Adaptation of algorithms for efficient execution on GPUs

Bulavintsev, Vadim G.; Zhdanov, Dmitry D.

DOI
10.1117/12.2601619

Publication date
2021

Document Version
Accepted author manuscript

Published in
Optical Design and Testing XI

Adaptation of Algorithms for efficient execution on GPUs

Vadim G. Bulavintsev[a] and Dmitry D. Zhdanov[b]

[a]Delft University of Technology, Delft, The Netherlands
[b]ITMO University, St. Petersburg, Russia

ABSTRACT

We propose a generalized method for adapting and optimizing algorithms for efficient execution on modern graphics processing units (GPU). The method consists of several steps. First, build a control flow graph (CFG) of the algorithm. Next, transform the CFG into a tree of loops and merge non-parallelizable loops into parallelizable ones. Finally, map the resulting loops tree to the tree of GPU computational units, unrolling the algorithm's loops as necessary for the match. The method provides a convenient and robust mental framework and strategy for GPU code optimization. We demonstrate the method by adapting a backtracking search algorithm to the GPU platform and building an optimized implementation of the ResNeXt-50 neural network.

Keywords: GPU, SIMD, control flow graph, loop optimization, DPLL, resnet

1. INTRODUCTION

Modern graphics processing units (GPU) execute computer vision and "big data" processing tasks efficiently. Those tasks belong to the class of "embarrassingly parallel" problems, which perfectly matches the "single instruction, multiple data" (SIMD)[1] hardware architecture of GPU. Unfortunately, adapting an algorithm to the GPU platform is generally hard. Moreover, it can be hard to get good performance out of a GPU because of GPU hardware quirks. GPU programming research typically focuses on optimizing a single aspect of GPU programming, leaving the big picture out of the scope. The present work seeks to fill this gap by formalizing a method and strategy for adapting and optimizing arbitrary algorithms to the GPU platform. The paper consists of the following sections:

Section 1 consists of a survey of prior research regarding GPU code optimization;

Section 2 discusses prior works and explains the philosophy behind the method;

Section 3 describes the method as a sequence of steps;

Section 4 provides a detailed example of applying the method to adapt a complex backtracking search algorithm to the GPU platform and assessing the resulting implementation performance;

Section 5 demonstrates optimizing execution of artificial neural network (ANN) on a GPU using the proposed method;

Section 6 concludes the paper by briefly discussing the method's performance and prospects.

(Send correspondence to V.G.B.)
V.G.B.: E-mail: v.g.bulavintsev@gmail.com,
D.D.Z.: E-mail: ddzhdanov@mail.ru

2. METHOD IDEA

2.1 Prior works

While trying to abstract hardware details, GPU programming frameworks[2–4] either force the programmer to use rigid library primitives (e.g. matrix multiplication) or add too much abstraction.[5] GPU research community tries solving this by using one of the following strategies:

- design a perfect programming language for programming GPUs;[6]
- make the compiler smart enough to optimize the GPU code without intervention from the programmer.[7–9]
- strike a balance between the two strategies above by extending an existing programming language.[10]

In contrast, we focus on providing a robust mental model of the optimization process.

2.2 Computation as a tree of operations

To understand and manipulate complex algorithms, humans break those down into smaller subroutines and compress repeating steps using loops.[11] Every algorithm written within the paradigm of structured programming[12] can be expressed as a tree of subroutines or subformulae. Also, every loop in an algorithm can be unrolled into a fixed sequence of repeating operations.[13] Thus, all finite algorithms an formulae can be unfolded into a tree of operations.

2.3 GPU as a tree of computation units

GPU architecture can be though of as a hierarchy of organizational levels (OL), containing computational units of the same type such as multiprocessors of SIMD architecture.[1] Thus, a GPU can be represented as a tree of computational units.

Consequently, GPU programming *relies on the programmer* exposing the parallelism of the executed algorithm. GPU code execution strategy can be loosely matched to breadth-first visiting of the computation tree. In the GPU code, OLs existence is evident in the form of explicit and implicit synchronization primitives, warp shuffle instructions, atomic memory access instructions, thread identifiers and compiler intrinsics.

3. METHOD DESCRIPTION

Getting good performance from a GPU for a given algorithm is a matter of mapping the algorithm tree to the tree of the GPU's computational blocks efficiently. The method consists of the following steps:

Build the control flow graph (CFG) of the algorithm

Transform the CFG into a tree of parallelizable loops

Map the tree of loops to the tree of GPU computational units, according to the limitations of the GPU hardware.

3.1 Construct the control flow graph

The Control Flow Graph (CFG)[14] of a program consists of *basic blocks* (BB) connected by directed edges of control flow. Each BB represents a sequence of instructions with a single input and a single output point. CFG starts with the *entry block* and ends with the *exit block*.

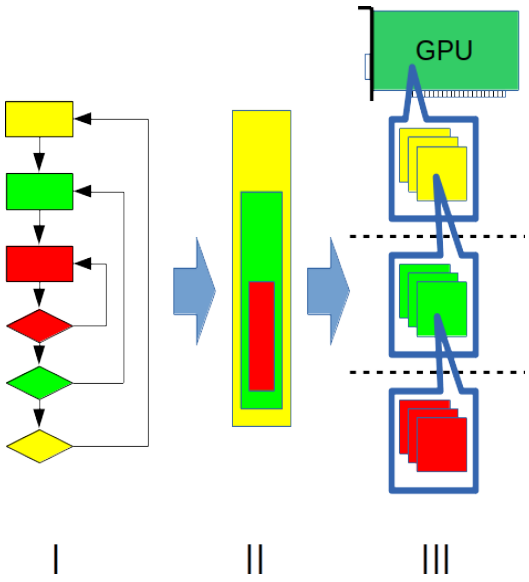

Figure 1. Algorithm adaptation stages

3.2 Build the tree of parallelizable loops

Informally, natural loop is as cycle formed by a single back edge, such that no edges from other parts of the CFG are leading into the loop body. A CFG that contains only natural loops is a *reducible* CFG. Any CFG can be transformed into a reducible one by node splitting. For the rest of the paper, we will discuss only reducible CFGs and natural loops.

After identifying natural loops, the CFG should be reduced to a tree of natural loops without data dependencies. To do this, we merge nested loops with data dependencies into parent parallelizable ones. Next, for every *parallelizable* loop, we should note its number of iterations (or estimate an approximate count of iterations). The result is a tree of parallelizable loops.

3.3 Map the algorithm tree to the hardware tree

At this step, we map the algorithm tree to the hardware tree, applying the loop unroll transformation[14] to split the loop nodes as necessary. Mapping a loop to a hardware level means assigning each iteration of the loop in a way that avoids waiting for results of computation performed by the other units of the same level. The primary optimization strategy is to position the loop nodes and their variables as deep into the hardware tree as possible to use faster, on-chip memory.

4. ALGORITHM ADAPTATION EXAMPLE

We demonstrate our method by adapting to a GPU platform the DPLL algorithm.[15] DPLL is the most popular algorithm for solving the Boolean formula satisfiability problem expressed in conjunctive normal form (CNF). Effectively, DPLL is a type of backtracking tree walk. The core idea of DPLL remained the same over the years:

1. guess a variable and assign a value to it
2. simplify the problem according to the guess
3. repeat the above steps until either the solution found or a contradiction is discovered, in which case
4. backtrack and try a different value for the guessed variable

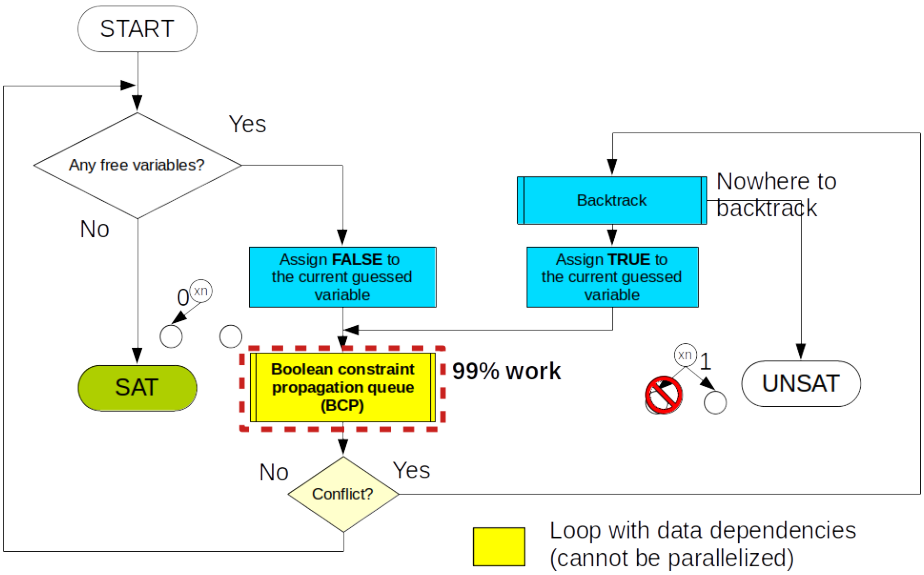

Figure 2. DPLL algorithm flowchart

4.1 Step 1, build a CFG

A simplified CFG for DPLL algorithm is shown at Fig. 2. 99% of computation happens inside the Boolean Constraint Propagation (BCP) procedure, that is thoroughly optimized in modern SAT solvers.[16]

4.2 Step 2, convert the CFG into a tree of loops

DPLL CFG can be viewed as a hierarchy of loops with an unpredictable number of iterations:

1. the top-level loop of guessing a variable value, i.e. tree walk;
2. the BCP loop, i.e. simplifying the formula;
3. checking the clauses of a variable;
4. checking the literals of a clause for conflicts or logic inference opportunities.

Note that level 2 loop iterations are interdependent because of every iteration changing the state of the formula's variables. As a result, the loops tree of DPLL contains a single branch in each level, making it look like a hierarchy. Our estimation of the average iteration count is based on parameters of problems used in the yearly SAT race competition.[17]

4.3 Step 3, map the algorithm tree to the GPU tree

For our analysis, the most important aspects of the architecture are the warp size and the maximum number of in-flight warps. Some important challenges in implementing DPLL on a GPU are:

- on average, the number of iterations at level 3 and level 4 loops are not enough to fill more than a couple of warps;
- it is impossible to predict when loops of level 3 or level 4 stop, resulting in SIMD ALUs waiting on complete branches;
- a typical CNF takes a couple of megabytes of memory;
- threads searching for solutions tend to stop abruptly at dead-ends.

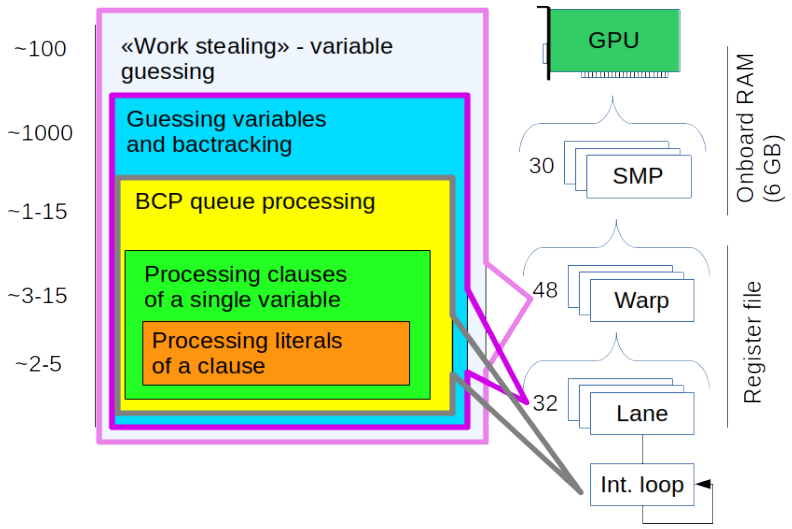

Figure 3. Matching DPLL loops hierarchy to a GPU's computational layers

We came up with the following solutions for the challenges outlined above:

- store the CNF only once, sharing common data between threads;
- apply loops merge transform[18] to level 3 and level 4 loops to decrease the SIMD branching-related performance deterioration;
- parallelize over level 1 loop (i.e. guessing variables[19]) to fill all the warps evenly;
- use work-stealing[20] to replenish the loop 1 task pool.

The resulting mapping is shown in Fig. 3.

4.4 Performance of the GPU-adapted DPLL

We implemented* the DPLL variant described above for NVIDIA GPUs in CUDA C language. To estimate the efficiency of the adaptation method, we also built a CPU version of the same algorithm. Table 1 contrasts the Boolean constraints propagation speed of the GPU-adapted DPLL variant running on a GPU to the performance of the same code running on a CPU. The performance is measured in millions of literals checked per second.

One can see that the adaptation procedure enabled efficient execution of the DPLL algorithm on a GPU. However, GPU performance is very dependent on the structure of the underlying problem.

Table 1. GPU vs CPU performance of the adapted DPLL algorithm

Benchmark problem	Constraints propagation speed (mln literals / second)	
	CPU (Core i7-8700k, 1 thread)	GPU (RTX2060)
"hole8.cnf" (pigeonhole problem)[21]	46	**166 (+260%)**
"dubois25.cnf" (synthetic problem)[21]	55	**436 (+690%)**

5. NEURAL NETWORK OPTIMIZATION EXAMPLE

ResNet[22] is a popular family of Artifical Neural Networks (ANN) used for image recognition tasks. Since the inception of the original ResNet-50[22] in 2015, many improved variations of the ANN have arose, such as ResNeXt-50.[23]

*https://github.com/ichorid/ringsat

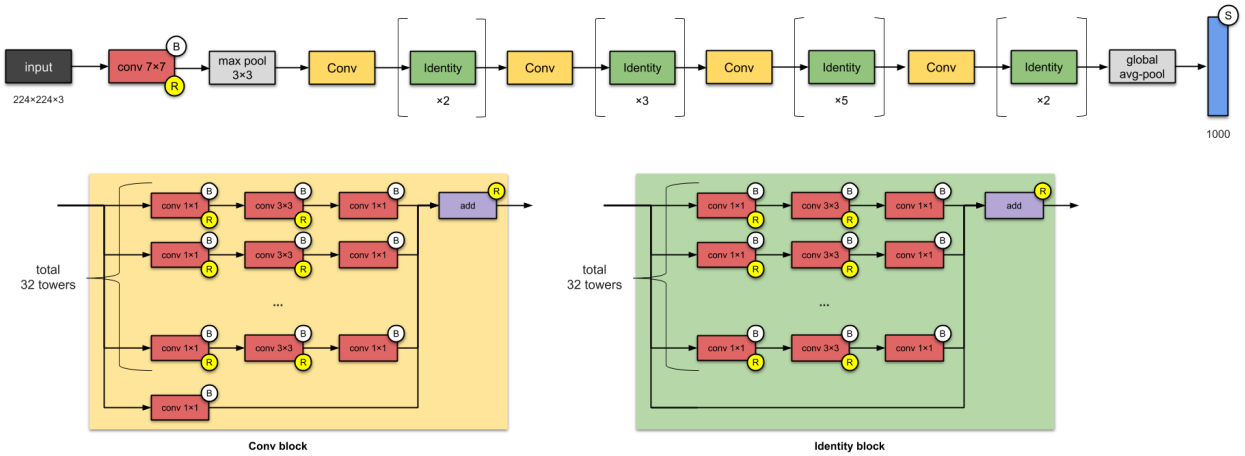

Figure 4. ResNeXt-50 neural network

5.1 ResNeXt-50 neural network

ResNeXt-50 adds parallel paths ("towers"), network-in-network style,[24] within each module (see visualisation[†] at Fig. 4). A typical implementation of ResNeXt-50 in a high-level framework[25] may show less than optimal inference speed, especially in case of processing of a single image, since frameworks are typically optimized for learning speed, simplicity of usage and high throughput. Low-latency, single-image inference speed can be critical in real-life applications, such as autonomous vehicles and augmented reality systems.

5.2 Optimizing ResNeXt-50 for low-latency execution on GPU

The case for ANN optimization is radically different from a typical looping algorithm: the network is essentially a deep sequence of matrix multiplications interleaved with aggregation-pooling procedures. To apply or method, we split the chain of execution into blocks as necessary. Existence of 32 "towers" in ResNeXt-50 gives us a perfect opportunity for GPU-specific optimization. One obvious way to optimize ResNeXt-50 for lower-latency execution is to map its "towers" to individual multiprocessors (Fig. 5), warps of a single multiprocessor (to improve caching efficiency), or even individual lanes. This way, all the "towers" could be executed simultaneously for a single input image, resulting in shorter processing times.

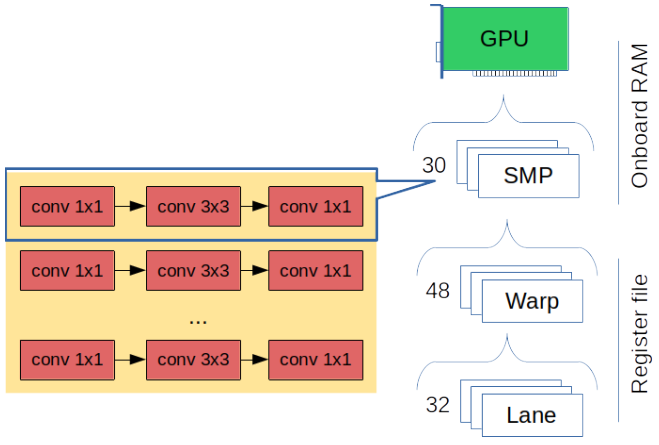

Figure 5. Mapping ResNeXt-50 "towers" to GPU hardware

[†]Illustration by Raimi Karim: https://towardsdatascience.com/illustrated-10-cnn-architectures-95d78ace614d

6. CONCLUSION

We proposed a method for adapting the algorithm to GPU architecture and demonstrated it by adapting a complex search algorithm (DPLL) to GPU. Our method is based on the mental model of matching the computation tree to the hardware tree. The model allows the programmer to convert an algorithm to GPU hardware while avoiding iterations of trial and error and guiding architectural choices towards optimal performance. The method is not bound to any particular programming language, but instead based on common notions of loops and structured programming, providing a convenient mental framework for thinking about GPU code optimization. One possible direction for future research is creating extensions for existing profilers and IDEs to provide the programmer with hints about the estimated performance of the GPU code.

REFERENCES

[1] Flynn, M. J., "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers* **C-21**, 948–960 (Sept. 1972).

[2] Stone, J. E., Gohara, D., and Shi, G., "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science Engineering* **12**(3), 66–73 (2010).

[3] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X., "TensorFlow: A System for Large-Scale Machine Learning," in [*12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*], 265–283, USENIX Association, Savannah, GA (2016).

[4] NVIDIA, Vingelmann, P., and Fitzek, F. H., "CUDA, release: 10.2.89," (2020).

[5] Dolbeau, R., Bodin, F., and de Verdiere, G. C., "One OpenCL to rule them all?," in [*2013 IEEE 6th International Workshop on Multi-/Many-core Computing Systems (MuCoCoS)*], 1–6, IEEE (2013).

[6] Baghdadi, R., Beaugnon, U., Cohen, A., Grosser, T., Kruse, M., Reddy, C., Verdoolaege, S., Betts, A., Donaldson, A. F., Ketema, J., Absar, J., Van Haastregt, S., Kravets, A., Lokhmotov, A., David, R., and Hajiyev, E., "PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming," in [*2015 International Conference on Parallel Architecture and Compilation (PACT)*], 138–149, IEEE, San Francisco, CA (Oct. 2015).

[7] Verdoolaege, S., Carlos Juega, J., Cohen, A., Ignacio Gómez, J., Tenllado, C., and Catthoor, F., "Polyhedral parallel code generation for CUDA," *ACM Transactions on Architecture and Code Optimization* **9**, 1–23 (Jan. 2013).

[8] Liou, J.-Y., Wang, X., Forrest, S., and Wu, C.-J., "GEVO: GPU Code Optimization Using Evolutionary Computation," *ACM Transactions on Architecture and Code Optimization* **17**, 1–28 (Dec. 2020).

[9] Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A., "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," in [*13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*], 578–594, USENIX Association, Carlsbad, CA (2018).

[10] Kataev, N. A. and Chernykh, S. A., "Automation of program parallelization in SAPFOR," 362–376 (2020).

[11] Abelson, H. and Sussman, G. J., [*Structure and interpretation of computer programs*], The MIT Press (1996).

[12] Böhm, C. and Jacopini, G., "Flow diagrams, turing machines and languages with only two formation rules," *Communications of the ACM* **9**, 366–371 (May 1966).

[13] Aho, A. V. and Ullman, J. D., [*Principles of compiler design*], Addison-Wesley series in computer science and information processing, Addison-Wesley, Reading, Mass., world student series ed ed. (1977).

[14] Allen, F. E., "Control flow analysis," *ACM SIGPLAN Notices* **5**, 1–19 (July 1970).

[15] Davis, M. and Putnam, H., "A Computing Procedure for Quantification Theory," *Journal of the ACM* **7**, 201–215 (July 1960).

[16] Eén, N. and Sörensson, N., "An Extensible SAT-solver," in [*Theory and Applications of Satisfiability Testing*], Goos, G., Hartmanis, J., van Leeuwen, J., Giunchiglia, E., and Tacchella, A., eds., **2919**, 502–518, Springer Berlin Heidelberg, Berlin, Heidelberg (2004). Series Title: Lecture Notes in Computer Science.

[17] Balyo, T., Heule, M., and Jarvisalo, M., "SAT Competition 2016: Recent Developments," *Proceedings of the AAAI Conference on Artificial Intelligence* **31** (Feb. 2017).

[18] Bulavintsev, V., "Flattening of data-dependent nested loops for compile-time optimization of gpu programs," *International Journal of Open Information Technologies* **7**(9), 7–13 (2019).

[19] Zaikin, O. S. and Kochemazov, S. E., "On black-box optimization in divide-and-conquer SAT solving," *Optimization Methods and Software* , 1–25 (Nov. 2019).

[20] Blumofe, R. D. and Leiserson, C. E., "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)* **46**(5), 720–748 (1999). Publisher: ACM New York, NY, USA.

[21] Hoos, H. H. and Stützle, T., "SATLIB: An online resource for research on SAT," *Sat* **2000**, 283–292 (2000).

[22] He, K., Zhang, X., Ren, S., and Sun, J., "Deep residual learning for image recognition," in [*Proceedings of the IEEE conference on computer vision and pattern recognition*], 770–778 (2016).

[23] Xie, S., Girshick, R., Dollár, P., Tu, Z., and He, K., "Aggregated residual transformations for deep neural networks," in [*Proceedings of the IEEE conference on computer vision and pattern recognition*], 1492–1500 (2017).

[24] Lin, M., Chen, Q., and Yan, S., "Network in network," *arXiv preprint arXiv:1312.4400* (2013).

[25] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al., "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems* **32**, 8026–8037 (2019).