JANUS: A Compilation System for Balancing Parallelism and Performance in OpenVX

Hossein Omidian[1,*] **and Guy G. F. Lemieux**[1]

[1]Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, BC, Canada

E-mail: *hosseino@ece.ubc.ca

Abstract. Embedded systems typically do not have enough on-chip memory for entire an image buffer. Programming systems like OpenCV operate on entire image frames at each step, making them use excessive memory bandwidth and power. In contrast, the paradigm used by OpenVX is much more efficient; it uses image tiling, and the compilation system is allowed to analyze and optimize the operation sequence, specified as a compute graph, before doing any pixel processing. In this work, we are building a compilation system for OpenVX that can analyze and optimize the compute graph to take advantage of parallel resources in many-core systems or FPGAs. Using a database of prewritten OpenVX kernels, it automatically adjusts the image tile size as well as using kernel duplication and coalescing to meet a defined area (resource) target, or to meet a specified throughput target. This allows a single compute graph to target implementations with a wide range of performance needs or capabilities, e.g. from handheld to datacenter, that use minimal resources and power to reach the performance target.

1. Introduction

In recent years both industrial and academic communities have focused on implementing Computer Vision (CV) applications on different embedded platforms. For the most computationally intensive CV algorithms, using parallelism techniques for implementing them is considered a must. Although in theory CPU and GPU platforms can perform such intensive computations, power remains the main challenge. Previous studies suggest custom hardware implementations [1, 2] as well as programmable many-core systems [3, 4, 5, 6, 7, 8] can be good alternatives since they are able to perform such intense computations within the power limits. Moreover, most CV applications can be define as streaming applications (i.e. each stage receives stream of image pixels, rows or frames, processes them, and sends the results as stream of data to the next stage). This means we can describe CV applications as compute graphs or Synchronous Data Flow Graphs (SDFGs) [9]. Previous studies have shown custom hardware implementations on FPGA as well as reconfigurable many-core systems have the potential to dramatically increase the performance/Watt for implementing computationally intensive SDFGs. The target embedded systems (for CV applications) can have on-chip resources (e.g. computational, power, area, etc) as large as an autonomous car or as small as a battery operated device. In both cases the final goal for developers is maximizing performance while decreasing the power. Previous studies such as Chen et. al. [10] report the off-chip DRAM traffic dominates the energy consumption and show eliminating the DRAM access reduces energy by up to 150.31x. This means in order to save power, we need to keep data locally on chip as long as

possible and avoid dumping/restoring data to/from main memory. Traditional CV programming systems such as OpenCV [11] operate on entire image frame at each step, which means for every node in the compute graph (e.g. ConvertColor node), the application needs to read the entire frame from off-chip memory, do the required process on the whole frame and then provide the results for the next node by dumping the results back to the off-chip memory. In contrast, OpenVX [12] has been introduced as a cross-platform standard which a computation can be tiled and the workload can be run on those tiles independently while keeping several tiles in on-chip memory concurrently.

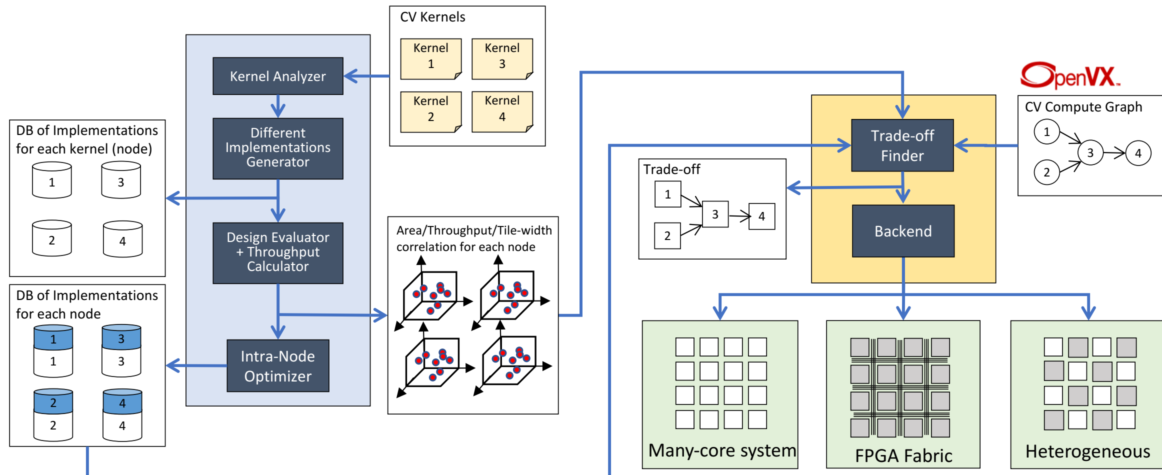

Figure 1. Tool flow

In this work we introduce a compilation system for OpenVX style compute-graphs. Our tool is able to automatically explore the space/time tradeoff problem for these compute graphs and find optimum solutions satisfying different throughput targets and area budgets while targeting a wide range of FPGAs and programmable many-core systems.

Every node (kernel) in the compute graph is analyzed to find all degrees of parallelism based on different loop transformation strategies [13, 14], pipeline opportunities and data dependencies. Based on those, our tool generates different implementations with different area, IO and throughput characteristics for each node in the compute graph. Moreover, using different optimization techniques such as node combining and node replication, our tool is able to widen the problem space by covering more possible solutions. After finding variety of different solutions for each node in the compute graph, the tool needs to find an optimum overall solution based on defined area budget or throughput target. Previous research has shown finding the optimum solution for this space/time tradeoff is an NP-complete optimization problem. It can be defined as Integer Linear Programming (ILP) problem and solved using ILP solver [15].

Although the ILP approach can be useful in some cases, using the ILP optimization model within the tool prohibits the use of certain optimizations. In contrast, using heuristic approaches allows us to perform object coalescing which cannot be done as an ILP formulation. Using heuristic approach leads to area saving and less runtime compared to the ILP approach [16].

There have been several recent studies on implementing image processing and OpenVX applications on reconfigurable platforms and exploring the area/throughput trade-off for them [17, 18, 19, 20]. These existing approaches either use a specific programming model which requires the user to learn a new programming language, or implement a soft multi-core platform on FPGA and then schedule and run the application on it. Automatically exploring area/throughput tradeoffs, automatically finding optimum implementations for different area budgets or throughput targets as well as achieving better results in faster runtime compared to

the existing ILP approaches, make our approach unique. Moreover our approach can be used for both FPGA and many-core platforms as well as heterogeneous platforms such as GRVI Phalanx with accelerators[8].

2. SDFG based HLS for CV applications

Figure 1 illustrates the detailed flow of the proposed tool. Our tool targets two different backends: 1) a Java-based language inspired by Ambric's KPN programming model and many-core architecture [7], and 2) a C-based OpenVX library system built using Xilinx's Vivado HLS tool. Since Synchronous Data Flow (SDF) is a restriction of Kahn process networks (KPN) [21], it is possible to transform any SDFG to a KPN model which means our tool works on SDFG for both of these backends.

2.1. OpenVX Programming Model

OpenVX is a cross-platform C-based API standard for Computer Vision applications. OpenVX can be a great programming model for embedded systems since it enables performance and power-optimized CV processing. It specifies a higher level of abstraction which allows us targeting different computing architectures. Most CV applications can be described as a set of vision kernels (nodes) which communicate through input/output data dependencies (e.g. AXI stream). OpenVX describes this set of vision kernels in a graph-oriented execution model (Compute Graph) based on Directed Acyclic Graphs (DAGs). Figure 2 shows an OpenVX code example (Canny) and Figure 3 shows the corresponding graph for it.

```
//Canny example
vx_node nodes[] = {
   vxColorConvertNode(graph,rgb, gray),
   vxGaussian3x3Node(graph,gray,gauss),
   vxSobel3x3Node(graph,gauss,gradx,grady
   vxMagnitudeNode(graph,gradx,grady,mag)
   vxPhaseNode(graph,gradx,grady,phase),
   vxNonMaxima(graph,mag,phase,nm),
   vxThreshold(grpah,nm,output)
};
```

Figure 2. OpenVX source code.

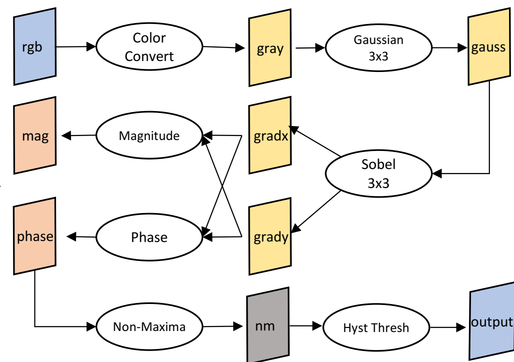

Figure 3. Canny edge detection graph representation.

2.2. Finding Different Implementations

Consider an application described as a compute graph with N nodes $f_1, f_2, ..., f_N$. For each node f_m our tool tries to find different implementations $P_m^1, P_m^2, ..., P_m^{S_m}$ where each implementation P_m^s can perform functionality of f_m with area cost $A(P_m^s)$, number of pixels it can consume/produce $NP(P_m^s)$ each firing, initial interval $II(P_m^s)$ and "inverse throughput" $\vartheta_{in/out}(P_m^s)$ for each input/output channel [16]. For most CV kernels, their input/output channels have matched throughput, so we can use a single variable $\vartheta_{IO}(P_m^s)$ which allows us to define "kernel throughput" $\Theta(P_m^s)$ as number of pixels consumed/produced in each clock cycle [2]. The initiation interval is the number of time units before another firing of the inputs is permitted; it can be just 1 for fully pipelined execution or > 1 for partially pipelined execution.

The Different Implementation Generator (DIG) module in our tool automatically finds the above mentioned implementations. The DIG needs to be able to automatically find a wide

range of different implementations with different area cost, throughput and tile-width to cover the solution space as much as possible. DIG uses different loop transformations, data dependency analysis, pipeline opportunities as well as changing the image tile-size (in both kernel IO and kernel core). In addition, an Intra-node Optimizer step in the tool generates a wider range of implementations. Intra-node Optimizer replicates and combines existing implementations of nodes in order to widen the solution space. Node replication can be used to either increase the throughput or widen the tile-width [2]. All of the abovementioned techniques are used find a wide range of implementations for each kernel which, widens the solution space for the area/throughput scaling problem. Below we discuss our trade-off formulation and solutions.

3. Trade-off Finding Problem Definition and Solutions

The trade-off finding problem can be defined in two different ways.

- Given a throughput target Θ_{tgt}, and different implementations for each node f_j, which implementation P_j^i should be selected and how many replicas nr_j^i are needed in order to minimize area cost A_A subject to the constraint the application throughput Θ_A is bigger than Θ_{tgt}.
- Given an available area on chip A_C and different implementations for each node f_j, which implementation P_j^i should be selected and how many replicas nr_j^i are needed in order to maximize application throughput Θ_A subject to the constraint the application area cost A_A is not bigger than A_C.

For our baseline, we modeled the optimization problem as ILP formulations similar to previous studies such as Cong et.al. [15] and used the GLPK open-source ILP solver [22]. Although ILP solvers are able to find an optimum solution for the defined problem, we need to define the problem beforehand which prohibits the use of certain optimization such as graph manipulation (node combining/splitting) during runtime. Moreover, as ILP finds the optimum solution by going through all different possible solutions, finding an optimum solution for a problem with a large search space is quickly becomes highly time inefficient. Our proposed heuristic approach addresses both of these shortcomings.

3.1. Using Heuristic Approach for Trade-off Finding Problem

Trade-off finding problem can be solved using heuristic approaches such as Omidian et.al. [16]. Our heuristic approach uses there following steps to find a good tradeoff:

- Throughput Analysis
- Application Throughput Propagation and Balancing
- Bottleneck Optimizer

In contrast to the ILP approach, a heuristic approach enables us to use additional optimization opportunities such as Inter-node optimization. Using heuristic approach also can improve the runtime for trade-off finding problem.

4. Experimental Results

Our experiments are carried out in two parts. We evaluate our heuristic approach for both different throughput target and area budgets (different FPGA architectures as well as different throughput targets for many-core system) and compare it to the ILP approach. Then we evaluate the throughput results by implementing CV benchmarks on Zedboard development platform with a Xilinx Zynq SOC. To show the capabilities of our approach, we have utilized the following benchmarks implemented as OpenVX compute graph:

- *Sobel* is a Sobel-filter based edge detection with 5 nodes.

- *Canny* implements Canny edge detector with 7 nodes.
- *Harris* implements Harris corner detector with 6 nodes.
- *JPEG* implements JPEG encoding with 4 nodes.

Table 1. Implementation Library for JPEG encoder.

module	Color Conversion				DCT					Quantization					Encoding
Version	v1	v2	v3	v4	v1	v2	v3	v4	v5	v1	v2	v3	v4	v5	v1
Inverse Throughput	1	2	4	8	1	2	4	6	32	1	2	4	8	128	512
Area	512	256	128	64	800	400	224	160	50	512	256	128	64	4	22

Table 2. Heuristic vs ILP for many-core system.

Method	Inverse Throughput	Color Conversion		DCT		Quantization		Encoding		Fork/join Overhead	Total Area
		impl	rep	impl	rep	impl	rep	impl	rep		
ILP	1	v1	1	v1	1	v1	1	4	512	10880	23968
Heuristic		v1	1	v5	32	v5	128	224	512	640	13888
ILP	2	v2	1	v2	1	v2	1	v1	128	5376	11920
Heuristic		v2	1	v5	16	v5	64	v1	128	256	7456
ILP	4	v3	1	v3	1	v3	1	v1	64	2688	5984
Heuristic		v3	1	v5	8	v5	32	v1	64	128	3600
ILP	8	v4	1	v4	1	v4	1	v1	32	1280	2976
Heuristic		v4	1	v5	4	v5	16	v1	32	0	1736

We evaluated our tool by implementing JPEG on the many-core system targeting different throughput targets. Our tool is able to find 11 different implementations for Color Conversion and Quantization modules, 17 different implementations for DCT, and only one implementation for Encoding. Table 1 shows a selection of these implementations. Both ILP and Heuristic approaches have been used by our tool in order to find a trade off between area and throughput for different inverse throughput targets for JPEG. Table 2 shows the results generated by these two approaches for given throughput targets. We list the selected implementation and number of replicas for each module. As we can see our heuristic approach finds better area/throughput trade-off compare of the ILP approach. For example, for an inverse throughput target of 2, our heuristic approach used 37% less area compare to ILP.

We also evaluated our tool with different Xilinx 7-series FPGAs. Both heuristic and ILP approaches could fill over 95% of the chip area on average. The heuristic approach however is able to achieve better throughput while improving the runtime up to 3.6x compared to the ILP approach. Figure 4 shows the runtime result improvement achieved through using our heuristic approach compared to the ILP for different Xilinx FPGAs. In addition, in order to examine the area saving achieved through using heuristic approach (using inter-node optimization), we first used the ILP approach to fill the chip and achieved the best throughput possible for different Xilinx FPGAs, then we used the achieved throughput on each FPGA as a throughput target for the heuristic approach. Figure 5 shows the area cost comparison between the heuristic and the ILP. The heuristic approach saves 19% area on average while decreasing the throughput only by less than 2%. Figure 6 shows the percentage of throughput reduction for heuristic compared to ILP and Figure 7 shows the results for setting different throughput targets and implement the tradeoff solutions on Zedboard development platform. As it's shown our tool is able to hit all the throughput targets while increasing the area cost linearly. The tool is able to achieve up to 5.5GigaPixel/sec throughput for implementing Sobel on a small size FPGA.

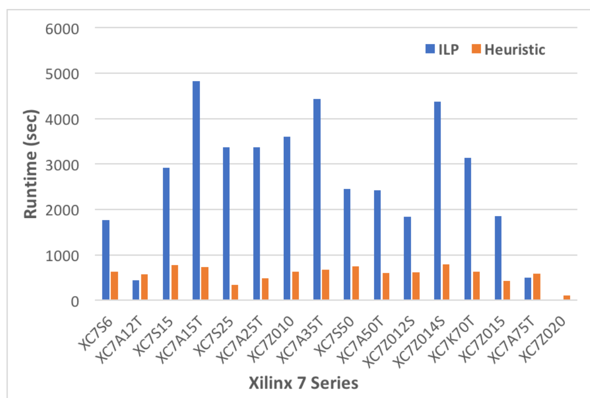

Figure 4. Heuristic vs ILP runtime speedup for Harris corner detection.

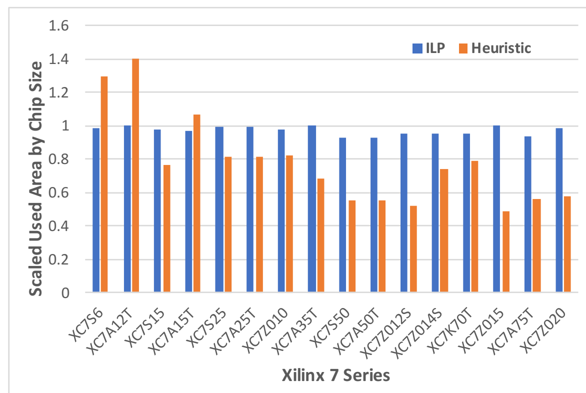

Figure 5. Area cost results for Harris using Heuristic and ILP approaches.

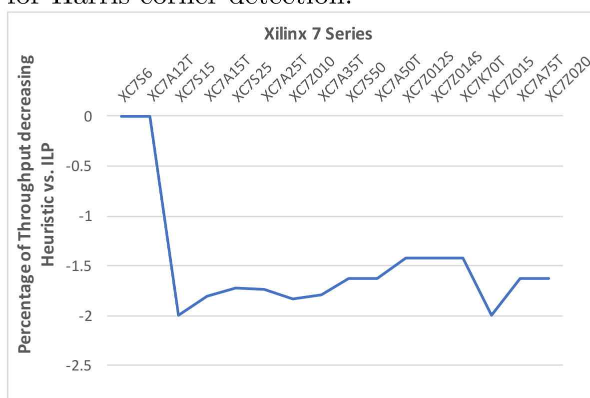

Figure 6. Heuristic vs ILP throughput results for Harris corner detection.

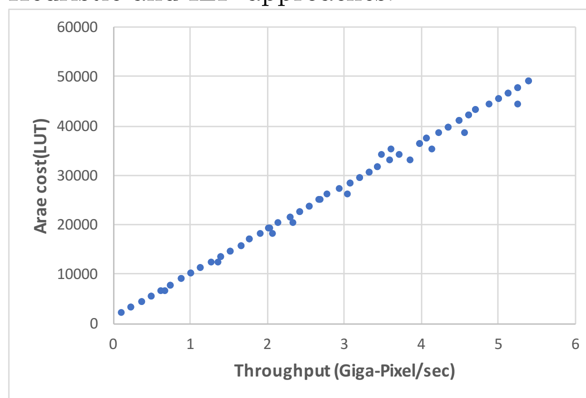

Figure 7. Area/Throughput results for implementing Sobel on Xilinx Zedboard.

5. Conclusion

In this work we introduce a compilation system for OpenVX style compute-graphs. Our tool is able to automatically explore the space/time tradeoff problem for these compute graphs and find optimum solutions satisfying different throughput targets and area budgets while targeting a wide range of FPGAs and programmable many-core systems. Our approach is differentiated from existing approaches as it automatically investigates finding different implementations, and it also combines module selection and replication methods as well as changing tile-size with node combining and splitting in order to automatically find a better area/throughput tradeoff. This approach was verified with different OpenVX benchmarks targeting different FPGA sizes. Our tool is able to achieve over 95% of the target area budget while improving the throughput. Using Inter-node Optimizer step, our heuristic tradeoff finder is able to hit the same throughput targets while reducing the area cost by 19% on average compared to existing ILP approaches. Hitting different throughput targets as well as getting up to 5.5 GigaPixel/sec for a small FPGA target shows our tool is efficient in managing on chip resources efficiently in order to hit different throughput targets.

References

[1] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010, pp. 37–47.

[2] H. Omidian and G. G. F. Lemieux, "Exploring automated space/time tradeoffs for openvx compute graphs," in *2017 International Conference on Field-Programmable Technology (FPT)*, Dec 2017.
[3] A. Severance, J. Edwards, H. Omidian, and G. Lemieux, "Soft vector processors with streaming pipelines," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, ser. FPGA '14, 2014, pp. 117–126.
[4] Adapteva-Inc, "Epiphany-iv 64-core 28nm microprocessor," 2014. [Online]. Available: http://www.adapteva.com/products/silicon-devices/e64g401/
[5] L. Benini, E. Flamand, D. Fuin, and D. Melpignano, "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator," in *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2012, pp. 983–987.
[6] KALRAY-Corporation, "Massively parallel processor array." [Online]. Available: http://www.kalray.eu/
[7] M. Butts, A. M. Jones, and P. Wasson, "A structural object programming model, architecture, chip and tools for reconfigurable computing," in *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*, 2007, pp. 55–64.
[8] J. Gray, "Grvi phalanx: A massively parallel risc-v fpga accelerator accelerator," *International Workshop on Overlay Architectures for FPGA (OLAF)*, vol. abs/1606.03717, 2016.
[9] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, pp. 1235–1245, 1987.
[10] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 609–622.
[11] *The OpenCV Reference Manual*, 2nd ed., Itseez, April 2014.
[12] Khronos-Group, "Openvx," 2017. [Online]. Available: https://www.khronos.org/openvx/
[13] A. W. Lim and M. S. Lam, "Maximizing parallelism and minimizing synchronization with affine transforms," in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '97, 1997, pp. 201–214.
[14] V. Sarkar and R. Thekkath, "A general framework for iteration-reordering loop transformations," *SIGPLAN Not.*, pp. 175–187, 1992.
[15] J. Cong, M. Huang, B. Liu, P. Zhang, and Y. Zou, "Combining module selection and replication for throughput-driven streaming programs," in *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2012, pp. 1018–1023.
[16] H. Omidian and G. G. F. Lemieux, "Automated space/time scaling of streaming task graph," *International Workshop on Overlay Architectures for FPGA (OLAF)*, vol. abs/1606.03717, 2016.
[17] G. Tagliavini, G. Haugou, A. Marongiu, and L. Benini, "Adrenaline: An openvx environment to optimize embedded vision applications on many-core accelerators," in *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, 2015, pp. 289–296.
[18] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan, "Darkroom: Compiling high-level image processing code into hardware pipelines," *ACM Trans. Graph.*, pp. 144:1–144:11, 2014.
[19] J. Hegarty, R. Daly, Z. DeVito, J. Ragan-Kelley, M. Horowitz, and P. Hanrahan, "Rigel: Flexible multi-rate image processing hardware," *ACM Trans. Graph.*, pp. 85:1–85:11, 2016.
[20] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz, "Programming heterogeneous systems from an image processing dsl," *ACM Trans. Archit. Code Optim.*, pp. 26:1–26:25, 2017. [Online]. Available: http://doi.acm.org/10.1145/3107953
[21] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," in *Information Processing '74: Proceedings of the IFIP Congress*, 1974, pp. 471–475.
[22] GNU-project, "Gnu linear programming kit," 2017. [Online]. Available: https://www.gnu.org/software/glpk/