Flattening of Data-Dependent Nested Loops for Compile-Time Optimization of GPU Programs

V. G. Bulavintsev

***Abstract* — Modern Graphics Processing Units (GPUs) belong to the "Single Instruction Multiple Data" (SIMD) computational architecture class. Due to inefficient execution of divergent branches, SIMD devices can lose performance on nested loops with data-dependent exit conditions. A specialized compile-time Control Flow Graph (CFG) transformation routine can solve this problem. The routine reduces loop nest level by merging the inner loop with the outer loop. The transformed program remains logically equivalent to the original one, while its branching pattern becomes better suited for execution on a SIMD device. The routine is implemented as a Low-Level Virtual Machine (LLVM) Transformation Pass. Depending on the dataset and nested loops parameters, the transformation reduces the worst-case running time of a specialized GPU benchmarking application up to 24 times.**

***Keywords*— Branch Divergence, Control Flow Analysis, LLVM, Nested Loops, SIMD.**

I. Introduction

SIMD devices, such as GPUs, suffer significant performance loss when running an algorithm that relies on unpredictable conditional jumps. This inefficiency stems from the fact that SIMD devices serialize execution of divergent branches [OPT2008]. However, the divergence problem can be solved for some types of branching constructs by applying a corresponding code transformation routine. In this paper, we build such a routine for the case of nested loops with data-dependent number of iterations. We develop a proof-of-concept version of the transformation for the LLVM compiler platform and apply it to a benchmark utility designed to simulate nested loops constructs that can be found in real-life applications.

In the rest of the Section I we describe the SIMD branching model and the problem of nested loops in greater detail. In Section II, we describe some concepts used in this work, such as control flow graph and natural loop. Section III introduces the actual transformation routine. Section IV describes the proof-of-concept LLVM Transformation Pass implementing the routine. Section V contains an experimental evaluation of the transformation's effect on a benchmarking application. Section VI contains information on related works. Section VII concludes the paper with the discussion of the results.

Manuscript received 27.05.2019.
V. G. Bulavintsev is with the Technical University of Delft, Mekelweg 5, 2628 CD Delft, the Netherlands (e-mail: v.g.bulavintsev@gmail.com).

A. SIMD architecture and conditional branch execution problem in GPUs

SIMD acronym stands for Single Instruction Multiple Data, a class of parallel computers [2]. SIMD machines are often described as *multiprocessors*, in contrast to traditional processors that belong to the Single Instruction Single Data (SISD) class. In a SIMD machine, a single Control Unit (CU) dispatches the same single instruction to multiple Processing Units (PU) belonging to the same working group, called the SIMD group. Working in a «lockstep» mode with other PUs of the group, each PU applies the same instruction to a different element of a dataset (Fig. 1). This parallel execution method makes SIMD architecture very efficient at vector processing.

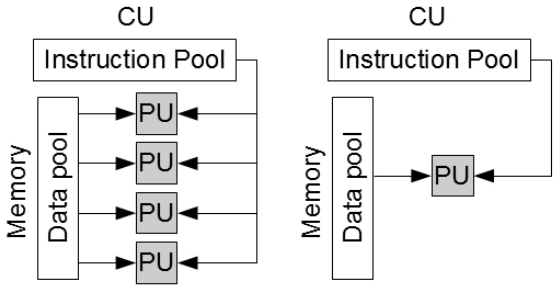

Fig. 1. SIMD and SISD architectures.

Unfortunately, SIMD design has a drawback that becomes evident during the execution of conditional expressions: different branches of the program have to run different instructions, but only a single instruction type can be issued by CU at once to all PUs of the SIMD group.

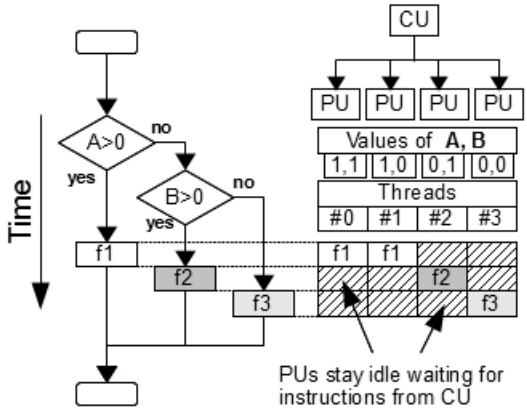

Fig. 2. Serialization of branch execution on SIMD platforms.

Modern GPUs handle this situation by serializing the execution of divergent branches (Fig. 2). According to the

branching condition, a group of computational threads executed within a single SIMD group (*a warp* in NVIDIA's terminology [3]) is split into two subgroups *L* and *R*. The threads in group *L* run (their PUs are active) while the threads in group *R* wait in the «frozen» state (their PUs are idle). When group *L* reaches a *reconvergence point* in the program, it stops and waits in the «frozen» state for group *R* to catch up. After group *R* reaches the same reconvergence point, the execution continues as usual. In the case of a single-level conditional branch, up to ½ of PUs can stay idle at any moment, waiting for other threads to reach a common point in program execution.

Unfortunately, nested conditional branches (e.g., nested "if-else" statements or nested "while" loops) lead to nested serialization, thread subgroups splitting to sub-subgroups, until a subgroup consists of a single thread, corresponding to a single active PU (Fig. 2). This effect can decrease GPU performance up to *n* times, where *n* is the warp size (32 or 64 in modern GPUs [3], [4]).

B. The problem of nested loops

Consider a GPU program that contains two "while" loops *A* and *B*, such that loop *B* is nested inside loop *A*. Loop *A* processes a list of elements of type *a*. Loop *B* processes elements of type *b*. Processing one element of type *a* requires sequential processing of several elements of type *b* (e.g., each *a* element is a container that contains some *b* elements). If the amount of *b* elements is different for each *a* element processed in a SIMD group, the loss of performance from branching is inevitable. Intuitively, this happens because the threads that had already processed all *b* elements in their current *a* element and thus had become ready to leave the *B* loop (and get the next *a* element) must wait for the other threads of their SIMD group to catch up. Only then the group can step into the next iteration of loop *A* and get new *a* elements for processing. Combining *A* and *B* into a single loop prevents this effect by forcing a thread to check if it is ready to process the next *a* element every time it finishes processing any *b* element [5].

II. Preliminaries

A. Control flow graph

Control Flow Graph (CFG) represents all paths of execution of a program [6]. It gives a natural way to study and manipulate the effects of control statements and conditional jumps.

CFG consists of *Basic Blocks* (BBs), connected by directed edges that represent control flow (execution sequence) of the program. A basic block represents a straight-line sequence of instructions with only one entry point (the start of the block) and only one exit point (the end of the block). Execution of instructions in a basic block always starts at its first instruction and ends with its last instruction, which is always a (conditional or unconditional) jump instruction. In a CFG, the *entry* and *exit* blocks represent the beginning and the end of the control flow.

In a flow graph, node *d* is said to *dominate* node *n* if every path from entry to *n* goes through *d*. An edge from node *N* (*latch*) to node *H* (*header*) is said to be *a back edge* if *H* dominates *N*. If all nodes are still reachable from entry node after removal of all back edges, the CFG is said to be *reducible*. Structured programming always produces reducible CFGs. Any irreducible CFG can be transformed into its reducible equivalent using *node splitting*. All loops of a reducible CFG are *natural loops* (Fig. 3).

B. Natural loops

A natural loop L of a back edge $(N \to H)$ is the smallest set of nodes satisfying the following properties:

1. $H, N \in L$;
2. for any two nodes $n, n' \in L$, there exists a path from n to n';
3. for any node $n \in L$, the set of its predecessors $Pred(n) \subseteq Pred(H)$.

Informally, a natural loop is a cycle in a graph that is formed by a single back edge, and that has no jumps into the middle of its body from other parts of the program. Control flow enters a natural loop only through its header and leaves it through one or more blocks with successors outside the loop (exit blocks).

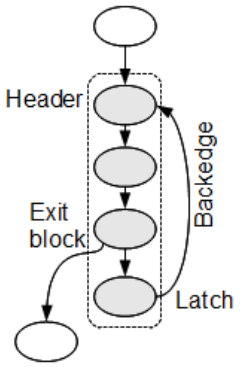

Fig. 3. An example of a natural loop.

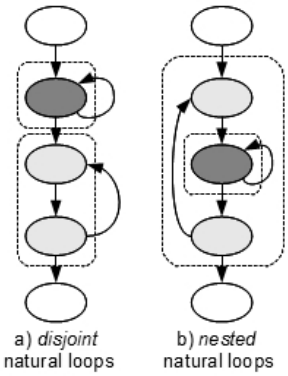

Fig. 4. Disjoint and nested natural loops.

Natural loops of a reducible graph form a tree hierarchy, where any pair of natural loops is either *disjoint* or *nested* (Fig. 4). For the rest of this paper, we will discuss only reducible graphs and natural loops.

III. Loop nest reduction

A. Transformation procedure

We presume the graph *G* to be reducible and loops *A* and *B* to be nested natural loops (*B* is nested in *A*). For simplicity, we consider loop *B* as a single basic block, holding the roles of the header, the latch, and the exit blocks simultaneously. Loop *A*'s header block A_{wh} is non-empty (e.g., it contains some "payload" instructions). It precedes loop *B*. Loop *A*'s exit/latch block A_e follows loop B immediately (Fig. 5a). The variables controlling *A* and *B*'s exit conditions are named accordingly: *a* and *b*.

The basic idea of the loop nest reduction transformation is to use *A*'s back edge to "emulate" *B*'s back edge, and then isolate instructions exclusive to *A* by putting them under conditional bypass (the latter procedure is known as *predication*). The procedure starts with the original CFG

(Fig. 5a) and goes as follows:

1. Split A_{hw} into the empty header block A_h and the "work" block A_w (Fig. 5b);
2. Put a new empty latch block A_l on A's back edge, thus removing the latch role from A_e (Fig. 5b);
3. Redirect B's back edge to point to A_l (Fig. 5c);
4. Create pre-header A_p with "store *false* into variable b" instruction (Fig. 5c);
5. Add conditional jump instruction to A_h, such that if b is *false*, A_w will be executed next, and if b is *true*, execution will jump straight to the B loop header (Fig. 5c). The resulting CFG is shown in Fig. 5c.

We do not describe here the transformation procedures for every possible CFG configuration that include nested loops since these can be trivially derived from the one described above. For example, if the outer loop contains several independent inner loops instead of one, steps 3-5 can be applied to each of these inner loops individually. In general, it is possible to apply the procedure iteratively to reduce the depth of the natural loops tree to any desired level.

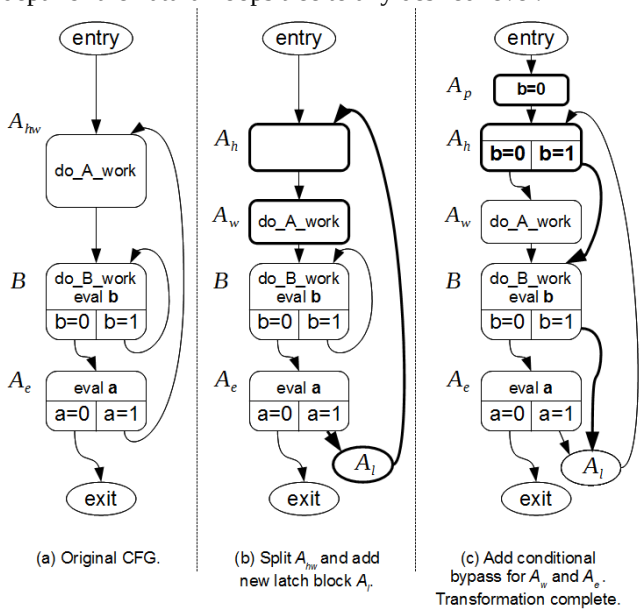

Fig. 5. Nested loops fusion transformation.

B. *The correctness of the transformation*

Let us compare the execution of the original and transformed CFGs. In original CFG we go straight to the execution of A's "payload" instructions inside A_{hw} block. In the transformed CFG, we instead first initialize b variable to *false* at A_p and proceed to check its value at A_h. This check leads to the execution of A_w where A's "payload" is done, as in original CFG.

Then we proceed into loop B's body basic block and compute b condition value. Assume the condition b is *true* and we need to take another iteration of the loop B. In the original CFG, we immediately jump back to the beginning of loop B's basic block. In the transformed CFG we jump to loop A's new latch block A_l and immediately make an unconditional jump to A_h. Next, because b is *true*, we take the conditional jump into loop B's basic block, as if executing the original CFG. Notice that we haven't executed "payload" instructions specific to loop A, because they were moved to A_w, and we "jumped over it" via conditional $A_h \to B$ edge. Thus, the execution of A_w is impossible when b is *true*.

When b becomes *false*, the execution process must leave loop B and take on a new iteration of loop A. In both variants of CFG after b becomes *false*, the conditional jump instruction at the end of loop B's basic block leads to A_e. There, we execute instructions specific to loop A, evaluate condition a (in this example, to *true*) and take conditional jump directly to A_{hw} (in case of original CFG) or to A_l and then to A_h (in the transformed one). Original CFG then executes A's "payload" instructions contained in A_{hw}. Transformed CFG checks b's value at A_h block, sees it is *false* and jumps to A_w where it executes A's "payload" instructions too.

The case when a and b are both *false*, resulting in an escape from the loop A, can be traced in a similar manner. It is easy to see that, in regards to "payload" processing, both CFGs execution *results* are equivalent, though their execution *paths* differ. The semantics of the transformed CFG is always equivalent to the semantics of the original CFG. The instructions that were added in the process of the transformation do not interfere in any way with the original data and stay local to the loops' bounds.

The transformed CFG's pattern of making jumps to A_h after each execution of B adds some overhead. However, for SIMD architectures this pays off because it makes all threads of a SIMD group regularly "check for new work" for cycle B by speculatively executing cycle A's "payload" instructions in A_w.

C. *Estimation of potential benefit*

Sometimes, the overhead of speculative checks can exceed the potential benefits of the higher usage of PUs. To describe the conditions that make the transformation viable, let us first introduce the following notation:

T_a – the time to execute A_w basic block (the external cycle's "payload" instructions);

T_b – the time to execute B's instructions;

N – the total number of threads/PUs in a SIMD group (size of a warp in NVIDIA's terms);

n – the number of idle threads in a SIMD group;

$(N-n)T_a$ - the "cost" of going to the outer loop to get new work for the idle threads;

nT_b – the potential payoff from getting new work for the idle threads (i.e., how much work they would do at the next iteration of B if we get new work for them).

At the end of each iteration of the internal loop, it makes sense to go for the new work if the potential gain in the number of payload elements processed at the next cycle outweighs the costs: $nT_b - (N-n)T_a > 0$. Or, put another way:

$$\frac{n}{N} > \frac{T_a}{T_a + T_b}; (1)$$

Formula (1) can be interpreted as follows: the "heavier" the instructions in the inner cycle relative to the instructions exclusive to the outer cycle, and the more threads would on

average stay idle in the untransformed program, the more speed up the transformation will provide. Note that the potential benefit of the transformation is limited by warp size N because of $n \leq N$.

IV. Implementation

A. Low-Level Virtual Machine framework

Low-Level Virtual Machine (LLVM) [7], [8] framework provides a set of compilers, translators, and other tools working with the LLVM Intermediate Representation language (IR) [7]. IR is a strongly typed low-level language for a virtual machine with an infinite number of registers, expressed in Single State Assignment (SSA) form [9]. LLVM is comprised of "front-ends" that enable compilation from high-level programming languages into IR, and "back-ends" that translate IR into binary machine code for different hardware platforms. In between these stages, a program in the IR form may undergo optimizations implemented within a framework of so-called "*transformation passes*."

B. Transformation pass

We developed a proof-of-concept LLVM IR transformation pass that implements the loop nest reduction procedure described above. We also developed a specific benchmark (see the next subsection) that emulates unpredictable nested work queues. The source code for both transformation pass and test application can be found at [10]. We use NVIDIA CUDA [3] as a GPGPU language of choice in our test application, but the transformation pass is not limited to CUDA or any other hardware platform since it works directly on IR. It can be applied without any changes to any program translated to IR form.

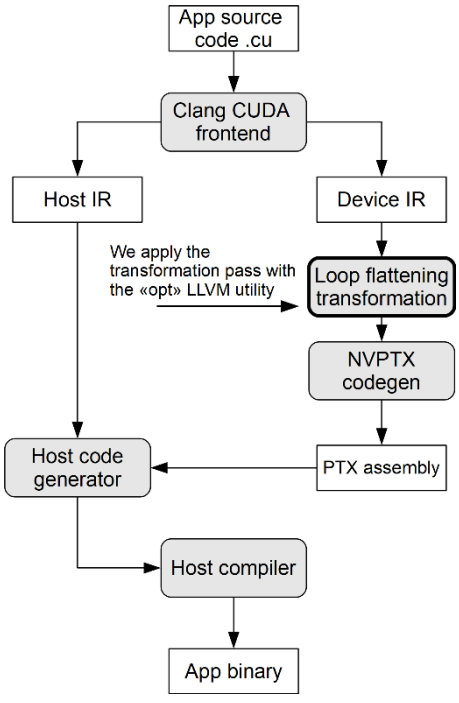

Fig. 6. CUDA program compilation and transformation process with gpucc (clang) compiler and LLVM tools.

C. Applying the transformation

The program should be first translated to the IR form. We achieve this by using the open-source GPGPU compiler "gpucc" [11]. We invoke it via the Clang [12] compiler infrastructure by hand and stop the compilation process after the GPU, and host parts of the program are translated to IR. Next, we apply our loop nest reduction pass to the GPU part of the IR code with the "*opt*" utility [7]. Then compilation is continued as usual (Fig. 6).

Disassembly of the NVPTX[1] binary [3], [11] confirms that only one back edge remains in the CFG after applying the transformation.

V. Experimental evaluation

A. Benchmarking application description

Formula (1) suggests that the transformation's efficiency for a given algorithm depends on two factors:

a) the ratio of "weights" of the inner loop's instruction sequence compared to the outer loop;
b) the average relative number of threads leaving the inner loop early.

(a) mostly depends on the properties of the algorithm itself, while (b) depends on the properties of a typical dataset processed by the algorithm. To investigate the effects of these factors on the viability of the loop nest reduction, we developed a specialized test application that simulates different ratios of loop "weights" and threads' early exits.

The benchmark application emulates the behavior of a two-level work queue that can be found, for example, in some search algorithms [5], [13], [14]. The GPU part of the program consists of a pair of nested loops: outer loop *A* and inner loop *B*, similar to the example given in section II.A.

Loop *B*'s body consists of a dummy computational routine that stands for "useful work" and a command to increase the "useful processing" counter by 1. Loop *B* stops when at least one of the two following conditions becomes true:

1. the loop has been repeated for a fixed number of times;
2. the "early exit" condition has been met.

The "early exit" sets up a situation when the thread has no work at the inner loop and must get to the outer loop to obtain new data.

Loop *A* runs for a fixed number of iterations. At the beginning of the first iteration of loop *A*, a simple pseudo-random number generator determines which particular threads will meet the "early exit" condition. The total number of threads which will meet the "early exit" condition at any iteration of loop *A* is controlled by parameter *k*. This scheme allows for unbiased distribution of "early exits" inside the SIMD group. Also, it simulates the worst possible case for SIMD performance, as the threads receive the "early exit" signal immediately, thus for

B. Experimental setup, performance measurement methods

Experiments were conducted on Ubuntu Linux 18.04 with

[1] Nvidia Parallel Thread Execution virtual machine and instruction set architecture.

LLVM-9.0 and CUDA SDK 10.1. Program optimization was enabled with "-O3" for both GPU and CPU code. CUDA assembler was forced to generate Compute Capability 7.0 code [3]. GPU part of the benchmark application was run on RTX 2060 NVIDIA GPU.

Three variants of the original benchmark application were produced, with different ratios of computational instructions between the external and the internal loops ("*A/B* ratios" equal to 10/1, 1/1, 1/10, 1/100). Next, the transformation procedure was applied to produce the corresponding transformed benchmark applications. The execution times of each of these original and transformed applications were then measured for all 32 possible values of parameter $k, 0 \leq k \leq 31$.

C. Experimental results

In the case of the original application, when $k = 0$, SIMD device lose no performance from branch divergence effects, because no "early exit" conditions are generated. When $k = 31$ (32 is the size of a SIMD group of NVIDIA GPUS), "early exits" force all but one thread to stay idle. Overall, parameter k allows us to simulate processing of a dataset which has any desired probability of a thread meeting an "early exit" event.

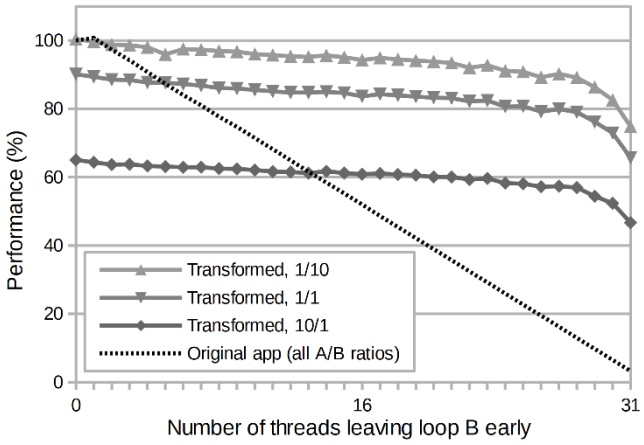

Fig. 7**.** Performance of the benchmark application variants after applying loop flattening transformation. Performance is given relative to corresponding non-transformed variant. Different data lines show benchmark variants with different A/B (outer loop / internal loop) "computational costs" ratios.

Fig. 7 shows the performance of the benchmark application variants for different values of k. Performance is shown relative to that of the "original" (non-transformed) application with $k = 0$.

One can immediately see that the execution time for the *original* application does not depend on k and always stays on the same level. This is expected, because even if all the threads except a single one would meet the "early exit" condition, the SIMD architecture will not let any threads get to the outer loop. Contrarily, *transformed* applications will always check for the new work, which results in shorter execution times for lower values of k. This speculative work-checking strategy introduces an overhead that makes transformed applications slower than the original for small to medium values of k, for higher A/B ratios. Overall, the transformation provides good speedups for applications with low A/B ratios, if the value the of k is high i.e., the dataset has a high probability of generating an "early exit" event. These observations confirm the results of the analysis conducted in Section II.D.

At $k = 31$, the original application loses 97% of performance. However, the transformed versions lose only 24%-56% of performance, depending on their *A/B* ratio. Thus, transformed applications perform up to 24x times faster than the original at this dataset. Due to the overhead of the speculative checks, for *A/B* ratio of 10/1, the transformation starts to pay off only from $k = 14$. That roughly corresponds to 50% probability that a thread will leave the loop B early. Starting from *A/B* ratio of 1/10, the overhead has no effect, and for every k, the transformed application runs faster than the original one. Curiously, for $0 \leq k \leq 3$ the transformed application shows the performance of up to 102%. This can be explained by transformation making the CFG more suitable for optimization at the later stages of the compilation process.

VI. Related works

The general method of converting control dependence to data dependence was described in 1983 by John R. Allen et al. [15]. "Loop flattening" term was coined in 1992 by K. Kennedy in [16], where he described the basic idea of reducing a pair of nested loops into a single loop, although in a Fortran-specific way. In 1995, the technique was studied, still in Fortran context, dissertation [17], and the context of sparse matrix-vector multiplication in [18]. The community's interest in optimizing execution of code on SIMD architectures resurged with advances in GPGPUs programming features [1].

During the last decade, the scientific effort was focused on solving the SIMD branching problem with both hardware and software solutions. The hardware-based approach is exemplified by dynamic thread regrouping [19], advanced methods of mapping control flow to SIMD processors within a bounded region of the program [20], dual-path execution model [21] and lane permutation [22], execution cycle compression [23], hardware-accelerate flattened loops execution [24]. The software-based solutions for the divergence problem included merging similar execution paths via the genetic sequencing algorithm [25], building a divergence metrics [26], analyzing SIMD group's size effect on the problem [27] and linearization of control flow in SIMD programs [28]-[30]. Work [31] uses predication to enable loop unrolling for data-dependent. Work [32] by T.D. Han and T. S. Abdelrahman is of particular interest because it analyzes the same loop flattening transformation as we do. Our work is similar to [32] in regards to implementing the transformation described in [16] within the LLVM framework. In [32], transformation is evaluated experimentally using a synthetic benchmark that simulates a distribution of "early exit" events and on some real-life applications. However, [32]'s authors do not publish the source code for the transformation. Our work provides the source code, focuses on the synthetic benchmark that simulates the worst case performance, and establishes an analytical expression for

assessing the transformation feasibility (Section III.C).

It is important to note that there is some confusion in the literature in regards to naming the transformation presented in [16]. The authors of [32] name their transformation "Loop Merging" instead of "Loop Flattening" and do not cite [16]. "Loop Flattening" transformation that is a very limited variant of the transformation described in [16] is described in [33]. It targets optimization of simple nested loops with known iteration counts (i.e., arrays processing) and is included in both LLVM and GCC [34]. To honor the transformation name used in [16] and distinguish it from the current LLMV and GCC variants, we decided to use the name "Loop Flattening of Data-Dependent Nested Loops" in our work.

VII. Conclusion

Experimental data presented in Fig. 7 shows that applying the loop nest reduction transformation to the benchmarking application increases its speed up to 24x times. However, the transformation results in significant slowdowns when the probability of an early exit from the internal loop is small, or when the algorithm's external loop instructions take more time than those in the internal loop. Thus, the transformation should be applied sparingly: careful analysis of real-world workloads should justify its usage in each particular case.

These observations give a clear direction for further research: the transformation should be applied automatically by the compiler as a part of the program optimization routine, driven by profiling information and static code analysis.

Experimenting with this transformation on real-world applications proved to be difficult because of the "chicken-and-egg" problem: the algorithms that had already been successfully ported to GPUs will not benefit from it, but algorithms that could benefit from it had not been ported to GPUs yet, because these do not function well there. Overall, loop nest reduction transformation can provide a way to adapt algorithms that rely on intricate branching patterns to GPU platforms.

References

[1] S. Ryoo et al., "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," *Proc. 13th ACM SIGPLAN Symp. Principles and practice of parallel programming*, pp. 73-82, 2008.

[2] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Trans. on Computers*, vol. C-21, pp. 948–960, 1972.

[3] *CUDA Toolkit Documentation v10.1*, Nvidia corp., Santa Clara, CA, 2019.

[4] *OpenCL User Guide v3.0*, AMD corp.: Santa Clara, CA, 2015.

[5] V. G. Bulavintsev, "An evaluation of CPU vs. GPU performance of some combinatorial algorithms for cryptoanalysis," *CMSE Bulletin of the South Ural State University*, vol. 4(3), pp. 67–84, 2015.

[6] F. E. Allen, "Control flow analysis," *ACM SIGPLAN Notices*, vol. 5, no. 7, pp. 1–19, Jul. 1970.

[7] C. Lattner, V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," *Proc. 2004 Int. Symposium on Code Generation and Optimization*, Washington DC, IEEE Computer Society, pp. 75-86, 2004.

[8] *The LLVM Compiler Infrastructure Project*, 2019, http://llvm.org.

[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Programming Languages and Systems*, TOPLAS 13.4, pp. 451–490, 1991.

[10] V. G. Bulavintsev, *NestedLoopsFusion GitHub project*, 2019, https://github.com/ichorid/nestedloopsfusion.

[11] J. Wu, et al., "gpucc: an open-source GPGPU compiler", *Proc. 2016 Int. Symp. Code Generation and Optimization*, New York, ACM, pp. 105–116, 2016.

[12] *Clang: a C language family frontend for LLVM*, 2019, http://clang.llvm.org.

[13] V. G. Bulavintsev, A. A. Semenov, "GPU-based implementation of DPLL algorithm with limited non-chronological backtracking", *Prikladnaya Diskretnaya Matematika, Suppl.*, vol. 6, pp. 111-112, 2013.

[14] O. Zaikin, "Application of parallel SAT solving algorithms for crypt-analysis of the shrinking and self-shrinking keystream generators," *International Journal of Open Information Technologies (INJOIT)*, vol.6, no. 10, pp. 29-33, 2018.

[15] J.Allen, K. Kennedy, C. Porterfiel and J. Warren, "Conversion of control dependence to data dependence," *Proc. 10th ACM SIGACT-SIGPLAN Symp. Principles programming languages*, pp. 177-189, 1983.

[16] K. Kennedy, "Relaxing SIMD control flow constraints using loop transformations," *Proc. ACM SIGPLAN 1992 conf. Programming language design and implementation*, vol. 27, no. 7, pp. 188-199, 1992.

[17] R.Von Hanxleden, "Compiler support for machine-independent parallelization of irregular problems," Doctoral diss., Rice Univ., 1995.

[18] A. Ghuloum and A. Fisher, "Flattening and parallelizing irregular, recurrent loop nests," *ACM SIGPLAN Notices*, vol. 30, no. 8, pp. 58-67, 1995.

[19] W. Fung, I. Sham, G. Yuan, and T. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," *Proc. 40th Annual IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, pp. 407-420, 2007.

[20] G. Diamos et al., "SIMD re-convergence at thread frontiers," *Proc. 44th Annual IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, pp. 477-488, 2011.

[21] M. Rhu and M. Erez, "The dual-path execution model for efficient GPU control flow," *Proc. 19th Int. Symp. High Performance Computer Architecture (HPCA)*, pp. 591-602, 2013.

[22] M. Rhu and M. Erez, "Maximizing SIMD resource utilization in GPGPUs with SIMD lane permutation," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 356-367, 2013.

[23] A. Vaidya, A. Shayesteh, D. Woo, R. Saharoy and M. Azimi, "SIMD divergence optimization through intra-warp compaction," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 368-379, 2013.

[24] J. Lee, S. Seo, H. Lee and H. Sim, "Flattening-based mapping of imperfect loop nests for CGRAs," *Proc. 2014 Int. Conf. Hardware/Software Codesign and System Synthesis*, p. 9, 2014

[25] B. Coutinho, D. Sampaio, F. Pereira and W. Meira Jr., "Divergence analysis and optimizations," *2011 Int. Conf. Parallel Architectures and Compilation Techniques*, IEEE, pp. 320-329, 2011.

[26] Z. Cui, Y. Liang, K. Rupnow and D. Chen, "An accurate GPU performance model for effective control flow divergence optimization," *26th International Parallel and Distributed Processing Symposium*, IEEE, pp. 83-94, 2012.

[27] T. Schaub, S. Moll, R. Karrenberg and S. Hack, "The impact of the SIMD width on control-flow and memory divergence," *ACM Trans. Architecture and Code Optimization (TACO)*, vol. 11, no. 4, p. 54, 2015.

[28] H. Wu, G. Diamos, S. Li and S. Yalamanchili, "Characterization and transformation of unstructured control flow in gpu applications," *1st International Workshop on Characterizing Applications for Heterogeneous Exascale Systems*, 2011.

[29] J. Anantpur and R. Govindarajan, "Taming control divergence in GPUs through control flow linearization," *Int. Conf. Compiler Construction*, Springer, Berlin, Heidelberg, pp. 133-153, 2014.

[30] T. Han and T. Abdelrahman, "Reducing branch divergence in GPU programs," *Proc. 4th Workshop on General Purpose Processing on Graphics Processing Units*, ACM, p. 3, 2011.

[31] A. Carminati, R. Starke and R. de Oliveira, "Combining loop unrolling strategies and code predication to reduce the worst-case execution time of real-time software," *Applied computing and informatics*, vol. 13, no. 2, pp. 184-193, 2017.

[32] T. Han and T. Abdelrahman, "Reducing divergence in GPGPU programs with loop merging," *Proc. 6th Workshop General Purpose*

Processor Using Graphics Processing Units, ACM, pp. 12-23, ACM, 2013.

[33] S. Pop, R. Yazdani and Q. Neill, "Improving GCC's auto-vectorization with if-conversion and loop flattening for AMD's Bull-dozer processors," *GCC Developers' Summit*, p. 89, 2010

[34] *GNU Compiler Collection*, 2019, http://gcc.gnu.org.