DawnCC: Automatic Annotation for Data Parallelism and Offloading

GLEISON MENDONÇA, BRENO GUIMARÃES, and PÉRICLES ALVES, UFMG
MÁRCIO PEREIRA and GUIDO ARAÚJO, Unicamp
FERNANDO MAGNO QUINTÃO PEREIRA, UFMG

Directive-based programming models, such as OpenACC and OpenMP, allow developers to convert a sequential program into a parallel one with minimum human intervention. However, inserting pragmas into production code is a difficult and error-prone task, often requiring familiarity with the target program. This difficulty restricts the ability of developers to annotate code that they have not written themselves. This article provides a suite of compiler-related methods to mitigate this problem. Such techniques rely on symbolic range analysis, a well-known static technique, to achieve two purposes: populate source code with data transfer primitives and to disambiguate pointers that could hinder automatic parallelization due to aliasing. We have materialized our ideas into a tool, DawnCC, which can be used stand-alone or through an online interface. To demonstrate its effectiveness, we show how DawnCC can annotate the programs available in PolyBench without any intervention from users. Such annotations lead to speedups of over 100× in an Nvidia architecture and over 50× in an ARM architecture.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Computing methodologies** → **Parallel computing methodologies;**

Additional Key Words and Phrases: Automatic parallelization, static analysis

ACM Reference Format:
Gleison Mendonça, Breno Guimarães, Péricles Alves, Márcio Pereira, Guido Araújo, and Fernando Magno Quintão Pereira. 2017. DawnCC: Automatic annotation for data parallelism and offloading. ACM Trans. Archit. Code Optim. 14, 2, Article 13 (May 2017), 25 pages.
DOI: http://dx.doi.org/10.1145/3084540

1. INTRODUCTION

Heterogeneous architectures formed by clusters of CPUs and GPUs now give us a de facto standard for high-performance computing. And currently, directive-based annotation systems stand out among the different techniques used to program these machines. Examples of such systems include OpenMP [Jaeger et al. 2015], OpenACC [OpenACC Standard 2013], OpenHMPP [Andión et al. 2016], OpenMPC [Lee and Eigenmann 2010], and OpenSs [Meenderinck and Juurlink 2011]. The annotation-based programming model is simple yet appealing: annotations are a metalanguage, which give developers the ability to grant parallel semantics to syntax originally written to run sequentially. Hence, developers can reap all of the benefits from the modern parallel hardware without having to worry too much about details of the target architecture—such inconveniences are left to the compiler. Success stories of such annotation systems

This work was sponsored by LG Electronics and partially supported by FAPEMIG, FAPESP, CNPq, and CAPES.
Authors' addresses: G. Mendonça; email: gleison.mendonca@dcc.ufmg.br; B. Guimarães; email: brenosfg@dcc.ufmg.br; P. Alves; email: periclesrafael@dcc.ufmg.br; M. Pereira; email: marcio.machado.pereira@gmail.com; G. Araújo; email: guido@ic.unicamp.br; F. M. Q. Pereira; email: fernando@dcc.ufmg.br.

DOI: http://dx.doi.org/10.1145/3084540

abound. Combined with modern accelerators, they have led to substantial performance gains [Bertolli et al. 2014; Reyes et al. 2012; Wienke et al. 2012].

Automatic parallelization has been the focus of much research, to the point that compilers are now able to detect parallel loops with high accuracy [Baskaran et al. 2010]. Nevertheless, annotating code to run in an accelerator device is still a difficult task, which often requires familiarity with the target program that must be transformed. Two challenges are particularly daunting: the disambiguation of pointers and the estimation of memory bounds. All of these problems have been attacked in one way or another by the programming languages community. Pointer disambiguation is a well-studied problem, with fully static [Andersen 1994; Pereira and Berlin 2009], fully dynamic [Duck and Yap 2016], and hybrid [Rus et al. 2003] solutions. And recent work has brought new light onto the problem of estimating the bounds of memory regions [Nazaré et al. 2014; Alves et al. 2015]. However, despite all of these advances, we still do not have the necessary equipment that enables developers to annotate code automatically and correctly with acceleration directives. In this article, we provide a suite of static analyses that solve this problem. To this effect, we introduce a tool, DawnCC, which automatically removes from programmers the burden of annotating code. DawnCC reuses the lattice of Rugina and Rinard [2000] to derive symbolic bounds for memory regions such as arrays, structs, or unions allocated in C programs. By *symbolic*, we mean that these bounds are written as symbols (e.g., variable names), present in the source code of the program itself. Such bounds let us annotate code with data copy directives, which move data between host and accelerator. The contributions of this new technology are the following.

Correctness. Section 3 shows how we have reused the techniques introduced by Alves et al. [2015] and Campos et al. [2016] to disambiguate pointers at runtime. We establish conditions that ensure the absence of aliasing between memory references, hence implementing *pointer restrictification* at the source code level. This disambiguation enables the discovery of more parallel regions in the sequential source code, because it reduces dependencies between data. Runtime pointer disambiguation is not new; however, it is the first time that such technique is used to ensure that acceleration code is produced correctly. We emphasize that all previous works on this area thus far rely on the programmer explicitly marking pointers with the `restrict` type modifier. DawnCC frees the programmer from this burden.

Engineering. Section 4 explains how we map information collected at the assembly level back into source code. Our analyses run on programs in the static single assignment (SSA) representation. Yet annotations are inserted onto source code. Far from being a "pretty-printing" problem, recovering high-level information from an optimized low-level language is not trivial. We have chosen this approach to be able to use two key features of an industrial-quality compiler: LLVM's scalar evolution analysis (see p. 18 of Grosser et al. [2012]) and that compiler's points-to analysis. The former lets us use symbolic intervals to bound memory regions, hence giving us the ability to analyze nonaffine regions. The latter lets us eliminate as many restrictification checks as possible while still producing correct code.

Optimizations. Section 5 shows how we use the program dependence graph of Ferrante et al. [1987] to eliminate redundant data transfer primitives. This optimization, which we call *copy fusion*, lets us remove transfer operations between parallel regions marked as kernels. Furthermore, as a by-product of the analyses that we perform, it lets us avoid moving read-only data from accelerator back to host. Implementing this optimization in a low-level intermediate representation

```
1  void
2  saxpy_serial(int n, float alpha, float *x, float *y) {
3      for (int i = 0; i < n; i++) {
4          y[i] = alpha*x[i] + y[i];
5      }
6  }
```

(a)

```
1  __global__ void
2  saxpy_parallel(int n, float alpha, float *x, float *y) {
3      int i = blockIdx.x * blockDim.x + threadIdx.x;
4      if (i < n) {
5          y[i] = alpha * x[i] + y[i];
6      }
7  }
8  ...
9  // Allocate x and y, and copy them to the GPU ...
10 // Invoke the parallel kernel:
11 int nblocks = (n + 255) / 256;
10 saxpy_parallel <<<nblocks, 256>>>(n, 2.0, x, y);
11 // Copy y back to the CPU ...
```

(b)

Fig. 1. (a) Standard C implementation of the single-precision $AX + Y$ (SAXPY) kernel. (b) Same algorithm written in C for CUDA.

(IR) is not difficult: it resembles lazy code motion [Knoop et al. 1992] but is applied on symbolic intervals. However, mapping the results of this optimization back into source code was not trivial. To map the control flow graph (CFG) back into the program's abstract syntax tree, we used a data structure that we call the *scope tree*.

The final product that stems out of these contributions is a tool, henceforth called DawnCC, that frees developers from the tedious and error-prone task of inserting copy directives in programs. This tool inserts OpenACC or OpenMP 4.0 annotations into programs, which are usually more optimized than those that would be inserted by a person, although equally readable. Annotated loops run on an accelerator. This transformation does not require any intervention from users—it is completely automatic. To validate the ideas discussed in this article, we have used DawnCC to transform programs present in PolyBench. As we explain in Section 6, our static analysis is able to bound 98% of all load and store operations present in these benchmarks. Whenever we can bound every memory access instruction within a loop, we can insert directives that move its data to and from an accelerator. In this case, we say that the loop is *analyzable*. In PolyBench, we are able to parallelize up to 95% of all loops. The net result is performance: by annotating loops automatically, we have been able to observe speedups of up to 105× on a CPU-GPU based architecture. Our tool is currently available through an online interface at http://cuda.dcc.ufmg.br/dawn/.

2. OVERVIEW

We use the *single-precision AX + Y* (SAXPY) kernel in Figure 1(a) to illustrate the contributions of this article. This kernel is a standard function in the the Netlib BLAS Standard.[1] It performs a combination of multiplication by scalar plus addition between corresponding cells of two vectors. It runs in linear time on a sequential machine. However, it is $O(1)$ in the parallel random-access machine (PRAM) model because there is no dependency between different iterations of the loop. In the high-performance computing jargon, the SAXPY loop is called a *doall*.

Figure 1(b) shows a direct translation of SAXPY to C for CUDA. CUDA's syntax is very similar to C's; however, its semantics is substantially different. Part of it (e.g., lines 1 through 7) is meant to run on a GPU; the rest (e.g., lines 9 through 11) is meant to run on a host CPU. The code that runs on the GPU will be instantiated multiple times, once per each logical thread. In this case, we have one thread per each valid index in the input vectors. Even though C for CUDA is becoming commonplace among

[1]http://www.netlib.org/blas/.

```
1  void saxpy_serial(int n, float alpha, float *x, float *y) {
2    long long int tmp[2];
3    tmp[0] = n -1;
4    tmp[1] = ((tmp[0] > 0) ? tmp[0] : 0); // upper bound
5
6    char x_y_alias_free = ((x >= y + tmp[1] + 1) ||
7                           (y >= x + tmp[1] + 1));
8
9    #pragma acc data pcopy(y[0:tmp[1]]) \
10                  pcopyin(x[0:tmp[1]]) \
11                  if(x_y_alias_free)
12   #pragma acc kernels loop independent \
13                  if(x_y_alias_free)
14   for (int i = 0; i < n; i++)
15     y[i] = alpha*x[i] + y[i];
16 }
```

(a)

```
1  void saxpy_serial(int n, float alpha, float *x, float *y) {
2    long long int tmp[2];
3    tmp[0] = n -1;
4    tmp[1] = ((tmp[0] > 0) ? tmp[0] : 0); // upper bound
5
6    char x_y_alias_free = ((x >= y + tmp[1] + 1) ||
7                           (y >= x + tmp[1] + 1));
8
9    #pragma omp target data map(to:x[0:tmp[1]]) \
10                  map(tofrom:y[0:tmp[1]]) \
11                  if(x_y_alias_free)
12   #pragma omp parallel for if(x_y_alias_free)
13   for (int i = 0; i < n; i++)
14     y[i] = alpha*x[i] + y[i];
15 }
```

(b)

Fig. 2. (a) SAXPY annotated with OpenACC pragmas. (b) SAXPY annotated with OpenMP pragmas. The gray area denotes code created automatically.

developers of parallel applications, having to worry about concurrent semantics and communication between multiple devices still restricts the use of this language.

To make GPUs more accessible to the everyday developer, the high-performance computing community has designed several annotation systems. An *annotation system* is a metalanguage that changes the semantics of a host language. In our setting, the host language is C, and the metalanguage is either OpenACC or OpenMP. Figure 2 shows the sequential SAXPY kernel annotated with (a) OpenACC and (b) OpenMP pragmas. Our DawnCC compiler inserts these pragmas, plus all code necessary for them to work, automatically.

DawnCC is a source-to-source compiler: it reads a C program and produces a version of that program with annotations. In this process, DawnCC solves two problems. First, it recognizes *doall* loops. Second, it inserts primitives to copy data to and from the GPU. To deal with the first problem, the identification of *doall* loops, we use standard compiler analysis techniques (see Chapter 6 of Wolfe [1995]). Because these techniques are already commonplace in the compiler's literature, in this article we focus on the second problem.

To insert data movement directives, such as `pcopy` in Figure 2(a) and `map` in Figure 2(b), we need to infer the bounds of memory regions. In this example, memory regions are the arrays `x` and `y`. We use static analysis techniques, further described in Section 3, to recover these limits. More importantly, we do it using symbols present in the program code itself. For instance, at line 3 of Figure 2(a) and (b), we are using the symbol `n` to rebuild the limits of the arrays `x` and `y`. We store such symbols in an array `tmp` of auxiliary values. These limits not only give us a way to copy data around, but they also let us show that pointers do not overlap. In our example, the tests at line 6 of Figure 2(a) and (b) give us this information. Whenever either of the inequalities $y >= x+n+1$ or $x >= y+n+1$ is true, we are sure that the vectors `x` and `y` do not overlap. We can only presume that the loop is parallel once we are under this assumption. We emphasize that these annotations have been produced without any intervention from a user. How we perform such deed is a subject for the next section.

3. STATIC ANALYSES

DawnCC is built around a suite of static analyses. These analyses exist on top of two well-known techniques: symbolic range analysis [Rugina and Rinard 2000; Alves et al. 2015] and dependence analysis [Ferrante et al. 1987]. In the rest of this section, we explain how these compilation techniques work. Figure 3 shows how these different analyses are related to each other, and where, within this article, they are discussed.

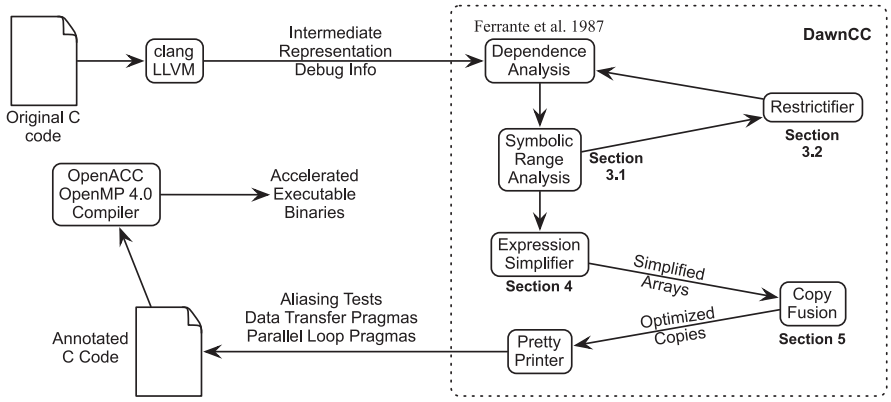


Fig. 3. Overview of DawnCC.

3.1. Symbolic Range Analysis

Our ideas rely on the compiler's ability to estimate the range of values covered by the integer variables used in a program. We perform this estimation via *symbolic range analysis*, as defined by Rugina and Rinard [2000]. In this article, we use the algorithm described by Alves et al. [2015], which works on programs in SSA form [Cytron et al. 1991]. SSA is a classic program representation in which each variable has one single definition point [Cytron et al. 1991]. The goal of the symbolic range analysis is to compute a map R, which gives, for every integer variable in a program, a pair $[l, u]$, representing its lower (l) and upper (u) bounds. A *symbol* is any variable whose value we cannot reconstruct as a function of other variables. Limits l and u are symbols, integer constants, or the special elements $+\infty$ and $-\infty$. If l and u are integer constants, then a range is well formed when $l \leq u$.

Figure 4 shows a subset of the range inference rules that we have used in this work. Our full implementation contains rules for every arithmetic instruction in LLVM, and every control flow construct, including LLVM's switch instruction. These rules show how the ranges of variables change during the process of finding a suitable map R. Such changes are guided by a relation $\mathbf{rg} \vdash (S \times R) \mapsto R'$, which receives a sequence of program statements S and a map R, and produces a new map R'. R' contains the result of updating R with the facts that we learn from S's syntax. Each syntactic construction affects R in a particular way. For instance, Rule `[Const]` determines that the abstract state of a variable v that has been assigned a constant n is a new interval that is large enough to contain n. Our most complicated rule is `[Lte]`, which determines how less-than-or-equal comparisons change the abstract states of variables. The ranges of variables in the "then" and "else" part of the branch are evaluated differently. The union of these ranges gives us the ranges at the end of the conditional. Different relational operators (e.g., $<$, $>$, and $\geq$) are handled in a similar way.

To determine R, we iterate the rules in Figure 4 until we achieve a fixed point. The fixed point operator is given by the relation **fp**. Due to loops, this analysis might not converge. Thus, we use a widening operator ∇ to ensure convergence. This analysis is linear on the size of the program, measured as the number of uses and definitions of variables within its text. The rules in Figure 4 show a flow-insensitive analysis. In other words, the range of a variable does not depend on the program point where that variable is used. We achieve flow sensitiveness by applying those rules onto programs in SSA form.

[Const] $$\frac{R_2 = R_1 \setminus v \mapsto (R_1(v) \sqcup [n, n]) \qquad \mathbf{rg}(S, R_2) = R_3}{\mathbf{rg}(\{v = n; S\}, R_1) = R_3}$$

[Sym] $$\frac{R_2 = R_1 \setminus v \mapsto (R_1(v) \sqcup [s_v, s_v]) \qquad \mathbf{rg}(S, R_2) = R_3}{\mathbf{rg}(\{v = \bullet; S\}, R_1) = R_3}$$

[Var] $$\frac{R(v_1) = [l, u] \qquad R_2 = R_1 \setminus v \mapsto (R_1(v) \sqcup [l, u]) \qquad \mathbf{rg}(S, R_2) = R_3}{\mathbf{rg}(\{v = v_1; S\}, R_1) = R_3}$$

[Add] $$\frac{\begin{array}{c} R(v_1) = [l_1, u_1] \qquad R(v_2) = [l_2, u_2] \\ l = l_1 + l_2 \qquad u = u_1 + u2 \qquad R_2 = R_1 \setminus v \mapsto (R_1(v) \sqcup [l, u]) \qquad \mathbf{rg}(S, R_2) = R_3 \end{array}}{\mathbf{rg}(\{v = v_1 + v_2; S\}, R_1) = R_3}$$

[Lte] $$\frac{\begin{array}{c} R(v_a) = [l_a, u_a] \\ R(v_b) = [l_b, u_b] \qquad R_t = (R_1 \setminus v_a \to [l_a, \min(u_b - 1, u_a)]) \setminus v_b \to [\max(l_a + 1, u_b)] \\ \mathbf{rg}(S_t, R_t) = R'_t \qquad R_f = (R_1 \setminus v_a \to [\max(l_a, l_b), u_a]) \setminus v_b \to [l_b, \min(u_a, u_b)] \\ \mathbf{rg}(S_f, R_f) = R'_f \qquad R_2 = R_t \sqcup R_f \qquad \mathbf{rg}(S, R_2) = R_3 \end{array}}{\mathbf{rg}(\{\texttt{if}(v_a \leq v_b)\ S_t\ \texttt{else}\ S_f; S\}, R_1) = R_3}$$

[Cond] $$\frac{\mathbf{rg}(S_t, R_1) = R_t \qquad \mathbf{rg}(S_f, R_1) = R_f \qquad R_2 = R_t \sqcup R_f \qquad \mathbf{rg}(S, R_2) = R_3}{\mathbf{rg}(\{\texttt{if}(\_)\ S_t\ \texttt{else}\ S_f; S\}, R_1) = R_3}$$

[Whl] $$\frac{\mathbf{fp}(\{\texttt{if}(v_a \leq v_b)\ S_1\ \texttt{else}\ \{\}\ \}, R_1) = R_2 \qquad R_3 = R_1 \sqcup R_2 \qquad \mathbf{rg}(S, R_3) = R_4}{\mathbf{rg}(\{\texttt{while}(v_a \leq v_b)\ S_1; S\}, R_1) = R_4}$$

[Base] $$\mathbf{rg}(\{\}, R) = R$$

[Stop] $$\frac{\mathbf{rg}(S, R) = R}{\mathbf{fp}(S, R) = R}$$

[Iter] $$\frac{\mathbf{rg}(S, R_1) = R_2 \qquad R_1 \neq R_2 \qquad \mathbf{fp}(S, R_2) = R_3}{\mathbf{fp}(S, R_1) = R_3}$$

$$[l_1, u_1] \nabla_r [l_2, u_2] = [l, u], \text{ where } \begin{cases} l \text{ is } l_1 \text{ if } l_1 = l_2 \text{ or } l \text{ is } -\infty \text{ otherwise} \\ u \text{ is } u_1 \text{ if } u_1 = u_2 \text{ or } u \text{ is } +\infty \text{ otherwise} \end{cases}$$

Fig. 4. Constraints for the symbolic range analysis.

Concerning Figure 4, we use $R \setminus v \to [l, u]$ to denote a new function $\lambda x.(x = v)?[l, u] : R(x)$. We use $\bullet$ to denote a symbol. Upon evaluating an assignment $v = \bullet$, Rule [Sym] creates a new interval $[s_v, s_v]$, where s_v is a fresh symbolic name (i.e., a name not used anywhere in the program text). We write $R_1 \sqcup R_2$ to denote a function $\lambda x.R_1(x) \sqcup R_2(x)$. We assign the following semantics to the operator $\sqcup$, used to merge dataflow facts: $[l_1, u_1] \sqcup [l_2, u_2] = [\min(l_1, l_2), \max(u_1, u_2)]$.

Example 3.1. Range analysis applied on `saxpy_serial`, in Figure 1(a), gives us that $R(\texttt{i})$ is $[0, 0]$ before the loop, $[0, \texttt{n} - 1]$ inside it, and $[\texttt{n}, \texttt{n}]$ after. We derive $R(\texttt{i})$ inside the loop by combining the rules in Figure 4. Successive evaluations of Rule [Whl] force us to apply Rule [Add] multiple times onto i. Widening gives us that $R(\texttt{i})$ tends toward $[0, +\infty]$. However, this limit stabilizes before, due to Rule [Lte], which constrains $R(\texttt{i})$ to be within $[0, \texttt{n} - 1]$ within the loop. In abstract interpretation's jargon, [Lte] implements a form of *narrowing*, which recovers precision lost by *widening*.

Inference of array access bounds. Once we have a map R, built via Figure 4's rules, we proceed to determine bounds for each array in the target program. The limits of

an array v, which we denote by $B(v)$, are determined as the union of the ranges of all variables used to index v. In other words, if a program contains n memory accesses such as $v[i_1], v[i_2], \ldots, v[i_n]$, then the access bounds of v are given by the expression: $B(v) = \&v + R(i_1) \sqcup R(i_2) \sqcup \ldots \sqcup R(i_n)$, where $\&v$ is the base address of v.

Example 3.2. Our array size inference, when applied onto `saxpy_serial`, in Figure 1(a), yields that $B(\texttt{x}) = [\&\texttt{x}, \&\texttt{x} + (\texttt{n} - 1)]$. We find such limits because the only variable that indexes `x` is `i`, which has range $[0, \texttt{n} - 1]$ within the loop.

3.2. Restrictification

The automatic insertion of copy primitives in imperative languages finds a tremendous adversary in pointer aliasing. Quoting page 144 of Jablin et al. [2011]: "Automatically managing communication based on compile-time static analysis is impossible for general C and C++ programs. In C and C++, any argument to a GPU function could be cast to a pointer, and an opaque pointer could point to the middle of data-structures of arbitrary size." If DawnCC cannot prove statically that some memory region is only accessed through one base pointer, then it does not try to find the limits of this region. This omission is common among loop-optimizing compilers, and is the case, for instance, of PolyLLVM [Verdoolaege et al. 2013]. However, the fact that we can infer the sizes of arrays gives us the possibility to disambiguate pointers, hence mitigating the aforementioned limitation. This disambiguation—also known as restrictification—lets us eliminate spurious dependencies in the source program due to pointer aliasing. Thus, it lets us potentially increase the number of parallel regions that we can find. If $B_1 = [l_1, u_1]$ and $B_2 = [l_2, u_2]$ are estimations for the regions dereferenced by two different pointers p_1 and p_2, then we say that they will not overlap if

$$p_1 + l_1 \geq p_2 + u_2 + 1 \text{ or } p_2 + l_2 \geq p_1 + u_1 + 1.$$

This test ensures that the regions covered by offsets that use p_1 and p_2 as base pointers have empty intersection. For instance, if each element pointed by p_1 takes 8 bytes, p_2 cannot point to the 4th byte of the last memory position accessed through p_1, whenever either of the preceding inequalities hold.

Both OpenACC and OpenMP give us the equipment necessary to use this information. Such equipment consists of *conditional directives*. A conditional directive only takes effect if a given predicate is valid at runtime. Thus, whenever we might have aliasing between different pointers within a loop, we use conditional tests to disambiguate them, as we illustrate in Example 3.3.

Example 3.3. The loop in `saxpy_serial`, in Figure 1(a), contains accesses to two pointers: `x` and `y`. As we have seen in Example 3.2, both have bounds $[0, \texttt{n} - 1]$. Thus, they will not alias if $\texttt{x} >= \texttt{y} + (n - 1) + 1$, or if $\texttt{y} >= \texttt{x} + (n - 1) + 1$. This test is implemented at lines 6 and 7 of Figure 2(a) and (b). The conditional pragmas that use the result of this test appear at lines 9 through 12 of the annotated programs.

4. FROM SOURCE TO IR, AND BACK AGAIN

DawnCC is built on top of the LLVM [Lattner and Adve 2004] compilation framework, whose intermediate representation is used as input for the static analyses presented in this section. We chose to perform our analyses on the IR of LLVM because we could, in this way, reuse already available techniques to compute bounds of variables. However, such benefit comes with a challenge: there exists a gap between LLVM's IR and the source code, and although we analyze the former, our annotations must, ultimately, be inserted in the latter. This section explains how we have bridged this gap. We focus on details of the LLVM's IR as a means to provide the reader with concrete examples;

(a)

```
1  void saxpy_3off(int n, float alpha, int *x, float *y) {
2    for (int i = 3; i < n; ++i)
3      y[i-3] = alpha*x[i] + y[i-3];
4  }
```

(b)

```
1  void saxpy_3off(int n, float alpha, int *x, float *y) {
2    char x_y_alias_free = ((x+3) >= (y + max(0,n-4) + 1)) || (y >= (x + max(3,n-1) + 1));
3
4    #pragma omp target map(x[3:max(3,n-1)-3+1], y[0:max(0,n-4)+1]) if(x_y_alias_free)
5    #pragma omp parallel for if(x_y_alias_free)
6    for (int i = 3; i < n; ++i)
7      y[i-3] = alpha*x[i] + y[i-3];
8  }
```

Fig. 5. (a) An example adapted from our original SAXPY code (Figure 1(a)). (b) A version of it annotated with OpenMP 4.0 pragmas to run on a GPU.

(a)

```
1  ; read x[i]
2  %2 = sext i32 %i to i64
3  %3 = getelementptr i32, i32* %x, i64 %2
4  %4 = load i32, i32* %3, align 4
5
6  %5 = sitofp i32 %4 to float
7  %6 = fmul float %alpha, %5
8
9  ; read y[i-3]
10 %7 = sub nsw i32 %i, 3
11 %8 = sext i32 %7 to i64
12 %9 = getelementptr float, float* %y, i64 %8
13 %10 = load float, float* %9, align 4
14
15 %11 = fadd float %6, %10
16
17 ; write y[i-3]
18 %12 = sub nsw i32 %i, 3
19 %13 = sext i32 %12 to i64
20 %14 = getelementptr float, float* %y, i64 %13
21 store float %11, float* %14, align 4
22
23 br label %inc
```

(b)

```
1  ; lower bound for x: &(x[3])
2  %1 = getelementptr i32, i32* %x, i64 3
3  %2 = bitcast i32* %1 to i8*
4
5  ; upper bound for x: &(x[max(3,n-1)])
6  %3 = add i32 %n, -1
7  %4 = icmp sgt i32 %3, 3
8  %5 = select i1 %4, i32 %3, i32 3
9  %6 = add i32 %5, -3
10 %7 = zext i32 %6 to i64
11 %8 = shl nuw nsw i64 %7, 2
12 %9 = getelementptr i32, i32* %x, i64 3
13 %10 = ptrtoint i32* %9 to i64
14 %11 = add i64 %8, %10
15 %12 = inttoptr i64 %11 to i8*
16
17 ; lower bound for y: &(y[0])
18 %13 = bitcast float* %y to i8*
19
20 ; upper bound for y: &(y[max(0,n-4)])
21 %14 = add i32 %n, -1
22 %15 = icmp sgt i32 %14, 3
23 %16 = select i1 %15, i32 %14, i32 3
24 %17 = add i32 %16, -3
25 %18 = zext i32 %17 to i64
26 %19 = shl nuw nsw i64 %18, 2
27 %20 = ptrtoint float* %y to i64
28 %21 = add i64 %19, %20
29 %22 = inttoptr i64 %21 to i8*
```

Fig. 6. (a) LLVM instructions generated for the body of the loop in line 3 of Figure 5 (a). (b) LLVM assembly representing the symbolic access bounds of arrays x and y.

however, our ideas fit scenarios made of high-level languages other than C/C++ and low-level languages other than LLVM bytecodes.

In low-level assemblies, array indexing expressions consist of *load* or *store* instructions. These instructions take as operand a memory address. Thus, finding the access bounds for a given array means finding the bounds for the target addresses of each load and store operation that can manipulate that array. Figure 5(a) shows a small variation of the SAXPY code seen in Figure 1(a). Figure 6(a) outlines the LLVM instructions generated for the loop body in line 3 of Figure 5(a). Names preceded by % represent virtual registers created by the compiler. The instruction `GetElementPtr` computes the actual address of an element, given a base address and the element's index. The *load* and *store* operations in lines 13 and 21 of Figure 6(a) represent the accesses to array y. The *load* in line 4 was translated from the access to x. If we apply the techniques from Section 3.1 on this program, then we get the code in Figure 6(b).

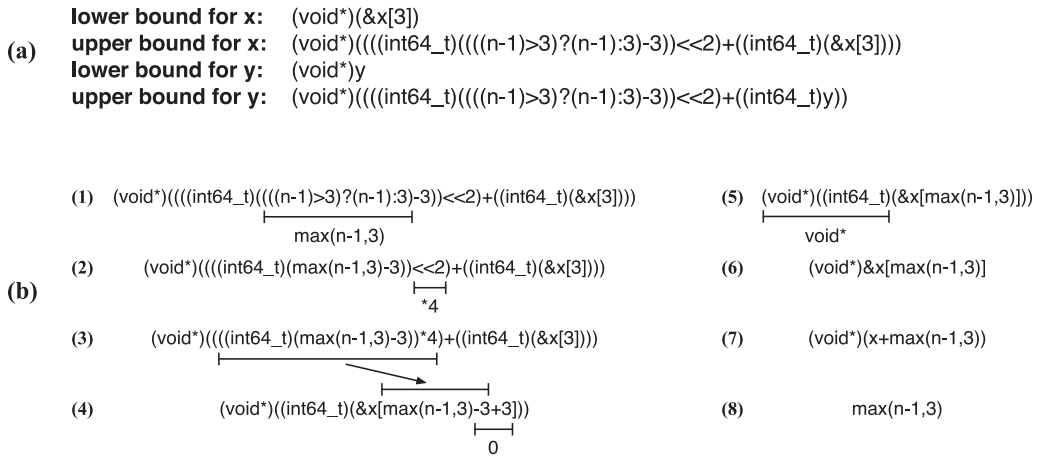


Fig. 7. (a) C code generated for the symbolic bounds in Figure 6 (b). (b) Steps used to improve the readability of expressions that describes the upper bound of x.

These assembly instructions, when executed at runtime, yield the lowest and highest addresses referenced through arrays x and y in our example (i.e., their access bounds).

Converting the low-level representation of array access bounds back into source code is relatively easy: most operators in LLVM IR have a one-to-one mapping in C or can be emulated by a small set of C operations. Using a bottom-up recursive conversion strategy over the assembly in Figure 6(b), we get the equivalent parenthesized C expressions in Figure 7(a). Although these expressions meet the goal of expressing access bounds in C code, they have two shortcomings. First, they are hard to read, a fact that would decrease the maintainability of code automatically annotated. Second, expressions used in OpenACC/OpenMP's memory transfer directives need bounds given in terms of base addresses and integer access indexes. However, the expressions in Figure 6(b) use pointer arithmetic to compute actual addresses. In what follows, we explain how we overcome these two problems.

Simplifying bound expressions. To make limit expressions more readable, we perform a series of static simplifications. Most of these simplifications take advantage of operations that are common to pointer arithmetic and address manipulation in C. Notice that our simplifications happen at the source code level, not in the LLVM format, because they are meant to be read by humans. It would be difficult to implement them on the LLVM IR and then map the simplified expressions back into source code, because this approach would require us to preserve debugging information. We explain in the following the transformations that we perform, exemplifying some of them in Figure 7. This figure shows, step by step, how the expression that computes the upper bound for array *x* (Figure 7(a)) can be simplified. The list of simplifications is as follows:

—*Conversion to* min *and* max *operations*: Limit expressions usually involve several *minimum* and *maximum* operations. These, however, are usually represented as a *less than* or *greater than* comparison followed by a selection instruction in LLVM IR (e.g., lines 7 and 8 of Figure 6(b)) or an equivalent ternary operation in C. Reducing this representation to a simple *min* or *max* operation makes the code easier to read. Figure 7(b) step (1) shows an example of this reduction.
—*Static resolution of conditionals*: Oftentimes our range analysis generates conditional operations based on relational expressions that can be trivially solved (e.g.,

($n < n + 1$)). Whenever possible, we solve these conditionals statically, eliminating the remaining dead branch.
—Shift *to* mul *and* div *conversion*: Most compilers convert multiplication or division by powers of 2 into shift operations. Whenever possible, we undo this optimization, as explicit multiplications and divisions make it easier for us to identify pointer manipulation patterns. Step (2) of Figure 7(b) illustrates this simplification.
—*Simplification of array indexing*: Computing the address of an element in a one-dimensional array involves (i) finding a base address and (ii) computing an offset. If we identify this pattern in a set of instructions, then we can replace it by the equivalent C expression using the brackets notation. Figure 7(b)'s step (3) demonstrates this simplification when applied onto the array x.
—*Constant propagation*: We solve any arithmetic operation that can be resolved statically (e.g., Figure 7(b) step (4)).
—*Extraction of common subexpressions*: Sometimes the symbolic access bounds of different arrays use the same intermediate subexpressions. When this is the case, we extract such expressions to temporary variables.

From symbolic bounds to annotations. The expressions generated by our analysis compute the lowest and highest addresses that can be accessed in an array. To insert the final directives, however, we need the lowest and highest integer values used to index the array. After simplification, we often end up with a simple indexing expression (Figure 7(b) step (6)) from which we can easily tell the integer index apart (steps (7) and (8)). Even for cases where the final bound expression is not as simple, we can still compute the limit index by subtracting the base address of the array and diving the result by its type size. Once we obtain readable integer indexes for all of the access bounds, we proceed to generate the annotated code seen in Figure 5(b).

The aliasing test in line 2 of Figure 5(b), which checks that arrays x and y do not overlap, can be obtained by inlining the symbolic access bounds that we have just computed into the restrictification inequality defined in Section 3.2. For data transfer pragmas, however, the lowest and highest index are not enough: mapping clauses in both OpenACC and OpenMP 4.0 determine the memory to be copied by specifying a start index and an integer length. This offset determines how many memory positions of an array should be transferred to the device, counting from the starting index. Although the start index will be the same as the lowest access index that we got using our analysis, the length will be the number of memory positions between the access limits. In other words, the length is the difference between the highest and lowest access indexes plus one. The *map* clauses in line 4 of Figure 5(b) show the resulting transfer ranges derived for our example.

5. DATA TRANSFER OPTIMIZATIONS

Instead of mimicking the work of programmers, accurate static analyses let us go beyond what a human user can achieve with code annotation systems. The usual workflow followed by a developer when annotating large programs for GPU parallelization is to (i) reason about each loop nest in separate, (ii) decide if it should or not be sent to the external acceleration device, and (iii) insert data transfer and parallel pragmas accordingly. This modus operandi is justified because loop nests contain most of the parallelization opportunities in a program and are usually small enough to be amenable to human reasoning. Nevertheless, complex syntax and intricate iteration spaces might cause developers to use redundant annotations in the effort to parallelize programs. The function `corr` in Figure 8(a), which was adapted from the *correlation* benchmark in the PolyBench suite, gives us the opportunity to illustrate some of these difficulties, as we explain in Example 5.1.

```
1  void corr(float *A, float *MEAN, float *STDEV, int m, int n) {
2    for (int i = 0; i < m; i++) {
3      for (int j = 0; j < n; j++)
4        MEAN[i] += A[i*n+j];
5
6      MEAN[i] /= n;
7    }
8
9    for (int i = 0; i < m; i++) {
10     for (int j = 0; j < n; j++)
11       STDEV[i] += (A[i*n+j] - MEAN[i]) * (A[i*n+j] - MEAN[i]);
12
13     STDEV[i] = sqrt(STDEV[i] / n);
14   }
15 }
```

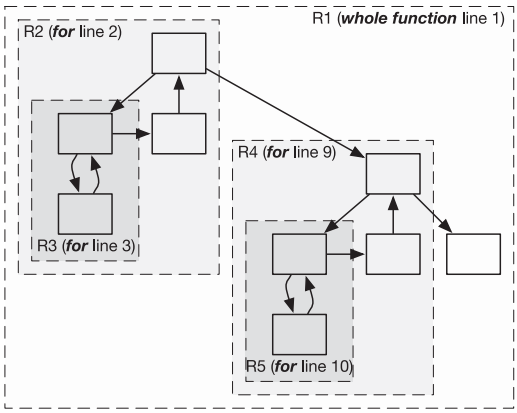
(a)

```
1  void corr(float *A, float *MEAN, float *STDEV, int m, int n) {
2    #pragma omp target map(to: A[:m*n]) \
3                       map(tofrom: MEAN[:m])
4    #pragma omp parallel for
5    for (int i = 0; i < m; i++) {
6      for (int j = 0; j < n; j++)
7        MEAN[i] += A[i*n+j];
8
9      MEAN[i] /= n;
10   }
11
12   #pragma omp target map(to: A[:m*n], MEAN[:m]) \
13                       map(tofrom: STDEV[:m])
14   #pragma omp parallel for
15   for (int i = 0; i < m; i++) {
16     for (int j = 0; j < n; j++)
17       STDEV[i] += (A[i*n+j] - MEAN[i]) * (A[i*n+j] - MEAN[i]);
18
19     STDEV[i] = sqrt(STDEV[i] / n);
20   }
21 }
```

(b)

```
1  void corr(float *A, float *MEAN, float *STDEV, int m, int n) {
2    #pragma omp target data map(to: A[:m*n]) \
3                            map(tofrom: MEAN[:m], STDEV[:m])
4    {
5      #pragma omp target
6      #pragma omp parallel for
7      for (int i = 0; i < m; i++) {
8        for (int j = 0; j < n; j++)
9          MEAN[i] += A[i*n+j];
10
11       MEAN[i] /= n;
12     }
13
14     #pragma omp target
15     #pragma omp parallel for
16     for (int i = 0; i < m; i++) {
17       for (int j = 0; j < n; j++)
18         STDEV[i] += (A[i*n+j] - MEAN[i]) * (A[i*n+j] - MEAN[i]);
19
20       STDEV[i] = sqrt(STDEV[i] / n);
21     }
22   }
23 }
```

(c)



(d)

Fig. 8. (a) Program adapted from the CORR benchmark. (b) Code produced following a simple per-loop annotation approach. (c) Code produced using the *data environment* feature of OpenMP 4.0. (d) CFG for the program divided into SESE regions.

Example 5.1. The approach described in Section 4, once applied onto function `corr` in Figure 8(a), gives us the code in Figure 8(b). This second version of `corr` executes all loop nests in the accelerator. As indicated by the directive in lines 2 and 3 of the new program, the contents of arrays *MEAN* and *A* are sent to the device before the loop in lines 5 through 10 of Figure 8(b). *MEAN*, the output array, is brought back after that loop finishes. For the second loop nest, *MEAN*, *A*, and *STDEV* are sent to the GPU and *STDEV* is brought back. This gives us a total of seven transfer operations.

Annotating loop nests as completely separate objects may cause a developer to miss optimization opportunities. One such opportunity is the reuse of memory transfer operations across different nests of loops. Data transfer operations may impose a prohibitive overhead when offloading code to a GPU [Gregg and Hazelwood 2011]. Not sending to the external device the contents of arrays used solely as computation output, or not bringing back input data, can reduce this overhead. In this article, we go one step beyond: we coalesce data transfers of loops that operate on the same data. For instance,

in Figure 8(b), both parallel loop nests operate over the array *MEAN*; thus, it would be desirable to keep this memory region in the external device during the execution of the whole function rather than bringing it back between the loops, contrary to what has been shown in Example 5.1. To achieve this end, both OpenMP 4.0 and OpenACC allow the user to explicitly define a data environment in the target device. A *data environment* is a syntactic region that determines a set of memory mappings between host and device, which are valid for any parallel region within the environment block. In C/C++, data environments are *scoped blocks* (i.e., a region delimited by braces). Example 5.2 shows the benefits of this optimization.

Example 5.2. Figure 8(c) contains a data environment ranging from line 4 until line 22. In this new version of `corr`, the pragma in lines 2 and 3 states that arrays *MEAN*, *STDEV*, and *A* must be sent to the GPU, and that *MEAN* and *STDEV* must be brought back. Transfers to the GPU happen, semantically, at line 4. Transfers from the GPU happen at line 22.

The reimplementation of Example 5.1, seen in Example 5.2, saves two data transfer operations. This example shows a specific case of a general optimization, which we call *coalescing of data transfer operations*. The goal of this optimization is to automatically encompass as many different parallel loops as possible in the same data environment, hence reducing the amount of data that needs to be transferred between host and device. There are two main steps involved in this process: (i) finding which loops should be surrounded by the same data environment (Section 5.1) and (ii) inserting the actual copy block in the annotated source file (Section 5.2).

5.1. Deciding Which Loops Can Be Merged into Common Transfer Blocks

To decide which loops should have their transfer operations coalesced, we first divide the program's CFG into single-entry single-exit (SESE) regions [Ferrante et al. 1987].[2] A *SESE region* is a subgraph S of a program's CFG G, containing two special nodes: H and L. Any path from $G - S$ to S passes through H, and any path from S to $G - S$ passes through L. Example 5.3 clarifies these notions. To achieve such effect, we resort to the symbolic range analysis seen in Section 3.1, plus classic data dependence analysis [Ferrante et al. 1987]. These two technologies, once combined, let us find the regions that we can optimize.

Example 5.3. Figure 8(d) shows the structure of the CFG for the program in Figure 8(a), with SESE regions highlighted (regions represented by a single basic block or that are not meaningful to the example are not represented in the figure).

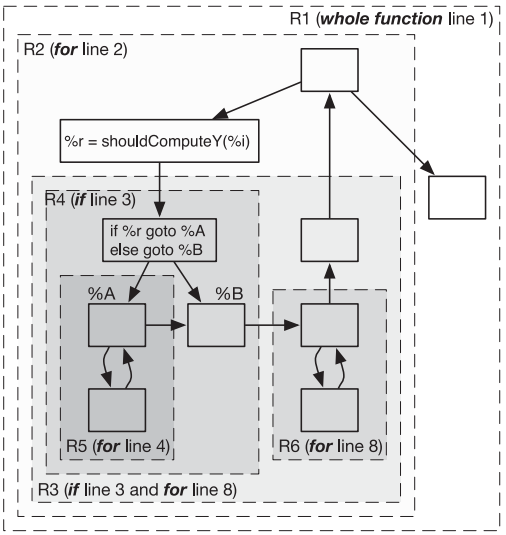
We can analyze a region if we are able to define the access bounds of all arrays used within its scope. An array has bounded accesses if we can determine symbolic ranges for all the expressions used to index it within the region of interest. A region of interest contains only parallel loops, which we can run on the target accelerator. We want to enclose the largest of such regions on a data environment. Example 5.4 provides some insight on this observation.

Example 5.4. Going back to our running example of Figure 8, every array access has limits known right at the start of region *R1*, the largest region in the function (Figure 8(b)). DawnCC places data transfer operations around this region, hence

[2]Our dataflow analyses require a statically known CFG. Hence, the compiler must be able to resolve indirect function calls, for instance. The benchmarks that we use in Section 6 do not sport this feature. Nevertheless, there are well-known techniques to discover the target of such invocations [Shivers 1988].

```
1  void fdtd_2d(float *Y, float *X, float *H, int m, int n) {
2    for (int i = 1; i < m; i++) {
3      if (shouldComputeY(i)) {
4        for (int j = 0; j < n; j++)
5          Y[i*n+j] = Y[i*n+j] - 0.5*(H[i*n+j] - H[(i-1)*n+j]);
6      }
7
8      for (int j = 1; j < n; j++)
9        X[i*n+j] = X[i*n+j] - 0.5*(H[i*n+j] - H[i*n+(j-1)]);
10   }
11 }
```

(a)



(b)

```
1  void fdtd_2d(float *Y, float *X, float *H, int m, int n) {
2    for (int i = 1; i < m; i++) {
3      #pragma omp target data map(to: H[:m*n]) \
4                  map(tofrom: Y[n:(m-1)*n], X[n:(m-1)*n])
5      {
6        if (shouldComputeY(i)) {
7          #pragma omp target
8          #pragma omp parallel for
9          for (int j = 0; j < n; j++)
10           Y[i*n+j] = Y[i*n+j] - 0.5*(H[i*n+j] - H[(i-1)*n+j]);
11       }
12
13       #pragma omp target
14       #pragma omp parallel for
15       for (int j = 1; j < n; j++)
16         X[i*n+j] = X[i*n+j] - 0.5*(H[i*n+j] - H[i*n+(j-1)]);
17     }
18   }
19 }
```

(c)

Fig. 9. (a) Program adapted from the *fdtd_2d* benchmark. (b) SESE regions in the program. (c) Unsafe annotations.

arriving at the code seen in Figure 8(c), which is semantically equivalent to the original program seen in Figure 8(a).

5.2. Carrying On with Copy Fusion into Source Code

As our intent is to insert annotations in the original source code, the IR over which we run our static analyses should be as close to the input program's text as possible. To this extent, when generating a program's CFG, we disable any optimization that can move code across different points of said CFG. One particular optimization, however, cannot be disabled in most cases: the conversion to SSA form [Cytron et al. 1991]. Most mainstream compilers will convert a program to SSA form when generating its IR. The virtual names inserted in this process make it harder to map instructions in a program's CFG to its original text—a fact that Example 5.5, adapted from PolyBench's `fdtd_2d`, clarifies.

Example 5.5. Figure 9(b) shows the SESE regions of function `fdtd_2d`, seen in Figure 9(a). The call to `shouldComputeY` in line 3 can yield side effects. Thus, dependence analysis tells us that the outer loop might not be parallel. The two inner loops in lines 4 and 5 and 8 and 9, however, are stencil kernels that can be annotated to run in the accelerator. In the CFG, the largest region for which we have full symbolic range information (*R3*) goes from right before the *if* statement in line 3 to right after the *for* loop in lines 8 and 9. However, the call to `shouldComputeY` is syntactically inside our target region in the program's text (lines 3 through 9) but falls outside of it in the program's CFG, due to the virtual name *%r* inserted during the SSA transformation.

Example 5.5 shows that a SESE region in a program's CFG in SSA form may not directly map to a SESE region in the original program text. In Example 5.5, surrounding region *R3* with a data transfer block in the source program (as seen in Figure 9(c)) could change its semantics. This modification happens if the call to `shouldComputeY` modifies the values stored in arrays H, Y, or X after they have been sent to the GPU.

To ensure that our fusion strategy is correct under SSA conversion, we resort to a simple restriction: the data environment created to cover an analyzable SESE region goes from the point right before its first parallel loop (including its preheader) to the point right after its last parallel loop. SSA conversion does not change the structure of loops nor moves computation across them. Thus, all computation inside the data environment defined by the preceding restriction will be the same in both a program's text and in its CFG. Example 5.6 provides some intuition on why such an approach is safe.

Example 5.6. If we were to surround region *R3*, in Figure 9(b), with a data transfer block, then this region should go from the loop in line 4 to the end of the loop in lines 8 and 9 in Figure 9(a). Thus, this data transfer block does not include the call `shouldComputeY`. This exclusion holds in both the source text (Figure 9(a)) and the CFG (Figure 9(b)).

Despite being safer, our augmented fusion approach still suffers from one problem: it is not always syntactically valid to surround different loops within the same region with a data transfer block. Inserting a block around both loops in Example 5.6 is not syntactically valid. The beginning of the block would fall inside the *if* in line 3 of Figure 9(a), and its end would lay outside. To avoid such problems, we use a last verification step. This check uses a data structure that we call a *scope tree*.

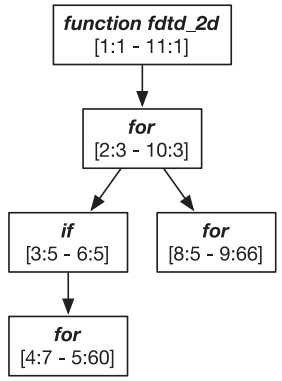
Ensuring safety with scope trees. A scope tree is a tree-like data structure in which nodes represent statements in the source program that can create SESE regions (statements such as *for*, *if*, and *switch cases*). Each node contains its text range: the coordinates in the source code (line and column of the character in the text) that represent its first and last character. For instance, the *if* statement in Figure 9(a) ranges from line 3 column 5 to line 6 column 5. This data structure can be constructed using a C parser. The main property of a scope tree, which makes it useful to our purposes, is that the range of any node falls either completely inside or completely outside another node's range. For instance, it has the *property of balanced parentheses*: if node c_1 is a child of node c_0, then the lines that c_1 cover lay within the lines that c_0 represents. Example 5.7 illustrates this property.

Example 5.7. Figure 10(b) shows the scope tree for the function `fdtd_2d`, seen in Figure 10(a). This tree has the property of balanced parentheses. For instance, the only node *if* spans from lines 3 to 6. Its child, a node *for*, spans from lines 4 to 5.

We say that a data environment block is safe to be inserted when it preserves the property of balanced parentheses. This additional verification tells us that enclosing the loops in region *R3* of Figure 9(b) is not syntactically safe, as we explained earlier. The largest regions of our example that follow all desired properties and meet all safety restrictions are now *R5* and *R6* of Figure 9(b). Therefore, in this example, each parallel loop should be annotated with separate data transfer operations. DawnCC produces, in this case, the code seen in Figure 10(a). Notice that the existence of a well-defined scope tree is not essential for DawnCC: if this data structure is not available or cannot be generated, the tool will use uncoalesced memory transfers to parallelize code.

```
1  void fdtd_2d(float *Y, float *X, float *H, int m, int n) {
2    for (int i = 1; i < m; i++) {
3      if (shouldComputeY(i)) {
4        #pragma omp target map(to: H[(i-1)*n:2*n]) \
5                           map(tofrom: Y[i*n:n])
6        #pragma omp parallel for
7        for (int j = 0; j < n; j++)
8          Y[i*n+j] = Y[i*n+j] - 0.5*(H[i*n+j] - H[(i-1)*n+j]);
9      }
10
11     #pragma omp target map(to: H[i*n:n]) \
12                          map(tofrom: X[i*n:n])
13     #pragma omp parallel for
14     for (int j = 1; j < n; j++)
15       X[i*n+j] = X[i*n+j] - 0.5*(H[i*n+j] - H[i*n+(j-1)]);
16   }
17 }
```

(a)



(b)

Fig. 10. (a) Annotations produced for program in Figure 9(a). (b) Scope tree.

6. EXPERIMENTS

We have evaluated our techniques using two different compilers and architectures to answer the following question: can our annotations produce speedups on typical kernels without intervention from developers? In what follows, we describe our methodology and discuss our results.

Benchmarks. For our experiments, we chose the PolyBench programs used by Grauer-Gray et al. [2012] with linearized multidimensional arrays. Benchmarks come with five input sizes: *mini*, *small*, *medium*, *large*, and *huge*.

Hardware. We experimented with the following setups:

—*Desktop*: Intel Xeon CPU E5-2620, with six cores of 2.00GHz and 16GB of RAM (DDR2), running Linux Ubuntu 12.04 3.2.0, equipped with a GPU model GeForce GTX 670, with 2GB of RAM (CUDA Compute Capability 3.0).
—*Phone*: Exynos 7420 AArch64 processor with 4GB of RAM running Android 5.1.1 and equipped with a GPU model ARM Mali-T760 with 913MB of RAM and eight parallel compute units.

Compilers. We used the following compilers to translate annotated code to binaries:

—*GPUClang*: An LLVM/Clang-based compiler that implements the OpenMP Accelerator Model. It adds an *OpenCL runtime library* to LLVM 3.5.0 that supports OpenMP offloading to devices like GPGPUs and FPGAs. The kernel functions are extracted from the OpenMP region and are dispatched as OpenCL or SPIR[3] code to be executed by the device. This whole process is transparent and does not require any programmer intervention. The OpenCL runtime library has two main functionalities: (a) it hides the complexity of the OpenCL code from the compiler, and (b) it provides a mapping from OpenMP directives to the OpenCL API, thus avoiding the need for device manufacturers to build specific OpenMP drivers for their GPUs or FPGAs. GPUClang also leverages on ISL polyhedral model optimizations [Verdoolaege et al. 2013] to transform the extracted loops so that they can be tiled and mapped to the blocks and threads in the OpenCL kernel code. GPUClang runs on the phone setup.
—*pgcc*: PGI C Compiler version 16.1 64 bits. Runs on the desktop setup and translates OpenACC to parallel code.

[3]https://www.khronos.org/spir.

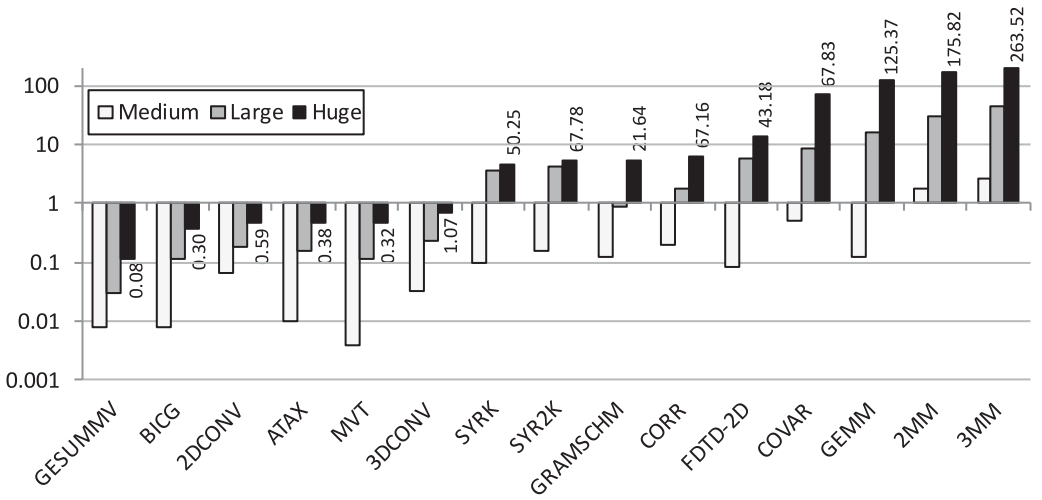


Fig. 11. Desktop (DawnCC+pgcc vs. pgcc). We compare speedup due to the annotations inserted by DawnCC to execution of sequential code on the desktop setup. Both programs, original and annotated, have been compiled with pgcc. The *y*-axis shows speedup in number of times. The higher the bar, the better. Numbers represent absolute runtime, in seconds, of benchmarks with largest inputs, running on the CPU.

6.1. Runtime Results

*Desktop setup (*DawnCC + *pgcc [OpenACC])*. Figure 11 shows the relative runtime of annotated code compared against the same code, without our annotations. Binaries are produced by pgcc -O3. Programs run five times; bars show averages. When probing the GPU's execution time, we include the time to transfer data to and from the GPU. Variance is negligible, and hence we will not provide error intervals. We observe very large speedups in four benchmarks: 2MM, 3MM, COVAR, and GEMM. These are embarrassingly parallel applications, which benefit substantially from the SIMD execution model of a GPU. We have observed slowdowns in six benchmarks. These slowdowns happen in benchmarks that run for very short times. To emphasize this point, we show absolute runtimes for largest inputs next to each benchmark in Figure 11.

*Phone setup (*DawnCC + *GPUClang [OpenMP])*. Figure 12 shows the speedup that we obtain when comparing the annotated code, compiled with GPUClang, against Clang 3.5 -O3, on the phone setup. Each program has been executed five times, and bars show averages. In this setup, we observe speedups in more benchmarks, although we have not gotten results as dramatic as those seen in the desktop setup. Again, GEMM is the benchmark where we got the more noticeable gains. The less impressive speedups are due to the fact that the difference, in terms of number of available cores, between the Mali GPU and the Exynos CPU is smaller than the difference between the GTX GPU and the Xeon CPU.

One of the issues observed in the runtime results is that while in the desktop setup the speedups are as expected, increasing with the size of the inputs, in the phone setup some results do not show this behavior. For example, in 2DCONV, ATAX, 3DCONV, and SYR2K, speedups do not always increase with input sizes. Careful analysis reveals that while in the desktop the cost of buffer allocation is almost constant, this is not the case in the phone setup. To support this statement, GPUClang was used to profile our benchmarks. This profile divides the total execution time of each application into three parts: (i) kernel computation, (ii) data offloading, and (iii) OpenCL management. This last parcel includes, for instance, context creation, queue management, and the creation

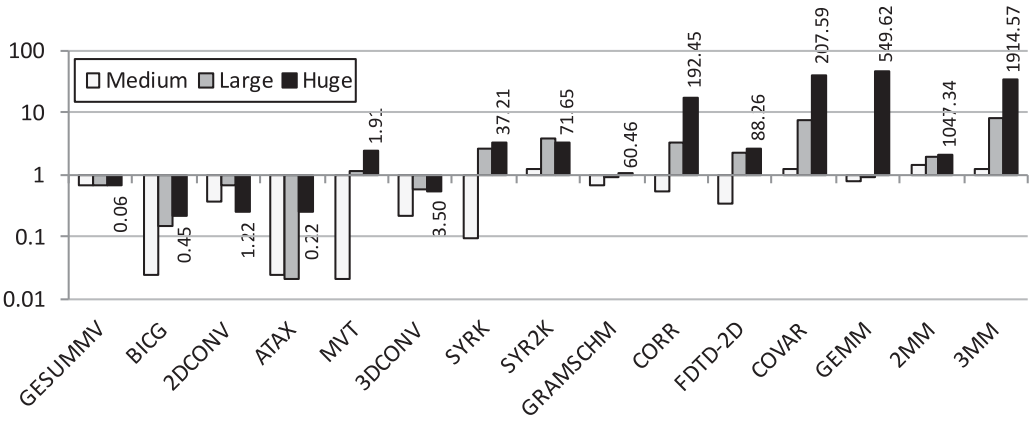
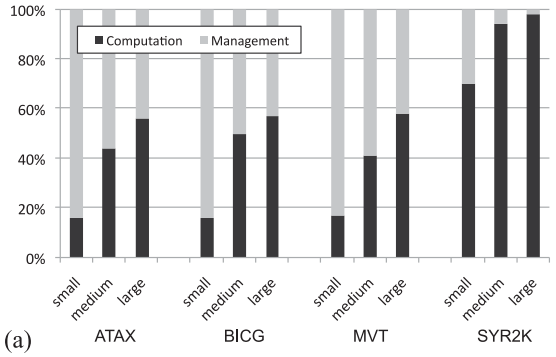


Fig. 12. Phone (DawnCC+GPUClang vs. GPUClang). We compare the code that DawnCC has annotated with OpenMP pragmas and compiled with GPUClang to the benchmarks without the annotations. The *y*-axis shows speedup in number of times. The higher the bar, the better. Numbers represent absolute runtime, in seconds, of benchmarks with largest inputs, running on the CPU.



Size (MB)	ARM/Mali (s)	Intel/Iris (s)
16	0.01	0.0002
64	0.251	0.00025
256	1.385	0.00021
320	1.623	0.00024
384	2.054	0.00027
448	2.502	0.00025

(b)

Fig. 13. (a) Percentage of management time compared to the time actually spent into data movement plus computation observed in the execution of different benchmarks in the phone setup. (b) Time required for buffer creation in different architectures.

of the buffers that will receive the data. Profiling reveals that the OpenCL driver takes a considerable share of the total execution time on most PolyBench programs, as Figure 13 shows. Buffer creation time varies with its size and has a large impact on program performance, particularly when the time required to offload data approaches the time needed to perform the actual computation. Longer executions times tend to amortize this overhead. We compared these results to another heterogeneous device that GPUClang targets, an Intel Core i5 processor with an integrated Intel/Iris GPU. Figure 13(b) shows the result of this comparison. Whereas times spent in the creation of buffers is almost constant in the latter, for different input sizes, it increases in the former. These observations seem to indicate that the OpenCL driver used for the ARM/Mali architecture can still be improved.

6.2. The Impact of Copy Fusion

Figure 14 shows results produced by the optimization described in Section 5. The reduction in the number of pragmas indicates how many regions in the code had their transfer operations coalesced. The number of copies account for how many arrays are transferred to the accelerator in each version. DawnCC has been able to eliminate

Benchmark	# of pragmas Orig.	Opt.	# of copies Orig.	Opt.	Runtime (sec) Orig.	Opt.
GESUMMV	1	1	7	7	•	•
BICG	3	1	10	7	0.80	0.70
2DCONV	1	1	3	3	•	•
ATAX	3	1	10	6	0.81	0.70
MVT	2	1	8	7	0.66	0.63
3DCONV	1	1	3	3	•	•
SYRK	2	1	5	3	10.89	10.87
SYR2K	1	1	4	4	•	•
GRAMSCHM	1	1	6	6	•	•
CORR	4	4	14	14	•	•
FDTD-2D	1	1	7	7	•	•
COVAR	3	1	9	6	0.92	0.90
GEMM	1	1	4	4	•	•
2MM	2	1	8	7	1.08	1.06
3MM	3	1	12	10	1.33	1.32
Total	29	18	110	94		

Fig. 14. Results produced by the copy fusion optimization of Section 5 in the desktop setup. "Orig." denotes the original program, and "Opt." is its optimized version. Runtime is the GPU's, for the largest input size.

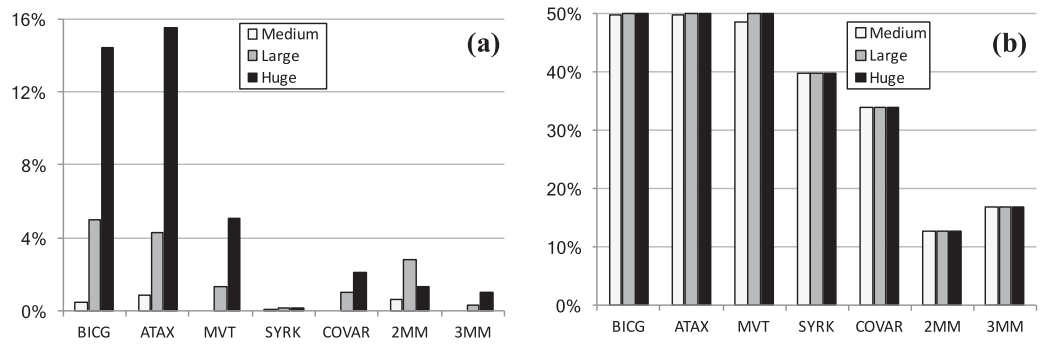


Fig. 15. (a) Performance improvement due to the copy fusion optimization discussed in Section 5. Bars show percentage of speedup of optimized over nonoptimized code. The higher the bar, the better. (b) Percentage of copy time saved. X% is the reduction in the time spent copying data. The higher the bar, the better.

redundant copies in 7 out of the 15 benchmarks available. Manual inspection of the untouched benchmarks reveal the absence of further opportunities for copy coalescing. In two of the benchmarks, ATAX and COVAR, DawnCC could produce a transfer block that surrounds the entire program kernel. In the latter case, three loop nests, containing seven loops, have been placed within a single transfer block. The final result of this optimization is performance, as the two right columns in Figure 14 show.

Figure 15(a) shows the percentage of speedup that we obtain with copy fusion. These numbers are modest for most of the benchmarks, but they refer to only the time to copy data between host and device. The higher the asymptotic complexity of the kernel, the lower will be the gains produced by copy fusion, because the copy cost is amortized on the program's runtime cost. Nevertheless, if we consider only the time to move data, then the results produced by copy fusion are more noticeable, as Figure 15(b) shows. This figure outlines only the percentage of time saved to copy data. In a few benchmarks (e.g., ATAX, BICG and MVT), copy fusion could reduce in almost 50% the time spent in memory transfers. Notice that there exist situations that prevent DawnCC from fusing copies. The most important is due to function calls. Because DawnCC runs

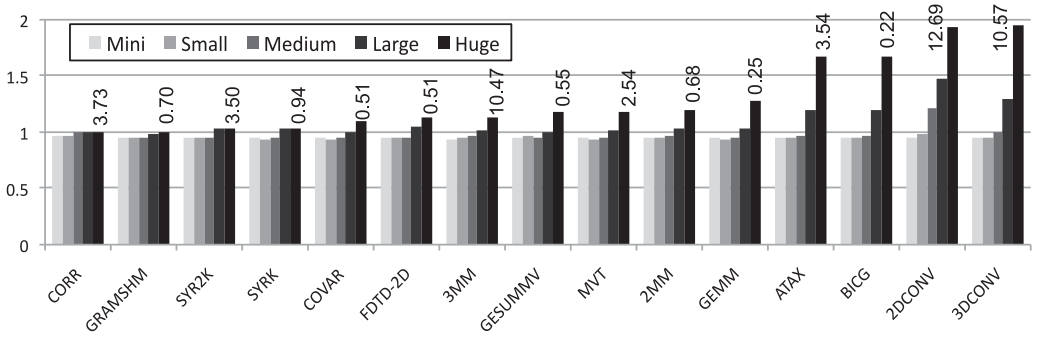


Fig. 16. Comparison between DawnCC and CUDA Unified Memory. The *y*-axis shows speedup, in number of times, of DawnCC over unified memory. The higher the bar, the better. Numbers represent absolute runtime, in seconds, of programs produced by DawnCC, when executed with largest inputs available.

intraprocedurally, it never moves copies around function invocations. To mitigate this limitation, users can specify side-effect-free functions.

6.3. DawnCC Versus CUDA Unified Memory

DawnCC inserts explicit copy directives in programs, moving data between host and device. In 2013, Nvidia released a new technology, called *CUDA Unified Memory*, which makes such copies unnecessary. This capability consists in a runtime system that manages buffer allocation and data coherence between the CPU and the GPU memories. Following the taxonomy of Figure 1 of Jablin et al. [2011], CUDA Unified Memory is a fully automatic management system. Its main benefit is productivity, as programmers no longer need to be concerned about data movement when developing GPGPU applications. In this section, we investigate how DawnCC's automatically inserted pragmas perform when compared to CUDA Unified Memory. In this experiment, we focus on the desktop setup because the unified memory feature is not available on the phone setup.

Figure 16 compares both approaches: explicit copies and unified memory. Binaries in the former group are produced with DawnCC, plus the following invocation of the PGI compiler: `pgcc -acc -ta=tesla`. Binaries in the latter category are produced with the PGI compiler only, via the command `pgcc -acc -ta=tesla:managed`. We ran each sample five times. Variance was less than 10% for the *mini* input size and negligible for inputs of *huge* size. The figure lets us conclude that explicit copies tend to improve performance for large inputs. This fact is evident, once we observe that, for the largest available inputs, every one of DawnCC's programs has performed better than those that use managed memory. Speedups range from no gain (in CORR) to 1.92× (in 3DCONV). For small inputs, CUDA Unified Memory tends to outperform explicit copies. Indeed, for the smallest inputs available, managed memory outperformed explicit copies in every benchmark—gains ranging from 8% (in GRAMSCHM) to 3% (in CORR). We emphasize that the runtime difference between *mini* and *huge* inputs is substantial. For instance, in CORR, the smallest input leads to an execution time of 0.1592 seconds, with the explicit copies produced by DawnCC. However, the largest input gives us 10.4706 seconds of execution.

6.4. DawnCC Versus PPCG

PPCG is a source-to-source optimizer implemented by Verdoolaege et al. [2013] that translates sequential C programs into CUDA programs. PPCG performs a very extensive suite of optimizations on the target program, which includes tiling, unrolling, and some kind of fission, as the same loop may be broken into separate kernels. In

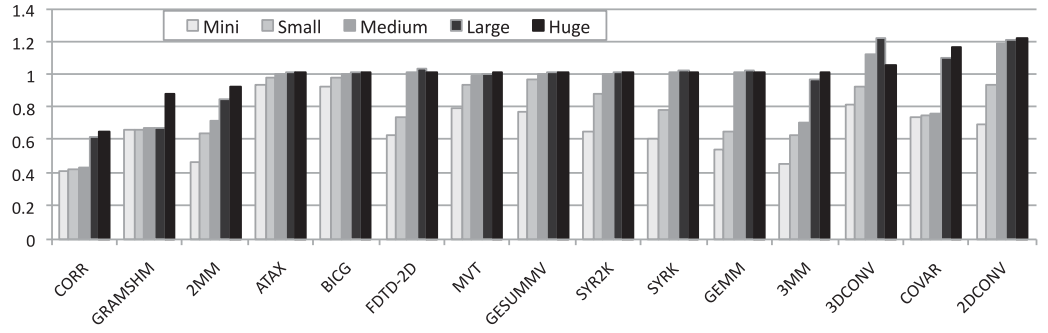


Fig. 17. Ratio between the time to perform data movement between copies inserted by PPCG and by DawnCC. The higher the bar, the best for DawnCC.

this section, we compare the quality of the data movement code produced by PPCG and DawnCC for the desktop setup. Generation of copies is the only feature common between these two tools. In other words, they do not solve the same problems: PPCG optimizes CUDA code, and DawnCC inserts data movement primitives into C programs. Moreover, they use different compiler back-ends: in the desktop setup, DawnCC uses pgcc to produce CUDA binaries, whereas PPCG first transforms the program, using polyhedron-based optimizations, outputs CUDA, and then invokes nvcc to produce machine code.[4] PPCG also requires us to mark which program regions should be compiled into CUDA. In this experiment, we reuse the annotations available in the benchmarks evaluated by Verdoolaege et al [2013]. Therefore, we shall restrict this study to the time that each code—produced by either PPCG or DawnCC—takes to move data between host and device on the desktop setup.

Figure 17 summarizes the result of this comparison. For small inputs, usually the copies produced by PPCG run faster. As input sizes escalate, DawnCC catches up. For some benchmarks, such as COVAR and 2DCONV, DawnCC achieves speedups of 20% for the largest inputs we had. Such gains are due to two optimizations performed by DawnCC: the ability to copy only part of an array and the fusion of copies. Manual inspection of the binaries produced by PPCG does not indicate that it applies such optimizations. Quoting its original description in Section 7.1 of Verdoolaege et al. [2013]: "any accessed array is allocated in its entirety on the device."

The overhead of DawnCC, which we have observed for small inputs, is due to the restrictification checks and the boilerplate code necessary to compute memory regions. This overhead tends to disappear for large inputs, as Figure 17 shows. Partial copies tend to pay off once inputs become larger. And restrictification is necessary for correctness. It is possible to create programs that would produce incorrect outputs, once parallelized by PPCG. The saxpy_3off kernel seen in Figure 5 is such an example: to obtain wrong outputs, it suffices to pass the same array in place of arguments x and y. Therefore, we believe that the techniques discussed in this article could be used to improve PPCG's performance and ensure its soundness.

6.5. DawnCC Versus Manual Code Annotation

In this section, we compare the code that DawnCC inserts against human-made annotations. To this end, we use a publicly available version of PolyBench, which contains OpenMP 4.0 annotations,[5] and we also use three larger programs from Rodinia [Che

[4]For the interested reader, PPCG produces code that is, on average, 47% faster than that of pgcc. For PolyBench, we have Min (FDTD-2D) = 1.03×, Max (CORR) = 4.96×, Median = 1.09×, and GeoMean = 1.47×.
[5]https://github.com/mattoslf/UniBenchV2.

et al. 2009] and Parboil [Stratton et al. 2012]. These programs let us illustrate some shortcomings of manual code annotation. The annotations in PolyBench differ from those we insert automatically in several ways. First, the human developer always copies the entire array, using as its size the information available at the point where the array is declared. In compiler jargon, this is a *forward region analysis* [Nazaré et al. 2014]. DawnCC performs a *backward analysis*: the way an array is indexed determines the region that is copied. The backward approach has two advantages: it lets us copy parts of an array, and it handles library code. Concerning this last point, DawnCC does not need to have access to the point where the array is created, as it does not need its size at the moment of allocation. Thus, we can annotate functions even when we do not have access to the entire program's source code. Performing a backward analysis is difficult for a person, as it might involve combining several different ranges bounded by complex expressions. This is not a serious issue for PolyBench, but it becomes relevant once we consider larger benchmarks. For instance, DawnCC annotates the core loop of Rodinia's LUD. This loop contains two nested loops, where we find nine accesses to an array a, indexed by seven different expressions, all of them involving three variables: size, i, and j. Finding minimum bounds for this array requires DawnCC to insert 63 lines of code in the program—a tedious task. Yet the annotated program, on the desktop setup, runs in 2.50 seconds, whereas its original version runs in 25.70 seconds. Notice that this result is still far from the optimum. Continuing with this example, LUD has a version manually implemented in C for CUDA. When compiled with Nvidia's nvcc, this executable runs in 0.07 seconds. Visual inspection of the annotations that DawnCC inserts does not reveal any obvious way to achieve this speedup, and thus we believe that pragma-based parallelization, in general, still has room for improvement.

The human-annotated version of PolyBench also does not include restrictification checks. It is difficult for a person to create these checks. As an example, we consider MRI_Q, one of Parboil's benchmarks. This program has 1,171 lines of code. DawnCC annotates its main kernel, ComputeQGPU. The annotated region contains six different arrays, three of which are arrays of structs. The restrictification checks perform 12 different comparisons. The manual insertion of such tests imposes on developers a burden that automatic tools such as DawnCC can handle well. And in terms of performance, their overhead pays off. The parallelized version of MRI_Q runs in 0.80 seconds on the desktop setup against 21.80 seconds of the sequential program.

The manually annotated version of PolyBench misses some opportunities for fusing copies. This omission happens, for instance, in 2MM. In the main kernel, the human developer could have coalesced copies of one particular array: (C). DawnCC does it. Nevertheless, being an automatic tool, sometimes DawnCC performs transformations that lead to worse runtimes. As an example, Rodinia's LAVAMD contains one function, kernel_cpu, which has four nested loops. DawnCC annotates the second-innermost loop. Our annotations contain 140 lines of C code, encompassing computation of bounds, and restrictification. Because this code runs within two loops, the final performance of the parallel program is worse: it executes in 10.30 seconds, whereas the sequential program runs in 6.90 seconds. We believe that identifying situations like this, statically, is an interesting and difficult problem, which we hope to study in the future.

7. RELATED WORK

Much work has been done to bring general-purpose GPUs closer to software developers. In this section, we briefly review some of this work, giving particular emphasis to source-to-source translators aimed at automatic parallelization of programs.

Annotation systems. Annotation systems such as OpenMP 4.0 [Jaeger et al. 2015], OpenSs [Meenderinck and Juurlink 2011], and Open-ACC Standard [2013] are a

simple, yet powerful alternative to the development of high-performance software. Such systems are not a programming language per se; rather, they work as a *metalanguage*, which, once combined with a host language, typically Fortran, or C, let developers inject parallel semantics into its standard syntax. The emergence of such systems has led to a resurgence of interest in parallelizing compilers. OpenACC, for instance, has been a target of several different compilers, such as AccUll [Reyes et al. 2012], ipmacc [Lashgar et al. 2014], OpenARC [Lee and Vetter 2014], and pgcc [Ghike et al. 2014]. Similarly, OpenMP 4.0 is already supported by several mainstream compilers, including gcc 4.9.0 (for C/C++), gcc 4.9.1 (for Fortran), icc 15.0 (C/C++/Fortran), and LLVM's Clang 3.7, which offers partial support to OpenMP 4.0 for C/C++. Our tool, DawnCC, is not an OpenACC or OpenMP 4.0 compiler, and hence it does not compete against the technologies that we have just mentioned. Instead, DawnCC works one level up: inserting the annotations that will be later translated by an OpenACC or OpenMP 4.0 compliant compiler.

Static analyses. There exists an enormous corpus about the automatic parallelization of software. For an overview about the classic techniques, we recommend the book of Wolfe [1995]. The goal of our work is not to parallelize programs; instead, we want to give programmers the tools to benefit from latent parallelism already available in code. Thus, our interval analysis is needed only when we use pragmas to offload code to an external device. We do not need data copy directives when using OpenMP to produce parallel code meant to run on multicore CPUs, for instance.

The range analysis that we use in this work is also not new. We took the implementation of Alves et al. [2015]. They have used this technique to restrictify pointers in C programs with the goal to enable more compiler optimizations. Several other researchers have used similar techniques with different purposes, such as enabling the discovery of more parallelism in programs [Rus et al. 2003]. Our goal is not to design better range analyses, nor to infer more parallel regions in programs. Instead, we use this technique to find the boundaries of memory regions that need to be transferred to an accelerator.

Source-to-source parallelization. The compiler literature describes several tools that contain features also present in DawnCC: PPCG [Verdoolaege et al. 2013], par4all [Amini et al. 2012; Guelton et al. 2012], C-to-CUDA [Baskaran et al. 2010], Bones [Nugteren and Corporaal 2014], and the CAPS Compiler [Andión et al. 2016]. One essential difference between these tools and DawnCC is the latter's focus on correctness in face of pointer aliasing. Much of our implementation decisions were necessary to keep our code transformations sound. That is the main reason behind our restrictification checks and the intraprocedural scope of our analyses. Previous work requires the programmer to certify the absence of aliasing in the target programs, for instance, using the `restrict` keyword in C. Consequently, all of these five previous publications contain examples that would not be transformed correctly in case memory regions overlap. For an example, see the discussion at the end of Section 6.4.

Furthermore, most of the previous work focuses on the implementation of loop transformations that increase parallelism and locality. Performing such transformations is not a goal of DawnCC. For instance, several optimization frameworks based on the polyhedral model have been used for automatic generation of OpenMP and GPU code [Verdoolaege et al. 2013; Baskaran et al. 2010]. These tools generate calls to data copy functions by inspecting the iteration domains of arrays used within parallel loops. The symbolic limits generated by such frameworks and the ones generated by the analysis chosen for this work have similar purposes [Alves et al. 2015], each having specific advantages. For instance, on our side, the method applied in this

article can handle nonaffine regions of code. However, the analyses implemented in polyhedral-based tools are able to perform more aggressive transformation in loops, such as tiling and fission. This extra power comes at the cost of a higher compilation time.

Automatic insertion of offloading primitives. There are several techniques to insert offloading primitives in programs [Alias et al. 2013; Guelton et al. 2012; Nugteren and Corporaal 2014; Verdoolaege et al. 2013]. They differ with relation to the static analysis used to infer memory bounds. Although some of the descriptions are not explicit, most of them are restricted to affine regions. By relying on a symbolic analysis, DawnCC handles intervals containing multiplication between variables, for instance. Nevertheless, each of those tools exercise some aspects of static analysis that makes it unique. For instance, Bones [Nugteren and Corporaal 2014] relies on an ingenious two-phase approach. First, patterns are mined, classified, and annotated in the source code. Then, in a second stage, a compiler translates such annotations into acceleration code. Par4all [Guelton et al. 2012], in turn, finds the boundaries of arrays, using an abstraction of memory regions called *convex array regions* [Triolet et al. 1986]. Convex regions, a notion in place in the PIPS compilation framework [Amini et al. 2012], are an elegant notation to describe affine spaces. One of the interesting aspects of Par4all is its interprocedural analysis, which lets it move copies outside the boundaries of functions. Finally, PPCG [Verdoolaege et al. 2013], although unable to optimize memory transfers, is unbeatable on its generality and reach: the polyhedral transformations that it carries out without any intervention from users seem unparalleled among the currently available tools. Indeed, PPCG was the only compiler among these three that we have been able to use in all of our benchmarks without having to adapt them; the others seem to not be maintained anymore, and we have not been able to use them. Nevertheless, we believe that DawnCC's ability to extract information from the low-level IR of an industrial-strength compiler—and map it back into source code—singles it out among these tools. In this sense, DawnCC is not the first tool that analyzes the LLVM IR to produce parallel code. For instance, Raghesh's extension to PollyLLVM [Raghesh 2011] inserts OpenMP directives into LLVM's assembly, targeting multicore architectures. However, these directives only exist in the low-level representation, and hence it is difficult for programmers to edit or maintain them.

8. CONCLUSION

This article has presented a suite of static analysis techniques that annotates code automatically with acceleration pragmas. These techniques are currently available as part of a source-to-source compiler, DawnCC, which we have used to transform well-known benchmarks, thus obtaining speedups of over 100×. To the best of our knowledge, DawnCC is the only tool able to insert OpenACC/OpenMP pragmas in sequential C programs without any intervention of users, and ensure its correctness, even in face of pointer aliasing. In this process, DawnCC performs several analyses and optimizations over the target code, such as restrictification of pointers and fusion of redundant copies. The existence of a tool such as DawnCC opens up different doors for researchers. Future work includes, for instance, the study of heuristics to avoid annotating unprofitable code regions and the possibility to output hints to programmers to support more extensive code parallelization. Concerning this last point, there are two syntactic constructions that hinder DawnCC from bounding arrays: (i) indirect accesses (e.g., `x[y[i]]`) and (ii) function calls (e.g., `x[foo()]`). We believe that directives a là PENCIL [Baghdadi et al. 2015] could be used to handle such situations. DawnCC can be used through an online interface, available at http://cuda.dcc.ufmg.br/dawn/.

ACKNOWLEDGMENTS

We thank Michael Frank and Fabrício Ferracioli from LGE for reading our drafts. An earlier version of this work was presented at SBAC-PAD'16 [Mendonça et al. 2016]. We thank the referees of this first publication for insightful comments and suggestions.

REFERENCES

C. Alias, A. Darte, and A. Plesco. 2013. Optimizing remote accesses for offloaded kernels: Application to high-level synthesis for FPGA. In *Proceedings of the 2013 DATE Conference (DATE'13)*. 575–580.

Péricles Alves, Fabian Gruber, Johannes Doerfert, Alexandros Lamprineas, Tobias Grosser, Fabrice Rastello, and Fernando Magno Quintão Pereira. 2015. Runtime pointer disambiguation. In *Proceedings of the 2015 OOPSLA Conference (OOPSLA'15)*. ACM, New York, NY, 589–606.

M. Amini, C. Ancourt, F. Coelho, B. Creusillet, S. Guelton, F. Irigoin, P. Jouvelot, R. Keryell, and P. Villalon. 2012. *PIPS Is Not (Only) Polyhedral Software*. Technical Report. IMPACT.

Lars Ole Andersen. 1994. *Program Analysis and Specialization for the C Programming Language*. Ph.D. Dissertation. DIKU, University of Copenhagen.

José M. Andión, Manuel Arenaz, François Bodin, Gabriel Rodríguez, and Juan Tourino. 2016. Locality-aware automatic parallelization for GPGPU with OpenHMPP directives. *International Journal of Parallel Programming* 44, 3, 620–643.

R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, et al. 2015. PENCIL: A platform-neutral compute intermediate language for accelerator programming. In *Proceedings of the 2015 PACT Conference (PACT'15)*. IEEE, Los Alamitos, CA, 138–149.

M. M. Baskaran, J. Ramanujam, and P. Sadayappan. 2010. Automatic C-to-CUDA code generation for affine programs. In *Proceedings of the 2010 CC Conference (CC'10)*. 244–263.

Carlo Bertolli, Samuel F. Antao, Alexandre E. Eichenberger, Kevin O'Brien, Zehra Sura, Arpith C. Jacob, Tong Chen, and Olivier Sallenave. 2014. Coordinating GPU threads for OpenMP 4.0 in LLVM. In *Proceedings of the LLVM-HPC Conference (LLVM-HPC'14)*. IEEE, Los Alamitos, CA, 12–21.

Victor H. S. Campos, Péricles Rafael Oliveira Alves, Henrique Nazaré Santos, and Fernando Magno Quintão Pereira. 2016. Restrictification of function arguments. In *Proceedings of the 2016 CC Conference (CC'16)*. ACM, New York, NY, 163–173.

Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IISWC Conference (IISWC'09)*. IEEE, Los Alamitos, CA, 44–54.

R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems* 13, 4, 451–490.

Gregory J. Duck and Roland H. C. Yap. 2016. Heap bounds protection with low fat pointers. In *Proceedings of the 2016 CC Conference (CC'16)*. ACM, New York, NY, 132–142.

Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems* 9, 3, 319–349.

Swapnil Ghike, Ruben Gran, María Jesús Garzarán, and David A. Padua. 2014. Directive-based compilers for GPUs. In *Proceedings of the 2014 LCPC Conference (LCPC'14)*. 19–35.

S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *Proceedings of the 2012 InPar Conference (InPar'12)*. IEEE, Los Alamitos, CA, 1–10.

Chris Gregg and Kim Hazelwood. 2011. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *Proceedings of the 2011 ISPASS Conference (ISPASS'11)*. IEEE, Los Alamitos, CA, 134–144.

Tobias Grosser, Armin Größlinger, and Christian Lengauer. 2012. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters* 22, 4, 1–28.

Serge Guelton, Mehdi Amini, and Béatrice Creusillet. 2012. Beyond do loops: Data transfer generation with convex array regions. In *Proceedings of the 2012 LCPC Conference (LCPC'12)*. 249–263.

Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. 2011. Automatic CPU-GPU communication management and optimization. In *Proceedings of the 2011 PLDI Conference (PLDI'11)*. ACM, New York, NY, 142–151.

Julien Jaeger, Patrick Carribault, and Marc Pérache. 2015. Fine-grain data management directory for OpenMP 4.0 and OpenACC. *Concurrency and Computation: Practice and Experience* 27, 6, 1528–1539.

Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 1992. Lazy code motion. In *Proceedings of the 1992 PLDI Conference (PLDI'92)*. ACM, New York, NY, 224–234.

Ahmad Lashgar, Alireza Majidi, and Amirali Baniasadi. 2014. IPMACC: Open source OpenACC to CUDA/OpenCL translator. arXiv:1412.1127.

Chris Lattner and Sarita Adve. 2004. LLVM: A compilation framework for lifelong program analysis transformation. In *Proceedings of the 2004 CGO Conference (CGO'04)*. IEEE, Los Alamitos, CA, 75–86.

S. Lee and R. Eigenmann. 2010. OpenMPC: Extended OpenMP programming and tuning for GPUs. In *Proceedings of the 2010 SC Conference (SC'10)*. IEEE, Los Alamitos, CA, 1–11.

Seyong Lee and Jeffrey S. Vetter. 2014. OpenARC: Open accelerator research compiler for directive-based, efficient heterogeneous computing. In *Proceedings of the 2014 HPDC Conference (HPDC'14)*. ACM, New York, NY, 115–120.

Cor Meenderinck and Ben Juurlink. 2011. Nexus: Hardware support for task-based programming. In *Proceedings of the 2011 DSD Conference (DSD'11)*. 442–445.

Gleison Mendonça, Breno Guimaraes, Péricles Alves, Márcio Pereira, Guido Araújo, and Fernando Magno Quintao Pereira. 2016. Automatic insertion of copy annotation in data-parallel programs. In *Proceedings of the 2016 SBAC-PAD Conference (SBAC-PAD'16)*. IEEE, Los Alamitos, CA, 1–8.

H. Nazaré, I. Maffra, W. Santos, L. Barbosa, L. Gonnord, and F. M. Q. Pereira. 2014. Validation of memory accesses through symbolic analyses. In *Proceedings of the 2014 OOPSLA Conference (OOPSLA'14)*. ACM, New York, NY, 791–809.

Cedric Nugteren and Henk Corporaal. 2014. Bones: An automatic skeleton-based C-to-CUDA compiler for GPUs. *ACM Transactions on Architecture and Code Optimization* 11, 4, 35:1–35:25.

OpenACC Standard. 2013. *The OpenACC Programming Interface*. Technical Report. CAPS.

Fernando Magno Quintao Pereira and Daniel Berlin. 2009. Wave propagation and deep propagation for pointer analysis. In *Proceedings of the 2009 CGO Conference (CGO'09)*. IEEE, Los Alamitos, CA, 126–135.

A Raghesh. 2011. *A Framework for Automatic OpenMP Code Generation*. Master's thesis. IIT Madras.

R. Reyes, I. López-Rodríguez, J. Fumero, and F. Sande. 2012. AccULL: An OpenACC implementation with CUDA and OpenCL support. In *Proceedings of the 2012 Euro-Par Conference (Euro-Par'12)*. 871–882.

Radu Rugina and Martin Rinard. 2000. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM SIGPLAN Notices* 35, 5, 182–195.

Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. 2003. Hybrid analysis: Static and dynamic memory reference analysis. *International Journal of Parallel Programming* 31, 251–283.

O. Shivers. 1988. Control flow analysis in scheme. In *Proceedings of the 1988 PLDI Conference (PLDI'88)*. ACM, New York, NY, 164–174.

John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-Mei W. Hwu. 2012. *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing*. Technical Report. IMPACT.

Rémi Triolet, Francois Irigoin, and Paul Feautrier. 1986. Direct parallelization of call statements. In *Proceedings of the 1986 SIGPLAN Conference (SIGPLAN'86)*. ACM, New York, NY, 176–185.

Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization* 9, 4, 54:1–54:23.

Sandra Wienke, Paul L. Springer, Christian Terboven, and Dieter an Mey. 2012. OpenACC—first experiences with real-world applications. In *Proceedings of the 2012 Euro-Par Conference (Euro-Par'12)*. 859–870.

M. J. Wolfe. 1995. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Boston, MA.

Received November 2016; revised March 2017; accepted April 2017