Two-Echelon System Stochastic Optimization with R and CUDA

Witold Andrzejewski[1], Maciej Drozdowski[1]✉, Gang Mu[2,3], and Yong Chao Sun[2]

[1] Institute of Computing Science, Poznań University of Technology, Poland
{Witold.Andrzejewski;Maciej.Drozdowski}@cs.put.poznan.pl
[2] School of Mathematical Sciences, Tongji University, Shanghai, China
103644@tongji.edu.cn
[3] F. Hoffmann-La Roche AG, Basel, Switzerland Gang.Mu@roche.com

Abstract. Parallelizing of the supply chain simulator is considered in this paper. The simulator is a key element of the algorithm optimizing inventory levels and order sizes in a two-echelon logistic system. The mode of operation of the logistic system and the optimization problem are defined first. Then, the inventory optimization algorithm is introduced. Parallelization for CUDA platform is presented. Benchmarking of the parallelized code demonstrates high efficiency of the software hybrid.

Keywords: two-echelon problem, simulation-based optimization, CUDA, R

1 Introduction

Logistic systems are key elements of the contemporary economy. Optimizing operations of the logistic systems is essential for running distributed production facilities. The frights with goods are often managed by multi-level systems where facilities consolidate the requests, store the goods, and redistribute them to the subordinate levels, and customers. Such multi-level systems are called multi-echelon [2]. In this paper we examine a two-echelon system with one internal level and leaf facilities (cf. Fig.1). The operations of the system must be optimized by adjusting inventory levels and reorder sizes. Since multi-echelon systems have discrete event-driven nature, they are not susceptible to analytical solutions, and simulation is frequently used to analyze their performance. In this paper we report on parallelizing a two-echelon system simulator which is a core of the stochastic simulation-based optimization algorithm minimizing cost of the operations with the quality of service constraints. The optimization method applied here is an adaptation of [1]. Initially the optimization algorithm has been implemented in R language. R offers relative ease of algorithm prototyping and a wealth of data analysis libraries. However, the algorithm in R was very slow and it has been decided to parallelize its most time-consuming part: the simulations. Further organization of this paper is the following. In Section 2 the optimization problem is defined. The solution algorithm is outlined in Section 3. Parallelization of the simulation is presented in Section 4. Benchmarking results are given in Section 5. Conclusions are provided in the last section.

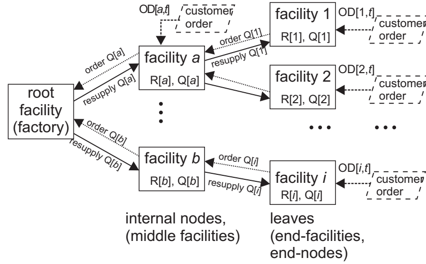

Fig. 1. Structure of the inventory system.

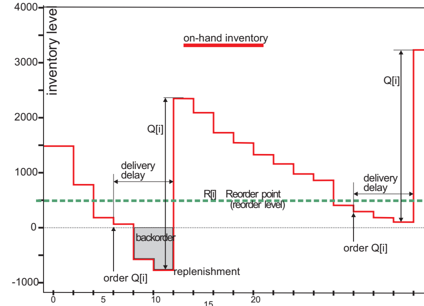

Fig. 2. Inventory level changes in the R, Q-policy.

2 Problem Formulation

The inventory system has a tree structure with the root facility, middle facilities, and leaves as depicted in Fig.1. A predecessor of facility i will be denoted $pred(i)$, the *set* of facility i successors will be denoted $succ(i)$. Customers submit orders in the middle facilities and leaves. If inventory levels are sufficient, the customers and the successors are immediately served. If the inventory level at some facility i is too low, then i submits a request to its predecessor $pred(i)$. If the inventory level at $pred(i)$ is too low to serve all the requests, then $pred(i)$ orders goods from its own predecessor $pred(pred(i))$. Thus, the requests may propagate to the root node. Processing the requests requires time. If the requests recursively propagate toward the root, then all the intermediate processing times must be included in the waiting time. Customers can be served immediately if high inventory levels are maintained. However, storing goods costs and there is a trade-off between quality of service and the cost of running the system.

Daily Bookkeeping. Let m denote the number of facilities and Nt the number of days in the simulation. The following events may happen at facility $i = 1, \dots, m$ on day $t = 1, \dots, Nt$: 1) delivery from $pred(i)$ in size of $Q[i]$ units, 2) a request of size $OD[i,t]$ from a local customer, 3) requests from subordinate nodes $j \in succ(i)$ are received in sizes $Q[j]$. Sizes of customer orders $OD[i,t]$ are generated from $\max\{0, N(\mu_i, \sigma_i)\}$, where $N(\mu_i, \sigma_i)$ is normal distribution with mean μ_i and standard deviation σ_i. Let $MoI[i,t]$ be the morning inventory level, and $EvI[i,t]$ the evening inventory level, at facility i on day t, We have $MoI[i,t] = EvI[i,t-1]$. $MoI[i,t]$ is increased by $Q[i]$ if a replenishment from $pred(i)$ arrives on day t. Let $DR[i,t-1]$ denote the total size of the earlier day demand remaining to be fulfilled at i at the beginning of day t. The aggregate demand of facility i on day t is $ROD[i,t] = OD[i,t] + DR[i,t-1] + \sum_{j \in succ(i)} Q[j]$. Let $FOD[i,t]$ be the size of fulfilled orders at node i on day t. We have: $FOD[i,t] = \min\{ROD[i,t], MoI[i,t]\}$. The size of requests remaining to be satisfied in the following days is $DR[i,t] = \max\{0, ROD[i,t] - MoI[i,t]\}$. At the end of day t the remaining inventory is $EvI[i,t] = MoI[i,t] - FOD[i,t]$.

R,Q–Policy. The R,Q–policy [4] guides inventory levels using two control parameters: reorder level $R[i]$ and reorder size $Q[i]$. The idea of R,Q–policy is shown in Fig.2. The *inventory on hand* (IOH) is the amount of goods actually available at facility i which can be immediately served. Requests which cannot be served are accumulated as *backorder*. When inventory on hand at facility i falls below $R[i]$ an order of size $Q[i]$ is placed in $pred(i)$. A new order to $pred(i)$ can be sent only after the previous one is executed.

Delivery Delays. The requests are executed immediately only if inventory levels are sufficient. Otherwise, the requests must wait until the next replenishment. A delivery delay has two components: facility *processing time* and a *recursive* component. Processing time is the interval between receiving an order and sending the replenishment. If facility i has insufficient inventory level, then the request from $succ(i)$ must additionally wait until the arrival of a shipment from $pred(i)$. This delay is represented by the recursive component. The recursive component accumulates the processing times of the preceding facilities. The root facility inventory level is unlimited. Let $pt[i,t]$ denote the processing time of the order submitted at facility i on day t. $pt[i,t]$ is generated from $U(Min_pt[i], Max_pt[i])$, where $U(a,b)$ is a discrete uniform distribution in range $[a,b]$.

Constraints and the Objective Function. Quality of the customer service is measured as the fraction of all orders served immediately in the whole volume of orders. For facility i it is $service_level[i] = \sum_{t=1}^{Nt} FOD[i,t] / \sum_{t=1}^{Nt} ROD[i,t]$. It is required that $service_level[i]$ for all facilities i be at least 0.9 with probability at least 0.95 according to Student t-distribution.

The cost of the logistic system has three components: holding, ordering, and the shortage costs. For facility i, $Holding_Cost[i] = Ch[i] \sum_{t=1}^{Nt} EvI[i,t]$, where $Ch[i]$ is the cost per unit of inventory per day. $Ordering_Cost[i]$ of i is the number of submitted orders times the cost of one order $Co[i]$. The cost of shortage is $Shortage_Cost[i] = Cs[i] \sum_{t=1}^{Nt} DR[i,t]$, where $Cs[i]$ is the cost of one unit of backorder per day. The cost is accumulated over all facilities: $Cost = \sum_{i=1}^{m} (Holding_Cost[i] + Ordering_Cost[i] + Shortage_Cost[i])$.

The problem consists in finding reorder levels $R[i]$ and sizes $Q[i]$ such that $Cost$ is minimum, subject to the above constraints on customer quality of service.

3 Solution Method

Structure of the Algorithm. The solution is a stochastic optimization method with sampling and linearization of the objective function and constraints. A method proposed in [1] has been adapted. A pseudocode of the algorithm is shown in Fig.3. Vector `Initial_IOH` provides initial inventory levels. Constants `countmax`, `deltamin` limit the run of the algorithm. Nmc is the number of samples generated by simulating the inventory system. `blackboxagent1(R,Q)` simulates the system for the given vectors of reorder levels R, reorder sizes Q and returns samples of cost and service levels concatenated in array `y`. `blackboxagent1(R,Q)` comprises two loops: over Nt days t, and m facilities i, generating user requests $OD[i,t]$, updating $MoI[i,t], EvI[i,t], Cost, service_level[i]$, etc.

```
Input: R,Q,Initial_IOH, Ch, Co, Cs, Min_pt, Max_pt, μi, σi
       Nmc=100,countmax,delta,deltamin,Nd=64,m,Nt=120,ρ       // default values of Nmc,Nd,Nt
Output: R,Q,Cost,service_level
1: repeat {                                                   // build a feasible solution
2:   for(i in 1:Nmc){y=rbind(y,blackboxagent1(R,Q))}                    // generate samples
3:   using samples y foreach i calculate from t-distribution probability
     p[i] of maintaining service_level[i]>=0.9;
4:   for(i in 1:m){if(p[i]<0.95)){Q[i]=Q[i]+50; R[i]=R[i]+50; }}
5: } until (forall i: p[i]>=0.95);
6: DM=FrF2(Nd,2m)                                          // use precomputed design matrix
7: counter=0; while (counter<countmax) {                                        // optimize
8:   counter=counter+1;
9:   for(j in 1:Nd) { for(i in 1:Nmc) {
     y=rbind(y,blackboxagent1(R+ρ*DM[j],Q+ρ*DM[j])) }};                 // generate samples
10:  using samples y calculate linear dependencies of Cost                 // linearization
     and service_level on R, Q;
11:  R',Q'=linear_program(Cost,service_level,delta)                 // linear programming
12:  for(i in 1:Nmc){y=rbind(y,blackboxagent1(R',Q'))}                 // generate samples
13:  if ((forall i: service level >=0.9 with probability >=0.95) AND
     (Cost increased with probability <=0.2)) {
14:    R=R'; Q=Q'; }                                        // R', Q' become a new solution
15:  else {
16:    if (delta<=deltamin) { return R,Q,Cost,service_level}
17:    else {delta=delta/2}}                              // retry in smaller neighborhood
18: }// end of while
```

Fig. 3. Pseudocode of the optimization method.

The algorithm can be divided into two parts. In lines 1–5 a feasible solution is searched for, while in lines 6–18 the solution is optimized. A feasible solution must guarantee that $service_level[i] \geq 0.9$ with probability at least 0.95 according to t–distribution. These constraints are verified in steps 3–5. If satisfied, then the algorithm proceeds to the cost optimization. Otherwise, $R[i]$ and $Q[i]$ are increased in the facilities i missing the constraint and the loop is reiterated.

In the second part of the code, a linear model of R, Q impact on $Cost$ and $service_level[i]$ is constructed in lines 9–10. For this purpose fractional factorial experiment design is used (explained in the following). The linear models of $Cost$ and $service_level[i]$ dependencies on R, Q are used in step 11 to formulate a linear program minimizing $Cost$ subject to the constraints on quality of service and range `delta` of R, Q changes (explained in the following). The linear program provides new values of R', Q' evaluated by simulation in line 12. If R', Q' meet the constraints on the quality of service and cost (line 13), then R', Q' become a new solution (line 14). Otherwise, the range of changes `delta` is verified. If `delta` falls below `deltamin` then the algorithm stops. In the opposite case `delta` is halved in line 17 and the algorithm reiterates. `blackboxagent1(R,Q)` is called Nmc times in each iteration of loop 1–5, and $Nmc(Nd + 1)$ times in loop 7–18. For the default setting these were 100, and 6500 calls, respectively. Hence, `blackboxagent1(R,Q)` is the biggest computational effort in the algorithm.

Linearization. Linearization method is based on fractional factorial design [3, 5]. As results linear functions linking factors (decision variables) $R'[i], Q'[i]$ with response variables $Cost, service_level[i]$ near the current values of $R[i], Q[i]$ are obtained, e.g., $Cost = c_0 + \sum_{i=1}^{m}(c_i(R'[i] - R[i]) + c_{m+i}(Q'[i] - Q[i]))$. Values of coefficients c_i are discovered by setting the factors into boundary values

$R[i] \pm \rho, Q[i] \pm \rho$, simulating the system, and checking values of *Cost*. However, the number of different boundary value settings is 2^m and it is not possible to verify them all. Which factor $R'[i], Q'[i]$ to set into which boundary value to obtain the best evaluation of the linear model of *Cost*, at the number of tests limited to Nd, is determined by a precomputed design matrix `DM` providing this information as ± 1 values. One test consists in collecting Nmc performance samples. In tests $j = 1 \ldots, Nd$ mean costs $cost_j$ are obtained. Then, coefficients c_i are calculated as $c_i = (\sum_{j=1}^{Nd} DM[j,i](cost_j - c_0))/(\rho \times Nd)$, where c_0 is the mean cost obtained in all tests. Analogously, $service_level[i] = q_{i0} + \sum_{k=1}^{m}(q_{ik}(R'[k] - R[k]) + q_{i,m+k}(Q'[k] - Q[k]))$. Coefficients q_{ij} are obtained from $q_{ij} = (\sum_{k=1}^{Nd} DM[k,j](sl[i,k] - q_{i0}))/(\rho \times Nd)$, where $sl[i,k]$ is the mean service level at facility i in test k, q_{i0} is the mean service level at i in all tests.

Linear Programming. The goal of linear optimization is to minimize *Cost* by adjusting R', Q' while obeying quality of service constraints. For this purpose an external library `Rglpk` [9], which is an interface to GNU Linear Programming Kit [6], has been used. Let $X' = (R', Q')$ denote concatenated vectors R' and Q', which are our decision variables. Let X be a vector of concatenated current values of R, Q which are constant values. *Cost* is optimized by the following linear program:

$$\text{minimize:} \quad \sum_{j=1}^{2m} c_j X'[j] \tag{1}$$

$$\text{subject to:} \quad \sum_{j=1}^{2m} q_{ij} X'[j] \geq 0.9 - q_{i0} + \sum_{j=1}^{2m} q_{ij} X[j] \quad i = 1, \ldots, m \tag{2}$$

$$X[i] - delta \leq X'[i] \leq X[i] + delta \quad i = 1, \ldots, 2m \tag{3}$$

Cost is minimized by objective (1). By inequalities (2) the quality of service is observed, while changes of $X' = (R', Q')$ are confined in range $delta$ by (3).

4 GPU Parallelization

The optimization algorithm presented in the previous section has been parallelized for the target of NVIDIA GPU architecture and the CUDA API [8].

Implementation Details. Function `blackboxagent1` is executed iteratively to collect samples of the two-echelon system performance. These iterations are independent of each other and can be run in parallel. We will call such iterations *Monte Carlo iterations* (MC iterations in short).

Each `blackboxagent1` instance (i.e. each MC iteration) executes Nt times a *day loop*. Since each day depends on the previous one, these iterations cannot be easily parallelized. Each iteration of the day loop performs three loops called *end-node loop*, *middle node loop* and *processing time loop*. In the end-node loop, each iteration refers to a different leaf of the distribution tree and this loop can be parallelized. Hence, `blackboxagent1` has been parallelized as follows: 1) iterations of the day loop are performed sequentially, 2) levels of the distribution

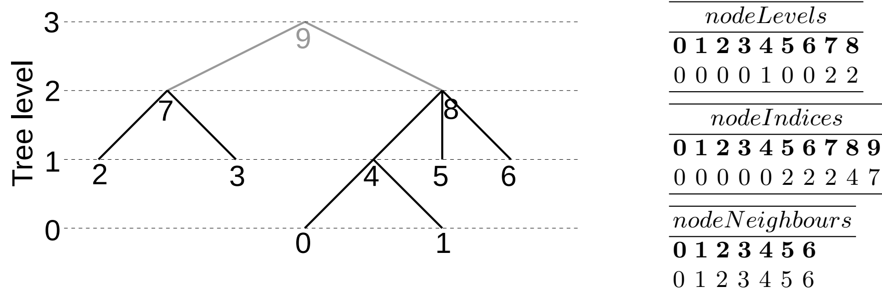

Fig. 4. Exemplary tree and the corresponding *nodeLevels*, *nodeNeighbours*, *nodeIndices* arrays. In the arrays the upper lines show the indices to guide the eye.

tree are processed sequentially, 3) nodes on the same tree level, are processed in parallel. Middle node and processing time loops are replaced with loops iterating over the range of tree levels. Costs and service levels are aggregated on the fly by every thread. After the day loop ends, threads in parallel compute their service levels and store them in the output array. Calculation of the total cost, however, requires aggregating partial costs of the threads. To compute this value a parallel segmented reduction algorithm has been employed [7]. At each iteration over the levels of the tree a different set of threads processes their nodes. The dependencies of parents on their children are retained, but synchronization between threads of a single MC iteration for each level of the tree on every day are required. This leads to the problem of thread allocation. Since currently the distribution tree is short (2 levels), it has been chosen to allocate a thread per tree node. A disadvantage of this approach is that it wastes thread resources as only threads of the currently processed tree level are working. Still, it allows to use registers and shared memory for storing processing state, and reduces the need for communication between threads. Due to synchronization, threads performing a single MC iteration have to be included in a single GPU block. Consequently, current implementation can process distribution trees of at most 1024 nodes, which is the largest number of threads within a block, e.g., for NVIDIA Tesla K80. Another issue was the assignment of MC iterations to blocks. Due to the nature of GPUs the threads are executed in warps. Hence, the block size should be a multiple of 32 and more than 64 to hide read after write dependencies. Assigning only one MC iteration per block is wasteful. Consequently, we have chosen to assign as many MC iterations per block as possible, at the maximum block size, which is an execution parameter. This introduced some unnecessary synchronizations (as only *all* threads within a block can be synchronized), but still it performed better than the other approaches.

In order to reduce the memory footprint of the `blackboxagent1` the following optimizations have been applied: Matrix MoI is referred to only for the current day t. Since the references to MoI are done to the entries of the local nodes only, this matrix has been substituted by thread private variables stored in registers. Similar optimizations have been done with EvI, ROD, FOD and DR matrices.

OD matrix has been removed in favor of computing pseudorandom values on the fly. The structure of the inventory system is represented by a number and three arrays (see Fig.4): $treeHeight$ holds the length of the longest path between the root and any leaf. Array $nodeLevels$ determines the order of processing the nodes. For each leaf, value 0 is always stored since the leaves are processed first. The root and middle nodes are assigned their levels in the tree. The level of a root is $treeHeight$ and for the middle nodes it is $treeHeight$ minus the distance to the root (cf. Fig.4). Arrays $nodeIndices$ and $nodeNeighbours$ are a GPU-friendly representation of a directed tree by a neighbor list. The neighbor lists are stored in the array $nodeNeighbours$. Array $nodeIndices$ stores for a node an index in $nodeNeighbours$ at which the node's neighbor list starts. The last entry in the $nodeIndices$ does not correspond to any node, but stores the length of the $nodeNeighbours$ array. Given node index x, the length of x neighbor list is $nodeIndices[x+1] - nodeIndices[x]$. The root is ignored because in the assumed logistic model root inventory is unlimited and stored at no cost, hence the root delivers the goods immediately, and no cost or quality of service need be calculated for it. Let us note that the CUDA code has been linked with R through the Rcpp mechanism.

5 Evaluation

Performance of the proposed solution has been evaluated in a series of experiments. Unless stated to be otherwise the reference instance had a root, one middle node, three leaves and the following parameter vales: $Nmc = 100, Nd = 64, Nt = 120, Ch = (1,1,1,1), Co = (100,100,100,500), Cs = (1000,1000,1000, 2000)$, `Initial_IOH`=(1E4,1E4,1E4,4E4), Min_pt=(4,5,4,7), Max_pt=(8,7,6,9), μ_i=(1E3,1E3,1E3,1E3],σ_i=(400,300,300,500). Tests have been performed on a PC computer with Intel Core i7 930 CPU with the clock at 2.8GHz, 24GB RAM, NVIDIA GeForce Titan 6GB RAM. The software platform were Arch Linux, NVIDIA CUDA Toolkit 7.5, gcc v4.9.

In the first series of experiments 10 instances of the inventory system have been examined. Beyond the reference instance, nine other instances were constructed by modifying one parameter of the default configuration at a time: Instance 2: Nmc=200, Instance 3: Nd=32, Instance 4: Nd=128, Instance 5: Nt=60, Instance 6: `Nt`=180, Instance 7: $Ch = (10,10,10,10), Co = (100,100,100,100)$, Instance 8: `Initial_IOH`=(9E3,9E3, 9E3,27E3), Instance 9: $Min_pt = (1,1,1,1), Max_pt = (10,10,10,10)$, Instance 10: $\mu_i = (1200,1200,1200,1200), \sigma_i = (500,500,500,500)$. The speedup in the R+GPU execution vs pure R has been evaluated. For the R code 8 execution time samples were collected, for the GPU code 10 samples were collected. Wall-clock times have been used. The results are shown in Fig. 5. Quartiles of speedups for each instance are shown in Fig. 5. The smallest speedup of 768 was obtained for instance 5 and the biggest equal to 10873.4 for instance 4. In terms of run-time it means a reduction from 2134s (average for all runs on all instances in

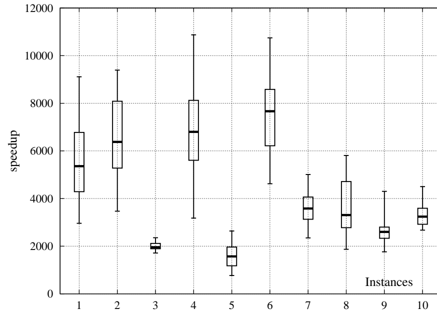

Fig. 5. Speedups in test instances.

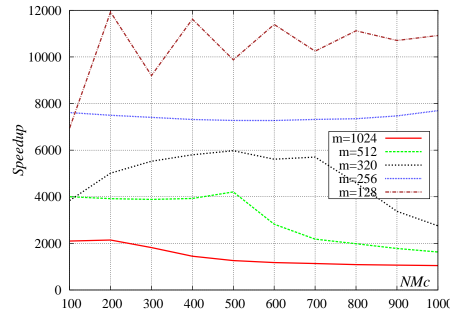

Fig. 6. Speedup vs Nmc, m.

R) to 0.485s (average for all runs on all instances in R+GPU hybrid). Thus, the optimization process time has been successfully reduced.

In the first series of experiments the whole code comprising simulation on GPU and optimization in R has been evaluated. This included $service_level[i]$ and $Cost$ linearization, linear programming, which are essentially sequential. Therefore, in the second series of experiments performance of `blackboxagent1` alone in R and in GPU implementations have been compared. The impact of three main parameters determining complexity of the application: Nmc – the number of MC iterations, Nt – the number of days, m – the number of facilities, and the block size have been tested (at $Nd = 1$). The results are collected in Figs 6–10. In Figs 6, 7 speedups with reference to R implementation are shown for 1024-thread CUDA blocks. Values of speedup presented in Figs 6, 7 should be taken with caution because they do not represent scalability analysis typical of parallel processing literature. Here the reference execution time has been measured for the algorithm implemented on a different software platform, namely, in interpreted language (R). This gives an indication of savings from abandoning R implementation in favor of R+CUDA hybrid. Moreover, more than the actual numbers, the tendencies can be informative.

It can be seen in Figs 6,7 that speedup decreases with the size of the simulation, namely, the number of facilities m and the number of MC iterations Nmc. This can be expected because the more the computational resources are oversubscribed, the higher the overall overheads costs. The impact of thread scheduling is visible in Fig. 6 as a saw-like shape of the speedup curve for $m = 128$ facilities. Yet, such effect is not visible for m=256. This behavior can be explained via a formula linking block size and the execution time, provided in the further text. For bigger number of facilities ($m \geq 320$), speedups decrease at roughly $Nmc > \lfloor 1024/m \rfloor * 256$ and then tend to a new constant value. This effect can be also seen in GPU execution time (not shown here) because for $m = 320$ and $Nmc > 768$, for $m \in \{384, 448, 512\}$ and $Nmc > 512$, for $m > 512$ and $Nmc > 256$ the growth of the execution times accelerates. The threshold of performance drop corresponds with 256 blocks executed in the computational grid. It can be guessed that big numbers of blocks waiting to be executed on a

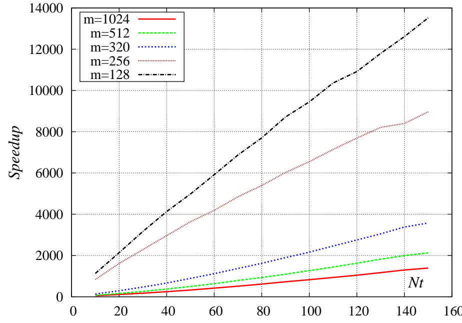

Fig. 7. Speedups vs Nt, m.

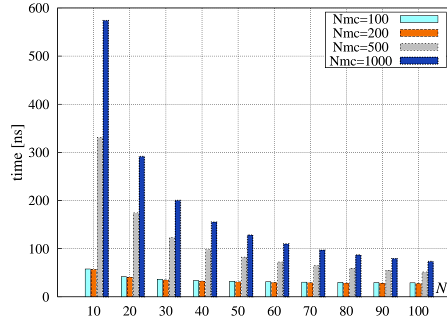

Fig. 8. Time of processing one facility-day on GPU vs Nt, Nmc.

streaming multiprocessor have negative impact on the performance. This drop in performance is exacerbated if the difference between 1024-thread block size and $m\lfloor 1024/m \rfloor$ is big, that is, when blocks have many idle threads.

In Fig.7 it can be seen again that speedup decreases with the size of the computation. The R code implementation checked each day, by iteration over the past days, whether the current day is an arrival day for a replenishment sent on some past day. Consequently, the R code computational complexity was proportional to $(Nt)^2$. In the CUDA code the loop in a loop was substituted with an array of replenishment arrival days, thus reducing complexity to the order of Nt. As a result linear speedup can be observed.

In Fig.8 the average runtime per day and per facility on GPU has been shown. It can be seen that with growing Nt the average time per facility and day decreases. On the one hand, with growing number of the days of simulation Nt fixed overheads are amortized because threads run longer before being dequeued from the streaming multiprocessors. On the other hand, overheads related to the size of simulation grow with Nmc and time of simulating facility day also grows (which confirms the earlier observations).

In Fig.9 impact of the block size on processing time is shown. As it can be verified, the way how simulations of the m facilities are scheduled in the blocks and the blocks on the streaming multiprocessors impacts performance of the computation. The saw-like execution time pattern in Fig.9 can be explained by the formula expressing the number of block executions: $\lceil \lceil Nmc/\lfloor bs/m \rfloor \rceil / (sm * \lfloor tpSM/bs \rfloor) \rceil$, where bs is block size, $tpSM = 2048$ is the number of resident threads per streaming multiprocessor, $sm = 14$ is the number of streaming multiprocessors.

In Fig.10 performance of the simulation in the sense of GFlops is shown. The obtained throughput values are far from the theoretical hardware maximum because our application is not constantly performing multiply-add operations, has complicated memory reference and thread execution patterns resulting from simulating a tree-like logistic network.

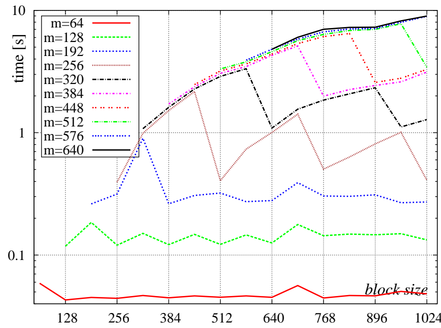

Fig. 9. Processing time vs block size, and m, $Nmc = 1000$.

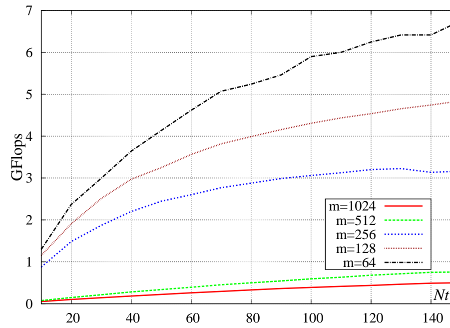

Fig. 10. Speed of processing vs Nt, m.

6 Conclusions

In this paper we reported on parallelization of two-echelon supply chain optimization method initially coded in R. The most time-consuming part of the algorithm has been ported to CUDA platform. The effects obtained demonstrate that a hybrid of R and CUDA combines ease of prototyping, wealth of data analysis tools with the speed of graphics processing units.

References

1. Chu Y, You F, Wassick JM, Agarwal A (2015) Simulation-based optimization framework for multi-echelon inventory systems under uncertainty, Computers and Chemical Engineering 73:1-16. doi: 10.1016/j.compchemeng.2014.10.008
2. Cuda R, Guastaroba G, Speranza MG (2015) A survey on two-echelon routing problems, Computers & Operations Research 55:185-199. doi: 10.1016/j.cor.2014.06.008
3. Groemping U (2016) Fractional Factorial Designs with 2-Level Factors, `https://cran.r-project.org/web/packages/FrF2/FrF2.pdf`
4. Hillier FS, Lieberman GJ (1990) Introduction to Stochastic Models in Operations Research, McGraw-Hill Publishing Company, New York
5. Jain R (1991) The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling. John Wiley and Sons, New York
6. Makhorin A (2012) GLPK (GNU Linear Programming Kit), `http://www.gnu.org/software/glpk/`
7. Martin PJ, Ayuso LF, Torres R, Gavilanes A (2012) Algorithmic strategies for optimizing the parallel reduction primitive in CUDA. In: Smari WW, Zeljkovic V (eds), HPCS, IEEE, pp 511–519. doi: 10.1109/HPCSim.2012.6266966
8. NVIDIA CUDA Programming Guide (2016) `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`
9. Theussl S, Hornik K, Buchta C, Schuchardt H (2016) R/GNU Linear Programming Kit Interface, `https://cran.r-project.org/web/packages/Rglpk/Rglpk.pdf`