Enhancing Performance through Control-Flow Unmerging and Loop Unrolling on GPUs

Alnis Murtovi
TU Dortmund University
Dortmund, Germany
alnis.murtovi@tu-dortmund.de

Giorgis Georgakoudis
Lawrence Livermore National Laboratory
Livermore, USA
georgakoudis1@llnl.gov

Konstantinos Parasyris
Lawrence Livermore National Laboratory
Livermore, USA
parasyris1@llnl.gov

Chunhua Liao
Lawrence Livermore National Laboratory
Livermore, USA
liao6@llnl.gov

Ignacio Laguna
Lawrence Livermore National Laboratory
Livermore, USA
lagunaperalt1@llnl.gov

Bernhard Steffen
TU Dortmund University
Dortmund, Germany
bernhard.steffen@tu-dortmund.de

2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) | 979-8-3503-9509-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/CGO57630.2024.10444819

***Abstract*—Compilers use a wide range of advanced optimizations to improve the quality of the machine code they generate. In most cases, compiler optimizations rely on precise analyses to be able to perform the optimizations. However, whenever a control-flow merge is performed information is lost as it is not possible to precisely reason about the program anymore. One existing solution to this issue is code duplication, which involves duplicating instructions from merge blocks to their predecessors.**

This paper introduces a novel and more aggressive approach to code duplication, grounded in loop unrolling and control-flow unmerging that enables subsequent optimizations that cannot be enabled by applying only one of these transformations.

We implemented our approach inside LLVM, and evaluated its performance on a collection of GPU benchmarks in CUDA. Our results demonstrate that, even when faced with branch divergence, which complicates code duplication across multiple branches and increases the associated cost, our optimization technique achieves performance improvements of up to 81%.

***Index Terms*—compiler, code duplication, LLVM, GPU**

I. Introduction

GPU architectures are widely deployed on HPC computing centers, cloud computing installations, and even embedded devices, for high performance and power efficiency. They greatly accelerate execution of a wide variety of applications, including HPC scientific codes and AI/ML workloads. Compiler optimization on GPU codes is essential for achieving the performance potential of those architectures. However, compiler optimization for these architectures has unique challenges given their unique characteristics: massive parallelism in the form of SIMT hierarchies, lockstep execution limited by branch divergence, and a complex memory subsystem. To account for those challenges, compilers, such as LLVM [1], modify their compiler optimization pipeline to introduce transformations explicitly targeting optimized code generation for GPUs. Nevertheless, devising new compiler optimizations to generate faster code on GPUs is an open problem, with significant possible impact given the multitude of applications necessitating GPU execution for high performance.

"Don't repeat yourself" (DRY) is a core principle in software development that emphasizes reducing redundancy and reusing code to improve maintainability, efficiency, and readability. Although this principle undoubtedly offers advantages from a software developer's point of view, introducing redundancy or duplicating code can prove beneficial from a performance point of view. Compilers use a wide range of advanced optimizations to improve the quality of the machine code they generate. Compiler optimizations often depend on the availability of precise analyses to be effective. Whenever a control-flow merge is performed, information is lost as it is not possible to precisely reason about the program anymore.

Furthermore, existing approaches [2]–[4] for optimizing GPU code generation intensely prioritize basic block and control-flow merging targeting to reduce performance degradation due to branch divergence. In brief, GPUs follow the Single-Instruction Multiple Threads (SIMT) execution model. Threads are organized into groups, called warps using NVIDIA's terminology, and all threads of a warp execute an instruction in lockstep. If threads in a warp evaluate a condition differently and therefore branch to different basic blocks, execution is serialized, with each subset of threads on the same path executing consecutively. Although such approaches address the performance problems by reducing branch divergence, they further obscure the compiler's view and analysis precision on the code.

By contrast, we propose a new optimization approach targeting complex loops, which are common hotspots in execution, that include branching in their bodies. Our approach combines unmerging control flow with unrolling, to effectively recover control-flow provenance information and facilitate precise analyses for additional, subsequent compiler optimizations. Control-flow unmerging leverages code duplication, i.e., moving instructions from merge blocks to their predecessors, which has been shown [5]–[8] to enable optimization, such as branch removal, constant propagation, and redundancy elimination. Further, combining it with loop unrolling amplifies its effectiveness by increasing the scope of analysis and optimization. Applying both those transformations in conjunction enables compiler optimizations that otherwise are missed if only one these transformations is applied in isolation.

```
1 while( length > 1 ) {
2     mid = lowerLimit + ( length / 2 );
3     if(A[mid] > quarry)
4         upperLimit = mid;
5     else
6         lowerLimit = mid;
7     length = upperLimit - lowerLimit;
8 }
9 return lowerLimit;
```

Listing 1. The binary search loop in XSBench.

As a motivating example consider the program in Listing 1. This is the binary search loop, taken from the HPC mini application XSBench [9], which illustrates how information lost through a control-flow merge prevents the compiler from performing additional optimizations. In Line 7, if the compiler could infer that the condition in Line 3 evaluated to true, thus Line 4 executed, it could avoid the subtraction operation because in this path $upperLimit$ is equal to mid, which is equal to $lowerLimit+(length/2)$, hence the subtraction result is $length/2$, i.e., $length$ is assigned $length/2$, which is pre-computed for the assignment in Line 2. Code duplication from control-flow unmerging enables this optimization opportunity by duplicating Line 7 in both branches of the if-condition. Additionally, applying loop unrolling further exposes similar optimization opportunities by expansion. In this example, our unroll and unmerge approach achieves $1.36\times$ speedup over a baseline O3 pipeline (more details in section IV). In summary, the contributions of our paper are:

- We present a new optimization approach, *unroll and unmerge (u&u)*, that performs loop unrolling and control-flow unmerging in tandem, to enable subsequent compiler optimizations that otherwise will not be enabled by performing only one of those transformations alone.
- We present a heuristic that decides on which loops to apply our optimization to increase performance while keeping code size and compile time inflation under control.
- We provide an LLVM-based implementation of our transformation, integrated in the recent version of LLVM 16, freely available to the community [10], to validate our claims by extensive experimentation using a production-level compiler.
- We perform a rigorous experimentation campaign using 16 GPU benchmarks from various application domains in HecBench [11], a benchmark suite carefully curated by the HPC community. Our extensive evaluation using CUDA implementations of those benchmarks on a state-of-the-art NVIDIA V100 GPU shows that our approach achieves speedups of up to 81%, while experimenting with different loops and unrolling factors.
- We provide in-depth analysis to explain how our transformation affects LLVM's compilation pipeline and programs' execution behaviour and derive new insight for compiler optimization on GPUs. Our analysis reveals that the observed significant performance improvements from our approach are thanks to unlocking aggressive redundant instruction elimination, crucially including data movement instructions, despite increased branch divergence in several cases.

The paper is structured as follows. Section II discusses related work. Section III presents our optimization and its implementation. Section IV shows the results of applying our optimization to a set of GPU benchmarks while Section V performs an in-depth analysis of representative results and discusses the strengths and limitations of our approach. Section VI concludes the paper and presents ideas for future work.

II. Related Work

This section briefly discusses related work and introduces necessary concepts required by the rest of our paper.

a) Loop Unrolling: Loop unrolling [12]–[14], a compiler optimization technique, reduces the overhead of loop control structures, such as the costs of incrementing the loop counter, evaluating the loop condition, and branching. Depending on the characteristics of a loop, there are different methods [15] of unrolling it, such as full, partial, or runtime. Loop unrolling exposes additional opportunities for instruction-level parallelism (ILP), allowing modern, superscalar, out-of-order processors to execute multiple instructions for enhanced performance. Modern compilers often employ heuristics or profile-guided information [16] to determine when and how much to unroll a loop. Our approach employs the concepts of loop unrolling and unfolds control flow graph branching inside the body of the loop, which we refer to as unmerging.

b) Function Inlining: Function inlining [17], a compiler transformation with similar motivation, enables subsequent optimizations like constant propagation or dead code elimination. Inlining can increase code size significantly and potentially reduce the application performance by negatively impacting instruction cache hit rates. Therefore, advanced heuristics have been implemented to decide whether inlining a function is profitable [18]–[21]. There are also approaches that use profile-guided information to decide whether it is worthwhile to inline a function. The heuristics used by these techniques can be modified and applied in our proposed method as our method can potentially increase code size.

c) Branch Divergence: GPUs employ a hierarchical Single Instruction Multiple Threads (SIMT) execution model. Threads are grouped in blocks and within a block, threads are structured in warps. Warps are the scheduling unit of execution. Threads in the warp execute instructions in lockstep, i.e., all threads execute the same instruction in parallel. If threads in the warp evaluate a condition differently and branch to different basic blocks, execution is considered diverged. During diverged execution, threads on the taken path execute first, while the remaining threads are idle until scheduled next. Branch divergence results in increased computation latency due to under-utilization of the hardware resources.

Branch divergence negatively impacts performance, thus several techniques propose mechanisms to mitigate it, such as dynamic warp formation [22], branch re-convergence [2]–[4], and predication [23]. By contrast, our approach aggressively

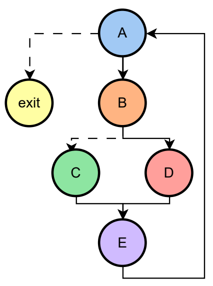


Fig. 1. A simple loop.

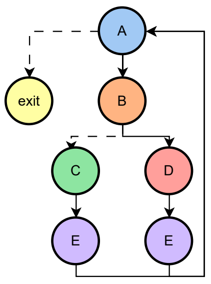


Fig. 2. Loop with unmerged control flow.

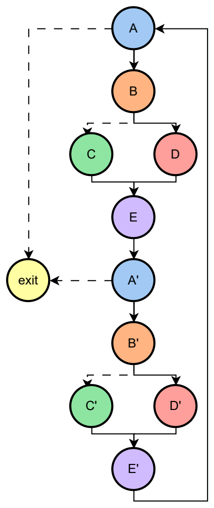


Fig. 3. Loop unrolled.

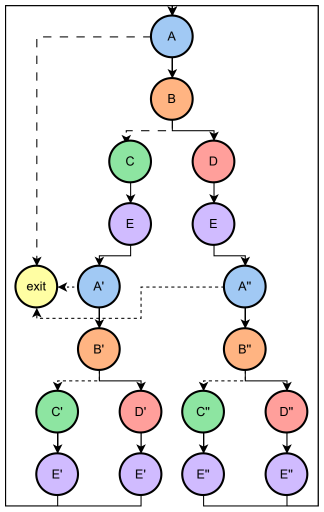


Fig. 4. Loop both unrolled and unmerged.

expands instead of merging control-flow, through unmerging and unrolling transformation, hence it possibly induces deeper diverged execution paths that increase branch divergence.

Other techniques, such as tail merging [2], branch fusion [3] and control-flow melding [4], identify same/similar instructions inside branches and meld basic blocks in order to reduce branch divergence. Our approach follows the opposite direction, as it includes additional branching to provide opportunities to the compiler for aggressive optimization. Despite the possible increases in branch divergence our evaluation in section IV indicates performance benefits.

d) Code Duplication: Code duplication techniques [8], [24] enable compiler optimizations in CPUs. The approaches target GraalVM [25] on Java/JVM benchmarks which have different requirements as they use just-in-time compilation. In [8] a duplication technique similar to the proposed one, unmerges only the direct successor basic block instead of a sequence of basic blocks as we do. Unmerging only the direct successor block limits the code size increase and branch-divergence inefficiencies. However, our evaluation shows more optimization opportunities to appear when unmerging more than just the direct successor block.

In [24] they present a technique for unrolling only the hot or fast path of a loop. They apply their technique to non-counted loops, i.e., loops for which the iteration count can neither be determined at compile time nor at run time. We think that we can apply similar ideas to our approach where we do not unroll and unmerge the whole loop, but those parts that are amenable for further optimizations or frequently executed.

III. Design And Implementation

A. High Level Design

Our optimization consists of two steps: loop unrolling and subsequent unmerging of control-flow paths. Figure 1 shows the control flow of a simple loop containing a branch in its body. Control flow branches from basic block B to either basic block C or D. The control flow is subsequently merged in basic block E. At E, it is not possible to know whether C or D was executed previously, as E has two predecessors.

1) Control Flow Unmerging: Our approach aggressively eliminates such merging blocks. Briefly, *unmerging* duplicates the basic block E, and sets the successor of C and D to separate E basic blocks. We apply *unmerging* in the example Figure 1 and, Figure 2 depicts the transformed control flow.

The resulting control flow enables subsequent compiler optimizations to eliminate redundancies, specifically: (i) in both E basic blocks it is known whether C or D was previously executed, and (ii) it is known how the condition in B was evaluated based on the branching decision leading to its path. This knowledge enables subsequent optimizations, such as constant folding and read elimination.

2) Loop Unrolling: Figure 3 shows the loop of Figure 1 after unrolling it with an unroll factor of 2, i.e., the new loop body now consists of two copies of the original loop body. Unrolling a loop is done in three steps: (1) create copies of the basic blocks of the loop; (2) rewire the backedge from E to A to point to the copied loop header A'; (3) rewire the backedge from E' to point to the original loop header A instead of A'. Similarly to unmerging the control flow, more information becomes available. For the copied basic blocks, the compiler knows that A, B and E have executed before, hence it can use this information to apply extra optimizations, such as read elimination and strength reduction.

3) Unrolling And Unmerging (u&u): Figure 4, illustrates the transformed control flow of our proposed approach, where the control-flow in the unrolled loop is also unmerged. The transformation provides additional information to the compiler which is absent when solely using either unmerging or unrolling. Specifically, *u&u* provides additional information to identify (post-)dominators and extract information from the evaluated conditions. Our approach aggressively duplicates the entire path leading to the initial loop header, instead of only duplicating the direct merge basic block. This design decision reveals as many previously obscured (partial) redundancies for elimination as possible in subsequent optimizations.

The aggressive duplication may result in substantial code size increase, however, this is rare. Our evaluation results empirically indicate that subsequent optimizations enabled by

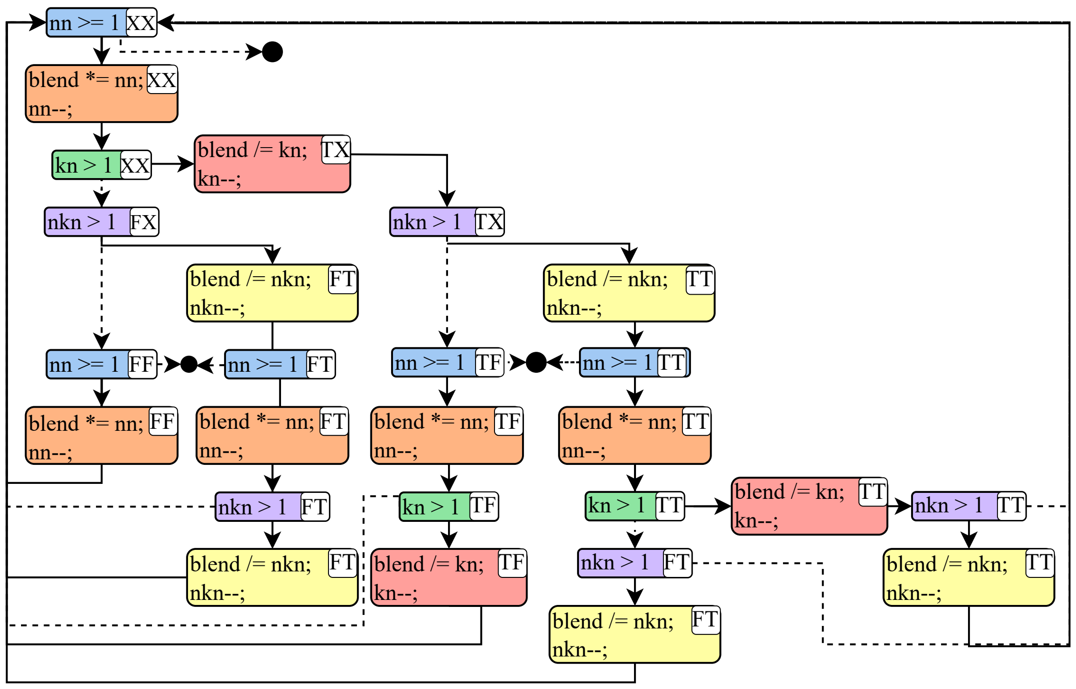


Fig. 5. Loop in *bezier-surface* after applying our optimization with unrolling factor of 2 and running $-O3$ LLVM default optimizations. Solid lines represent `True` or unconditional edges, while dotted lines represent `False` ones. Black dots represent loop exits. Each node in the graph includes a label indicating the provenance information on the two conditions of the node. Character `X` indicates no information for the condition at the respective position of the character, `T` or `F` indicate that the condition has evaluated to `True` or `False`, respectively.

```
while(nn >= 1) {
  blend *= nn;
  nn--;
  if(kn > 1) {
    blend /= kn;
    kn--;
  }
  if(nkn > 1)  {
    blend /= nkn;
    nkn--;
  }
}
```

Listing 2. Loop in the *bezier-surface* application.

our approach result in dead code elimination opportunities from conditions that evaluate always true or false. Nevertheless, the worst case code size increase can be analytically computed. If p is the number of paths, s the size of the unmerged loop, and u the unroll factor, the size of the loop after unrolling and unmerging is $f(p, s, u) = \sum_{i=0}^{u-1}(p^i * s)$.

Unrolling and unmerging creates longer continuous execution paths in the control flow of the application enabling the compiler to apply additional optimizations. Besides the code size increase, these longer execution paths can limit the parallelism efficiency of GPU warps since the optimization aggressively removes merging blocks. Although counter-intuitive, our approach benefits performance despite this inefficiency, by reducing the number of miscellaneous instructions and increasing instructions per cycle (IPC) overall.

B. Detailed Example

Listing 2 shows a code snippet of a loop of the *bezier-surface* program to motivate the effectiveness of the designed optimization. Once conditions $kn > 1$ (Line 4) and $nkn > 1$ (Line 8) are `False`, they remain `False`, since kn and nkn are only modified when these conditions are `True`. Consequently, the compiler can avoid re-evaluating them, once the compiler identifies these values to be set to `False`. Our transformation empowers the compiler to perform such transformations, reducing execution time by 30% in this example.

Figure 5 shows the CFG of the loop after applying LLVM compiler optimizations including ours. LLVM automatically removed the redundant condition checks in unrolled code due to our transformation. Essentially, *u&u* creates 4 duplications of the original loop, since the 2 conditions in the loop define 4 different, unmerged paths with an unrolling factor of 2. In 3 of those duplications, the compiler eliminates the redundant evaluation of conditions. For example, if $kn > 1$ is `False`, evaluating $kn > 1$ is avoided in unrolled code (see nodes in Figure 5 with labels `FT` or `FF`). Analogously, if $nkn > 1$ is `False`, re-evaluating this condition in unrolled code is eliminated (see nodes marked with labels `TF` or `FF`). If both of these conditions are `False` (node with label `FF`), then the unrolled loop body consists only of the initial assignments avoiding the evaluation of both conditions. Note that those eliminations mitigate the code size increase from unrolling.

Nevertheless, it is possible to further reduce code size by redirecting the edge that targets the loop header label with `TT`

TABLE I
OVERVIEW OF BENCHMARKS (L = #LOOPS, %C = % OF TIME SPENT IN COMPUTE KERNELS, RSD = RELATIVE STANDARD DEVIATION).

Name	**Category**	**Command Line**	**L**	**% C**	**Baseline Mean (ms) ± RSD**	**Heuristic Mean (ms) ± RSD**
bezier-surface	CV and image processing	-n 4096	3	67.18%	78.75 ± 4.07%	66.16 ± 3.47%
bn	Machine learning	result	11	97.28%	1322.07 ± 1.52%	1042.53 ± 1.47%
bspline-vgh	Simulation	*no CLI input*	1	11.69%	137.49 ± 6.46%	77.04 ± 6.64%
ccs	Bioinformatics	-t 0.9 -i Data_Constant_100_1 _bicluster.txt -m 50 -p 1 -g 100.0 -r 100	9	99.98%	1629.32 ± 0.2%	3462.97 ± 0.02%
clink	Machine learning	*no CLI input*	5	27.23%	1058.04 ± 0.12%	870.99 ± 0.03%
complex	Math	10000000 1000	1	99.91%	2199.23 ± 0.26%	2730.95 ± 0.1%
contract	Data compression/reduction	64 5	46	99.61%	5470.18 ± 0.76%	6570.50 ± 0.11%
coordinates	Geographic information system	10000000 1000	6	92.63%	744.91 ± 0.06%	744.33 ± 0.07%
haccmk	Simulation	2000	1	99.83%	5823.46 ± 0.01%	5105.43 ± 0.01%
lavaMD	Simulation	-boxes1d 30	1	66.52%	33.28 ± 0.08%	30.65 ± 0.07%
libor	Finance	100	8	99.99%	1422.20 ± 0.07%	1345.94 ± 0.03%
mandelbrot	CV and image processing	100	1	14.47%	15.60 ± 0.08%	13.21 ± 0.07%
qtclustering	Machine learning	*no CLI input*	19	99.14%	176.3 ± 1.9%	165.92 ± 0.2%
quicksort	Sorting	10 2048 2048	15	80.36%	518.19 ± 0.29%	502.68 ± 0.28%
rainflow	Simulation	100000 100	3	99.55%	7395.28 ± 0.18%	7089.02 ± 0.17%
XSBench	Simulation	-s small -m event	210	87.62%	137.21 ± 0.12%	121.72 ± 0.14%

to point to the initial loop header labeled with XX, since there are no redundancies to eliminate on that path. Though there are more ways to hand-optimize this specific loop, it is important that *u&u* automatically creates optimization opportunities.

C. Implementation

We implement our transformation as a LoopPass in LLVM, invoked on all loops discoverable through LLVM's loop analysis. For each loop, our pass inspects its body and identifies whether there exists control flow to unmerge. In the absence of such control flow, it immediately returns without applying any transformation. When it applies, our pass uses the existing loop information to re-flow transformed loops by duplicating basic blocks and rewiring control flow[1] on the new paths.

We modify the default pass manager and add our *unroll and unmerging* pass in the Clang optimization pipeline. The user, using command line parameters, can either include or exclude loops from our optimization pass. The pass assigns consistent, deterministic unique ids to loops in the code, which the user can use to transform specific loops. In the case of loop nests, the pass by default unrolls only the outer loop while inner loops are only unmerged, not unrolled. Unrolling only the outer loop still leads to optimization opportunities in the inner loops and we purposefully do not unroll inner loops to capture such effects. However, using configuration options, the pass is capable of unrolling nested loops as well.

Finally, we handle all loop types besides the ones containing *convergent* operations, such as `syncthreads()`, which cannot be made control-flow dependent. We use LLVM's convergence analysis to find out if a loop contains convergent operations, and do not apply our pass to such loops, as we cannot safely duplicate convergent operations.

We develop a heuristic that decides whether to *u&u* a loop and the unrolling factor. The heuristic takes into account the size of the loop s and its number of paths p to estimate the size of the loop after unrolling and unmerging, assuming a specific unroll factor u, using the formula $f(p, s, u)$ from Section III-A. The size of the loop is calculated by using LLVM's cost model, simlilarly to LLVM's loop unroll pass. The heuristic decides to *u&u* a loop if there is an unrolling factor $u' \geq 2$ such that $f(p, s, u') < c$, where c is a parameter of the heuristic. If such a u' exists, the heuristic chooses the largest $u' \leq u_{max}$ where u_{max} is another parameter of the heuristic. Enforcing a maximum unroll factor u_{max} and an upper bound c on the size of the loop limits potential instruction cache issues while keeping into check code size and compile time increases. For nested loops, we first try to *u&u* innermost loops and only *u&u* outer loops if the heuristic decided not to *u&u* any of its inner loops. The heuristic refrains from *u&u* loops annotated with explicit unrolling pragmas to not interfere with user-requested optimizations.

IV. EVALUATION

A. Hardware and Software Setup

We evaluated our transformation on a machine with an NVIDIA V100 GPU on a set of 16 applications from HeCBench [11], which contains a collection of GPU benchmarks curated by the HPC community. We deploy the CUDA implementations of those benchmarks, compiled with the modified Clang/LLVM version 16.0.1 including our transformation. Clang/LLVM internally uses CUDA version 11.6.1 toolchain

[1]We handle *phi* instructions due to the SSA [26] LLVM IR by unraveling control-/data-flow to match the re-flowed loop or replacing them if control decays to a single predecessor block.

to create a machine executable. Table I gives an overview of the applications we experiment with.

B. Methodology

We perform measurements with 5 different configurations of the compilation optimization pipeline for comparison:

- *baseline*: compiled with default `-O3`.
- *unroll*: `-O3` + just loop unrolling (without unmerging).
- *unmerge*: `-O3` + just unmerging (without loop unrolling).
- *u&u*: `-O3` + loop unrolling and unmerging.
- *u&u heuristic*: `-O3` + heuristic *u&u* ($c = 1024$, $u_{max} = 8$).

The baseline configuration measures performance by using compiler defaults without any modification. For *unroll* we use LLVM's existing loop unroll pass, for *unmerge* we use our pass setting the unroll factor to 1. Comparing *unroll*, *unmerge* against *u&u* shows the effects of applying those transformations in isolation, by contrast to our combined approach, to help reasoning that the effect of applying both transformations is greater than applying just one of them. All *u&u*, *unroll*, and *unmerge* are added at the same position, early in the compilation pipeline to maximize subsequent optimizations enabled through those transformations. Our pass enables subsequent optimizations to apply, hence a late position in the pipeline is ineffective, unless the whole pipeline is restarted which would untenably increase compilation time.

We use Nvprof [27] to perform our measurements, which reports the time spent for memory transfers (device to host and vice versa) and the time spent executing GPU kernels. Our transformation does not affect memory transfer time, so we use the sum of all GPU kernel execution times to calculate speedup. Table I shows the fraction of time spent in GPU kernels[2] to weigh in for calculating Amdahl's law end-to-end speedup. We apply our pass to one loop at a time to precisely measure the effect of applying our pass and comparators to each loop. For each application, loop, unrolling factor, and configuration, we run the application 20 times and report the median of the sum of all kernel execution times.

Note that the baseline compiler also performs loop unrolling if it deems it worthwhile. Besides full loop unrolling, it can also partially or runtime unroll a loop depending on its tripcount. Those ideas could also be applied to *u&u* and further improve the performance of our optimization. For full unrolling, LLVM employs a profitability based analysis which takes into account simplifications that may be performed after full unrolling. A similar approach may be helpful to help our heuristic decide whether it is profitable to *u&u* a loop.

C. Experimentation Results

We structure our evaluation into 3 research questions (RQ):

RQ1 Does *u&u* achieve speedup over `-O3` compilation?
RQ2 How does *u&u* affect compilation times and code size?
RQ3 Does *u&u* perform better than just unroll or unmerge?

We first summarize the results of our heuristic before going into details. Figures 6a, 6b and 6c show speedup, code size and compile time increase over baseline for the heuristic and for all loops and applications with different unroll factors of 2, 4, 8. Our heuristic improves performance for 13 out of 16 applications, achieving a maximum speedup of $1.81\times$ for *bspline-vgh*. In some cases, our heuristic is able to achieve higher speedup than those reported on individual loops because it applies to multiple well-performing loops choosing different, beneficial unrolling factors per-loop. The geometric means for speedup, code size and compile time increase over all applications for the heuristic are $1.05\times$, $1.7\times$ and $1.18\times$ respectively, which shows that the heuristic speeds up applications while avoiding extremes in code size and compile time.

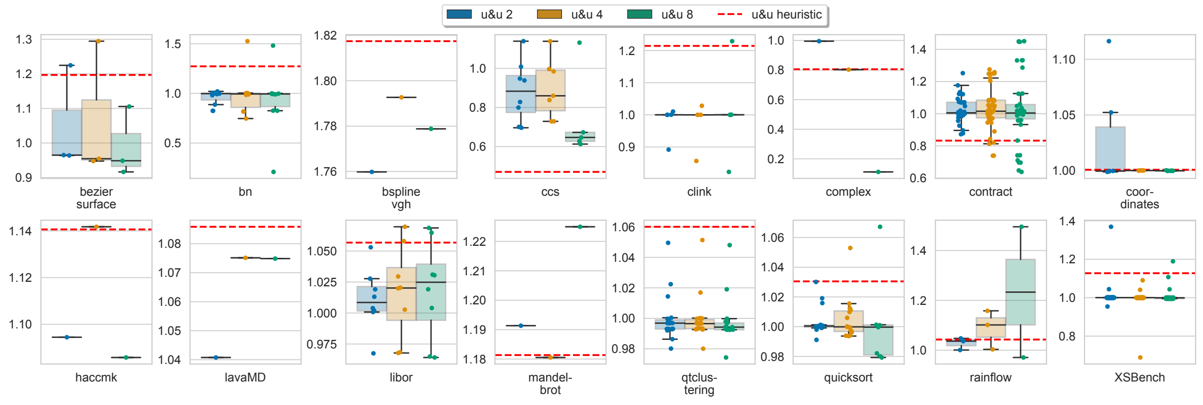
RQ1: Does *u&u* achieve speedup over `-O3` compilation? First of all, we want to find out if *u&u* outperforms the default `-O3` compilation pipeline. For all applications, except *complex*, there is at least one unroll factor for which our transformation is able to improve the execution time of one or more loops in the application. For one application, *complex*, we observe significant slowdowns that increase with the unroll factor, due to a branch that is frequently divergent leading to warp execution inefficiencies in the *u&u* version. Its maximum slowdown of $0.11\times$ occurs for an unroll factor of 8.

For brevity we will only highlight some interesting aspects. For *bezier-surface* and *rainflow*, *u&u* enables the elimination of condition checks that are either always true or false in the next iteration of the loop, depending on how the conditions were evaluated in the previous iteration. For *XSBench*, *u&u* is enabling subsequent optimizations that eliminates data movement and subtraction operations depending on how the condition inside the binary search loop (see Listing 1) was evaluated. Even though we are replacing predicated instructions by possibly divergent branches, we are able to speedup *XSBench* by up to $1.36\times$. For *coordinates*, we see a speedup of $1.11\times$ for an unroll factor of 2, but no speedup (nor slowdown) for other unrolling factors. The speedup occurs because in the baseline version the loop is fully unrolled by LLVM. When we include our pass in the pipeline, LLVM does not unroll this loop, which speeds up execution. To verify our observation, we also compiled a version by explicitly disable unrolling for this specific loop and measured similar speedup as to when applying our pass.

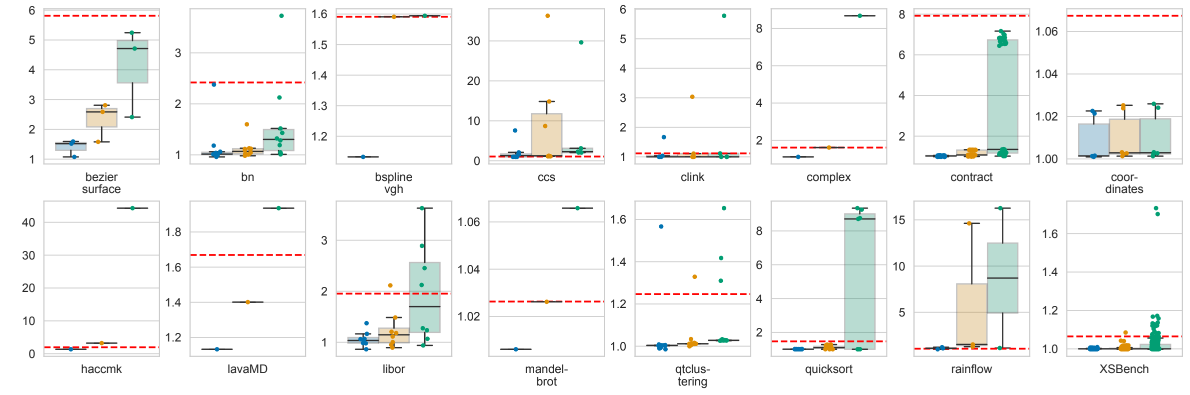
The *complex* application is an outlier experiencing high slowdowns with increased unrolling factors. These slowdowns occur because the baseline version avoids branches by using predicated instructions, while *u&u* introduces branches and increases the length of possibly divergent paths, thus serializing execution. Subsequent optimizations cannot speedup the application any further and thus we observe slowdowns.

While in most cases our heuristic is able to avoid performance degradation, there are still 3 applications where performance is noticeably reduced by *u&u*. For *ccs*, the heuristic chooses to *u&u* multiple small loops which leads to slowdown, as shown by the single loop measurements. Applying *u&u* disables beneficial runtime unrolling for those loops, which LLVM otherwise applies. While for *complex* and

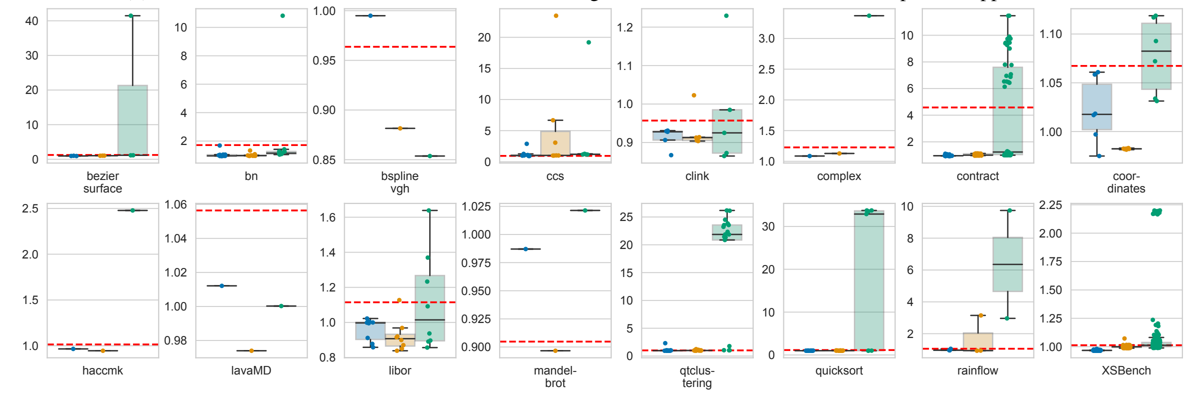
[2]The remaining time is spent performing memory transfers.



(a) Speedup of *u&u* with different unrolling factors over baseline for each loop in all applications.



(b) Code size increase of *u&u* with different unrolling factors over baseline for each loop in all applications.



(c) Compile time increase of *u&u* with different unrolling factors over baseline for each loop in all applications.

Fig. 6. Speedup, code size and compile time increase for *u&u*.

contract the heuristic does not avoid slowdown, it contains it by choosing a small unrolling factor. To avoid degradation, *u&u* can be enhanced with runtime unrolling or the heuristic should better predict possible optimizations and avoid the existing *u&u* transformation that bars them. By selectively unmerging only those parts of the loop that enable subsequent optimizations, the heuristic could further improve performance by reducing inefficiencies in warp execution and instruction caching. We believe these extensions to our approach will lead to performance increases for those remaining three applications.

Note that each data point in Figure 6a represents the median of 20 runs. Table I shows the relative standard deviation (RSD) for the baseline and heuristic measurements[3]. The variability observed is low across the board. The variability for *bezier-surface* and *bspline-vgh* is relatively higher (4.07% and 6.46% respectively), but speedup from *u&u* is statistically significant, as it is much higher than the RSDs and we also report medians, which removes outliers.

RQ2: How does *u&u* affect compilation times and code size? With our second research question, we investigate how code size and compilation time change when *u&u* is performed.

[3]To ensure stable measurements and avoid hardware induced noise, we use *nvidia-smi* (NVIDIA's system management interface) to verify GPUs are at the highest performance setting (maximum clock frequency), with no adaptive power management.

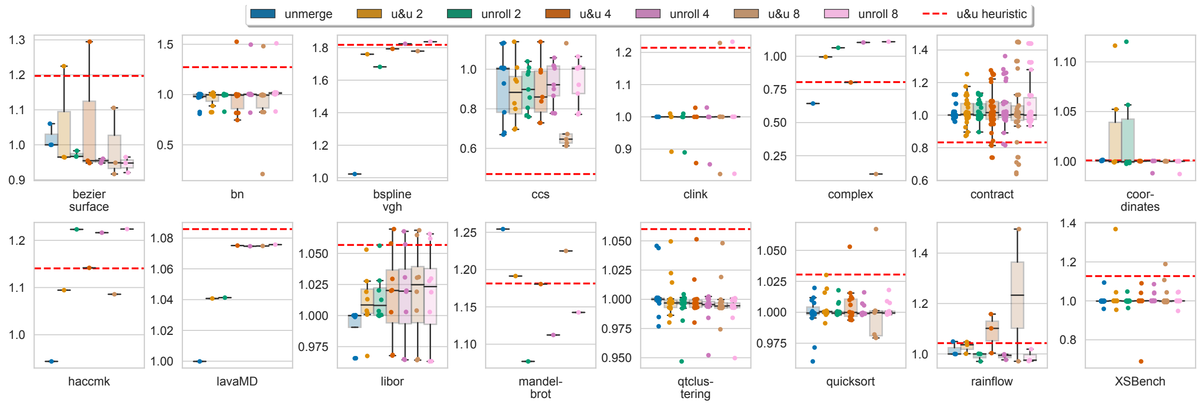


Fig. 7. Speedup achieved by *u&u* per application and unroll factor vs. just applying *unroll*, or *unmerge*.

Figure 6b shows by how much code size increases with different unroll factors. It is important to note that we compared the size of the binaries, since this is the end product of compilation following backend's lowering to machine instructions. If an application is large such as *XSBench* and *quicksort*, the relative code size increase will not be large even if the size of the loop is increased by a lot. The optimized loops of *ccs*, *complex*, *haccmk*, and *rainflow* dominate the code size, thus when our optimization further increases their size we observe significant overall code size increase.

Typically, code size increases with the unroll factor, but we can also observe some irregularities. For *bspline-vgh*, the loop's trip count is 4, so the loop is fully unrolled for an unroll factor of 4 and 8, which explains why the code size is the same for an unroll factor of 4 and 8. For *ccs*, there are 4 loops for which the compilation process times out after a limit of 5 minutes, which also explains why the maximal code size increase is observed for an unroll factor of 4.

The heuristic completely avoids extreme code size increases, as seen for *ccs*, *clink*, *complex*, *quicksort* and *rainflow*, by choosing unrolling factors based on the expected size of the loop after *u&u*, excluding loops whose expected loop size is higher than the threshold. In few cases, such as *bezier-surface*, *contract* and *coordinates*, the code size increases are slightly higher than the highest single loop code size increase as the heuristic chooses to *u&u* multiple loops.

Figure 6c shows by how much the compile time increased with different unroll factors. The compile time increase is slightly slower than code size increase. For two applications, *bezier-surface* and *qtclustering*, the compile time increases are much higher than the code size increase. Most of the time is spent in the *IPSCCPPass* (interprocedural constant propagation pass), i.e., 86% for *bezier-surface* and 72% for *qtclustering*. While most of the compile time is not spent within our pass, the time spent in other passes is increased, as they have to process the duplicated code our pass generates.

Just as for code size, the heuristic completely avoids all cases of extremely high compile time. The highest compile time increase of $4.58\times$ for the heuristic is for *contract*, because the heuristic decides to *u&u* many of its loops. Lower unrolling factors in those cases reduce compile time while still enabling performance benefits from *u&u*.

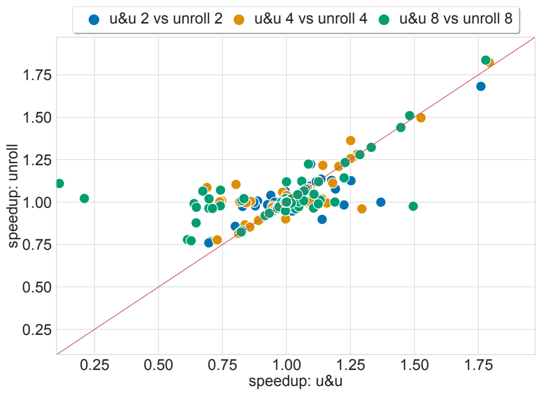
While in many cases, compile time and code size increase is moderate, there are some outliers. This is to be expected, because, as discussed in Section III, the size of the loop increases exponentially with the unroll factor when our transformation is applied. A remedy, as already mentioned to increase speedup, unmerging only those control-flow merges that lead to subsequent optimization opportunities, could also keep code size and compile time under control.

RQ3: Does *u&u* perform better than just unroll or unmerge? Our transformation performs unrolling and unmerging together. We explore the speedup from just unrolling or just unmerging, to contrast it with their combined application in our approach. Figure 7 compares *u&u* and *unroll* with unroll factors of 2, 4, and 8, and *unmerge* for all loops.

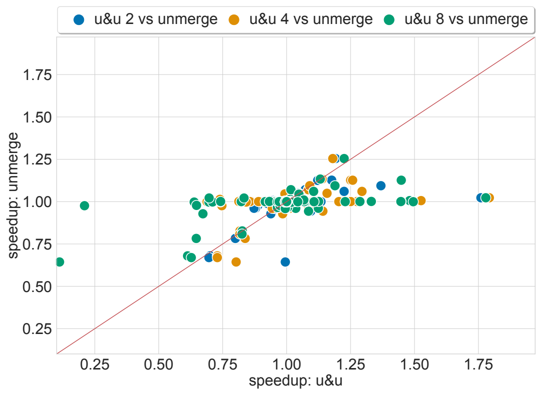
Our *u&u* method achieves higher speedups than *unroll* and *unmerge* for most applications. Only for *mandelbrot*, *unmerge* achieves higher speedup than both *unroll* and *u&u*, although *u&u* performs better than *unroll*. The worst performing *u&u* benchmark is *complex*, and it leads to significant performance degradation compared to *unroll* or *unmerge*. In some benchmarks, *u&u* shows only slightly higher speedup such as for *ccs*, whereas in few cases *unroll*'s speedup is slightly higher, such as for *haccmk*.

Following, we examine more closely on benchmarks that *u&u* achieves the highest speedup. For *bezier-surface* and *rainflow*, *u&u* eliminates condition checks in iteration $i + 1$ of a loop by knowing how the conditions were evaluated in iteration i, thus the higher speedup than just unrolling. For *rainflow*, *u&u* also eliminates load instructions. As we discussed earlier throughout the text, for *XSBench*, *u&u* eliminates both data movements and subtraction operations, depending on how the condition inside the binary search loop is evaluated. For *haccmk*, the speedups achieved by *unroll* are slightly higher than *u&u*, due to an increasing number of stalls related to instruction fetching for *u&u*.

Figures 8a and 8b compare *u&u* with *unroll* and with *unmerge* respectively. Each data point represents one loop. The x-axis shows the speedup obtained by applying *u&u* to



(a) *u&u* vs *unroll*.



(b) *u&u* vs *unmerge*.

Fig. 8. Speedup for all loops.

a single loop, whereas the y-axis the speedup by applying *unroll*, or *unmerge* respectively on the same loop. Loops on the diagonal red line have the same speedup for both *u&u* and *unroll* (*unmerge*). Loops above the diagonal have greater speedup from applying *unroll* (*unmerge*) alone. Conversely, loops below the diagonal have greater speedup thanks to *u&u*.

Observing Figure 8a, there are several loops below that diagonal that *u&u* optimizes when *unroll* fails or even slows them down, across different unroll factors. A large number of loops is on or near the diagonal, where speedup from *u&u* and *unroll* is similar. In rare occasions *u&u* leads to significant slowdown whereas *unroll* does not.

The highest unroll factor of 8 shows both greater speedup and greatest slowdown for loops that only benefit from *u&u* but not *unroll*. This is due to exponential increase on code size by *u&u*, when it is not mitigated by subsequent optimization, resulting in instruction cache or warp execution inefficiencies. Nevertheless, moderate unroll factors of 2 or 4 in *u&u* avoid severe slowdown while achieving noticeable speedup.

Solely applying *unmerge* fails to achieve speedup for the majority of the loops as depicted in Figure 8b. The majority of the points are either below or align with the diagonal and therefore strongly indicate that *unmerge* is typically ineffective unless composed with unrolling to result in benefits.

```
while( length > 1 ) {
  mid = lowerLimit + ( length / 2 );
  if(A[mid] > quarry)
    upperLimit = mid;
  else
    lowerLimit = mid;
  length = upperLimit - lowerLimit;
}
return lowerLimit;
```

Listing 3. Binary search loop in XSBench.

V. In-Depth Analysis

In this section we analyze in-depth certain applications to highlight the strengths and weaknesses of our approach. We use Nvprof [27] to collect GPU hardware performance counters to identify the root causes of the observed trade-offs. For *XSBench* we report how our optimization eliminates computation instructions, while for *rainflow* it eliminates load instructions, among others, which shows that both compute-intensive and memory-intensive workloads profit from it.

XSBench: Listing 3 shows the C++ code of the binary search loop in *XSBench*. Our transformation applied on this loop results in speedups of up to 36%. Listing 4 shows the original NVPTX code, NVIDIA's assembly language, generated for the code presented in Listing 3. The assembly code includes predicate-based select `selp` instructions[4]. The instruction `selp.b64 dst, src1, src2, cond` store to the destination register `dst` either the value of register `src1` or `src2` depending on the predicate flag `cond`. The `selp` instruction at Line 8 represents the assignment `lowerLimit = mid` which is only performed if `A[mid]>quarry`.

Listing 5 shows part of the equivalent assembly code after our transformation has applied with an unroll factor of 2. Specifically, it shows the part executed when two iterations of the loop are executed and the condition `A[mid]>quarry` is true in both iterations. We observe several differences: (1) conditionally executed jumps replace `selp` instructions, which happens because, as we can see in Figure 4, the depth of nested branches increases; (2) the subtraction is eliminated in our version, because `upperLimit - lowerLimit` is `length/2` if `A[mid]>quarry` is true; (3) there is only one `mov` instruction in the *u&u* version, for this path which represents 2 iterations of the loop, compared to 4 `selp` instructions, executing in 2 iterations of the baseline version. The value `mid` does not move into `upperLimit`, as it is unused outside of the loop, and the register that holds `length/2` is reused. But it has to be moved in Line 2, to update `upperLimit` if `A[mid]>quarry` is false. The hardware performance counters reflect those changes. Although *warp_execution_efficiency*[5] goes down from 62.88% to 18.91%, *inst_misc*[6] is reduced by 55%, while IPC increases by 1.88×, after applying *u&u* with an unroll factor of 8.

[4]These are similar to CMOV in the x86 assembly language.
[5]Ratio of the average active threads per warp to the maximum supported.
[6]Number of miscellaneous instructions executed by non-predicated threads.

```
$L__BB7_2:
  shr.u64      %rd13, %rd17, 1;
  add.s64      %rd14, %rd13, %rd20;
  shl.b64      %rd15, %rd14, 3;
  add.s64      %rd16, %rd10, %rd15;
  ld.f64       %fd2, [%rd16];
  setp.gt.f64 %p2, %fd2, %fd1;
  selp.b64     %rd19, %rd14, %rd19, %p2;
  selp.b64     %rd20, %rd20, %rd14, %p2;
  sub.s64      %rd17, %rd19, %rd20;
  setp.gt.s64 %p3, %rd17, 1;
  @%p3 bra     $L__BB7_2;
$L__BB7_3:
  st.param.b64 [func_retval0+0], %rd20;
  ret;
```

Listing 4. PTX for the XSBench binary search loop.

```
$L__BB7_2:
  mov.u64    %rd4, %rd33;
  shr.u64    %rd22, %rd34, 1;
  add.s64    %rd6, %rd22, %rd28;
  shl.b64    %rd23, %rd6, 3;
  add.s64    %rd7, %rd19, %rd23;
  ld.f64     %fd2, [%rd7];
  setp.leu.f64    %p2, %fd2, %fd1;
  @%p2 bra    $L__BB7_8;
  setp.lt.u64   %p5, %rd34, 4;
  @%p5 bra    $L__BB7_6;
  shr.u64    %rd34, %rd34, 2;
  add.s64    %rd33, %rd34, %rd28;
  shl.b64    %rd26, %rd33, 3;
  add.s64    %rd27, %rd19, %rd26;
  ld.f64   %fd4, [%rd27];
  setp.leu.f64     %p6, %fd4, %fd1;
  @%p6 bra    $L__BB7_7;
  bra.uni    $L__BB7_5;
$L__BB7_5:
  setp.gt.s64    %p7, %rd34, 1;
  @%p7 bra    $L__BB7_2;
  bra.uni    $L__BB7_6;
```

Listing 5. PTX after *u&u*, true-true path.

Rainflow: In Listing 6, we can see one of two loops in the *rainflow* application, that *u&u* is able to optimize[7]. Let the following conditions (i) a := `x[i] > y[j]` (ii) b := `x[i] > x[i+1]` (iii) c := `x[i] < y[j]` (iv) d := `x[i] < x[i+1]` (v) e := `y[++j] = x[i]`. In this example, having knowledge about how the conditions were evaluated previously, allows the elimination of loads, and condition checks. There are 7 paths in this loop, abe, $a\bar{b}cde$, $a\bar{b}c\bar{d}$, $a\bar{b}\bar{c}$, $\bar{a}cde$, $\bar{a}c\bar{d}$ and $\bar{a}\,\bar{c}$. We observe the following: (1) If a or c are true, then `x[i+1]` must be loaded to check if b or d are true. If a or c evaluate to true, the load of `x[i]` can be eliminated in the next iteration since it is equal to `x[i+1]` from the previous iteration. (2) On paths starting with $a\bar{b}$, c is always false, because $a \Rightarrow \bar{c}$. (3) If e has executed, `y[j]` is equal to `x[i]`, thus loading `y[j]` in the next iteration is redundant. (4) After the abe path, in the next iteration, a is false and c must be true, so the condition needs to check only if d is true, since b and e imply c in this next iteration. (5) Analogously, on

[7]Note that x and y are marked with the `__restrict__` keyword.

```
for (int i = 1; i < len-1; i++) {
  if ((x[i] > y[j] && x[i] > x[i+1]) ||
      (x[i] < y[j] && x[i] < x[i+1])) {
      y[++j] = x[i];
  }
}
```

Listing 6. Loop in the *rainflow* application optimized by *u&u*.

```
1 while (n > 0) {
2   if (n & 1) {
3     a_new *= a;
4     c_new = c_new * a + c;
5   }
6   c *= (a + 1);
7   a *= a;
8   n >>= 1;
9 }
```

Listing 7. Loop in *complex* slowed down by *u&u*.

the paths that end with cde, in the next iteration a must be true (and therefore c is false), hence the condition only needs to check if b is true. These are all partial redundancies, i.e., they exist only in some, but not all, paths and therefore cannot be eliminated only by unrolling since it lacks information about how the conditions were evaluated in the previous iteration. However, *u&u* makes these partial redundancies explicit for removal. These optimizations are reflected in the performance metrics. For an unroll factor of 4, *inst_misc* is reduced by 77%, *inst_control*[8] by 45% and *gld_throughput*[9] by 17%. While *warp_execution_efficiency* goes down from 28.42% to 13.79%, IPC is increased by 2.04×.

Complex: Listing 7 shows the loop in *complex* that significantly slows down by *u&u*. We trace the reason for this slowdown to branch divergence. The variable `n` is a function parameter, that is set to the global thread id, i.e., `threadIdx.x + blockIdx.x * blockDim.x`. The condition in Line 2 checks if `n` is an odd number. Because the thread ids are unique within a warp, there is a high chance that the branch will diverge. For an unroll factor of 8, the *warp_execution_efficiency* goes down from 100% to 19.37%. At the same time *stall_inst_fetch*[10] goes up from 3.72% to 79.59%. Unrolling and unmerging increases the penalty incurred by branch divergence because threads stay divergent longer due to the longer paths before reconvergence. There are also no optimizations from *u&u* to offset this. We could avoid such cases by employing a taint analysis that checks whether a condition depends on the values of e.g., `threadIdx`, and not apply our transformation in these cases.

VI. Conclusion

This work introduces a novel and aggressive transformation to unroll loops in the code and duplicate control flow by unmerging to expand conditional execution flow. Subsequent

[8]Number of control-flow instructions executed by non-predicated threads.
[9]Global memory load throughput.
[10]Percentage of stalls occurring because of delayed instruction fetching.

compiler optimization exploits the transformed control flow to enable optimizations that cannot be enabled without our transformation. We prototype our optimization in LLVM, and evaluate its performance on a collection of GPU benchmarks. Although our optimization induces branch divergence in SIMT architectures, such as the GPUs we use, the technique can achieve performance improvements of up to 81%.

As future work we plan to extend our method with partial unmerging and partial/runtime unrolling. Also, we will improve our heuristic's cost model to predict the benefits of subsequent optimization, including divergence analysis [28], [29], to better identify loops for applying our approach.

Acknowledgment

This work was prepared by LLNL under Contract DE-AC52-07NA27344 (LLNL-CONF-849354) and was supported by the LLNL LDRD Program under Project No. 21-ERD-018. The views and opinions of the authors do not necessarily reflect those of the U.S. government or Lawrence Livermore National Security, LLC neither of whom nor any of their employees make any endorsements, express or implied warranties or representations or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of the information contained herein. The United States Government retains, and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.

Appendix

A. Abstract

Our artifact provides everything that is necessary to reproduce Table 1 and Figures 6 - 8. This includes a fork of LLVM that contains our code to perform unrolling and unmerging, the benchmarks used in our paper and scripts to run the measurements, and create the plots.

B. Artifact Check-List (Meta-Information)

- **Algorithm:** Unmerging and Unrolling.
- **Program:** All 16 benchmarks used in this paper are included in the artifact.
- **Compilation:** A C/C++ compiler is required to build our fork of LLVM that is then used to build the benchmarks.
- **Run-time environment:** A system that has docker or podman installed. Otherwise, a system with cuda, cmake, ninja, a C/C++ compiler and python3.
- **Hardware:** An NVIDIA GPU is required. We used an NVIDIA V100 GPU.
- **Metrics:** Execution times, measured as the sum of all kernel execution times, compilation time, and code size.
- **Output:** Figures 6a, 6b, 6c, 7, 8a, 8b and Table 1.
- **How much disk space required (approximately)?:** At least 45GB.
- **How much time is needed to prepare workflow (approximately)?:** Approximately 1 hour.
- **How much time is needed to complete experiments (approximately)?:** Approximately 50 hours.
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: LLVM is licensed under Apache License v2.0 with LLVM Exceptions.**
- **Data licenses (if publicly available)?: HeCBench is licensed under the BSD 3-Clause License. XSBench is licensed under the MIT License.**
- **Archived (provide DOI)?: 10.5281/zenodo.10205186**

C. Description

1) How Delivered: Our artifact can be downloaded from Zenodo: https://zenodo.org/doi/10.5281/zenodo.10205186.

2) Hardware Dependencies: An NVIDIA GPU is required.

3) Software Dependencies: Our artifact can either be run using container technologies, e.g. docker, podman or singularity. Otherwise, the following software has to be installed: cuda, cmake, ninja, a C/C++ compiler, python3.

D. Installation

Download the zip from https://zenodo.org/doi/10.5281/zenodo.10205186, unpack it, and step into the unpacked folder `cgo-uu` afterward. The unpacked folder contains a `README.md` (also

```
tar -xzvf cgo-uu.tar.gz
cd cgo-uu
```

`README.pdf`) that gives detailed instructions and help on potential issues. We provide a Dockerfile that can be used to build an image that contains all dependencies that are needed to run our artifact. The *README.md* contains detailed instructions on how to build and run the docker image.

Alternatively, the artifact can also be run without using container technologies such as Docker. In this case, one has to install all dependencies on their own machine. We provide a script that checks whether all necessary software dependencies can be found:

```
bash scripts/check_dependencies.sh
```

After installing all necessary dependencies or building the image and starting a container, a version of LLVM has to be built that contains our Unrolling and Unmerging pass:

```
cd scripts
. build_llvm.sh
```

E. Experiment Workflow

Once LLVM has been built, it is possible to start the measurements. Execute the following two commands to run all the measurements and create Table 1 and Figures 6, 7 and 8.

```
bash run_all.sh
bash plot/generate_all.sh
```

Alternatively, it is possible to run measurements separately:

```
# Run the measurements for uu-heuristic
bash run_heuristic.sh
# Create Table 1
bash plot/create_table.sh

# Run u&u with an unroll factor of 2
bash run_uu.sh 2
# Run u&u with an unroll factor of 4
bash run_uu.sh 4
# Run u&u with an unroll factor of 8
bash run_uu.sh 8
# Create Figures 6a, 6b, 6c
```

```
bash plot/fig6.sh

# Run config 'unroll' with an unroll factor of 2
bash run_unroll.sh 2
# Run config 'unroll' with an unroll factor of 4
bash run_unroll.sh 4
# Run config 'unroll' with an unroll factor of 8
bash run_unroll.sh 8

# Run config 'unmerge'
bash run_unmerge.sh

# Create Figure 7
bash plot/fig7.sh
# Create Figure 8a and 8b
bash plot/fig8.sh
```

F. Evaluation and Expected Result

After running the experiment workflow, the `cgo-uu/results/` folder contains Table 1 (`table1.txt`) and Figures 6a, 6b, 6c, 7, 8a, and 8b (`fig6a.pdf`, `fig6b.pdf`, `fig6c.pdf`, `fig7.pdf`, `fig8a.pdf`, `fig8b.pdf`).

G. Notes

Refer to the `README.md` for in-depth instructions on how to use the artifact, how to use the provided Dockerfile, and for help with potential issues that may arise.

References

[1] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 2004, pp. 75–88. [Online]. Available: https://doi.org/10.1109/CGO.2004.1281665

[2] W. Chen, B. Li, and R. Gupta, "Code compaction of matching single-entry multiple-exit regions," in *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, ser. Lecture Notes in Computer Science, R. Cousot, Ed., vol. 2694. Springer, 2003, pp. 401–417. [Online]. Available: https://doi.org/10.1007/3-540-44898-5_23

[3] B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. M. Jr., "Divergence analysis and optimizations," in *2011 International Conference on Parallel Architectures and Compilation Techniques, PACT 2011, Galveston, TX, USA, October 10-14, 2011*, L. Rauchwerger and V. Sarkar, Eds. IEEE Computer Society, 2011, pp. 320–329. [Online]. Available: https://doi.org/10.1109/PACT.2011.63

[4] C. Saumya, K. Sundararajah, and M. Kulkarni, "DARM: control-flow melding for SIMT thread divergence reduction," in *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2022, Seoul, Korea, Republic of, April 2-6, 2022*, J. W. Lee, S. Hack, and T. Shpeisman, Eds. IEEE, 2022, pp. 1–13. [Online]. Available: https://doi.org/10.1109/CGO53902.2022.9741285

[5] F. Mueller and D. B. Whalley, "Avoiding unconditional jumps by code replication," in *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI), San Francisco, California, USA, June 17-19, 1992*, S. I. Feldman and R. L. Wexelblat, Eds. ACM, 1992, pp. 322–330. [Online]. Available: https://doi.org/10.1145/143095.143144

[6] F. Mueller and D. B. Whalley, "Avoiding conditional branches by code replication," in *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI), La Jolla, California, USA, June 18-21, 1995*, D. W. Wall, Ed. ACM, 1995, pp. 56–66. [Online]. Available: https://doi.org/10.1145/207110.207116

[7] R. Bodík, R. Gupta, and M. L. Soffa, "Complete removal of redundant expressions (with retrospective)," in *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation 1979-1999, A Selection*, K. S. McKinley, Ed. ACM, 1998, pp. 596–611. [Online]. Available: https://doi.org/10.1145/989393.989453

[8] D. Leopoldseder, L. Stadler, T. Würthinger, J. Eisl, D. Simon, and H. Mössenböck, "Dominance-based duplication simulation (DBDS): code duplication to enable compiler optimizations," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, Vösendorf / Vienna, Austria, February 24-28, 2018*, J. Knoop, M. Schordan, T. Johnson, and M. F. P. O'Boyle, Eds. ACM, 2018, pp. 126–137. [Online]. Available: https://doi.org/10.1145/3168811

[9] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "XSBench - the development and verification of a performance abstraction for Monte Carlo reactor analysis," in *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, Kyoto, 2014.

[10] A. Murtovi, G. Georgakoudis, K. Parasyris, C. Liao, I. Laguna, and B. Steffen, "Artifact for "Enhancing Performance Through Control-flow Unmerging and Loop Unrolling on GPUs" (CGO 2024)," Nov. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.10205186

[11] Z. Jin, "Hecbench," https://github.com/zjin-lcf/HeCBench/, accessed: 2023-09-01.

[12] J. W. Davidson and S. Jinturkar, "An aggressive approach to loop unrolling," Citeseer, Tech. Rep., 1995.

[13] G. S. Murthy, M. Ravishankar, M. M. Baskaran, and P. Sadayappan, "Optimal loop unrolling for GPGPU programs," in *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*. IEEE, 2010, pp. 1–11. [Online]. Available: https://doi.org/10.1109/IPDPS.2010.5470423

[14] M. Booshehri, A. Malekpour, and P. Luksch, "An improving method for loop unrolling," *arXiv preprint arXiv:1308.0698*, 2013. [Online]. Available: https://doi.org/10.48550/arXiv.1308.0698

[15] M. Kruse, "Loop transformations using clang's abstract syntax tree," in *ICPP Workshops 2021: 50th International Conference on Parallel Processing, Virtual Event / Lemont (near Chicago), IL, USA, August 9-12, 2021*, F. Silla and O. Marques, Eds. ACM, 2021, pp. 21:1–21:7. [Online]. Available: https://doi.org/10.1145/3458744.3473359

[16] R. C. O. Rocha, V. Porpodas, P. Petoumenos, L. F. W. Góes, Z. Wang, M. Cole, and H. Leather, "Vectorization-aware loop unrolling with seed forwarding," in *CC '20: 29th International Conference on Compiler Construction, San Diego, CA, USA, February 22-23, 2020*, L. Pouchet and A. Jimborean, Eds. ACM, 2020, pp. 1–13. [Online]. Available: https://doi.org/10.1145/3377555.3377890

[17] T. Theodoridis, T. Grosser, and Z. Su, "Understanding and exploiting optimal function inlining," in *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds. ACM, 2022, pp. 977–989. [Online]. Available: https://doi.org/10.1145/3503222.3507744

[18] A. H. Ashouri, M. Elhoushi, Y. Hua, X. Wang, M. A. Manzoor, B. Chan, and Y. Gao, "Work-in-progress: Mlgoperf: An ML guided inliner to optimize performance," in *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2022, Shanghai, China, October 7-14, 2022*. IEEE, 2022, pp. 3–4. [Online]. Available: https://doi.org/10.1109/CASES55004.2022.00008

[19] T. Theodoridis, T. Grosser, and Z. Su, "Understanding and exploiting optimal function inlining," in *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds. ACM, 2022, pp. 977–989. [Online]. Available: https://doi.org/10.1145/3503222.3507744

[20] M. E. Weingarten, T. Theodoridis, and A. Prokopec, "Inlining-benefit prediction with interprocedural partial escape analysis," in *Proceedings of the 14th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages, VMIL 2022, Auckland, New Zealand, 5 December 2022*, C. Kotselidis and A. Prokopec, Eds. ACM, 2022, pp. 13–24. [Online]. Available: https://doi.org/10.1145/3563838.3567677

[21] S. Kulkarni, J. Cavazos, C. Wimmer, and D. Simon, "Automatic construction of inlining heuristics using machine learning," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013*. IEEE Computer Society, 2013, pp. 9:1–9:12. [Online]. Available: https://doi.org/10.1109/CGO.2013.6495004

[22] W. W. L. Fung, I. Sham, G. L. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *40th Annual IEEE/ACM International Symposium on Microarchitecture*

(MICRO-40 2007), 1-5 December 2007, Chicago, Illinois, USA. IEEE Computer Society, 2007, pp. 407–420. [Online]. Available: https://doi.org/10.1109/MICRO.2007.30

[23] J. Anantpur and R. Govindarajan, “Taming control divergence in gpus through control flow linearization,” in *Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, ser. Lecture Notes in Computer Science, A. Cohen, Ed., vol. 8409. Springer, 2014, pp. 133–153. [Online]. Available: https://doi.org/10.1007/978-3-642-54807-9_8

[24] D. Leopoldseder, R. Schatz, L. Stadler, M. Rigger, T. Würthinger, and H. Mössenböck, “Fast-path loop unrolling of non-counted loops to enable subsequent compiler optimizations,” in *Proceedings of the 15th International Conference on Managed Languages & Runtimes, ManLang 2018, Linz, Austria, September 12-14, 2018*, E. Tilevich and H. Mössenböck, Eds. ACM, 2018, pp. 2:1–2:13. [Online]. Available: https://doi.org/10.1145/3237009.3237013

[25] G. Duboscq, L. Stadler, T. Würthinger, D. Simon, C. Wimmer, and H. Mössenböck, “Graal ir: An extensible declarative intermediate representation,” in *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, 2013, pp. 1–9.

[26] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, 1991. [Online]. Available: https://doi.org/10.1145/115372.115320

[27] “Nvidia corp. 2021. nvprof: Cuda toolkit documentation.” http://docs.nvidia.com/cuda/profiler-users-guide/index.html., accessed: 2023-09-01.

[28] R. Karrenberg and S. Hack, “Improving performance of opencl on cpus,” in *Compiler Construction - 21st International Conference, CC 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, ser. Lecture Notes in Computer Science, M. F. P. O’Boyle, Ed., vol. 7210. Springer, 2012, pp. 1–20. [Online]. Available: https://doi.org/10.1007/978-3-642-28652-0_1

[29] J. Rosemann, S. Moll, and S. Hack, “An abstract interpretation for SPMD divergence on reducible control flow graphs,” *Proc. ACM Program. Lang.*, vol. 5, no. POPL, pp. 1–31, 2021. [Online]. Available: https://doi.org/10.1145/3434312