Problem C: GPU Accelerated Logic Rewriting

Ghasem Pasandi, Sreedhar Pratty, David Brown

NVIDIA Corp., Santa Clara, CA

Introduction

Logic synthesis is an important step in design and implementation flow of digital chips with a big impact on final Quality of Results (QoR). A given netlist to a logic synthesis tool goes through multiple optimization steps, technology-independent, before it gets mapped into a network of logic gates. *Logic rewriting* is a greedy technique to optimize a given circuit in the technology-independent phase of the logic synthesis. Logic rewriting optimization function can operate on different graph types such as And-Inverter Graphs (AIGs), And-Or-Inverter Graphs (AOIGs), and Majority-Inverter Graphs (MIGs). In the logic rewriting, the given graph is traversed usually in a topological order and Boolean function of a node based on inputs of its k-feasible cuts [1] is rewritten and is replaced with the original one. This transforms the original graph into a new one and results in increasing or decreasing (desired) its node size or keeping it unchanged. Due to the local scope and greedy nature of the rewriting function, multiple rounds of rewriting operation usually led to a further decrease in the total node count of the graph. For more information on rewriting, please see the following papers [2] and [3]. The main goal of this contest problem is to implement rewriting optimization function using CUDA in order to increase its speed through GPU acceleration.

The GPU accelerated computing platform has also advanced significantly over the last 5-10 years, enabling advancements in deep learning by fueling its tremendous demand for computing power. We will make NVIDIA T4 GPUs [4] available to contestants via cloud GPU instances for development and benchmarking. These GPUs support 8.1 TFLOPS of peak single-precision floating-point performance with 16 GB of GDDR6 memory and 300 GB/s of memory bandwidth. T4 tensor cores support a peak of 130 TOPS of 8-bit integer performance and 260 TOPS of 4-bit integer performance, which contestants may be able to leverage for additional speedups. We also plan to support CUDA 10.0+, including newer CUDA features [5] such as unified memory, cooperative groups, independent thread scheduling, CUDA graphs, and more to help enable more flexible and efficient parallelization of existing code base. Finally, recent progress in deep learning frameworks such as PyTorch can enable programming GPUs in Python and provide GPU-optimized library functions useful to EDA. For example, DREAMPLace [6] achieved a 40x speedup on VLSI global placement leveraging several optimization functions available in PyTorch.

Standard Logic Rewriting

In this section, we briefly explain the standard logic rewriting operation. For more details please see [2] and [3]. Before start of the rewriting process, 4-feasible cuts (cuts with up to 4 inputs) are computed for each node. Then, in a topological ordering traversal, the AIG form of the given circuit is traversed and for each node, different re-written versions of the function of the node based on a cut input are tried. There are around 65,000 4-input functions that are classified into 222 NPN classes. NPN stands for negation and permutation of inputs and negation of output. For example, !a+b*c, a*(!b+!c), a+!b*c, and !b+a*c are all in the same NPN classes. This is because the second Boolean expression is achieved by negating output of the first one; the third one is achieved by negating two inputs of the first expression, and finally the last one is obtained by permuting inputs. These functions are stored in a hash-table and are used for logic rewriting during the above-mentioned graph traversal. The logic rewriting in its default setting preserves the logic level and tries to reduce the node count of a sub-graph corresponding to the cut under analysis. At the end, when all nodes are visited and a rewriting attempt is done on them, a full episode of a rewriting operation for the whole AIG finishes.

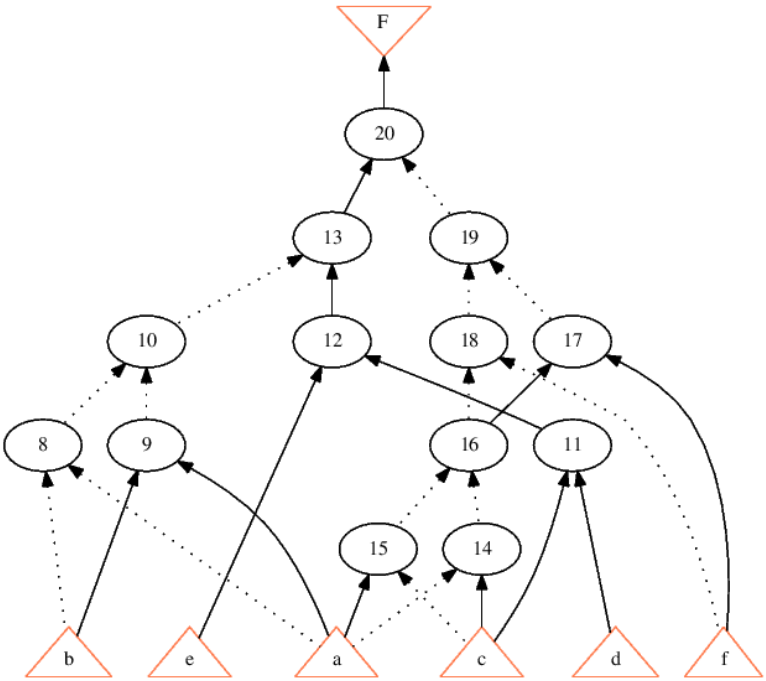

Fig. 1: An example AIG with 13 nodes for the following 6-input 1-output Boolean expression: F = (!(!(!a*!b)*!(a*b))*(e*(c*d)))*!(!(!(!(a*!c)*!(!a*c))*!f)*!((!(a*!c)*!(!a*c))*f)), where ! means NOT or inversion and * means AND.

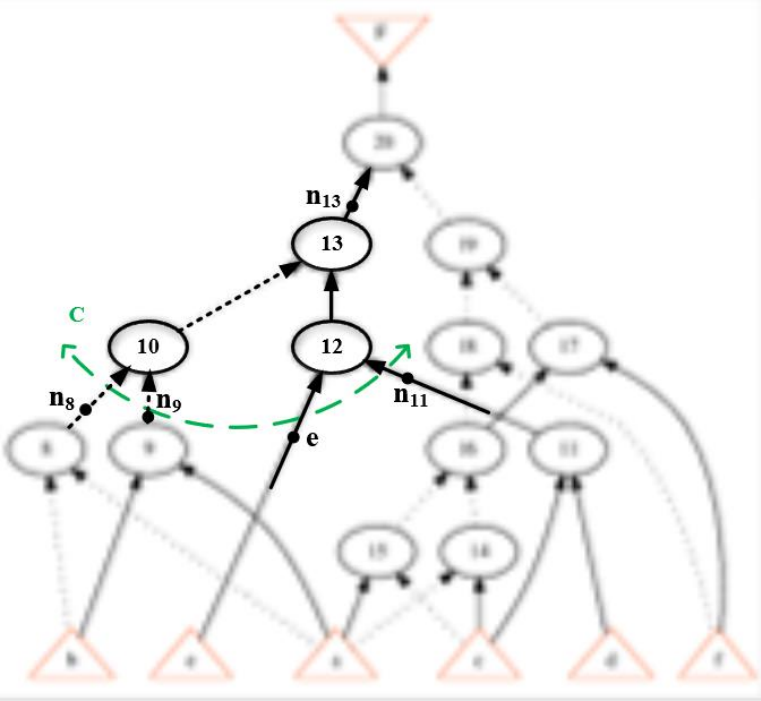

Fig. 2: A 4-input cut (4-cut) of node 13 (n_{13}) and its current Boolean function based on inputs of this cut: $n_{13} = !(!n_8 * !n_9) * (e * n_{11})$. The corresponding sub-graph of this cut is also shown.

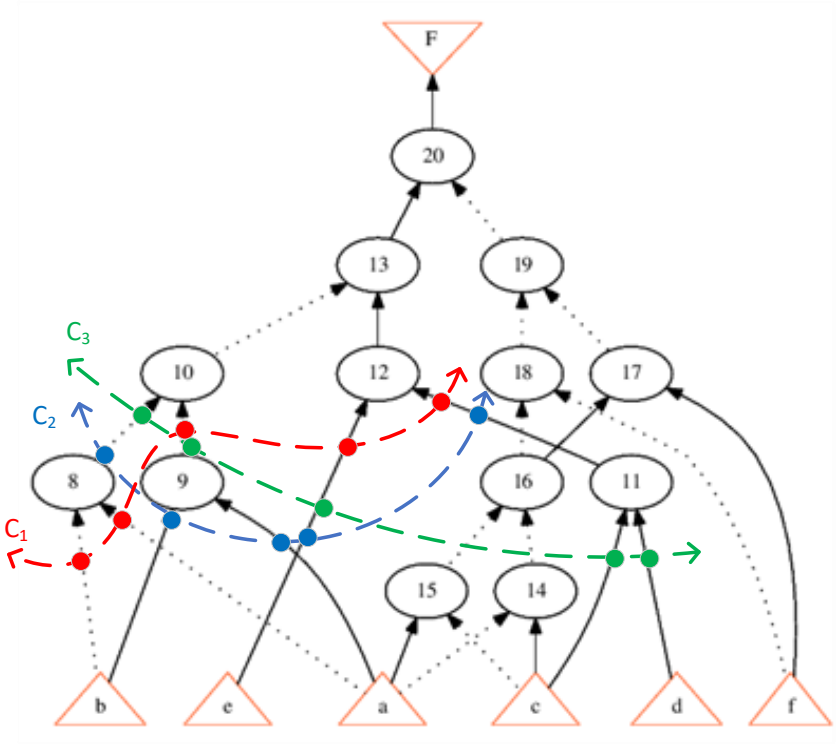

Fig. 3: Showing three 5-cuts of node 13. Inputs of each cut are shown by small circles; for example, inputs of cut C_3 are: c, d, e, node 9, and node 8.

In the following, we explain what is meant by a full episode of rewrite. Fig. 1 shows the AIG from of an arbitrary circuit with six primary inputs (a to f) and one primary output (F). This AIG has 13 nodes that are labeled from 8 to 20. In an AIG, nodes are 2-input AND gates, and dashed lines indicate inverted edges (functionally, there is an inverter on a dashed edge). A full episode of rewrite consists of (i) computing 4-feasible cuts for all nodes, (ii) visiting each node and trying different re-written versions for the function of a node based on inputs of its cuts. Fig. 2 shows the moment that node 13 is being visited. A 4-input cut of this node is shown by a green color in this figure. The function of node 13 based on inputs of this cut is: $!(!n_8*!n_9)*(e*n_{11})$ where ! indicates NOT or inversion and * is an AND operation. The corresponding sub-graph for this cut is clearly shown in Fig. 2, which consists of nodes 10, 12, and 13.

The rewrite command of ABC uses a precomputed database of 4-input functions for the rewriting purposes. In other words, once the functionality of a node based on a 4-cut is computed (as in Fig. 2 for node 13), this Boolean function is looked up in the database and different rewritten versions of this function, which are available in the database, are tried. An AIG sub-graph for each re-written version of the Boolean function is generated inside the rewrite function. If the node count of this new sub-graph is fewer than the original one (or if it is the same in case of using rewrite -z), the rewrite operation is accepted, and the sub-graph is substituted. To have a better idea of cuts with more than 4 inputs, three 5-input cuts of node 13 are shown in Fig. 3.

CUDA Programming

CUDA is a parallel computing platform and programming model developed by NVIDIA for general computing on Graphics Processing Units (GPUs). CUDA enables developers to speed up compute-intensive applications by harnessing the power of GPUs for the parallelizable part of the computation. There are some differences between CUDA and regular C/C++ coding including function definition and declaration, compiling, and memory management. Fig. 3 shows a Hello World code for both C and CUDA.

C

```c
void c_hello(){
    printf("Hello World!\n");
}

int main() {
    c_hello();
    return 0;
}
```

CUDA

```cuda
__global__ void cuda_hello(){
    printf("Hello World from GPU!\n");
}

int main() {
    cuda_hello<<<1,1>>>();
    return 0;
}
```

Fig. 4: Hello world in C and CUDA.

As seen, the main differences are existence of __global__ specifier and <<<…>>> syntax in CUDA implementation. __global__ indicates that the cuda_hello function will run on a GPU and the <<<1,1>>> syntax specifies execution configuration of the GPU.

To compile the above program, a CUDA compiler called nvcc in a CUDA toolkit [8] should be used. To learn more about the CUDA programming and to see more examples, please look at the websites in the following references: [9-11]. To practice more, you can find nine great homework with solutions in the following link: https://github.com/olcf/cuda-training-series/tree/master/exercises.

Problem Description

The standard rewriting function operating on AIGs is available as part of an open-source logic synthesis and verification tool called ABC [7]. You can clone ABC using the following link: https://github.com/berkeley-abc/abc. The name of the rewriting command in ABC is *rewrite* or *rw*. Fig. 5 shows options of rw command and how it is applied to the i10 benchmark circuit. The source code for the rewrite command is located at src/base/abci/abc.c, and the name of the function is Abc_CommandRewrite().

Fig. 6 shows a snippet of this code. Inside Abc_CommandRewrite(), another function is called which is mainly responsible for the rewriting operation: Abc_NtkRewrite(). Source code of the latter function is located at src/base/abci/abcRewrite.c.

Similarly, you can track all the functions used in the rewrite command of ABC and find their source codes. You may also check the following files:

src/base/abci/abc.c

src/base/abc/abc.h

src/opt/rwr/rwr.h

src/opt/rwr/rwrEva.c

```
(base) bash-4.2$ ./abc
UC Berkeley, ABC 1.01 (compiled Jan 22 2021 14:47:51)
abc 01> read i10.aig
abc 02> print_stats
i10                        : i/o =  257/  224  lat =    0  and =   2675  lev = 50
abc 02> rw
abc 02> print_stats
i10                        : i/o =  257/  224  lat =    0  and =   2096  lev = 48
abc 02>
```

Fig. 5: Reading i10.aig circuit in ABC environment and applying standard rw command to it. As seen, after applying rw, number of *and* nodes is decreased from 2675 to 2096.

```
***********************************************************************/
int Abc_CommandRewrite( Abc_Frame_t * pAbc, int argc, char ** argv )
{
    Abc_Ntk_t * pNtk = Abc_FrameReadNtk(pAbc);
    int c;
    int fUpdateLevel;
    int fPrecompute;
    int fUseZeros;
    int fVerbose;
    int fVeryVerbose;
    int fPlaceEnable;
    // external functions
    extern void Rwr_Precompute();

    // set defaults
    fUpdateLevel = 1;
    fPrecompute  = 0;
    fUseZeros    = 0;
    fVerbose     = 0;
    fVeryVerbose = 0;
    fPlaceEnable = 0;
    Extra_UtilGetoptReset();
    while ( ( c = Extra_UtilGetopt( argc, argv, "lxzvwh" ) ) != EOF )
    {
        switch ( c )
        {
        case 'l':
            fUpdateLevel ^= 1;
            break;
        case 'x':
            fPrecompute ^= 1;
            break;
        case 'z':
            fUseZeros ^= 1;
            break;
        case 'v':
            fVerbose ^= 1;
            break;
        case 'w':
            fVeryVerbose ^= 1;
            break;
        case 'p':
            fPlaceEnable ^= 1;
            break;
        case 'h':
            goto usage;
        default:
            goto usage;
        }
    }
```

Fig. 6: A code snippet for the high-level function of rewrite command in ABC.

The rewrite command of ABC only operates on AIGs, meaning that if the input to this optimization function is not an AIG (e.g., i10.v instead of i10.aig), it will not work and will return an error message. To convert a circuit to an AIG format, you should use *strash* or *st* command inside an ABC environment.

The current implementation of rewrite command in ABC works only on CPUs. In this contest problem, we are asking you to implement the rewrite command in CUDA. Therefore, you should locate all functions that are used in rewrite command of ABC and implement as much of them as possible using CUDA. Implementation of the rewrite command in CUDA will make it possible to parallelize the rewriting operation and to run it on GPUs, resulting in an increase in the speed of the whole rewriting process. At the end of your CUDA-based rewrite command, the rewritten AIG should be written into a file using a command like the following: *write_aiger output_circuit.aig*

Benchmark Suite

The benchmark circuits will be chosen from the following suites: ISCAS, EPFL, IWLS. We will provide a Google drive folder for accessing the benchmarks. We encourage contestants to test their codes using a combination of several different test circuits to exercise their codes better. The purpose of sourcing from several different benchmark suites is to provide a range of small to large designs that cover a wide range of circuit structures for evaluating the parallelism and bigger cut sizes in the new rewriting function. We suggest contestants to use smaller *designs* for code development, exploration, and validation, while larger *designs* should be used for code evaluation. We will use a combination of open-source benchmark circuits as well as some (hidden) internal industrial level benchmarks for evaluation purposes. The purpose of using unknown benchmark circuits for contest's final evaluation and scoring is to encourage contestants to design a 'universal' rewriting module that can handle all different kinds of circuits and provide speed improvements for an unseen circuit, instead of designing their code to perform well only on the given benchmarks.

Evaluation

The speed and GPU utilization are both important in this contest problem. In fact, one of the reasons we encourage implementation of the rewriting algorithm using CUDA is to enable massive number of rewrite operations to be done for the same or better run-time compared to the baseline. We use a scoring function that considers both the run-time and the GPU utilization:

$$\text{Each benchmark score} = \alpha\left(\frac{t_{baseline}}{t_{yours}}\right) \times \beta\left(\text{Util}_{GPU}\right) \quad (1)$$

$$\text{Final score} = \sqrt[p]{\prod_i^{p=number\ of\ test\ benchmarks} score_i} \quad (2)$$

where $t_{baseline}$ is the run-time of a rewrite operation of ABC plus the time elapsed for reading the input circuit and writing it into a file. For example, for i10.aig input circuit, we time the following operations: "read i10.aig; rw; write_aiger i10_rw.aig;" This will give $t_{baseline}$.

t_{yours} is the run-time of your implementation including both CPU and GPU run times. We encourage you to run most of your code on GPUs for further speedup purposes; we will check the GPU utilization and use a normalized value of that (Util_{GPU}) in our scoring function. Therefore, if you use GPU more, chances are that you will get better scores. α and β are two functions used for normalization. For now, you can assume α(x)=x and β(x)=$x^{1.5}$. Finally, note that the evaluation will be based on average of multiple runs to take the stochastic nature of run-time values into account.

To make the scoring strategy crystal clear, we bring an example here. Suppose that there are three imaginary teams, and their codes are being evaluated on two benchmark circuits. Table 1 shows break down of scores.

Table 1: Score details for the scenario of having three teams and using two benchmark circuits.

team	Circuit 1 $t_{baseline}$ = 1.1			Circuit 2 $t_{baseline}$ =18.6			Final score
	t_{yours}	$Util_{GPU}$	score	t_{yours}	$Util_{GPU}$	score	
A	0.91	0.38	0.2832	17.75	0.74	0.6671	0.4347
B	0.99	0.41	0.2917	17.01	0.77	0.7388	0.4642
C	2.35	0.66	0.2510	15.18	0.81	0.8932	0.4734

In this table, going from team A to team C, more portion of the rewrite command is being implemented in CUDA. Having most of the rewrite command being implemented in CUDA may result in an increase in the total run-time for small circuits (as in the case of circuit 1 for team C) compared with the baseline; this is because the circuit is small and the overheads of operations like copying things between host and devices may exceed the saving that it offers through parallel execution on GPUs. However, this will be paid off on larger circuits (e.g., circuit 2). That is why, team C is getting a better final score than the other two teams.

The required code submission format is detailed in Fig. 7. You should compile all your code and any other tool that you use such as ABC into a single executable binary file. It should take as input one command line argument: input circuit in aig format. Your executable binary should be able to generate a ".aig" file for the re-written circuit upon execution of your code.

```
GPUrewrite.exe <input_circuit.aig>  [output_circuit.aig]
#Actual execution of your rewriting code. Execution of this code will be timed. The code
should receive an input AIG and dump the re-written AIG into a file. The name of the
output file should be the same as input plus "_rw".

#Example: input: GPUrewrite.exe i10.aig  output: i10_rw.aig
```

Fig. 7: Desired code submission format. To generate i10_rw.aig, your code should use this command at the end of the rewriting process: *write_aiger i10_rw.aig*

Regardless of the coding languages that are used in your implementation, submission of the code should be done in a binary executable format which take the above-described variables as command line arguments.

To profile your program and find GPU utilization, *nvprof* command should be used. For example, for the vector addition in [11] (assuming that it is named *vector_addGG*), you should use the following command: *nvprof --trace gpu ./vector_addG*. This will output something like below:

```
ubuntu@ip-172-31-12-83:~/scratch$ nvprof --trace gpu ./vector_addGG
==2560== NVPROF is profiling process 2560, command: ./vector_addGG
out[0] = 3.000000
PASSED
==2560== Profiling application: ./vector_addGG
==2560== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   94.21%  729.25ms         1  729.25ms  729.25ms  729.25ms  vector_add(float*, float*, float*, int)
                    3.65%  28.229ms         1  28.229ms  28.229ms  28.229ms  [CUDA memcpy DtoH]
                    2.14%  16.578ms         2  8.2892ms  8.2828ms  8.2956ms  [CUDA memcpy HtoD]
No API activities were profiled.
```

Fig. 8: Profiling a vector addition code.

To have CPU profiling as well, you may turn the cpu profiling switch on as below:

```
ubuntu@ip-172-31-12-83:~/scratch$ nvprof --cpu-profiling on --cpu-profiling-mode flat --trace gpu ./vector_addGG
==2611== NVPROF is profiling process 2611, command: ./vector_addGG
out[0] = 3.000000
PASSED
==2611== Profiling application: ./vector_addGG
==2611== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   93.95%  703.83ms         1  703.83ms  703.83ms  703.83ms  vector_add(float*, float*, float*, int)
                    3.82%  28.624ms         1  28.624ms  28.624ms  28.624ms  [CUDA memcpy DtoH]
                    2.23%  16.705ms         2  8.3525ms  8.2705ms  8.4344ms  [CUDA memcpy HtoD]
No API activities were profiled.

======== CPU profiling result (flat):
Time(%)      Time  Name
 66.19%  2.34125s  ???
 21.02%  743.57ms  cuMemcpyDtoH_v2
  5.40%  190.92ms  cuDevicePrimaryCtxRetain
  3.98%  140.68ms  cuInit
  2.84%  100.48ms  ???
  0.28%  10.048ms  cuMemcpyHtoD_v2
  0.28%  10.048ms  munmap

======== Data collected at 100Hz frequency
```

Fig. 9: Including CPU profile in execution of the vector addition code.

Warning!: as you can see above, *nvprof* has the luxury of listing execution time and percentage for different kernels and also memory operations. Therefore, if you use some dummy codes to just increase your GPU utilization, it will be evident. In such a case, you will be disqualified from this contest.

Conclusions

Logic rewriting is an effective optimization heuristic that is used to minimize the AIG from of digital circuits during technology-independent phase of the logic synthesis. Due to its local scope, it is beneficial to perform multiple parallel rewrites to further reduce the total AIG node count. Also, even though logic rewriting is fast on small and mid-size circuits, it can be very slow on industrial level circuits with millions of nodes. Implementing the standard rewriting function in CUDA and running it on GPU can help solving the said problems by enabling execution of many rewrite operations in parallel. This will result in increasing speed and improving QoR.

References

[1] J. Cong and Y. Ding, "Flowmap: An optimal technology mapping algorithm for delay optimization in lookup-table based fpga designs", *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 13, no. 1, pp. 1-12, Jan. 1994.

[2] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting a fresh look at combinational logic synthesis" In Proceedings of the 43rd annual Design Automation Conference, pages 532--535. ACM, 2006

[3] G. Pasandi and M. Pedram, "Balanced factorization and rewriting algorithms for synthesizing single flux quantum logic circuits". In Proceedings of the Great Lakes Symposium on VLSI. ACM, 183—188

[4] https://www.nvidia.com/en-us/data-center/tesla-t4/

[5] Stephen Jones, "CUDA New Features and Beyond", GTC, 2019

[6] Yibo Lin et al., "DREAMPlace: Deep Learning Toolkit Enabled GPU Acceleration for Modern VLSI Placement", DAC, 2019

[7] A. Mishchenko et. al, "ABC: A system for sequential synthesis and verification" Berkeley Logic Synthesis and Verification Group, 2018. Available online: https://github.com/berkeley-abc/abc.git

[8] NVIDIA, "CUDA Toolkit: Develop, Optimize and Deploy GPU-Accelerated Apps", available online: https://developer.nvidia.com/cuda-toolkit

[9] GPU Hackathons, available online: https://www.gpuhackathons.org/technical-resources

[10] M. Harris, "An even easier introduction to CUDA", available online: https://developer.nvidia.com/blog/even-easier-introduction-cuda/

[11] P. Sakdnagool, "CUDA Tutorial", available online: https://cuda-tutorial.readthedocs.io/en/latest/