GPU-accelerated Evolutionary Design of the Complete Exchange Communication on Wormhole Networks

Jiri Jaros
Faculty of Information Technology
Brno University of Technology
Bozetechova 2
612 66 Brno, Czech Republic
jarosjir@fit.vutbr.cz

Radek Tyrala
AT&T Mobility
Palachovo namesti 7262
625 00 Brno, Czech Republic
radek.tyrala@att.com

ABSTRACT

The communication overhead is one of the main challenges in the exascale era, where millions of compute cores are expected to collaborate on solving complex jobs. However, many algorithms will not scale since they require complex global communication and synchronisation. In order to perform the communication as fast as possible, contentions, blocking and deadlock must be avoided. Recently, we have developed an evolutionary tool producing fast and safe communication schedules reaching the lower bound of the theoretical time complexity. Unfortunately, the execution time associated with the evolution process raises up to tens of hours, even when being run on a multi-core processor. In this paper, we propose a revised implementation accelerated by a single Graphic Processing Unit (GPU) delivering speed-up of 5 compared to a quad-core CPU. Subsequently, we introduce an extended version employing up to 8 GPUs in a shared memory environment offering a speed-up of almost 30. This significantly extends the range of interconnection topologies we can cover.

Categories and Subject Descriptors

C.1.2 [**Processor architectures**]: Multiple Data Stream Architectures—*Interconnection architectures, Parallel processors, multiple-data-stream processors (SIMD)*
D.2.2 [**Software engineering**]: Design Tools and Techniques—*Evolutionary prototyping*

General Terms

Algorithms, Performance, Design.

Keywords

Complete exchange communication; Collective communications; Communication scheduling; Evolutionary design; GPU-based acceleration; Multi-GPU systems.

GECCO'14, July 12–16, 2014, Vancouver, BC, Canada.

http://dx.doi.org/10.1145/2576768.2598315 .

1. INTRODUCTION

A recent trend in high performance computing (HPC) has been towards the use of parallel processing to solve computationally-intensive problems. Nowadays, with the enormous transistor budgets of 32-nm and 22-nm technologies on a silicon die, it is feasible to place large CPU clusters on a single chip (System on Chip, SoC). Such systems are based on both x86 compute cores (Intel Xeon Phi[1]) and ARM based cores (Paralella[2], Calxeda[3], etc). The memory of many-core systems is physically distributed among computing nodes that communicate by sending data through a Network on Chip (NoC) [5]. With an increasing number of processor cores, memory modules and other hardware units in the latest chips, the importance of communication among them and of related interconnection networks is steadily growing.

Communication operations can be either point-to-point, with one source and one destination, or collective, with more than two participating processes. Some embedded parallel applications, like network or media processors, are characterized by independent data streams or by a small amount of inter-process communications [6]. However, many general-purpose parallel applications display a bulk synchronous behaviour: the processing nodes access the network according to a global, structured communication pattern.

The performance of these collective communications (CC) has a dramatic impact on the overall efficiency of parallel processing. The most efficient way to switch messages through the network connecting multiple processing elements makes use of pipelined wormhole (WH) switching [12]. Wormhole switching reduces the effect of the path length on the communication time, but if multiple messages exist in the network concurrently (as it happens in CCs), contention for communication links may be a source of congestion and waiting times. To avoid congestion delays, it is necessary to organize CC into separate steps in time and to put into each step only such pair-wise communications whose paths do not share any links. The contention-free scheduling of CCs is therefore important.

In our previous work, we developed an evolutionary tool capable of creating optimal communication schedules for various communication patterns on wide range of interconnection network topologies with the size of up to 256 nodes [7]. The only exception was the complete exchange (CE) pattern where we had been unsuccessful even on small meshes with 32 nodes. This was given by the overwhelming run time requirements on the order of days even when running on a quad-core CPU. Since the CE is very widely used in iterative finite difference methods, fast Fourier transformations, numeric PDE solvers, linear algebra and many other, we de-

[1]http://www.intel.com/
[2]http://www.parallella.org/
[3]http://www.calxeda.com/

cided to harness the potential of massive parallel architectures such as GPUs to accelerate the evolution and produce optimal schedules in more realistic times.

The rest of this paper is organised as follows. The section continues with discussing the main characteristics of GPUs and provides fundamental concepts for writing fast codes. Section 2 describes the problem of scheduling the complete exchange communication pattern, the evolutionary algorithm, and defines the solution encoding followed by the fitness function. Section 3 introduces the accelerated version of the EA, presenting the key design decisions. The experimental results are presented in section 4 for two different GPU cards and a multi-GPU system. Finally, a summary and directions for the future work are provided in the Conclusions.

1.1 GPU Architecture and Programming

Graphics Processing Units (GPUs) are massively parallel accelerators primarily targeted on speeding up the computer graphics with millions of independent polygons and pixels. Their architecture is therefore adapted for executing relatively simple algorithms on big datasets in parallel. Nevertheless, GPUs have become key part of many supercomputing systems (Titan in Oak Ridge) and the trend in their employment steadily grows. There are two basic APIs commonly used for writing the GPU-accelerated applications, namely CUDA[4] and OpenCL[5]. Roughly speaking they are more or less the same. In this work, we decided to implement the codes in OpenCL to be able to support a wider range of GPU cards (we intend to deploy the code on Intel Xeon Phi cards in the future), however, porting it on CUDA would be trivial.

As we mainly targeted the proposed code on NVIDIA GTX 580 cards, we are going to provide the fundamental architecture basics of the Fermi architecture in this subsection. The graphics card is divided into a graphics chip (GPU) and 1.5GB of main memory. Main memory, acting as an interface between the host CPU and GPU, is connected to the host system using a PCI-Express 2.0 bus. This bus can easily become a bottleneck as its bandwidth is only a fraction of what both GPU and CPU memories provide [9].

The GPU main memory is optimized for block transactions and stream processing providing very high bandwidth but also high latency. Hiding this latency is very important for keeping GPU executions units busy. The GTX 580 offers 768KB of fast on-chip L2 cache to allow reordering of the memory requests as well as on-chip shared memory, and large register fields to get the working data as close the execution units as possible. The GTX580 processor consists of 16 independent Streaming Multiprocessors (SM), each of which is further divided into 32 CUDA cores. SMs are based on the Single Instruction, Multiple Thread (SIMT) concept allowing them to execute exactly the same instruction over a batch of 32 consecutive threads (referred to as a warp) at a time. This concept dramatically reduces the control logic of SMs, but on the other hand, dictates strict rules on thread cooperation and branching. A few consecutive warps form a thread block that is the smallest resource allocation unit of SM. In order to fully exploit the potential of a given GPU, a few concepts must be kept in mind [13]:

- Thousands of threads are necessary to be executed concurrently on the GPU to hide memory latency.
- All the threads within a warp should follow the same execution path minimizing the thread divergence.
- All memory requests within a warp should be coalesced reading data from consecutive addresses.
- Synchronization and/or communication among threads can be done quickly only within a thread block.
- Working data set should be partitioned to fit on-chip shared memory to minimize main memory accesses.
- Data transfers between CPU and GPU memories can easily become a bottleneck given the low PCI-Express bandwidth.

[4]https://developer.nvidia.com/cuda-downloads
[5]http://www.khronos.org/opencl/

2. EVOLUTIONARY DESIGN

The selection of Evolutionary Algorithms (EA) for the scheduling problem has been justified already in [7]. Although the proposed methodology of designing near-optimal CC schedules is independent of the particular evolutionary algorithm, we restricted ourselves in this work only to a simple EDA evolutionary algorithm without gene dependencies (UMDA).

Univariate Marginal Distribution Algorithm (UMDA) [11] is a very simple EDA [10] (Estimation of Distribution Algorithm) which does not reflect any interaction between genes (variables/solution parameters). The main advantages of this algorithm are better mixing of genetic material than it is possible in standard GA [4], very simple implementation and much faster execution than more complex EDAs like BOA (Bayesian Optimization Algorithm [10]) algorithms. Of course, any other EA could be employed. Basic comparison of a success rate and execution time of other types of EA applied to CC scheduling problem can be found in [7].

2.1 Complete Exchange Communication

The Complete Exchange (CE) communication pattern, also referred to as all-to-all scatter (AAS) or personalised all-to-all broadcast is one of the most complex collective communication (CC) routine present in many parallel algorithms, such as BLAS, FFT, island model of EA and many others. This communication is often aligned with matrix transposition where the data is originally distrusted over processing elements by rows and after applying the operation by columns.

The complete exchange algorithm is usually defined in terms of a group of processes. In this case, we have P communication processes each of which is sending and receiving a message to all its $P-1$ partners. To complete one complete exchange, $P(P-1)$ point-to-point message transfers have to be carried out.

The simplest time model of point-to-point communication in direct Wormhole (WH) networks takes the communication time composed of a fixed start-up time t_s at the beginning (SW and HW overhead of a sender and a receiver), a serialization delay, i.e. the transfer time of m message units (words or bytes), and of a component that is a function of distance h (the number of channels on the route or hops a message has to do)

$$t_{p2p} = t_s + mt_1 + ht_r \quad (1)$$

where t_1 is per unit-message transfer time and t_r includes a routing delay, switching and inter-router latency. A relatively small dependence on h may be taken into account by including $h_{max}t_r$ into t_s, so that only two parameters t_s and mt_1 are sufficient.

In the rest of the paper we assume that all collective communication (CC) including CE in WH networks proceeds in synchronized steps. In one step of CC, a set of simultaneous packet transfers takes place along complete disjoint paths between source and destination node pairs. If the source and destination nodes are not adjacent, the messages go via some intermediate nodes, but processors in these nodes are not aware of it; the messages are routed automatically by the routers attached to processors.

Complexity of collective communication is determined in terms of the number of communication steps or equivalently by the number of "start-ups" τ_{CC} (upper bound). Provided that the term $h_{max}t_r$ is included in t_s and excluding contention for channels, CC time can be obtained approximately as the sum of start-up delays plus associated serialization delays $m_i t_1$ in individual communication steps

$$\tau_{CC} = \sum_{i=1}^{\tau_{CC}} (t_s + m_i t_i) = \tau^{CC}[t_s + mt_1]. \quad (2)$$

The above expression assumes that the nodes can only re-transmit/consume original messages, so that the length of messages $m_i = m$ remains constant in all communication steps. This is true in the so called non-combining model of communication; on the contrary, in the combining model the nodes can combine/extract partial messages with negligible overhead. The combining/non-combining model influences CC performance and either one can outperform the other in some cases. Further on we will consider the non-combining model only. Possible synchronization overhead involved in communication steps, be it hardware or software-based, should be included in the start-up time t_s. Let us note that with uniform messages and a single clock signal domain, one barrier synchronization before CC might be sufficient to synchronize the whole CC. Communication steps would then follow in the lockstep. According to frequency of CCs and an amount of interleaved computation (BSP model) in a certain application, efficiency of parallel processing can be estimated.

The lower bound of the time complexity (number of communication steps) for complete exchange can be obtained considering that half the messages from each processor cross the bisection, whereas the other half do not. There will be altogether $\lceil 2(\frac{P}{2})^2/BC \rceil$ of such messages in both ways, where BC is the channel bisection width [2]. Sometimes a stronger lower bound may be obtained considering the count of channels from all sources to all destinations (Σ) and the limited count Σ_1 of channels available for one step. In regular networks with constant node degree such as hypercube and torus $\Sigma_1 = Pk$, where k is the number of output channels in each node. In irregular networks such as mesh, as each node has to accept $P-1$ distinct messages, $\lceil (P-1)/min(k) \rceil$ bound has to be also obeyed.

2.2 Input Data Structure and Preprocessing

The input data structure maintains a description of the network topology, the definition of CC and sets of transmitters, receivers and intermediate routers. The topology description is saved in the form of node neighbour lists, where the nodes are considered to be neighbours only if they are connected by a simple direct link.

After the input file is loaded, the data have to be preprocessed. In the first phase, the preprocessor divides the set of all nodes V^* into a set of transmitters T and a set of receivers R. Then, a set of terminal nodes $V \subseteq V^*$ is determined as the union $T \cup R$. The terminal nodes can inject/consume messages to/from the network, while the non-terminal nodes (routers) can only retransmit the messages. Finally, all the sets are ordered based on the node index.

The preprocessor generates all the shortest paths (the set R_{xy}) between all transmitter-receiver pairs $x, y \in V$ and saves them into a specific data structure in the operating memory during the second phase. This task is performed by a modified well known Breadth-First Search (BFS) algorithm [3].

2.3 Solution Encoding

Let us define the chromosome encoding for the complete exchange communication. Considering CE communication between M transmitters from set T and N receivers from set R:

1. The complete exchange can be defined as a set CC of pairwise transfers src, dst originating in $src \in T$ and terminating in $dst \in R$, where $src \neq dst$.

$$CC = \{p2p_{src,dst} : src \in T, dst \in R, src \neq dst\} \quad (3)$$

2. A direct encoding can be designed for the CE schedules (i.e. an exact description of the schedule is stored in a chromosome). Chromosome can be formalized as n-tuples of genes:

$$chr = \begin{pmatrix} gene_{0,0} & \dots & gene_{0,N-1} \\ \vdots & \ddots & \vdots \\ gene_{M-1,0} & \dots & gene_{M-1,N-1} \end{pmatrix}, \quad (4)$$

where M is the number of transmitters and N is the number of receivers while n is the total number of genes. Notice that

$$M, N \leq P \wedge n = M \cdot N. \quad (5)$$

3. A gene $gene_{i,j}$ represents a single message transfer from the transmitter $x_i \in T$ to the receiver $x_j \in R$, where $x_i \neq x_j$. The source and the destination are identified by the genes' indexes i and j. A gene is the ordered couple:

$$\begin{array}{l} gene_{i,j} = [l_{i,j}, s_{i,j}], 0 \leq i < M, 0 \leq j < N, i \neq j \\ l_{i,j} \in R_{i,j} \\ 0 \leq s_{i,j} < Steps \end{array} \quad (6)$$

The first component $l_{i,j}$ represents a chosen path from the transmitter x_i to the receiver x_j stored in the set $R_{i,j}$. The second component $s_{i,j}$ determines a selected time slot (communication step) of the transfer. The total number of time slots is given as the predefined parameter $Steps$.

4. The (shortest) path is defined as an ordered set of unidirectional channels:

$$l_{i,j} = \{c_1, c_2, c_3, \dots, c_L\}, c_i = [a, b] \in V^* \times V^* \quad (7)$$

where $c_1 = [a_1, b_1] \wedge a_1 = x_i$ and $c_L = [a_L, b_L] \wedge b_L = x_j$.

5. Next, consider a set G containing all the genes included in a chromosome chr:

$$G = \{gene_{i,j} : 0 \leq i < M, 0 \leq j < N, i \neq j\}. \quad (8)$$

6. Finally, we can define bijective mapping f from set G into set CC meaning that each gene corresponds to a unique pairwise transfer and also vice versa:

$$\begin{array}{c} f : gene_{i,j} \in G \mapsto p2p_{src,dst} \in CC \iff \\ x_i = src, x_j = dst. \end{array} \quad (9)$$

2.4 Conflict-based Fitness Function

This section proposes a formal description of the fitness function.

1. Let SS (Same Slot/Step) be a binary relation on the set G. Let $a, b \in CC$ be message transfers represented by $gene_{i,j}$ and $gene_{k,l}$, then

$$gene_{i,j} SS gene_{k,l} \iff s_{i,j} = s_{k,l}. \quad (10)$$

Thus, two transfers are in relation SS if and only if they are executed during the same time slot.

Now, we show that SS is an equivalence relation:

(a) SS is reflexive, since no transfer can be performed in more than one time slot.

(b) SS is clearly symmetric considering $s_{i,j} = s_{k,l}$, then $s_{k,l} = s_{i,j}$.

(c) SS is transitive. Let a, b, c be elements of G. Whenever $aSSb$ and $bSSc$, then also $aSSc$ (a is executed during the same slot as c whenever a is executed during the same slot as b and b is executed during the same slot as c).

Thus SS is an equivalence relation.

2. The equivalence relation SS induces the partition on set G. Each equivalence class $[g_s]$ includes all transfers performed in the same slot s.

3. Let $E_{a,b}$ be a set of all channels shared by two transfers a, b represented by genes $gene_{i,j}, gene_{k,l} \in G$ going over paths $l_{i,j}$ and $l_{k,l}$. Then

$$E_{a,b} = l_{i,j} \cap l_{k,l}. \tag{11}$$

The number of conflicts between a and b can be obtained as the cardinality of the set $E_{a,b}$.

4. Define a multiset E_s including channels shared by all transfers within a given time slot, then

$$E_s = \bigcup_{a,b \in [g_s], a \neq b} E_{a,b}. \tag{12}$$

5. The multiset E, covering all shared channels within the whole CC, can be obtained as a union over all equivalence classes. Thus

$$E_s = \bigcup_{[g_s] \in G/SS} E_s. \tag{13}$$

6. The total number of conflicts can be obtained as the cardinality of multiset E. Thus

$$Fitness = |E|. \tag{14}$$

The valid communication schedule for a given number of communication steps must be conflict-free. Valid schedules are either optimal (the number of steps equals the lower bound) or suboptimal. Evolution of a valid schedule for the given number of steps is finished up as soon as fitness (number of conflicts) drops to zero. If it does not do so in a reasonable time, the prescribed number of steps must be increased.

2.5 Execution Time on Small Networks

In our previous work we have applied this approach on a wide range of network topologies targeted on networks on a chip with up to 256 nodes. However, the results for the complete exchange have not been satisfactory. Table 1 shows the time complexity of the evolved communication plans in term of communication steps followed by the lower bound. It can be seen that for bigger topologies (36 nodes and more) the EA was not successful and produced only suboptimal solutions. The table also shows the average runtime of the EA to find such a solution. It can be seen that the run time grows significantly, reaching the order of days for as low as 36 nodes. The optimisation of bigger networks was thus impractical.

Table 1: The time complexity of best schedules discovered and the theoretical lower bounds for the complete exchange communication along with the evolution time on a quad-core CPU.

Topology	Number of time steps	Time to solution
Mesh 3x3	6/6	2s
Mesh 4x4	16/16	22m 7s
Mesh 5x5	32/32	9h 20m
Mesh 6x6	56/54	3d 20h
Cube 3D	4/4	1s
Cube 4D	9/9	10m
Cube 5D	16/16	2d 8h
Cube 6D	35/32	2d 18h
Torus 3x3	4/4	1s
Torus 4x4	9/8	19m
Torus 5x5	16/15	7h
Torus 6x6	30/24	1d 12h

3. GPU ACCELERATION

The main disadvantage of the CPU implementation is the compute time required to find an optimal solution (a conflict-free plan with minimal number of time steps) that makes optimisation of more complex communication plans impractical. Thus, we decided to accelerate the evolution by Graphics Processing Units (GPU).

There are several strategies how to port the EA on the GPU platform with different difficulty and attainable speed-up [8]. Thus, we first profiled the original CPU implementation using the GNU profiler `gprof` on typical topologies. The flat profile (Fig. 1) reveals that almost 93% of the execution time is consumed by the fitness calculation, 2% are consumed by the random number generator, another 2% by the UMDA model building and sampling, and the rest can be attributed to support routines.

Taking into account the effort necessary to port the whole evolutionary algorithm on the GPU, the limits imposed on the EA by the GPU, and the negligible overhead of the EA in our case, we decided to only offload the fitness calculation onto the GPU. This makes the design much easier, allows to simply exchange the EA engine (e.g. GAUL[6] or NSGA-II[7]) and preserves the quality of the solution (many GPU-accelerated EAs need to make compromises to get good performance). On the other hand, the simple fitness offload limits the accessible speed-up. According to the Amdahl's law [1] with the non-accelerated part of the algorithm $\alpha = 0.07$, the theoretical speed-up is bounded by 14. Practically, the speed-up factor will be even lower because of the overhead of uploading the chromosomes for evaluation onto the GPU and downloading the fitness values back. Fortunately, the use of asynchronous transfers can hide this overhead by overlapping the calculation of the actual chromosome with transferring the following one.

3.1 Global Data Representation on GPU

In order to offload the fitness function evaluation on the GPU, we have to accommodate the global data on the GPU as well. This dataset maintains the list of all shortest paths between all transmitter-receiver pairs (R_{xy}). The preprocessor orders this set into a list of 4-tuples $T \times R \times R_{xy} \times l_{ij}$ stored in the CPU memory as a sparse 4D array. Since allocation, transfer and access to sparse multidimensional arrays are very inefficient on GPU, the array was restructured into a form of a simple 1D array with the size of $|T| * |R| * max(|R_{xy}|) * max(|l_{ij}|)$ and the shortest paths copied into it.

[6] http://gaul.sourceforge.net/

[7] http://www.iitk.ac.in/kangal/codes.shtml

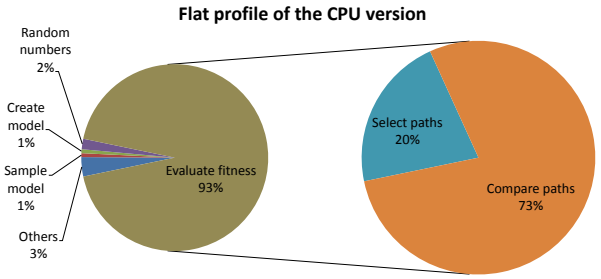

Figure 1: The flat profile of the CPU version of the code captured for a 7x7 2D mesh topology and 128 individuals in the EA population. The fitness evaluation takes more than 93 % of the execution time.

Considering that neither all shortest paths are of the same length nor there are the same number of shortest paths between all T-R pairs, this data layout introduces a non-negligible memory overhead that limit the maximum topology size to 192-256 nodes.

3.2 Fitness Evaluation on GPU

The fitness function evaluation consists of two phases. First, the path used in the G are partitioned into the equivalence classes $[g_s]$ using the equivalence relation SS in eq. (10). Second, the number of conflicts in each equivalence class is counted up and eventually summed up into the fitness function value.

Looking at Fig. 1, the path partitioning takes about 20% of the execution time while the search for conflicts does almost 80%. The CPU code for partitioning G traverses through the chromosome and classifies the path's ID into a particular equivalence class (C++ vector). Although this is a trivial operation, it requires dynamic memory allocation because it is not a priori known how many paths belongs into a given class $[g_s]$. Moreover, doing so in parallel implies the use of mutual exclusion (memory locks) when adding a path into a given class. Both these features have very limited support on GPUs and are always strongly performance penalised. Thus, this part of the fitness function evaluation is better to be processed on the CPU. The second phase, comparing the pairs of paths and seeking for conflicts, is completely data independent not requiring any dynamic memory allocation and/or synchronisation. That's why it is a good candidate for GPU acceleration.

The basic idea of the GPU-accelerated fitness function evaluation is as follows:

1. The CPU partitions the paths in G into equivalence classes and stores them in separate memory arrays.
2. The CPU uploads an equivalence class on the GPU.
3. The GPU mutually compares all path pairs within the class in parallel and stores the number of conflicts for each pair.
4. The GPU performs a parallel reduction to get the total number of conflicts within a given equivalence class.
5. The CPU downloads the number of conflicts for $[g_s]$ and adds them to the total sum from other equivalence classes.
6. The previous steps are repeated until all equivalence classes have been processed.

3.3 GPU Thread Model

The thread model describes how the work is distributed over many thousands of lightweight GPU threads. Ideally, the work should be evenly distributed, minimising the need for communication and synchronisation and maximising the data reuse. For identifying all path conflicts (shared channels), all path pairs have to be mutually compared. Having N paths in a given class $[g_s]$ yields $N(N-1)/2$ path comparisons. Considering the global exchange communication pattern, and a topology with $P = |T| = |R|$ nodes, $P^2(P^2-1)/2$ path comparisons have to be performed in the worst case. Considering even a small 4x4 mesh topology, this stands for 32 640 path comparisons.

This finding suggests that each thread could be assigned one pair of paths to compare since there are enough path pairs in larger topologies to employ hundreds of GPU threads. The GPU thread can be identified by a 1D index (ID) within a GPU kernel function. The ID determines which pair of paths from $[g_s]$ is assigned to the thread as follows:

$$\begin{aligned} &\text{1st path:} \quad \text{ID} \bmod |[g_s]| \\ &\text{2nd path:} \quad (\text{ID} + \tfrac{\text{ID}}{|[g_s]|} + 1) \bmod |[g_s]| \end{aligned} \tag{15}$$

The threads are then grouped into thread blocks of a specific size (usually 512 or 1024).

3.4 On-chip Memory Utilisation

As the path comparison involves no floating point operations and only a tiny amount of fixed point arithmetic (comparisons and indexing), the GPU kernel is strongly memory bound. Moreover, the memory access patter is not regular but scattered. Therefore, the memory accesses must be carefully designed to get reasonable performance.

Let's recall the path is stored as list of network nodes the message goes through on the way from the transmitter to the receiver, see Eq. (7). Every thread is assigned a path pair and processes it channel by channel, see Fig. 2. Since the threads are grouped into warps executed in the SIMT (Single Instruction Multiple Thread) fashion, a thread warp processes 32 path pairs simultaneously. Thus, when reading the first component of the channel (say path1[i]), the GPU actually reads first components of 32 different paths from distant places scattered over the global memory. This breaks one of the fundamental rules of the SIMT processing (neighbour threads read from neighbour memory locations), see Fig. 2.

The second issue is that the second path is read multiple times. Since there are no or very small caches on GPU, it is almost sure that all accesses will be served by global memory, which further decreases the performance.

```
1  for (int i = 0; i < path1Size - 1; i++)
2  {
3    for (int j = 0; j < path2Size - 1; j++)
4    {
5      if ((path1[i] == path2[j]) &&
6          (path1[i+1] == path2[j + 1])
7        )
8        conflicts[threadID]++;
9    }
10 }
```

Figure 2: Code snippet of the conflict seeking GPU kernel on a pair of paths. To detect conflicts, all channels has to be mutually compared.

One way how to lighten the GPU memory load would be to use shared on-chip memory to rearrange memory accesses and reuse the second path data. The idea is to use multiple threads to load a path from main memory exploiting the SIMT-friendly coalesced memory access pattern. After all paths are loaded, call a barrier

and then redistribute the paths stored in shared memory to threads for processing. Although looking promising, this idea holds many problems. First, since the paths are not of the same size, a mapping function to decide what threads are assigned to load particular paths would be necessary. This would require an expensive parallel prefix sum. Second, the paths wouldn't squeeze into the small shared memory. Considering even a small 4x4 mesh, the longest shortest path has a length of 7. Having 1024 threads in a thread block would require 2*28KB of shared memory, which is more than current GPUs can offer. Reducing the number of threads would lead to low occupancy and reduced performance. Considering all these obstacles, we concluded it would be better not to use shared memory as we did not want to limit the size of the topology optimised.

The memory bandwidth can yet be reduced by proper use of registers to hold the segments of the path for the time they are needed and not to read them from the global memory repeatedly. According to the principle of path comparing, every segment of the first path (two nodes) is compared with all segments of the second path. Thus, it will be read only once, stored in a pair of registers and compared with all segments of the second path. When moving to the flowing segment of the first path, we preserve the endpoint of the previous channel, declare it as the start point of the following channel and read only the endpoint from the maim memory. This sort of software pre-fetching reduces the memory bandwidth by 40%. The same node pre-fetching can be then applied on the second path, which further reduces the memory bandwidth down to 30%. The code snippet can be seen in Fig. 3.4.

```
node1_1 = path1[0];

for (int i = 0; i < path1Size - 1; i++)
{
  node1_2 = path1[i + 1];
  node2_1 = path2[0];

  for (int j = 0; j < path2Size - 1; j++)
  {
    node2_2 = path2[j + 1];
    if ((node1_1 == node2_1) &&
        (node1_2 == node2_2)
       )
         conflicts[threadID]++;

    node2_1 = node2_2;
  }
  node1_1 = node1_2;
}
```

Figure 3: Code snippet of the optimised conflict seeking GPU kernel on a pair of paths. The paths are loaded using software pre-fetching and stored in registers.

The numbers of conflicts identified by particular threads are stored in a shared on-chip memory array. The array has the same size as the thread block (usually 1024). Then, the parallel reduction kernel is used to sum the number of conflicts within the thread block. Finally, the first thread of the blocks adds its value to the others' one in global memory using an atomic add operation.

3.5 Multidimensional Kernel Organisation

The use of 1D GPU kernels and thread blocks imply that that a chromosome is processed in multiple stages, each communication slot and its class $[g_s]$ in one stage. Although this enables fine-granularity, the PCI-E transfers are too small to efficiently use the PCI-E bandwidth since only path indices are transferred (a few KBs). Moreover, the kernels have only a little work and has to be started many times, which introduced additional overhead.

To alleviate the overhead associated with the kernel launch and PCI-E transfers, the GPU kernels were redesigned to be multidimensional. The first dimension is associated with the paths within one communication slot $[g_s]$. The second dimension allows to process multiple communication slots. If necessary, the third dimension can be used to process multiple chromosomes at once. This introduces a high degree of freedom and allows for better workload balancing.

3.6 Multi-GPU Processing

Since many high-end desktops and supercomputer nodes can house multiple GPU accelerators, we investigated the possibility of employing multiple GPUs. Fortunately, both OpenCL and CUDA have simple extensions to support this feature.

The population of candidate solutions is divided into small batches, each of which including one or more chromosomes. Then, the CPU code spawns as many CPU threads as GPUs in the system and links each CPU thread with a single GPU. After that, two processing streams are opened on each GPU. These are used for double-buffering. When one work batch is processed using the first stream, another batch is uploaded on the GPU using the second stream, and vice-versa. This virtually hides the overhead associated with the data transfers and allows good scaling.

4. EXPERIMENTAL RESULTS

The proposed GPU implementation was tested on two machines. First, a standard desktop equipped with a quad-core Intel i7-920 CPU and 12GB of RAM was used to run the parallel CPU implementation and provide a reference point. This machine housed two NVIDA GPUs; GTX 285 - a high-end graphics of the pre-Fermi architecture released in 2009 and GTX 580 - a high-end graphics card of the Fermi architecture released in 2011. The main differences between these two cards are the presence of L2 cache and twice as high raw performance in the case of the GTX580. The memory bandwidth, the expected limitation of the proposed implementation, is comparable. The second machine was a single node of a GPU cluster equipped with two hex-core Xeons X5660 and eight NVIDIA GTX580 GPU cards. This machine served as a platform for multi-GPU tests.

Several instances of the most common topologies were used to evaluate the benefit of using the GPU accelerated implementation. We selected 2D meshes, 2D tori and multidimensional hypercubes topologies with the number of nodes varying from 4 to 64, see Table 1. The parameters of evolutionary algorithms were carefully set in our previous work as follows: the population consisted of 256 individuals, the model of the UMDA algorithm was built using 50% of the population selected by the binary tournament selection, 50% of new individuals were generated and evaluated every generation. Although not typical in EDA algorithms, a small amount of mutation was used to preserve population diversity. Let's mention that bigger populations could offer more work for GPU and thus a higher speed-up, however the population would converge slower [7]. This implies that unnecessary work would be done.

4.1 Tuning the Block and Grid Layout

In order to obtain the best possible performance we investigated the suitable thread block and grid layout and size. The performance evaluation was done on the most complex topology we had, a 2D mesh with 49 nodes containing the longest shortest paths (up to 13 hops). To spawn enough thread blocks on the GPU, we used a 2D grid layout where multiple multiple communication steps ($[g_s]$ of a

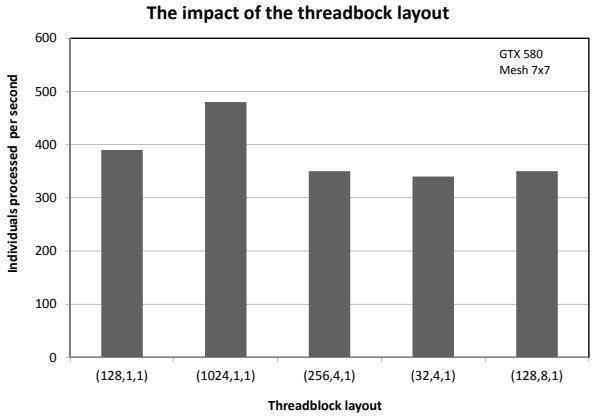

Figure 4: The influence of the GPU thread block layout on the performance on an NVIDIA GTX580 and Mesh 7x7. The fastest configuration seems to be a 1D thread block with 1024 threads.

Table 2: The number of individuals processed per second on various topologies (speed-up factor vs. CPU) using an NVIDIA GTX 285 and 580 GPUs.

Topology	GPU GTX 285	GPU GTX 580	CPU i7 920
Mesh 2x2	92,692 (0.16x)	537,895 (0.93x)	581,530
Mesh 3x3	48,369 (0.54x)	113,188 (1.26x)	89,572
Mesh 4x4	16,741 (1.05x)	22,981 (1.44x)	15,930
Mesh 5x5	3,810 (1.06x)	7,563 (2.11x)	3,588
Mesh 6x6	750 (1.49x)	2,372 (4.71x)	504
Mesh 7x7	108 (0.96x)	539 (4.81x)	112
Cube 3D	52,454 (0.36x)	85,384 (0.59x)	145,077
Cube 4D	18,621 (0.72x)	23,456 (0.91x)	25,845
Cube 5D	1,835 (1.34x)	4,152 (3.03x)	1,369
Cube 6D	51 (0.78x)	222 (3.42x)	65
Torus 4x4	18,619 (0.95x)	23,551 (1.21x)	19,498
Torus 5x5	6,336 (1.71x)	8,353 (2.26x)	3,695
Torus 6x6	1,230 (1.84x)	2,709 (4.06x)	677
Torus 7x7	351 (1.99x)	725 (4.11x)	176

single solution were being evaluated simultaneously and two GPU streams were feeding the GPU with data and overlapping communication and computation.

The question we addresses here is whether it is better to use a 2D thread block, where multiple communication steps are processed simultaneously or keep only one communication step per block. Fig. 4 shows that the best performance is obtained when the biggest possible 1D thread block of 1024 threads is used. The most likely cause of this behaviour the only one parallel reduction calculated in the 1D thread block while there are multiple reductions in 2D thread blocks. When smaller 1D thread blocks are used, the GPU is not fully exploited, and the parallel reduction with logarithmic time complexity plays a key role here. The performance of all tested 2D block configurations is comparable. We can thus conclude that a 1D thread block of 1024 threads is the best layout and this will be used in future tests.

4.2 Evolution Time on Single GPU

When the computation was optimally distributed over GPU threads and blocks, the acceleration of the EA was investigated using a number of different topologies. Table 2 shows the number of individuals processed per second. This covers not only the fitness function evaluation but the whole run of the UMDA algorithm including all the GPU-CPU transfers. The number of generations per second can thus be obtained by dividing numbers in the table by the value of 128 (the number of individuals evaluated per generation).

The first general observation says the bigger the topology the higher the performance benefit. This is natural, given by the substance of massively parallel architectures like GPUs. Hence, accelerating the communication on a 2x2 mesh cannot bring any speed-up and must deteriorate the performance. Here, there are only 16 communications and the longest path only has two hops, which gives us 120 comparisons. A speed-up factor of as low as 0.16 can be seen on a GTX 280. However, what was a congenial surprise was the performance of GTX 580 which almost met a quad-core CPU.

The achieved performance on GTX 285 was quite modest. The peak speed-up observed was about 2. On the other hand, GTX 580 showed its potential and reached a speed-up of almost 5. We suppose that the main reason was the presence of L2 cache memory that helps in reorganizing the scattered memory accesses and offers better buffering of the path. When accessing the first element of the path, the cache buffers the whole memory transaction of 128B. This is usually enough to store 32 elements of the path for long enough. The following memory accesses are thus served much faster.

There is also a notable variation in the speed-up observed on different topologies. The best one was measured for 2D meshes, followed by 2D tori and finally by n-dimensional hypercubes. It is believed that this is given by the topology diameter. The bigger the diameter is, the longer paths must be compared and the more work per thread is available reducing the overhead associated with offloading the fitness function evaluation.

When comparing the observed speed-up factors with the theoretical asymptotic ones, we can conclude that reaching 5 out of 14, considering a part of the fitness function still remains on the CPU although partially overlapped with the GPU work, is significant and helps a lot. Please remember that whole evolutionary runs for a topology with 64 nodes used to take couple of days.

4.3 Evolution Time on Multiple GPU

This section investigates the sustainable performance of a multi-GPU server under the proposed evolutionary toolkit. The parameters of the EA remained the same as in the previous section but the UMDA model was constructed and sampled by twelve CPU threads. This offered about three times higher performance than the desktop machine (the both CPUs run at the same frequency). The newly sampled individuals were then uniformly distributed over up to 8 GPUs for fitness evaluation.

Fig. 5 shows the strong scaling of the multi-GPU implementation (the population size is fixed). The performance scaling is almost ideal up to 4 GPUs. Beyond, the saturation of the system can be observed and the performance stagnates. There are three main reasons for this behaviour. First, the population is really small. When having 8 GPUs, a GPU has only 16 individuals assigned for evaluation in every generation. This decreases the possibility of pipelining and communication hiding. Second, the PCI-E subsystem gets congested by so many data transfers. It should be also mentioned that two GPUs share one PCI-E link to the CPU. Third, 12 CPU cores may not be enough to feed 8 hungry GPUs.

We anticipate that having more complex topologies would have led to much better scaling. To comment on the absolute performance provided by a single 8-GPU machine, we now have at our

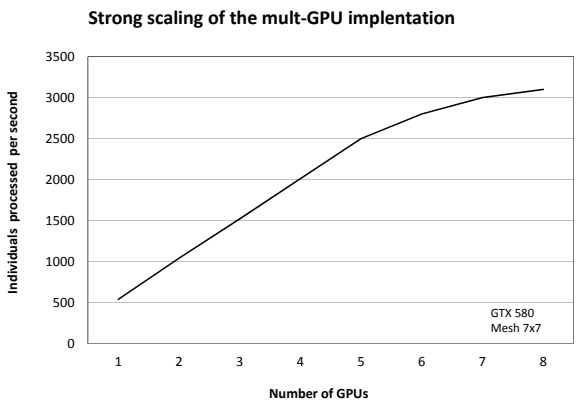

Figure 5: The strong scaling of the multi-GPU implementation tested on an 8-GPU machine equipped wit NVIDIA GTX 580 graphics cards. The GPUs are being fed by two hex-core Xeons here.

disposal a machine that accelerates the original EA algorithm running on a quad-core CPU by a factor of almost 30 (10 times compared to a two hex-core CPUs).

5. CONCLUSIONS

The importance of fast collective communications on interconnection networks of parallel computers steadily grows. Evolutionary design has been proven to be efficient in optimising the communication schedules on small scale networks, targeted mainly on networks on chips. However, the execution time necessary to evolve high quality communication schedules is still a limiting factor preventing the deployment of the evolutionary design on networks with more than 64 nodes [7].

In this paper, we have been investigating the possibility of accelerating the evolution by the use of Graphics Processing Units (GPUs). After profiling the original multi-thread CPU code, it was decided to only offload the most time consuming part of the fitness function and keep the rest of the algorithm on the CPU side. This allows us to simply exchange the evolutionary core by more advanced algorithms. On the other hand, the maximum attainable speed-up is limited due to the data transfers between CPU and GPU. Therefore, the best effort was made to hide the memory transfers and overlap them with computing.

The experimental measurements were divided into two sets. First, we investigated the performance of two NVIDIA GPU cards (GTX 285 and GTX 580). While GTX 285 showed quite poor performance, the GTX 580 reached a speed-up factor of almost 5. This reduces the typical evolutionary run on networks with 36 nodes from 30-50 hours down to 6-10. Second, we deployed the code on a multi-GPU server with 8 NVIDIA GTX 580 GPUs. Here, when supported by two hex-core Xeons, the speed-up factor reached almost 30. This reduces the execution time down to 1.5 hours, which is significant.

Let us point out the GPU acceleration has no influence on the quality of communication schedules found due to same evolutionary technique used. Naturally, the faster execution runs allow to run the evolution for longer time periods and find fast communication plans of the complete exchange pattern even for topologies we have not been successful on so far (64 nodes and more). The faster evolution also gives us a possibility to generate libraries of many irregular communication plans such as meshes/tori/cubes in faulty states where one or more links are broken in a reasonable time.

In the future work, we would like to focus on accelerating the evolution using the island model to employ a distributed GPU cluster that currently maintains 192 GPUs. We would also like to refactor the code to work with 8b integer values only, which is enough for topologies up to 256 nodes. This could reduce the memory requirements and memory bandwidth four times.

6. ACKNOWLEDGMENTS

This work was supported by the research project "Architecture of parallel and embedded computer systems", Brno University of Technology, FIT-S-14-2297, 2014-2016.

7. REFERENCES

[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[2] J. Duato and S. Yalamanchili. *Interconnection Networks - An Engineering Approach*. Morgan Kaufman Publishers, Elsevier Science, 2003.

[3] S. Gill. Parallel programming. *The Computer Journal*, 1:2–10, 1958.

[4] D. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley, Reading, MA, 1989.

[5] A. Ivanov and G. D. Micheli. Guest editorsâĂŹ introduction: The network-on-chip paradigm in practice and research. In *Proceedings of IEEE Design&Test of Computers*, pages 399–403. IEEE Los Alamitos CA, 2000.

[6] A. Jantsch and H. T. (Eds.). *Networks on Chip*. Kluwer Academic Publishers, 2003.

[7] J. Jaros. *Evolutionary Design of Collective Communications on Wormhole Networks*. Publishing house of Brno University of Technology VUTIUM, Brno, 2010.

[8] J. Jaros. Multi-gpu island-based genetic algorithm solving the knapsack problem. In *2012 IEEE World Congress on Computational Intelligence*, pages 217–224. Institute of Electrical and Electronics Engineers, 2012.

[9] J. Jaros, B. E. Treeby, and A. P. Rendell. Use of Multiple GPUs on Shared Memory Multiprocessors for Ultrasound Propagation Simulations. In J. Chen and R. Ranjan, editors, *Australasian Symposium on Parallel and Distributed Computing (AusPDC 2012)*, number AusPDC, pages 43–52, Melbourne, Australia, 2012. ACS.

[10] P. Larranaga and J. Loazano. *Estimation of Distribution Algorithms, A New Tool for Evolutionary Computation*. Kluwer Academic Publishers, 2002.

[11] H. Muhlenbein and G. Paas. From recombination of genes to the estimation of distributions i. binary parameters. In *Lecture Notes in Computer Science 1411: Parallel Problem Solving from Nature - PPSN IV*, pages 178–187, 1996.

[12] L. Ni and P. McKinley. A survey of wormhole routing techniques in direct networks. *Computer*, 26:62–76, 1993.

[13] NVIDIA. Cuda c best practices guide. Technical Report March, 2011.