Memory-access-aware Safety and Profitability Analysis for Transformation of Accelerator-bound OpenMP Loops

ARTEM CHIKIN, Intel Corporation, Canada
TAYLOR LLOYD, Amazon, USA
JOSÉ NELSON AMARAL, University of Alberta, Canada
ETTORE TIOTTO, IBM Canada, Canada
MUHAMMAD USMAN, University of Alberta, Canada

Iteration Point Difference Analysis is a new static analysis framework that can be used to determine the memory coalescing characteristics of parallel loops that target GPU offloading and to ascertain safety and profitability of loop transformations with the goal of improving their memory access characteristics. This analysis can propagate definitions through control flow, works for non-affine expressions, and is capable of analyzing expressions that reference conditionally defined values. This analysis framework enables safe and profitable loop transformations. Experimental results demonstrate potential for dramatic performance improvements. GPU kernel execution time across the Polybench suite is improved by up to 25.5× on an Nvidia P100 with benchmark overall improvement of up to 3.2×. An opportunity detected in a SPEC ACCEL benchmark yields kernel speedup of 86.5× with a benchmark improvement of 3.3×. This work also demonstrates how architecture-aware compilers improve code portability and reduce programmer effort.

CCS Concepts: • **Computer systems organization → Single instruction, multiple data**; **Multicore architectures**; **Heterogeneous (hybrid) systems**;

Additional Key Words and Phrases: GPUs, Heterogeneous Computing, OpenMP, Loop Transformations, Loop Collapsing, Loop Interchange, Memory Coalescing, Performance Portability

ACM Reference format:
Artem Chikin, Taylor Lloyd, José Nelson Amaral, Ettore Tiotto, and Muhammad Usman. 2019. Memory-access-aware Safety and Profitability Analysis for Transformation of Accelerator-bound OpenMP Loops. *ACM Trans. Archit. Code Optim.* 16, 3, Article 30 (July 2019), 26 pages.
https://doi.org/10.1145/3333060

1 PORTABLE PERFORMANCE DEMANDS STRONGER PROGRAM ANALYSES

High-Performance Computing (HPC) demands highly expressive, scalable parallel programming models. Prescriptive models such as Open Multi-Processing (OpenMP) have increasingly become the go-to solution to achieve these goals. The paradigm shift toward heterogeneous platforms with

Authors' addresses: A. Chikin, Intel Corporation, 150 Bloor Street West, Suite 400, Toronto, ON, Canada M5S 2X9; email: artem.chikin@intel.com; T. Lloyd, Amazon, 606-1942 Westlake Ave, Seattle, WA, USA 98101; email: taylor@taylorlloyd.ca; J. N. Amaral and M. Usman, Department of Computing Science, University of Alberta, Athabasca Hall, University of Alberta, Edmonton, AB, Canada T6G 2E8; emails: {jamaral, usman}@ualberta.ca; E. Tiotto, IBM Canada, 8200 Warden Ave, Markham, ON, Canada L6G 1C7; email: etiotto@ca.ibm.com.

1544-3566/2019/07-ART30
https://doi.org/10.1145/3333060

accelerator devices adds an additional dimension to the portable code generation problem. The OpenMP 4.0 standard allows programmers to offload regions of code for execution on a coprocessor, making no assumptions about its memory or execution model [26]. Accelerator architectures operate on different assumptions of memory layout and locality, each requiring highly specialized program code.

Common pitfalls in GPU programming can be avoided when generating code from high-level languages through code analysis and transformation that would be difficult in lower-level programming models. For instance, *coalescing of memory accesses* leads to higher performance when threads in a warp access memory locations that map to few cache lines so these accesses can be satisfied with fewer requests to the memory subsystem. A naive mapping of parallel loop nests to data-parallel code often results in non-coalesced accesses. A sufficiently capable optimizing compiler must be able to detect such code patterns and be able to reshape loop nests such that the resulting mapping exhibits better memory access characteristics.

This article introduces Iteration Point Difference Analysis (IPDA), which is able to capture OpenMP parallel loop access stride information and enable safety and profitability analyses that guide automatic interchange and collapse of loop nests. Loop-nest reshaping, informed by the results of IPDA, can yield dramatic performance improvements, demonstrated with an achieved speedup of up to 25.5× for a loop nest in the Polybench benchmark suite and up to 86.5× for a loop nest from the SPEC ACCEL suite. This new analysis builds on the ideas proposed in a novel static analysis framework called Arithmetic Control Form (ACF) [16]. ACF is unique in its ability to handle control-flow conditionals symbolically and statically determine access stride patterns in CUDA code. The IPDA framework introduces the ability to discover cross-iteration symbolic differences statically. Stronger data-access analysis also enables the generation of efficient parallel code *without* requiring programmers to provide hints to the compiler. The performance study in this article demonstrates that removing `collapse` clauses from OpenMP 4.x programs can increase performance across diverse accelerator architectures if the compiler is capable of inferring the profitability of loop collapsing automatically.

This article also demonstrates the versatility of the analysis framework by building a loop-dependence test based on IPDA: the IPDA Test is introduced as an Data Dependence Graph (DDG) pruning algorithm that enables safety proofs on more loop nests than originally possible in the experimental compiler setup. This article makes the following contributions: i) IPDA —A static analysis framework for the computation of cross-iteration symbolic differences among expressions contained in loops; ii) A novel DDG pruning technique based on constructing inequality proofs over symbolic iteration-point algebraic difference equations; iii) A static analysis that identifies inter-thread memory access stride of addressing expressions contained in parallel OpenMP loops; iv) Safety and profitability analyses to guide loop collapse and interchange transformations on OpenMP parallel loops intended for GPU execution.

2 PROGRAMMING MODEL, PROGRAM ANALYSIS, AND EXECUTION PLATFORM

GPUs are composed of a large number of Streaming Multiprocessors (SMs),[1] each capable of executing thousands of threads in parallel. Such massive parallelism requires a strict Single Instruction, Multiple Thread (SIMT) data-parallel programming model to achieve performance. The Nvidia V100 is a state-of-the-art Nvidia GPU for high-performance computing composed of 80 SMs. Each SM can issue an instruction for 128 threads per cycle [22]. The V100 has enough resources to maintain the state of thousands of threads, which gives each SM the ability to context-switch between threads with zero penalty—a key instruction-latency hiding mechanism.

[1]Nvidia-specific terms such as *warp* and *SM* are used throughout this article for the sake of clarity and consistency.

GPU threads are grouped into *warps*: all the threads that comprise a warp execute either the *same* instruction in lock-step or no instruction at all. Lock-step execution reduces the overhead of scheduling work across a large number of threads. Threads are further grouped into thread blocks. All threads within a thread block must execute on a single SM. They can both share intermediate results via access to SM's shared memory banks and synchronize their execution. Threads from different thread blocks have no means of direct communication or synchronization.

2.1 Memory Coalescing

In a typical data-parallel kernel, thread identifiers are used in memory-access addressing expressions to load/store the data items for each thread. The number of memory requests issued by a warp in a given cycle can be as large as the number of threads in a warp (32 in current architectures), because the threads belonging to it execute the same instruction simultaneously. The GPU global memory subsystem has a limited amount of bandwidth available. As a means to reduce the overall number of requests, the GPU *coalesces* multiple same-cycle accesses to memory within the same cache line into a single request. Coalescing memory accesses into fewer requests can dramatically improve memory throughput, because no thread in a warp can continue execution until the memory accesses of all threads have completed. Each global memory request requires hundreds of cycles to be completed; thus, structuring GPU programs to avoid non-coalesced memory accesses is paramount for performance [2].

2.2 Arithmetic Control Form Analysis

The ACF static analysis framework, introduced by Lloyd et al., is a way to capture linear and non-linear relationships between program statements [16].[2] ACF's main approach is to combine data and control flow by computing symbolic values for expressions of interest. Similarly to the work by Ferriere and Stoutchinin on ϕ-nodes [30] and prior efforts in if-conversion [18], ACF converts conditionally executed statements into predicated statements, capturing definitions across all potential traces through the program. Resulting ACF expressions consist of binary operations on constants and symbols representing compile-time unknowns.

In the context of data-parallel programs, ACF's key strength lies in its ability to compute an algebraic difference on the symbolic representation of a statement. For instance, consider a statement $\mathcal{S}$ that is executed by different threads, and assume that $\mathcal{S}$ contains an addressing expression `A[f(i)]`, where `i` is the identifier of a thread executing the code. ACF constructs an algebraic expression for the difference between the symbolic value of the function `f` computed by two distinct threads. Then, by substituting actual constant thread identifiers into symbolic expressions, ACF can determine the memory-access stride between threads by solving the difference to a constant.

ACF replaces variable references with their dominating definitions wherever possible during the construction of symbolic expressions. ACF can perform this replacement without any additional considerations for potential performance impact of this replacement, because ACF expressions are symbolic and are not actual Intermediate Representation (IR) of the program that will undergo transformation and code-generation.

2.3 OpenMP and Accelerator Programming

Directive-based programming models abstract accelerator hardware specifics from the developer. Being platform-agnostic, these models promise code portability across existing and future accelerators. The programmer specifies a `target` region, directing the compiler to offload the region to an accelerator device. The target region directive is annotated with data-transfer `map` clauses that indicate which arrays must be transferred to and from the device data environment. Full assortment

[2] ACF source code is available online [1].

of OpenMP parallelism constructs are supported inside `target` regions; however, task-level parallelism maps poorly to data-parallel devices such as GPUs. Thus, performance considerations limit the expression of parallelism in OpenMP for execution in GPUs to `parallel` and loop constructs.

OpenMP can express three levels of parallelism inside a `target` region: the `teams` construct declares a region of code to be executed by a league of threads, the `parallel` construct declares a task to be executed in parallel by threads within a league, and the `simd` construct declares vector-based execution of a loop. The first two provide a natural mapping to the GPU's notions of thread blocks and threads, respectively. Both `teams` and `parallel` constructs have associated loop work-distribution clauses: `teams distribute` and `parallel for`.

2.4 OpenMP 4.x GPU Code Generation

As implemented in OpenMP 4 for LLVM/Clang as well as in the IBM XL C/C++/Fortran compilers, `target` regions are outlined into separate procedures. The outlined procedure is cloned into two versions: a device version and a host fallback version. CPU code is generated for the fallback variant. A kernel suitable for GPU execution is generated for the device version. To best take advantage of the GPU, data-parallel code is generated in place of parallel loops. The resulting device kernel is translated into Nvidia's *Parallel Thread Execution* (PTX) pseudo-assembly language, using Nvidia's proprietary PTXAS assembler. The host code that previously contained the target region is rewritten to invoke the outlined device kernel through a runtime method call. The runtime performs the required setup and data transfer. Finally, the GPU runtime compiles the PTX code into machine instructions and launches kernel execution.

3 LOOP ITERATION POINT ALGEBRAIC DIFFERENCES

The Iteration Point Difference Analysis can symbolically calculate the difference in the expression's value across iterations of a loop and is especially useful for analysis of induction-value-dependent addressing expressions. A compiler can use the results of IPDA to make decisions regarding both safety and profitability of classical loop transformations. IPDA can improve the generation of code that will execute either in the CPU or the GPU.

IPDA uses an ACF-like approach to compute the loop access stride of an addressing expression. While ACF relies strictly on the presence of a direct source of thread-dependent behavior in the expression, IPDA uses the induction variables to examine iteration-point differences. In its evaluation prototype, ACF is applied to CUDA programs and is limited to stride-access analysis and branch-divergence detection on explicitly data-parallel programs. In that prototype, results are strictly used to advise CUDA programmers about potential opportunities for performance improvement.

Given a thread-dependent memory-addressing expression, IPDA computes the cross-iteration access stride. For instance, consider the code snippet in Figure 1.

As described in Section 2.2, and demonstrated in Figure 2, conditional expressions are encoded into the symbolic values as a sum of products of predicates multiplied by the value they imply. The symbolic expression computed for the address expression of the memory reference `B[idx]` in line 8 is:

$$\begin{aligned} IPD(\texttt{B[idx]}) = {} & (\texttt{[i]} < 32) \times (\texttt{[\&B]} + 8 \times (64 + \texttt{[i]})) \\ & + (\texttt{[i]} >= 32) \times (\texttt{[\&B]} + 8 \times \texttt{[i]}), \end{aligned}$$

where references to `idx` were replaced with their definition in terms of `i`, which is the source of induction-variable-dependent behavior. Symbolic propagation allows the analysis to compute the cross-iteration memory access stride by substituting constant parameters in place of induction

```
#define TSIZE 64
for (int i = 0; i < N; ++i) {
  int idx = 0;
  if (i < (TSIZE / 2))
    idx = TSIZE + i
  else
    idx = i
  B[idx] = foo()
}
```

Fig. 1. Example loop to be analyzed by IPDA with a conditionally defined indexing expression.

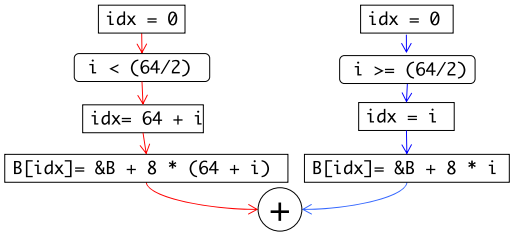

Fig. 2. Pictorial representation of ACF predicate encoding into a symbolic value.

variable identifiers and performing algebraic simplification:

$$\begin{aligned} IPD_1(\texttt{B[idx]}) - IPD_0(\texttt{B[idx]}) &= (1 < 32) \times (\texttt{[\&B]} + 8 \times 65) + (1 >= 32) \times (\texttt{[\&B]} + 8 \times 1) \\ &\quad -(0 < 32) \times (\texttt{[\&B]} + 8 \times 64) + (0 >= 32) \times (\texttt{[\&B]} + 8 \times 0) \\ &= (\texttt{[\&B]} + 520) - (\texttt{[\&B]} + 512) \\ &= 8. \end{aligned}$$

Inter-thread memory access stride is determined to be 8 bytes. In this example, constant iteration values of 0 and 1 are used. The actual analysis computes this difference for a sufficiently large number of iterations to arrive at a memory-access stride description. For example, when applying the analysis to calculate the inter-thread access stride of a GPU parallel loop, a number of iterations equal to the GPU thread-block size is tested. Existing OpenMP GPU runtimes select fixed-sized thread-block sizes based on the target GPU architecture (e.g., 128 for Pascal [17]).

A major strength of the IPDA framework is its reliance on symbolic value computation, which makes it highly independent of the transformation phase ordering: the analysis is likely to produce equally accurate results regardless of the current state of the loop code (e.g., before or after loop-invariant code motion). This has the effect of not only increasing the overall analysis accuracy but also its applicability at different stages of the compilation process.

Furthermore, IPDA's innovations make ideas first proposed in ACF relevant for non-data-parallel programs. Detection of induction-variable-dependent behavior is also particularly useful for the analysis of parallel OpenMP loops, because in such loops different iterations might be scheduled to be executed by different threads, affecting cache behavior.

3.1 Focusing on Loop-specific Analysis Demands

The transformation of loop nests often requires an analysis that can compute an induction-value-dependent difference between *distinct* addressing expressions computed at different iteration points of a loop nest. In contrast, the original design of ACF was intended only to compute the differences between the same expression as evaluated by multiple threads to detect divergent behavior. For example, let E_s be the expression used to compute the address of the source of a

loop-carried dependence relation that exists in the compiler's DDG and let E_t be the expression for the target of the same dependence. In many loops, E_s and E_t are similar enough that the difference between the ACF symbolic representations of E_t and E_s produces a simplified ΔE expression that yields useful information about potentially overlapping access ranges of the two statements. Such information can often enable a multitude of compiler transformations previously prevented by a conservative or insufficiently capable safety analysis. Moreover, for the cases where the symbolic difference simplification framework does not provide information to increase the precision of the dependence relations, it does not affect the soundness of the results. In the case of a ΔE expression that could not be simplified or does not provide meaningful access range insights, the dependence is left as-is in the DDG.

4 SYMBOLIC REPRESENTATION

IPDA computes the algebraic difference of expression instances as accessed in different loop iterations. It models each access as a tree of symbolic values consisting of constants, statically unknown values, and operators. The symbolic representation for an expression is constructed in a way that lends easily into computing differences across iterations. For instance, whenever possible, a load is replaced directly with its reaching definition.

An addressing expression of an arbitrary level of indirection is representable symbolically in a fashion similar to that of building an AST by traversing each index operation and creating offset addition operators for element accesses. Consider a data access expressed in C such as `A[4].x[2]`—the symbolic-expression tree is built by taking sums of each index operation (three in this case), each of which are offsets to the specified element.

Whenever there are multiple reaching defining expressions for a variable that is used in an IPDA address expression, a separate term is created for the IPDA expression for each reaching definition. Each term is multiplied by the set of predicates, extracted from the conditional statements along the path where the reaching definition lies, that must be true for that definition to reach the IPDA address expression. During the execution of the program, only one such path can be executed, and therefore only one such set of predicates can be true.

4.1 Algebraic Difference Cancellation

The following are the steps taken to compute a difference between two address expressions, which are now represented as canonicalized symbolic-expression trees:

(1) A new binary tree root is created, joining the two trees with a subtraction operator.
(2) The canonical sum-of-products form allows for a simple way to split each of the two subtraction operands into a series of constituting product summands to identify common expressions on both sides that can be cancelled.
(3) Once the cancelled sub-expressions are removed from the summand lists, the remaining expressions are rebuilt back into a canonical-form tree that is further simplified by applying various common expression-factoring techniques.

If the resulting tree has been resolved to a constant value, then IPDA returns the number of requests to global memory that an expression will incur.

4.2 Compiler Implementation

The IPDA analysis framework was implemented in the IBM XL C/C++/Fortran compiler's optimizer component. The backbone of the framework are the symbolic expression builder and simplifier components.

The builder constructs symbolic expressions for statements in the compiler's IR on-demand (e.g., when queried for an indexing expression of a memory access). The compiler's data-flow and control-flow graphs are used together to propagate dominating definitions where possible and create symbolic unknown symbols otherwise. Symbolic representation of conditionally defined values occurs when constructing a symbolic form for a value whose definition corresponds to a ϕ node in the data-flow graph. In this case, the framework computes the predicates of all the predecessor blocks of the ϕ and creates an expression tree consisting of the sum of predicates multiplied by their corresponding definition. The resulting symbolic value is a binary operation tree with constants and symbolic unknowns at the leafs. The builder also allows the framework to manipulate existing and construct arbitrary new symbolic expressions from scratch and based on existing IPDA values. For example, the framework's user may wish to manually create a difference of two existing symbolic values and pass the result on to the simplifier.

The symbolic expression simplifier is a custom-built collection of algorithms that implements basic algebraic rules and applies them to reduce a given IPDA expression. To ease the simplifier's job, symbolic expressions are first converted into a canonical sum-of-products form. The simplification algorithms range from capturing simple constant-value operations, such as division by 1 or multiplication by 0, to a fixed-point search for more complex constant-folding opportunities (e.g., $(2 + x) \times 4 \rightarrow 8 \times 4x$). Special-purpose routines are implemented in the framework to handle reduction of difference expressions. These routines canonicalize the difference operands and attempt to cancel out the operands' common sub-expressions. Difference reduction is also a fixed-point algorithm.

The aforementioned components are further abstracted via purpose-specific analyses. For example, the memory access coalescing analysis, described in Section 6, is simply queried with a load/store of interest. The analysis interacts with the IPDA framework: building symbolic expressions, instantiating iteration-point specific symbolic values, creating difference expressions, passing them on to the symbolic simplifier, and capturing the result. Such analysis is then used by other relevant compiler components; for example, a profitability analysis for a transformation such as the ones described in Section 7 need not be aware of the IPDA framework, because they query the coalescing analysis instead.

5 DATA DEPENDENCE GRAPH PRUNING WITH ITERATION POINT DIFFERENCES

5.1 Value Range Overlap Analysis

The IPDA test constructs address value ranges that encompass the full scope of memory locations accessed by a potential dependence source and sink expressions across all dimensions of the iteration space. Consider a potential dependence in a given single loop where the dependence source is a store instruction to a memory location whose address is bounded by $E_{src} = [a, b]$. Similarly, the sink of the dependence is a load instruction whose address is in the range $E_{sink} = [c, d]$. If the ranges E_{src} and E_{sink} do not overlap, then IPDA determines that source and sink operations do not create a dependence; thus, the potential dependence is false and can be pruned from the DDG.

To determine the value range of an address expression, IPDA propagates the value ranges of individual variables up the expression trees. Let $R1 = [a, b]$ and $R2 = [c, d]$ be two ranges. The following list outlines the various operations on ranges and how IPDA evaluates them to produce a resulting range:

- $R1 + R2 = [a + c, b + d]$
- $R1 - R2 = [a - d, b - c]$
- $R1 \times R2 = [min(min(ac, ad), min(bc, bd)), max(max(ac, ad), max(bc, bd))]$
- $R1 \div R2 = (a >= 0 \wedge b >= 0) \times [a/d, b/c] + (a < 0 \vee b < 0) \times [minInt, maxInt]$

```
1 for (i=0; i<6; i++) {
2   A[(i+8)*N] = A[i*N] + x;
3   x = ...;
4 }
```

Fig. 3. Example loop array access with a potential dependence.

- $R1\%R2 = (a >= 0 \wedge c >= 0)\times[0, d]+(a < 0 \vee c < 0)\times[minInt, maxInt]$
- $R1$ *and* $R2 = [max(a, c), min(b, d)]$
- $R1$ *or* $R2 = [min(a, c), max(b, d)]$
- $R1 = | \neq | < | \leq | > | \geq R2 = [0, 1]$.

The result range computations for arithmetic operators are trivial. For Boolean *or*, the result range computation is simplified to computing a union, with the lower bound being the minimum of the operands' lower bounds and the upper bound being the maximum of the operands' upper bounds. Similarly, Boolean *and* result range computation is similar to that of computing an interval intersect.

We outline the procedure IPDA uses to verify dependencies in non-nested loops, then describe the more complex case of nested loops. Figure 4 summarizes the steps of the algorithm.

5.2 Single-loop Dependence Checking

Let i and i' be two distinct values for the induction variable of a single-nested loop. The source and sink expressions of a potential dependence are formalized as functions of the loop induction variable. IPDA constructs symbolic, canonicalized expressions for $f(i)$ and $g(i')$ for the source and sink expressions, respectively. Functions f and g map the induction variable to an interval of memory addresses that may be accessed by the source and the sink. Therefore, IPDA difference $f(i) - g(i')$ is the interval difference of memory accesses by the source and the sink. If the difference is an empty interval, then distinct ranges of memory addresses are accessed by the source and by the sink and there is no real dependence. However, if the difference is resolved to a non-empty interval, then a range overlap exists.

Consider the illustrative example in Figure 3, where `A` is an array of integers and the value of `x` is reassigned in the body of the loop. There is a potential dependence whose source is the write of an element of `A` on the left-hand side of the first statement in the loop body and whose sink is the read of an element of `A` on the left-hand side of the same statement. IPDA begins by identifying the source and the sink of a dependence, then constructs their symbolic representations as explained in Section 4 (`constructSymbolicVal` method in Figure 4). The symbolic difference $f(i) - g(i')$ is constructed and then factored such that the induction variables appear exclusively as a term of difference on each other: $(i' - i)$ (lines 6–8 in Figure 4(a)). The difference simplification and factoring of the induction variable differences constitute the following algebraic manipulations:

$$
\begin{aligned}
f(i) &= IPD(\texttt{A[(}i\texttt{+8)*N]}) = \texttt{[\&A]} + 4 \times (i + 8) \times \texttt{[N]} \\
&= \texttt{[\&A]} + (4i + 32) \times \texttt{[N]} \\
&= \texttt{[\&A]} + 4i\texttt{[N]} + 32\texttt{[N]} \\
g(i') &= IPD(\texttt{A[}i'\texttt{*N]}) = \texttt{[\&A]} + 4i'\texttt{[N]} \\
f(i) - g(i') &= (\texttt{[\&A]} + 4i\texttt{[N]} + 32\texttt{[N]}) - (\texttt{[\&A]} + 4i'\texttt{[N]}) \\
&= 4i\texttt{[N]} + 32\texttt{[N]} - 4i'\texttt{[N]} \\
&= 4\texttt{[N]}(i - i') + 32\texttt{[N]} \\
&= 4\texttt{[N]}\Delta i + 32\texttt{[N]}.
\end{aligned}
$$

```
1  Function PruneLoopDep(sourceExpr, sinkExpr, i)
2      let i' ! = i;
3      F ← constructSymbolicVal(sourceExpr);
4      G ← constructSymbolicVal(sinkExpr);
5      replaceSymbolInExpr(G, i, i');
6      diffExpr ← (F − G);
7      factoredDiff ← factorIVDelta(diffExpr, i, i');
8      LO, RO, Op ←
         splitByTopOperator(factoredDiff);
9      LOR ← computeRange(LO);
10     ROR ← computeRange(RO);
11     return verifyNoOverlap(LOR, ROR, Op);
```

(a) Single-loop dependence checking

```
1  Function PruneNestDep(sourceExpr, sinkExpr, Loops)
2      prune ← TRUE;
3      LIv ← { };
4      LIv' ← { };
5      for Loop L ∈ Loops do
6          i ← L.index;
7          let i' ! = i;
8          LIv.append(i); LIv'.append(i');
9      F ← constructSymbolicVal(sourceExpr);
10     G ← constructSymbolicVal(sinkExpr);
11     for loopIndex ∈ 0...LIv.size() do
12         replaceSymbolInExpr(G, LIv[loopIndex],
             LIv'[loopIndex]);
13     diffExpr ← (F − G);
14     // Δi_k = (i_k − i'_k)
15     factoredDiff ← factorIVDelta(diffExpr, LIv, LIv');
16     for S ∈ 𝒫(Loops) do
17         for J ∈ Loops | J ∉ S do
18             replaceSymbolInExpr(factoredDiff, Δ J.index, 0);
19         LO, RO, Op ← splitByTopOperator(factoredDiff);
20         LOR ← computeRange(LO);
21         ROR ← computeRange(RO);
22         prune &= verifyNoOverlap(LOR, ROR, Op);
23     return prune;
```

(c) Loop-nest dependence checking

Fig. 4. Data Dependency Graph pruning algorithm summary.

With the simplified symbolic difference, the analysis verifies whether the difference can possibly equal to zero. The equation is rewritten into an inequality, as follows for our example: $4\texttt{[N]}\Delta i + 32\texttt{[N]} \neq 0$. Should this equation possibly have solutions, then a dependence exists because $f(i)$ overlaps with $g(i')$, i.e., $f(i) - g(i') \neq 0$. IPDA rewrites the inequality by splitting the equation into a right operand (*RO*) and a left operand (*LO*) and isolating either into one side of the inequality (`splitByTopOperator` method in Figure 4). In particular, the three cases that are checked by splitting into the two operands are as follows:

$$\begin{aligned} RO + LO \neq 0 \quad &\Rightarrow \quad RO \neq -LO \\ RO - LO \neq 0 \quad &\Rightarrow \quad RO \neq LO \\ RO \times LO \neq 0 \quad &\Rightarrow \quad RO \neq 0 \wedge RO \neq 0. \end{aligned}$$

In the example, the inequality $4\texttt{[N]}\Delta i + 32\texttt{[N]} \neq 0$ is evidently true iff $4\texttt{[N]}\Delta i \neq -32\texttt{[N]}$. Observe that if `N` equals 0, the inequality is proven to be false and a dependence exists. Suppose `N` is the size of a dimension of the array in question; then, a value range analysis would determine that the range of `N` is, conservatively $[1, maxint]$. This fact allows IPDA to further simplify the inequality into: $\Delta i \neq -8$. By definition of normalized loops, the induction variables initialize at 0 and exclusively increment. Should IPDA be able to statically determine the upper bounds of the loop, it can substitute in ranges for induction variables i and i'. Since the range is $[0, 5]$ for i in our example, then the value range of $(i - i')$ is either $[1, 5]$ or $[-5, -1]$, by construction of i'. The range of the right-hand-side expression scalar value is $[-8, -8]$. The dependence check is then reduced to verifying whether ranges $[1, 5]$ and $[-8, -8]$ or $[-5, -1]$ and $[-8, -8]$ overlap. This verification can be performed via a simple bounds check. Arbitrary precision integers are used in our implementation

```
1 for (i1 = 0; i1 < N1; i1++) {
2   for (i2 = 0; i2 < N2; i2++) {
3     ...
4       for (in = 0; in < Nn; in++)
5         x = A[i1*N1 + ... + in*Nn]
6         A[src] = x
7     ...
8   }
9 }
```

Fig. 5. Example *n*-degree loop nest with a potential loop dependence.

to handle various *maxint* range scenarios. In the case of intra-iteration dependencies, IPDA will not be concerned, because loop-independent dependencies may be executed in parallel and are therefore not tested by the analysis.

5.3 Loop-nest Dependence Checking

When dependence relation source and sink are contained in a loop nest, the IPDA Test must ensure that their access ranges do not overlap in *any* two points of the iteration space. To do so, IPDA repeatedly applies the access overlap test described in Section 5.1 to differences across all combinations of loops that contain the dependence (`verifyNoOverlap` method in Figure 4).

IPDA collects all induction variables, along with their upper bounds whenever possible, to create a set $I = \{i_1, i_2, \ldots, i_n\}$. Another set $I' = (\{i'_1, i'_2, \ldots, i'_n\} | i'_k \neq i_k)$, holds the set of induction variables that represent arbitrary values for the induction variables in the *n*-dimensional iteration space. The subscript indicates the IV of a specific loop in the loop nest.

In the same manner that functions f and g are used in Section 5.2, let $f(i_1, i_2, \ldots, i_n)$ and $g(i'_1, i'_2, \ldots, i'_n)$ be functions that map arbitrary iteration points to the location in memory being accessed by the source and sink, respectively. IPDA constructs symbolic, canonicalized expressions for f and g, joined by a symbolic subtraction operator that represents $f - g$. This total expression is factored such that each i_k and i'_k term appears *exclusively* as a difference $\Delta i_k = (i_k - i'_k)$.

To prove that f and g do not map to the same memory location in *any two iterations*, IPDA evaluates the access range overlap across *every* dimension in the iteration space as well as every combination of dimensions. Consider the n-degree loop in Figure 5. The iteration space is composed of n axes: $i_1, i_2, \ldots i_n$. A trivial example of data dependencies arises if the source expressions access memory at A[i_1*$N_1 + \cdots + i_n$*N_n-1]. In such a case, the next iteration in the innermost loop depends on its previous iteration, but the IV value for every other loop is constant. Consider also that a source could access memory *across* dimensions, unlike the previous example where the source and sink occur only in a single dimension (the innermost loop). Since dependencies may occur across any possible combination of dimensions in a nested parallel loop, IPDA performs range-overlap analysis (Section 5.1) on each combination by *fixing* the Δi_k values to 0 for each loop at depth k, which does not participate in the dependencies for the combination.

The power-set $\mathcal{P}(D)$ is the set of all subsets of D. For each set $S \in \mathcal{P}(D)$, each $\Delta i_k \in D \wedge \Delta i_k \notin S$ indicates a loop at depth k that is fixed ($\Delta i_k = 0$), so all $\Delta i_k \in S$ identify the combination of loops in S that are part of the current iteration space being evaluated for memory-access overlap. IPDA fixes Δi_k values not in a given S by substituting a value of zero for the difference that appears as $(i_k - i'_k)$ in the symbolic expression. The resulting reduced difference expression is then evaluated with the range analysis overlap method, as discussed in Section 5.2. If and only if, for every S, the analysis is able to prove via the range overlap analysis that the difference expression is *never* equal to zero, then the dependence relation is pruned from the DDG.

```
for(j=1;j<NJ-1;++j){
  for(i=1;i<NI-1;++i){
    for(k=1;k<NK -1;++k){
      B[i*(NK*NJ)+j*NK+k]
           = foo(j,i,(CIVK+1));
    }
  }
}
```

(a) Original Source Code

```
for(CIVJ=0;CIVJ<NJ-2;++CIVJ){
  j = CIVJ+1;
  for(CIVI=0;CIVI<NI-2;++CIVI){
    i = CIVI+1;
    for(CIVK=0; CIVK<NK-2;++CIVK){
      k = CIVK+1;
      B[i*(NK*NJ)+j*NK+k]
           = foo(j,i,(CIVK+1));
    }
  }
}
```

(b) After Loop-Normalization

```
for(CIVJ=0;CIVJ<NJ-2;++CIVJ){
  j = CIVJ+1;
  for(CIVI=0;CIVI<NI-2;++CIVI){
    i = CIVI+1;
    for(CIVK=0;CIVK<NK-2;++CIVK){
      B[i*(NK*NJ)+j*NK+(CIVK+1)]
           = foo(j,i,(CIVK+1));
    }
  }
}
```

(c) After Copy-Propagation

Fig. 6. Example Loop Nest at various stages of compilation/optimization.

5.4 Symbolic Differences of Control-dependent Expressions Improve Dependence Testing

The IPDA Test's ability to incorporate conditionally defined values into symbolic expressions, means to perform algebraic simplification on differences of such expressions, and ability to propagate variable definitions to their uses across control flow distinguishes it from other symbolic analyses.

Many competing analyses require the dependence source and sink expressions to be defined in terms of the induction variables of their containing loops. However, this reliance is often broken by other compiler transformations, such as the ones that canonicalize the representation of loops. For instance, Figure 6(b) shows the normalized version of a loop. After normalization, the addressing expressions are no longer expressed in terms of the canonical loop induction variables. Some compilers may rely on expression re-materialization to obtain the expressions in terms of the canonical induction variables; however, a cost function may prevent the propagation of expressions into the body of a hot loop. The result is addressing expressions still expressed in terms of a mix of both original and canonical induction variables. Such mixed indexing expressions may stymie dependence analyses, as is the case in the example in Figure 6(c), where propagating `CIVJ+1` and `CIVI+1` to the indexing expression in place of `i` and `j` would hurt performance. IPDA's symbolic propagation of variable definitions to their references eliminates the problem.

Common loop dependence analysis algorithms have difficulty processing addressing expressions with non-constant induction-variable coefficients. Consider the loop nest in Figure 8(a), and suppose `NI`, `NJ`, and `NK` are runtime parameters. The Greatest Common Divisor (GCD) Test, for example, checks dependences by verifying that the induction variable coefficients divide the constant factor of the respective Diophantine equation [33]. This test cannot be performed if one of its operands is an unknown runtime value. Similarly, the Banerjee Test is not able to compute the coefficient sums to evaluate the constraint condition inequality [3]. However, the dependence source and sink are often likely to contain the same induction-variable coefficients, and therefore the IPDA's difference calculation engine will factor and cancel them, leaving the resulting expression in terms of the induction variables and other constants.

5.5 Prototype Implementation Demonstrates that the IPDA Test Is Essential for Safety Analysis

The IPDA Test was implemented on top of the software framework described in Section 4.2. As in LLVM, XL's loop dependence analysis is based almost entirely on the seminal work by Goff et al. [10] and includes an assortment of exact and approximate tests such as the Lamport Test [14], the GCD Test [33], the Banerjee Test [3], and the Delta Test [10]. The IPDA Test is appended as an

additional step in the DDG pruning pipeline. The test's implementation re-uses symbolic expression infrastructure, augmenting it with specific expression manipulation primitives for constructing iteration point differences and a value range analysis. As it stands, the execution of the IPDA Test is highly specific to the compiler infrastructure it is built within. Thus, the only comparison to this first prototype is the original XL loop dependence analysis, which is a mature infrastructure from a major vendor. The IPDA Test became an essential component of the safety analysis for the loop transformations described in Section 7, reducing the DDG further than the compiler's existing analyses and allowing transformations previously deemed unsafe. The techniques described here are applicable in a much wider variety of applications. We invite researchers and developers to explore those.

6 IPDA GPU GLOBAL MEMORY COALESCING ANALYSIS ON PARALLEL OPENMP LOOPS

Equipped with the ability to calculate cross-iteration access stride of addressing expressions, the compiler can infer the inter-thread access pattern of an addressing expression contained in a parallel loop. If the loop in question is destined for GPU offloading, then the inter-thread access pattern can be used to determine the coalescing characteristics of the memory access. The original ACF, as implemented in GPUCheck, utilized explicit sources of thread dependence (i.e., thread identifiers) to examine the degree of coalescing in a given memory access operation. ACF employed taint analysis, where thread identifiers and their propagated uses were marked so thread-dependent accesses to global memory could be analyzed and therefore inform the programmer of possible non-coalesced accesses. The IPDA test makes use of a similar approach but with regards to memory accesses within the body of parallel loops.

Only OpenMP loops that specify a `schedule` clause set to `static` with a compile-time-known chunk size parameter are analyzed. In practice, this restriction does not seriously limit the usefulness of the analysis, because loops without a user-specified schedule are common and can be treated as having a schedule that the compiler deems beneficial. OpenMP requires that the parallel loop counts be known at runtime. Thus, the prototype only considers regular loops. This, however, is not a limitation of the analysis itself, because cross-iteration value differences can also be computed for loop nests with irregular geometry.

Taking the schedule chunk size into account, IPDA maps the induction variable of a parallel OpenMP loop to threads and employs a similar coalescing analysis as ACF. The analysis collects load and store instructions in the body of a given loop nest that are marked as tainted and outputs the number of memory requests that are required per warp to satisfy the memory access. Each tainted access instruction in a given loop is represented as a symbolic expression E. The difference $E_t - E_0$, as computed by IPDA for thread t, indicates the memory access stride for the instruction of interest. Algebraic simplification and difference cancellation techniques outlined in Section 4.1 are applied in an attempt to simplify inter-thread difference results to constant values. For instructions where IPDA is successfully able to compute constant-value access strides, the analysis employs the algorithm outlined in Figure 7 to greedily fit the solved results into a memory request.

The IPDA test creates a list holding potential constant *symbolic* differences and keeps a count of non-constant instances. The list `Requests` indicates the overall requests to global memory in the loop body, where each individual element depicts the range of accesses that compose a single coalesced access. An insertion into `Requests` creates an access of stride *sizeof*(*accesstype*) bytes, with a limit of 256, the size of the cache line in the current generation of Nvidia cards. This approach allows the analysis to account for varying access strides within a warp. For instance, if the constant-value difference between threads 0 through 15 of the warp differs from the constant-value difference between threads 16 through 31, the coalescing analysis would still compute the correct number of required hardware requests.

```
1  Constants ← { };
2  Unknowns ← 0;
3  for instruction I ∈ loop body do
4  |  E ← symbolic(I);
5  |  for thread t ∈ num(threads) do
6  |  |  if diff.isConstant() then
7  |  |  |  Constants.append(diff);
8  |  |  else
9  |  |  |  Unknowns += 1;
10 Function numRequests(Constants, Unknown)
11 |  Requests ← { };
12 |  for c ∈ C do
13 |  |  fit ← false;
14 |  |  for r ∈ Requests do
15 |  |  |  if c ≥ r.low && c ≤ r.high then
16 |  |  |  |  fit ← true;
17 |  |  |  else if c ≥ r.high - 256 && c ≤ r.high then
18 |  |  |  |  r.low ← c;
19 |  |  |  |  fit ← true;
20 |  |  |  else if c ≤ r.low + 256 && c ≥ r.low then
21 |  |  |  |  r.high ← c + 8;
22 |  |  |  |  fit ← true;
23 |  |  if fit ≠ true then
24 |  |  |  Requests.append( (low: c, high: c+8) );
25 |  return (Requests.size, Requests.size + Unknown);
```

Fig. 7. Computing the number of coalesced accesses.

For symbolic differences that cannot be solved to a constant value, IPDA coalescing analysis conservatively assumes distinct requests. For instance, no static analysis can deduce a value that is only known at runtime. It is possible to use the IPDA framework and determine the memory stride by evaluating the pre-computed symbolic differences at runtime. Such an approach could be used in a Just-in-Time compiler or in an inspector-executor framework where one among various versions of the code is selected at runtime. Implementation and evaluation of such approaches are left for future investigation.

7 IMPROVING GPU MEMORY ACCESS PATTERNS WITH LOOP TRANSFORMATIONS

Naively translating parallel OpenMP loops directly into data-parallel code can lead to an inefficient kernel that poorly utilizes the GPU memory subsystem. Consider the OpenMP target region extracted from the GEMM benchmark from the Polybench suite shown in Figure 8(a). The iteration space of the `i` loop is first divided into chunks in conformance with the `teams distribute` construct directive, and iterations of each chunk are then scheduled to run in parallel, as prescribed by the `parallel for` construct. Each thread executing an iteration of the `i` loop sequentially executes the `j, k` loop nest. In this example, for a given memory access, the inter-thread stride is the size of each array, which result in an inter-thread stride of 4,096 bytes and none of the accesses can be coalesced.

```
1  #pragma omp target teams distribute parallel for
2  for (i = 0; i < NI; i++) {
3    for (j = 0; j < NJ; j++) {
4      C[i*NJ + j] *= BETA;
5      for (k = 0; k < NK; ++k) {
6        C[i*NJ + j] += ALPHA * A[i*NK + k]
7                       * B[k*NJ + j];
8      }
9    }
10 }
```

(a) Original benchmark source code

```
#pragma omp target teams distribute parallel for
for (c = 0; c < NI * NJ; c++) {
  i = c / NI;
  j = c % NI;
  C[i*NJ + j] *= BETA;
  for (k = 0; k < NK; ++k) {
    C[i*NJ + j] += ALPHA * A[i*NK + k]
                   * B[k*NJ + j];
  }
}
```

(b) Collapsed `i-j` nest.

Fig. 8. Example `target` region from GEMM benchmark.

Memory access patterns for a GPU kernel that uses high-dimensional data structures or otherwise non-trivial addressing expressions are often not obvious even to experienced developers and require expert knowledge of the compiler's code-generation scheme and mapping from loop-parallel to data-parallel code. A particular loop layout may also benefit one accelerator architecture over others, leading to loss of performance portability no matter which selection is made. IPDA's memory-access analyses enable and guide loop transformations that improve GPU memory utilization by increasing access coalescing.

7.1 Loop Collapse

The OpenMP `collapse(n)` clause merges `n` nested loops into a single parallel iteration space. Collapsing parallel loops for execution on a GPU has two performance-sensitive effects: the number of parallel work items to be scheduled increases, and the thread memory access pattern changes.

- **Increased parallelism**
  The total number of iterations of the collapsed parallel loop is equal to the product of the trip counts of all the loops in the collapsed nest, increasing the amount of parallelism available. Figure 9 shows an example parallel loop nest and the mapping of iterations to units of parallel work and sequential iterations.
- **Improved memory access pattern**
  Collapsing changes the inter-thread access stride, because the outer loop induction variable no longer maps to the thread identifiers. If the sequential execution order of the collapsed loops is used to determine the order of the iterations in the collapsed iteration space, then the stride of the innermost collapsed loop becomes the inter-thread access stride.

Collapsing can be beneficial even when no `collapse` clause is present. For example, in the loop shown in Figure 8(a), consecutive iterations of the `j` loop access adjacent elements of arrays `C` at line 4, `C` at line 6, and `B` at line 7 and have an inter-thread stride of 4,096 bytes with no coalescing. After collapsing the `i-j` loop nest (Figure 8(b)) all accesses are perfectly coalesced.

7.1.1 Loop Collapse Safety. In the absence of a `collapse` clause, the compiler must prove that collapsing is safe for a loop nest of depth n with a parallel outermost loop. Such a nest must satisfy the following conditions to be safely collapsed:

- It must be perfect: all statements must be inside the innermost loop.
- Loop boundaries for all loops must not change after entry into the loop nest.
- There must be no loop-carried dependencies among iterations of any of the loops in the nest.

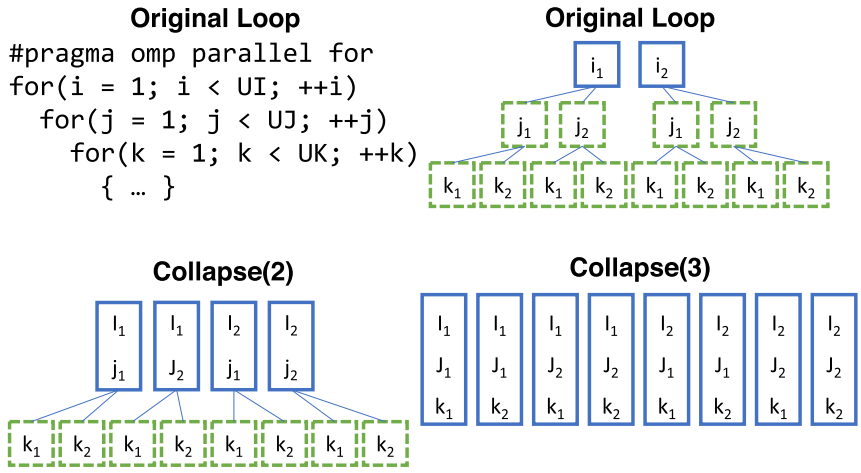

Fig. 9. Pictorial representation of a parallel loop nest's iteration space.
– Loop iterations comprising units of parallel work.
– Loop iterations executed sequentially.

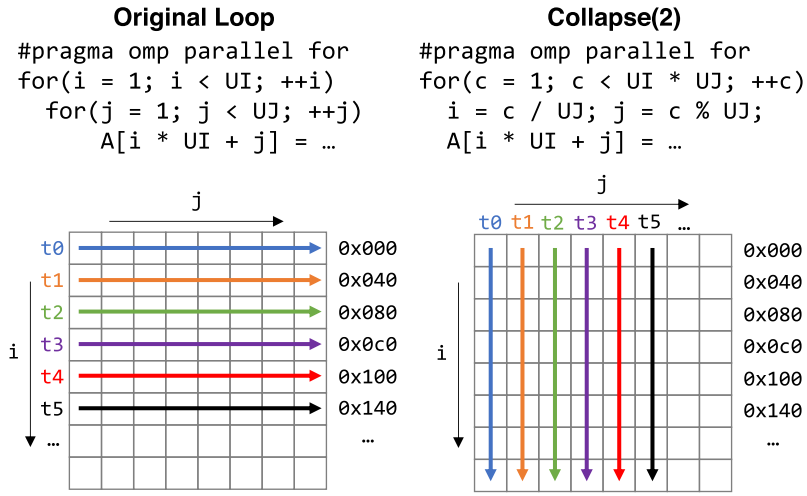

Fig. 10. Pictorial representation of a parallel loop nest's memory access pattern.

A parallel loop contained in a target region follows aliasing restrictions on data mapped into the device data environment. These restrictions often make dependence analysis feasible where it would not be for an identical loop not contained within a target region. For a given loop nest, all possible collapse depths are tested for safety. For the set of provably safe collapse depth levels, profitability analysis determines which, if any, should be performed by the compiler. Figure 10 illustrates how this collapse enables coalescing by changing the access to a two dimensional array from row-major order to column-major order.

7.1.2 Loop Collapse Profitability. The main performance benefit of loop collapse stems from improved access patterns. Thus, the reduction in the number of memory requests executed per

warp can be used to estimate the profitability of collapsing a loop nest. Using the IPDA framework, the profitability is computed as follows:

(1) Compute the number of memory requests for every access in the original loop nest.
(2) For each nest level κ, compute the number of memory requests for every access in the nest by rewriting the addressing expressions to emulate the effect of collapsing the nest to that level.
(3) A collapse is *profitable* if, for any nest level, the total number of memory requests per warp in the kernel is reduced.

These steps are performed for all collapse depth levels considered safe, and the most *profitable* level is chosen for actual code transformation. The current implementation of the analysis is focused on optimizing *only* parallel loops that target GPUs. While the results of the analysis can be adapted to guide a profitability heuristic targeting multicore CPUs, this target is not currently supported. Such a heuristic would place opposite demands on inter-thread access stride (minimize false-sharing and maximize each thread's locality).

7.2 Loop Interchange

Loop interchange is a classical loop transformation wherein the order of two iteration variables in a loop nest is exchanged. The aim of loop interchange here is to improve access coalescing by changing the mapping of loop induction variable differences to inter-thread memory access stride. For example, consider the loop nest excerpt shown in Figure 12. IPDA memory coalescing analysis finds that the resulting inter-thread stride leads to non-coalesced accesses. It also shows that loop `i`, when used as a source of thread-dependence (i.e., when iteration-point differences of the `i`-loop are treated as thread-difference values for addressing expressions), would result in perfectly coalesced loads and stores. However, the loop nest cannot be collapsed to a depth of three, as described previously, because it contains a loop-carried dependence across iterations of the `j` loop. An alternative method to achieve a mapping to data-parallel code that distributes individual iterations of the `i` loop to GPU threads, loops `i` and `j` can be first interchanged without affecting the semantics of the program. Post-interchange, the outermost two-dimensional `k-i` loop nest can be collapsed, according to the collapse profitability analysis outlined above.

7.2.1 Loop Interchange Safety. The proposed interchange applies to a loop nest where the outermost loop is labeled as parallel by a programmer and consists of moving a loop from inside that nest to the outermost level. Thus, the two outermost loops of the transformed nest can be collapsed, as described in Section 7.1. Finding loops within a given loop nest for which this transformation is legal requires finding loops within the nest that are independent. Moving an independent loop to the outermost level preserves loop-carried dependencies of all other loops in a nest. Similar to the analysis performed for loop collapse, aliasing restrictions on data mapped to the target region data environment often results in a more precise dependence analysis. In both collapse and interchange safety analysis, the compiler uses the IPDA Test to further reduce the DDG. Every loop found to be independent in a parallel loop nest is considered a candidate for loop interchange.

7.2.2 Loop Interchange Profitability. For all candidates, profitability is computed in a fashion identical to the profitability of collapsing the nest to depth of up to and including the loop in question. The profitability of loop interchange takes into account the subsequent collapse transformation, which is required to create a mapping from induction variable differences in addressing expressions to inter-thread stride that results in better coalescing characteristics. The most profitable of the candidate loops is selected for actual code transformation.

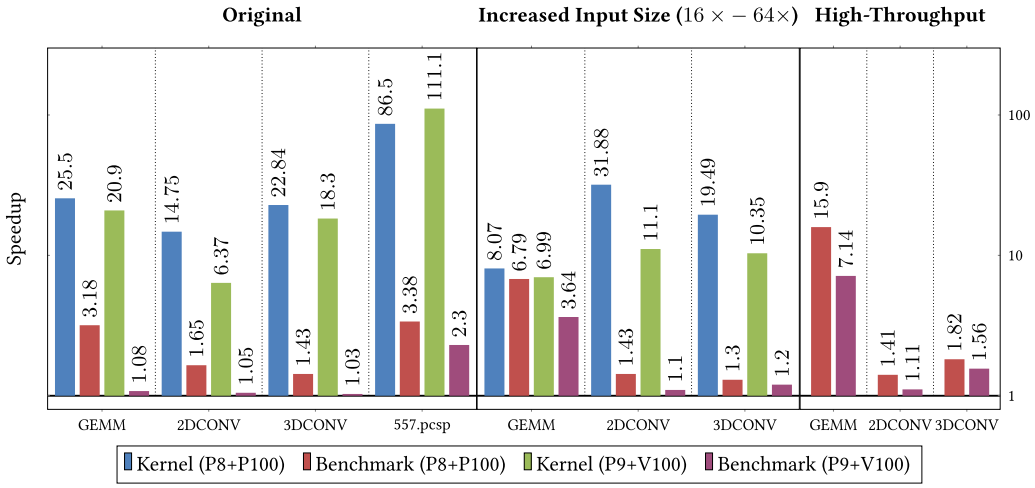

Fig. 11. Benchmark execution time speedup with automatic loop interchange and collapse-enabled compared to the default code-generation scheme.

8 EVALUATION

Coalescing of memory requests is a key performance consideration when writing or generating GPU code. It is increasingly difficult for developers to infer memory access characteristics of OpenMP GPU code as it gets translated into a data-parallel form. Moreover, explicitly committing the code to a specific access pattern that would suit a specific type of accelerator can hurt *performance portability*. Thus, high-level accelerator programming models make such considerations the prerogative of the compiler designer rather than the developers.

The efficacy of the analysis framework is evaluated in two ways: First, the potential performance impact of the two proposed loop transformations is demonstrated on a set of representative OpenMP 4.x programs. The second evaluation demonstrates that a compiler equipped with loop transformations informed by the IPDA analysis allows a higher degree of performance portability in OpenMP code. Generality and architecture-independence of OpenMP code can often be improved by removing developer-specified clauses intended as optimization prescriptions that commit generated code to specifically target GPU accelerators. The results of this evaluation demonstrate (1) that, equipped with the IPDA analysis and loop transformation framework, the compiler is able to re-capture the performance impact of such clauses by automatically performing the required optimization when generating GPU code; and (2) that omitting performance-guiding clauses results in a performance improvement when targeting other accelerator architectures.

8.1 Informed Loop Reshaping Performance Impact

The Polybench [28] and SPEC ACCEL [13] OpenMP 4 benchmark suites are used to evaluate the efficacy of the coalescing-analysis-informed loop reshaping of OpenMP 4.x parallel loop nests. Execution times are reported for two experimental setup machines: an IBM POWER8 host with an Nvidia P100 GPU and an IBM POWER9 host with an Nvidia V100 GPU accelerator. Figure 11 shows speedup of benchmarks with IPDA-guided collapse and interchange transformations enabled in the compiler. Each benchmark was executed 10 times, and the mean execution time is reported. Each presented kernel execution time is a mean of 10 average kernel execution times from each of the benchmark runs. The analysis detected transformation opportunities in three Polybench benchmarks: `MVT`, `2DCONV`, `3DCONV`, and one SPEC ACCEL benchmark: `557.pcsp`.

```
#pragma omp target teams distribute parallel for private(i,j,k,m,fac1,j1,j2)
for (k = 1; k <= gp2-2; k++) {
  for (j = 0; j <= gp1-3; j++) {
    j1 = j + 1;
    j2 = j + 2;
    for (i = 1; i <= gp0-2; i++) {
      fac1 = 1.0/lhsY[2][k][j][i];
      lhsY[3][k][j][i] = fac1*lhsY[3][k][j][i];
      lhsY[4][k][j][i] = fac1*lhsY[4][k][j][i];
      for (m = 0; m < 3; m++) {
        rhs[m][k][j][i] = fac1*rhs[m][k][j][i];
      }
      lhsY[2][k][j1][i] = lhsY[2][k][j1][i] - lhsY[1][k][j1][i] * lhsY[3][k][j][i];
      ...
    }
  }
}
```

Fig. 12. Excerpt from a target region in 557.pcsp.

The matrix multiplication GEMM contains a single 3-deep parallel loop nest. Collapsing the nest to depth 2 was found by the analysis to have the effect of transforming a kernel with completely non-coalesceable accesses into a kernel with perfectly coalesced accesses. The kernel execution time improves by a factor of 25.5× in the P100 and 20.9× in the V100. The benchmark execution time improves by a factor of 3.18× in the P100 and by 8% in the V100 machine.

2DCONV and 3DCONV convolution benchmarks contain a single parallel loop nest of depth 2 and 3, respectively. The original compiler failed to prove that the 3DCONV loop nest is free of loop-carried dependences. The IPDA Test reduced the DDG further, ultimately proving the nest as independent and safe to collapse. Profitability analysis on the two parallel nests indicated that automatic collapse would result in turning both from non-coalesceable into completely coalesced GPU kernels. The transformed code for 2DCONV improves kernel execution time by a factor of 14.75× and 6.37×, and benchmark execution time improves by a factor of 1.65× and 5% on the P100 and V100 machines, respectively. Transformed 3DCONV code results in kernel execution time speedup of 19.54× and 18.3× and benchmark execution time speedup of 2.03× and 3% on the P100 and V100 machines, respectively.

The lower speedup at the increased input size in GEMM is due to the kernel being memory-bound. At high input sizes, more parallel work is available. The runtime aims to maximize the GPU occupancy and launches kernels with a higher number of thread-blocks. Even though most accesses are coalesced, the device still spends the majority of its time waiting for memory loads/stores to be resolved, and the higher number of thread-blocks in-flight puts additional pressure on the memory subsystem. Using `nvprof` to monitor GEMM kernels shows that the GPU's memory throughput is at near 100% for most of the kernel's lifetime. A similar effect is seen for 3DCONV, but not 2DCONV, which is a less memory-heavy computation.

The SPEC ACCEL benchmark suite consists of highly tuned OpenMP code written in a way that maximizes GPU performance. Aggressive use of `collapse` clauses by the benchmark developers limits the opportunities available for automatically inferring the need to interchange or collapse loop nests. Still, our analysis identified one such opportunity. `557.pcsp`, a pentadiagonal software application, contains > 60 OpenMP `target` regions, all made up of parallel loops. One parallel loop, performing the forward elimination operation according to the Thomas algorithm, was identified by the IPDA analysis to be a safe candidate for transformation. An excerpt from the three-dimensional `k-j-i` parallel loop nest in question is shown in Figure 12. The

coalescing analysis identifies the opportunity to reduce the number of memory requests per warp present in the kernel by interchanging loops `j` and `i` and collapsing the resulting `k-i` nest. Observe that a `collapse(3)` transformation of the original code is not possible due to loop-carried dependencies across iterations of the `j` loop. Post `j-i` interchange and `k-i` collapse, the `j` loop is executed sequentially by each thread, preserving the `j`-loop-carried dependence. The IPDA Test employed by the transformation safety analysis removed a false-positive dependence from the DDG, thus proving that loop `i` is independent. The combination of transformations reshapes the resulting GPU kernel in a way that makes every memory access contained within fully coalesced. The transformed version yields an improvement in kernel execution time of 86.5× on an Nvidia P100 accelerator and 111.1× on an Nvidia V100. Across a run of the benchmark, the forward elimination kernel is invoked 401 times. The untransformed kernel's poor performance characteristics make it the biggest contributor to the overall benchmark execution time, of which it constitutes 41%. Applying the transformations described above to just 1 out of > 60 parallel loops present in the benchmark results in overall speedup of 3.38× in the P100 and 2.3× in the V100.

The dramatically lower improvement in benchmark execution time compared to kernel execution time in Polybench benchmarks stems from their small default input data sets. Thus, kernel execution time comprises only a small part of the total time, with kernel initialization and data transfer taking up most of the benchmark execution time. Higher improvements should occur with larger input sizes. For example, the `GEMM` benchmark performance improvement begins to approach the kernel speedup as input size is increased, as can be seen in Figure 11. As input size is increased, `CONV` benchmark speedups remain fixed, despite a massive improvement in kernel execution time, because memory transfer time dominates the overall execution time.

The relative decrease in overall benchmark execution time on a V100 is due to an increased overhead of kernel launch when using a Volta GPU with CUDA 9. The CUDA runtime creates a unifying context at first kernel launch to maintain state for consecutive launches. The context creation operations take up to 0.7s versus up to 0.1s for the machine with a Pascal GPU. We hypothesized that the difference is due to Volta's support for unified memory, which would require allocation of pinned host memory. The impact of this overhead is most pronounced in single-kernel programs with short runtimes. To test this hypothesis, we created high-throughput versions of the Polybench programs in which the benchmark kernel is invoked 100 times in a single launch on different data sets, amortizing the CUDA context creation overhead. Results are reported in the rightmost bar graph of Figure 11. The high-throughput version of `GEMM` confirmed our intuition with overall benchmark speedup of 15.9× and 7.14× on the P100 and V100, respectively. Memory-transfer dominated `3DCONV` also demonstrated Volta improvement start to approach Pascal figures with speedups of 1.82× and 1.56×. Based on these experiments, we believe that in real-world computation-heavy code, the improvement in overall program performance on Volta is likely to scale more closely with kernel performance improvement.

In benchmarks where no safe/profitable opportunities were found, performance remained unchanged; as such, their results are not presented in Figure 11. Benchmarks that do not contain `collapse` clauses and benchmarks that could not be compiled with the current compiler versions are not presented in Table 1. `557.pcsp` currently crashes when compiled with the ICC.

8.2 Code Portability Impact

The OpenMP `collapse` clause is prescriptive and requires the compiler to generate a specific code structure. Its goal is to exploit performance characteristics of a particular architecture. The issue is that memory access patterns that lead to maximum coalescing in a data-parallel GPU result in poor spatial locality on a multi-core CPU architecture.

Table 1. Loop collapse Clauses Were Removed from Benchmarks That Contain Them

Benchmark	Parallel OpenMP 4 loops	User-specified collapse clauses	IPDA automatic collapse transformations	No Collapse GPU Speedup	IPDA-enabled GPU Speedup	No Collapse MIC Speedup
3MM	3	3	3	0.03×	1.0×	1.26×
COVAR	3	1	1	0.98×	1.0×	1.04×
SYR2K	1	1	1	0.06×	1.0×	1.13×
SYRK	2	2	2	0.12×	1.0×	1.15×
503.postencil	1	1	0	0.33×	0.33×	1.83×
555.pseismic	14	13	0	0.09×	0.09×	1.01×
563.pswim	17	8	0	0.84×	0.84×	1.19×
570.pbt	42	36	26	0.78×	0.96×	0.97×
557.pcsp	60	60	50	0.04×	0.76×	na

This table shows the portion of the loop collapses re-discovered to be beneficial and automatically applied by the compiler. Execution ratio columns show the performance of the code stripped of collapse clauses versus the code with collapse clauses present. GPU code is executed on an Nvidia P100. The MIC (Many Integrated Core) relative execution column compares the performance of the same two versions of the kernel executed on an Intel Xeon Phi 7250 processor, compiled with ICC ver. 17.0.2.

Forcing developers to make such trade-offs in a prescriptive manner reduces *performance portability*. A capable compiler must be able to detect when such transformation is beneficial for the target architecture. This experimental evaluation provides evidence that the IPDA-based safety and profitability analyses can be a useful tool to increase performance portability and to allow program code to remain free from architecture-driven annotations.

To test this claim, we remove all collapse clauses from the Polybench and SPEC ACCEL benchmarks available to us. Then, we let the IPDA-based framework automatically perform the same transformations on GPU-bound loops. The evaluation shows that the resulting, more-generic, code has *the same* performance as the architecture-specific code and has *better* performance on platforms for which it was not hand-tuned—this evaluation uses the Intel MIC Xeon Phi 7250 accelerator. Table 1 shows the impact of removing the collapse clauses from benchmark code.

The implications of removing this directive are clearly demonstrated by the reduced GPU performance without it in 3MM at 0.03× speedup versus the original version of the benchmark that includes the clause. Similarly, in SYR2K with a speedup of 0.06×, when the directive is removed. COVAR slowdown is not as significant, because only one out of three parallel loops in the benchmark is annotated with collapse. The IPDA-enabled GPU Speedup column shows that the original performance is recovered when IPDA-based transformations are applied to the code from which the clauses were removed.

In multi-core CPU platforms, each processor has a local cache. Adjacent threads accessing adjacent memory locations, which is the effect of collapsing, results in false sharing, causing unnecessary coherence traffic and degrading performance. Intel's OpenMP performance guidelines recommend avoiding this usage pattern at all cost [12]. Yet, even highly tuned benchmark implementations have programmers inserting prescriptive clauses that maximize false sharing. The "No Collapse MIC Speedup" column of Table 1 shows that simply removing the collapse clauses can significantly improve performance on an x86-based accelerator (up to 26%). Analysis and transformation capabilities enabled by IPDA go a long way towards removing the need for specializing the code to a given architecture.

9 RELATED WORK

9.1 Performance Portability

The quest for performance portability of high-level parallel programming models has attracted much research attention to the areas of modeling and optimization of parallel-program performance [8, 19, 25, 32, 34]. In the context of OpenACC, Miles et al. argue that true performance portability can only be achieved through compiler transformations guided by the specific demands of the target platform [20]. They observe that parallel loop nests must be structured differently depending on whether they are to be executed on a homogeneous multi-core machine or on a highly parallel, throughput-optimized accelerator. Both paradigms currently coexist in the domains of high-performance and scientific computing; thus, the continued development of compiler technology is key to achieve performance portability.

9.2 OpenMP to GPGPU Optimization

Lee, Min, and Eigenmann presented an OpenMP to GPGPU compiler long before the programming model provided official constructs for accelerator offloading [15]. Their *stream optimizer* deploys similar interchange and collapse transformations to increase GPU inter-thread locality and thus enable coalescing of memory requests. *Stream optimizer* selects a loop whose induction variable increments a memory reference's indexing expression, using classical auto-parallelization techniques to build a list of candidate loops. However, they provide no details on how such analysis is performed and present no new parallelization or profitability metrics. Similarly, scant details are presented about the selection of loops for collapsing. There are concerns about the implementation, because some examples shown in the article apply a collapse transformation to a loop nest where loop interchange is not possible because of a loop dependence—collapsing such a loop nest is not a legal operation. This seminal work presents new ideas, but, as they state, "some advanced compiler optimizations using inter-procedural analysis were applied manually." In contrast, the IPDA framework is a fleshed-out technology demonstrated in a commercial compiler that is more robust and more general.

9.3 Symbolic Memory Reference Analysis

Symbolic analysis of loop code is a concept that dates as far back as 1976, when Cheatham and Townley proposed symbolic execution as a tool for loop analysis for the EL1 programming language [6]. This analysis expressed a set of facts about an execution of a loop across iterations captured as recurrence equations with symbolic unknowns. This research paved the way for decades of work that iterated on the idea. Haghighat et al. use numerical finite differences to detect generalized induction-variable expressions and to reduce cross-iteration access stride to a recurrence, solving that can yield dependence information (Paraphase 2 compiler) [11]. Gerlek et al. apply a technique based on a generalization of demand-driven constant propagation to detect strongly connected components in the SSA graph with the goal of identifying sequence variables in program code [9]. They demonstrate how solving recurrences that occur in loop expressions can be used to replace update statements with the respective closed form.

Motivated by the need to analyze addressing expressions that cannot be captured as a solvable recurrence, Rus et al. introduced a framework for analysis of memory reference sets addressed by induction variables without closed forms [29]. The framework relies on a data structure called the Value Evolution Graph (VEG). Based on *Gated Static Single Assignment* representation, the VEG augments the GSA data-flow graph by representing values as ranges of possible actual values. Sequences of data-flow edges $p \rightarrow \cdots \rightarrow q$ form *evolutions*, which are unioned across all paths from p to q to form an aggregate evolution. Similarly to IPDA, the VEG can be used to compute

iteration distance between two consecutively accessed elements. Representing evolutions as graph paths restricts the evolutions that it can represent and the kinds of operations that can be performed. IPDA's symbolic representation allows it to scale to large indexing expressions, because the algebraic differences lead to simpler expressions due to term cancellations.

Paek et al. introduced the term *coalescible* to describe an array access pattern in their presentation of Linear Memory Access Descriptors (LMADs)—an array region representation that abstracts data structure shapes of a programming language by relating them directly to linear machine memory [27]. An LMAD is a set of arbitrarily ordered memory locations. LMADs are encoded as a set of offsets from a common base and a set of constraints on the dimensional indices that correspond to the loops containing the array reference, forming a polytope and a set of strides through it. The symbolic manipulation and simplification techniques used in IPDA are similar to the ones used to simplify LMADs to identify and manipulate array access pattern. Like LMADs, IPDA infers access characteristics from *access dimensions*—changes in overall stride based on changes in a single dimension of a loop nest. Both LMADs and IPDA are able to characterize array access strides and aim to expose simple access footprints embedded in complex addressing expressions. However, the LMAD representation places restrictions on the capability of analyses it powers that the IPDA framework does not have. By design, LMAD is restricted to linear expressions. For example, an expression containing $i \times j$, where i, j are loop indices, cannot be expressed as an LMAD. IPDA framework places no such restrictions on the subscript expressions it can express. The symbolic manipulation of indices in IPDA allows an analysis to reason about access ranges of non-linear expressions (e.g., in the interest of dependence testing). Moreover, unlike LMADs, IPDA captures conditionally defined terms in subscript expressions. Such cases are common in practice and would prevent an LMAD-based analysis from improving the code.

Moon et al. proposed a technique called *predicated array data-flow analysis*, which associates predicates with data-flow values that represent control-flow paths taken to arrive at the values [21]. The predicates are formulated into executable program statements that form tests that guard parallelized versions of computation. They capture control-flow into the data-flow representation at runtime for a specific control-flow path. In contrast, The IPDA framework encodes facts about all possible control-flow paths into its symbolic representations of program statements without losing accuracy. A result similar to Moon et al.'s can, in principle, be achieved by the IPDA framework; however, the techniques described in this article are presented in a fully static context.

Oancea and Rauchwerger combined LMADs and runtime information to produce highly accurate dependence information [23]. They transform independence conditions into succinct predicates. They define Uniform Set Representation (USR) to represent array references, which is further reduced to a language of predicates. Logic inference is then used to factorize the predicate sets into a set of conditions that, if evaluated to true, signify that the loop is parallel. These predicates can be evaluated statically or at runtime. The hybrid nature of the analysis allows their work to handle complex control flow and nonlinear indexing. It is also what distinguishes their work from IPDA. IPDA foregoes array-abstraction set representations, opting for a symbolic evaluation framework that is more computationally expensive but requires arguably fewer conservative heuristics, as in the discussion on LMADs above. For instance, IPDA can statically handle complex predicated control flow in a path-insensitive fashion. This capability of IPDA could also be delivered to capture values only known at runtime, in a similar fashion to the runtime predicate evaluation by Oancea et al., by evaluating the succinct symbolic expressions produced by the dependence tests described in Section 5 and the profitability metrics described in Section 6 at runtime and using this evaluation to select a code version at runtime.

Predicated values in array subscripts were explored by Oancea with a flow-sensitive analysis that summarizes effects of conditional *Induction Variables* that often appear in programming

constructs such as filters [24]. The work builds on USR to allow representation of conditionally incremented index variables with the same goal of building predicate-based independence tests to be evaluated at runtime. In that work, handling of predicated values is only supported for monotonic index variables. In contrast, IPDA's symbolic encoding allows the expression of conditionals in a more general representation that is able to capture predicated values that are not index variables contained in generic control flow constructs. An analysis framework that operates on symbolic expressions instead of an array abstraction is inherently more amenable to a greater variety of applications. This work shows dependence and coalescing analyses built using this framework; however, other potential compiler applications are well suited to benefit from this representation, such as expression tree balancing and strength reduction.

9.4 Loop Dependence Analysis

Industrial-strength compilers, such as the LLVM Compiler Infrastructure and the IBM XL C/C++/Fortran Compilers, use a near-complete implementation of Goff-Kennedy-Tseng dependence testing [10]. For a single-index addressing expression, exact tests are typically used that treat most commonly occurring single-index expressions as special cases for which efficient closed-form solutions are implemented. For linear addressing expressions, dependence testing is often reduced to finding integer solutions to systems of linear Diophantine equations. Implementations of the Goff-Kennedy-Tseng work include a limited variety of "symbolic" tests. One such test processes addressing expressions that contain no index variables and can be symbolically tested for equality. Another handles expressions of the form $<ai + c_1, ai' + c_2>$ that contain a single index variable i, with loop-invariant symbolic additive constants c_1 and c_2, where the difference $c_2 - c_1$ can be reduced to a constant. These techniques are restricted to a small subset of addressing expressions that conform to a very specific format. In contrast, the IPDA analysis scales to arbitrary addressing expressions and to many index variables present in these expressions, enabling the analysis of deep loop nests.

The work most similar to our proposed DDG algorithm technique is the Range Test by Blume and Eigenmann [4]. The Range Test propagates ranges to symbolic values to determine potential overlap of two addressing expressions across iterations of a given loop. It then computes the minimum and the maximum of the corresponding ranges of addressing expressions across multiple loops and checks whether the maximum value for one expression is less than or equal to the minimum value of the other; whereas the IPDA Test first computes algebraic differences between symbolic representations of the two subscripts and then methodically reduces iteration point differences for all loop subsets in a given nest to verify if the difference can be zero. The compiler infrastructure used by the IPDA framework does not implement the Range Test, making a quantitative comparison challenging. A key advantage to the IPDA subscript difference approach is evident when handling conditionally defined variables. Although the abstract interpretation-based [7] range analysis used by the Range Test is able to capture ranges of conditionally defined values, it does so by conservatively combining ranges across control-flow paths at merge points, using the largest upper and lowest lower bounds. Similar range analysis, when run on a symbolic subscript difference computed by IPDA, is instead able to compute much narrower ranges due to the likelihood of the conditional sum-of-product expressions being cancelled out as a common term on both sides of the difference.

Engelen et al. propose a symbolic loop analysis framework for nonlinear dependence testing based on a representation of symbolic expressions with chains of recurrences (CRs) [31]. Their framework handles variable and pointer updates in conditional paths inside the loop body by constructing a set of CR forms for a conditionally defined variable, where each set element corresponds to the CR form of a given program path. Bounding functions for the range of the given variable

are then constructed for a set of CR forms, instead of an individual CR form. Indexing expression range analysis is then performed, similarly to the Range Test, over sets of characteristic functions. The CR-set technique may handle some code patterns described in the article that are common to DSP codes (boundary checks); however, range analysis over sets of characteristic functions has significant drawbacks that do not affect IPDA. Consider the case of:

```
int x;
if (c) { x = 0; }
...
if (c) { A[x] = ...; }
```

The set of CRs for the conditionally defined variable `x` through code paths that lead to the array access `A[x]` will have the value of `x` range both 0 or any other possible integer value (for the case that the if condition does not hold). As a result, the analysis would not be able to infer any information about the array access. IPDA analysis is capable of capturing variable references' dominating definitions and will determine the access to always be `A[0]`. Even without the definition propagation, the algebraic difference computation on the values of `A[x]` computed by different iterations would be solved to a 0, because the condition expressions would be canceled out. The symbolic difference simplification process often makes range computation much simpler.

Another popular methodology for dependence analysis is the Polyhedral model [5]. The Polyhedral model treats loop iterations within loop nests as points in a lattice inside a polytope. This representation allows geometric modelling of any affine functions of indices that comprise the polytope. Dependence relations can then be established based on overlap of the resulting polytopes of memory location subscripts. A key limitation of the polyhedral representation—one that does not impact IPDA —is its restriction to spaces of affine functions of index variables. Moreover, expressions containing variables defined in conditional execution paths are intractable by the Polyhedral model.

10 CONCLUSION

Architecture-specific compiler optimization is key for achieving performance portability for high-level parallel programs. Conflicting demands of current accelerator architectures when it comes to efficient use of memory hierarchies mean compilers demand stronger program analyses and heuristics to generate optimal code for a given target. This article introduced a static analysis framework capable of identifying memory access strides of parallel accelerator code using Iteration Point Difference Analysis. The evaluation of a prototype implementation of a framework that uses IPDA to guide the safety and profitability decisions required for improving performance through loop transformations demonstrated the potential for dramatic performance improvement in GPU-bound OpenMP code. Moreover, this article also demonstrated that informed compiler transformation can further advance the goal of performance portability by reducing the reliance on programmer hints used to hand-tune OpenMP loop code. Making such hints redundant both increases performance across a greater variety of target architectures and increases abstraction of the underlying computing platform, making parallel programs more generic and allowing the developer to focus instead on the problem at hand.

REFERENCES

[1] ACF-Coalescing-LLVM 2019. ACF static analysis framework source-code. Retrieved from: https://github.com/uasys/ACF-Coalescing-LLVM.

[2] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS'09)*. 163–174.

[3] Utpal K. Banerjee. 1988. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA.
[4] W. Blume and R. Eigenmann. 1994. The range test: A dependence test for symbolic, non-linear expressions. In *Proceedings of the Conference on Supercomputing*. 528–537.
[5] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the ACM-SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'08)*. 101–113.
[6] T. E. Cheatham and J. A. Townley. 1976. Symbolic evaluation of programs: A look at loop analysis. In *Proceedings of the Symposium on Symbolic and Algebraic Computation*.
[7] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'77)*. ACM, 238–252.
[8] Thomas Fahringer, Michael Gerndt, Graham Riley, and Jesper Larsson Träff. 2000. Formalizing OpenMP performance properties with ASL. In *Proceedings of the International Symposium on High Performance Computing*, Mateo Valero, Kazuki Joe, Masaru Kitsuregawa, and Hidehiko Tanaka (Eds.). 428–439.
[9] Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. 1995. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Trans. Prog. Lang. Syst.* 17, 1 (Jan. 1995), 85–122.
[10] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. 1991. Practical dependence testing. In *Proceedings of the ACM-SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'91)*. 15–29.
[11] M. Haghighat and C. Polychronopoulos. 1993. Symbolic program analysis and optimization for parallelizing compilers. In *Proceedings of the Workshop on Languages and Compilers and Parallel Computing (LCPC'93)*. 538–562.
[12] Intel. 2011. Avoiding and identifying false sharing among threads. Retrieved on March 13, 2018 from: https://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads.
[13] Guido Juckeland, William Brantley, Sunita Chandrasekaran, Barbara Chapman, Shuai Che, Mathew Colgrove, Huiyu Feng, Alexander Grund, Robert Henschel, Wen-Mei W. Hwu, Huian Li, Matthias S. Müller, Wolfgang E. Nagel, Maxim Perminov, Pavel Shelepugin, Kevin Skadron, John Stratton, Alexey Titov, Ke Wang, Matthijs van Waveren, Brian Whitney, Sandra Wienke, Rengan Xu, and Kalyan Kumaran. 2015. SPEC ACCEL: A standard application suite for measuring hardware accelerator performance. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation (LNCS)*, Vol. 8966. Stephen A. Jarvis, Steven A. Wright, and Simon D. Hammond (Eds.). 46–67.
[14] Leslie Lamport. 1974. The parallel execution of DO loops. *Commun. ACM* 17, 2 (Feb. 1974), 83–93.
[15] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. 2009. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In *Proceedings of the Symposium on Principles & Practice of Parallel Programming (PPoPP'09)*. 101–110.
[16] T. Lloyd, K. Ali, and J. N. Amaral. 2019. *GPUCheck: Detecting CUDA Performance Problems with Static Analysis*. Technical Report. University of Alberta, Edmonton, AB, Canada.
[17] T. Lloyd, A. Chikin, J. N. Amaral, and E. Tiotto. 2018. Automated GPU grid geometry selection for OpenMP kernels. In *Proceedings of the Workshop on Applications for Multi-Core Architectures (WAMCA'18)*. Retrieved from: https://webdocs.cs.ualberta.ca/amaral/papers/LloydWAMCA18.pdf.
[18] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. 1992. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the International Symposium on Microarchitecture (MICRO'92)*. 45–54.
[19] Jiayuan Meng and Kevin Skadron. 2009. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *Proceedings of the International Conference on Supercomputing (ICS'09)*. 256–265.
[20] D. Miles, D. Norton, and M. Wolfe. 2014. Performance portability and OpenACC. In *Proceedings of the Conference of Cray User Group (CUG'14)*.
[21] Sungdo Moon, Mary W. Hall, and Brian R. Murphy. 1998. Predicated array data-flow analysis for run-time parallelization. In *Proceedings of the 12th International Conference on Supercomputing (ICS'98)*. 204–211.
[22] Nvidia. [n.d.]. Nvidia Tesla V100 GPU architecture—The world's most advanced data center GPU. Retrieved from: http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.
[23] Cosmin E. Oancea and Lawrence Rauchwerger. 2012. Logical inference techniques for loop parallelization. In *Proceedings of the ACM-SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'12)*. 509–520.
[24] Cosmin E. Oancea and Lawrence Rauchwerger. 2015. Scalable conditional induction variables (CIV) analysis. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'15)*. 213–224.
[25] Michael F. P. O'Boyle, Zheng Wang, and Dominik Grewe. 2013. Portable mapping of data parallel programs to OpenCL for heterogeneous systems. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'13)*. 1–10.
[26] OpenMP Language Committee. 2013. OpenMP application program interface version 4.0. Retrieved on March 13, 2018 from: http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf.

[27] Yunheung Paek, Jay Hoeflinger, and David Padua. 2002. Efficient and precise array access analysis. *ACM Trans. Prog. Lang. Syst.* 24, 1 (Jan. 2002), 65–109.

[28] Daniel Rolls, Carl Joslin, and Sven-Bodo Scholz. 2010. Unibench: A tool for automated and collaborative benchmarking. In *Proceedings of the International Conference on Program Comprehension (ICPC'10)*. 50–51.

[29] Silvius Rus, Dongmin Zhang, and Lawrence Rauchwerger. 2004. The value evolution graph and its use in memory reference analysis. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'04)*. 243–254.

[30] Arthur Stoutchinin and Francois de Ferriere. 2001. Efficient static single assignment form for predication. In *Proceedings of the International Symposium on Microarchitecture (MICRO'01)*. 172–181.

[31] Robert A. van Engelen, J. Birch, Y. Shou, B. Walsh, and Kyle A. Gallivan. 2004. A unified framework for nonlinear dependence testing and symbolic analysis. In *Proceedings of the International Conference on Supercomputing (ICS'04)*. 106–115.

[32] Zheng Wang and Michael F. P. O'Boyle. 2009. Mapping parallelism to multi-cores: A machine learning based approach. In *Proceedings of the ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP'09)*.

[33] Michael Joseph Wolfe. 1982. *Optimizing Supercompilers for Supercomputers*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign, Champaign, IL.

[34] Zhong Zheng, Xuhao Chen, Zhiying Wang, Li Shen, and Jiawen Li. 2011. Performance model for OpenMP parallelized loops. In *Proceedings of the Conference on Transportation, Mechanical, and Electrical Engineering (TMEE'11)*.

Received August 2018; revised April 2019; accepted May 2019