlin Heidelberg, 2002.