# Automatic Insertion of Copy Annotation in Data-Parallel Programs

Gleison Souza Diniz Mendonça Breno Campos Ferreira Guimarães Péricles Rafael Oliveira Alves
Fernando Magno Quintão Pereira
*UFMG, Belo Horizonte, Brazil*
{gleison.mendonca,brenosfg,pedroramos,periclesrafael,fernando}@dcc.ufmg.br
Márcio Machado Pereira Guido Araújo
*Unicamp, Campinas, Brazil*
{mpereira,guido}@ic.unicamp.br

***Abstract*—Directive-based programming models, such as OpenACC and OpenMP arise today as promising techniques to support the development of parallel applications. These systems 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 paper provides one fundamental component in the solution of this problem. We introduce a static program analysis that infers the bounds of memory regions referenced in source code. Such bounds allow us to automatically insert data-transfer primitives, which are needed when the parallelized code is meant to be executed in an accelerator device, such as a GPU. To validate our ideas, we have applied them onto Polybench, using two different architectures: Nvidia and Qualcomm-based. We have successfully analyzed 98% of all the memory accesses in Polybench. This result has enabled us to insert automatic annotations into those benchmarks leading to speedups of over 100x.**

***Keywords*-compilers; parallelism; optimization**

## I. Introduction

Heterogeneous architectures formed by clusters of CPUs and GPUs give us, today, a *de facto* standard in terms of high-performing computing. To illustrate this statement, recently an implementation of conjugate gradients has been able to scale to 3.12 million heterogeneous cores on Tianhe-2, reaching 623 Tflop/s [1]. Currently, directive-based annotation systems standout among the several different techniques used to program such machines. Examples of such systems include OpenMP [2], OpenACC [3] and OpenSs [4]. The idea behind this programming model is simple, yet appealing: annotations work as a meta-language, which give developers the ability to grant parallel semantics to syntax originally written to run sequentially. Hence, developers can reap all the benefits from modern parallel hardware, without having to worry too much about minutiae of concurrent programming, such as race conditions and deadlocks – such barriers are left to the compiler. Success stories of such annotation systems abound, and, combined with modern accelerators, they have led to substantial performance gains [5].

Nevertheless, annotating code to run in an external accelerator device is still a difficult task, which often requires familiarity with the target program that must be transformed. Two challenges, in particular, are daunting: the discovery of parallel loops and the estimation of memory bounds. The former problem has received substantial attention from the programming languages community, to the point that compilers are able, today, to detect parallel loops with high accuracy [6]. However, the second challenge is still an unsolved issue. Languages such as C and C++ do not provide programmers with syntax to recover the sizes of memory regions such as arrays and structs. Developers themselves need to keep track of these bounds. Surpassing this second obstacle is necessary to correctly allocate data in the address space where it will be processed. For instance, on a CPU-GPU based system, developers must copy data explicitly from the host (CPU) to the de device (GPU). In this paper, we provide a suite of static analyses to solve the second problem.

The contribution of this paper is a static analysis that derives *symbolic* bounds for memory regions such as arrays, structs or unions allocated in C or 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. Armed with such bounds, we enable a source-to-source compiler to annotate code with data-copy directives. Such directives take charge of moving data between the different processors that constitute a heterogeneous parallel system. As a by-product of our analysis, we enable the *disambiguation* of pointers. In other words, we can establish conditions that ensure the absence of aliasing between them; hence, effectively implementing *pointer restrictification* [7], [8] at the source code level. This disambiguation enables

the discovery of more parallel regions in the sequential source code, because it reduces dependences between data. 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.

The ideas that we introduce in this paper let us transform a loop, which the developer has indicated as parallel, in such a way that it can run on an accelerator. This transformation does not require any intervention from users – it is completely automatic. In order to validate the ideas discussed in this paper, we have used DawnCC to transform programs present in Polybench[1]. Our static analysis is able to bound 98% of all the 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 the loops. The net result is performance: by annotating loops automatically, we have been able to observe speedups of up to 105x on a CPU-GPU based architecture. Our tool is currently available through an online interface: http://cuda.dcc.ufmg.br/dawn/.

## II. Overview

We use the *Single Precision AX + Y* (SAXPY) kernel in Figure 1 (a) to illustrate the contributions of this paper. This kernel is a standard function in Nvidia's BLAS library[2]. It simply 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. The program in Figure 1 (b) is written in a language whose syntax is very similar to C's. However, its semantics is substantially different. Part of it, e.g., lines 1-7 is meant to run on a GPU; the rest, e.g., lines 9-11, 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 developers of parallel applications, having to worry about concurrent

[1]http://cavazos-lab.github.io/PolyBench-ACC/

[2]https://devblogs.nvidia.com/parallelforall/six-ways-saxpy/

```
1  void  saxpy_serial(int n, float alpha, float *x, float *y) {
2     for (int i = 0; i < n; i++) {
3        y[i] = alpha*x[i] + y[i];
4     }
5  }
```
(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  // Invoke the parallel kernel:
10 int nblocks = (n + 255) / 256;
11 saxpy_parallel <<<nblocks, 256>>>(n, 2.0, x, y);
```
(b)

Figure 1: (a) Standard C implementation of the Single Precision $AX+Y$ (SAXPY) kernel. (b) Same algorithm written in C for CUDA.

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 a number of *annotation systems*. An annotation system is a meta-language that changes the semantics of a host language. In our setting, the host language is either C or C++, and the meta-language 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 the code necessary for them to work, automatically.

DawnCC is a source-to-source compiler: it reads an ordinary 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 [9, Ch.6]. Because these techniques are already commonplace in the compiler's literature, in this paper we focus on the second problem.

In order 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 III, to recover the limits of these regions. More importantly: we do it using symbols present in the program code itself. For instance, at line 3 of Figures 2 (a) and (b), we

```
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)

Figure 2: (a) SAXPY annotated with OpenACC pragmas. (b) SAXPY annotated with OpenMP pragmas. The grey area denotes code created automatically.

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 Figures 2 (a) and (b) give us this information. Whenever either of the inequations `y >= x+tmp[1] + 1` or `x >= y + tmp[1] + 1` are true, we are sure that the vectors `x` and `y` do not overlap. We can only assume 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.

## III. Static Analyses

DawnCCis built around a suite of static analyses. These analyses exist on top of two well-known techniques: symbolic range analysis [7], [10] and dependence analysis [11]. In the rest of this section we explain how these compilation techniques work.

### A. Symbolic Range Analysis

Most of our contributions rely on the compiler's ability to estimate the range of values covered by integer variables during the execution of a program. We perform this estimation via *symbolic range analysis*, as defined by Alves *et al.* [7]. A *symbol* is any variable whose value we cannot reconstruct as a function of other variables. $R(v)$, the range of a variable $v$, is a pair $[l, u]$, where $l$ 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 if, and only if, $l \leq u$. The goal of the symbolic range analysis is to compute the map $R$ for every integer variable in a program.

Figure 3 shows the range inference rules that we have used in this work. 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 $S$, 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.

To determine $R$, we iterate the rules in Figure 3 until we achieve a fixed point. The fixed point operator is given by the relation **fp**, as defined in Figure 3. Due to loops, this analysis might not converge. Thus, we use a widening operator $\nabla$ to ensure quick convergence. This analysis is linear on the size of the program, where the size of a program is measured as the number of uses and definitions of variables within its text. The rules in Figure 3 show a flow-insensitive analysis. That is to say: 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 Static Single Assignment (SSA) form. SSA is a classic program representation in which each variable has one single definition point [12].

Concerning Figure 3, 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, e.g., a value that cannot be built as function of other values. Upon evaluating an assignment $v = \bullet$, we create 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 $\sqcup$:

$$[l_1, u_1] \sqcup [l_2, u_2] = [\min(l_1, l_2), \max(u_1, u_2)]$$

*Example 3.1:* Range analysis, when applied onto function `saxpy_serial`, in Figure 1 (a), gives us that variable `i` has range $[0, 0]$ before the loop; range $[0, \mathtt{n} - 1]$ inside it, and range $[\mathtt{n}, \mathtt{n}]$ after the loop.

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

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

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

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

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

$$\frac{\begin{array}{c} \mathbf{rg}(S_t, R_t) = R_t \\ \mathbf{rg}(S_f, R_t) = R_f \qquad R_2 = R_t \sqcup R_f \qquad \mathbf{rg}(S, R_2) = R_3 \end{array}}{\mathbf{rg}(\{\texttt{if}(\_)\ S_t\ \texttt{else}\ S_f; S\}, R_1) = R_3}$$

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

$$\mathbf{rg}(\{\}, R) = R \qquad \frac{\mathbf{rg}(S, R) = R}{\mathbf{fp}(S, R) = R}$$

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

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

Figure 3: Constraints for the symbolic range analysis.

### B. Array Size Inference

Once we have a map $R$, built via the rules in Figure 3, 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 the variables used to index $v$. That is to say, if a program contains $n$ memory accesses such as $v[i_1], v[i_2], \ldots, v[i_n]$, then the bounds of $v$ are given by the expression: $B(v) = R(i_1) \sqcup R(i_2) \sqcup \ldots \sqcup R(i_n)$.

In this work, for simplicity, we only determine bounds for arrays that do not alias other pointers within a loop. In other words, if we cannot prove that a given memory region can only be accessed through a single base pointer within a given loop, then we do 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 [13].

*Example 3.2:* Our array size inference, when applied onto `saxpy_serial`, in Figure 1 (a), estimates the ranges covered by array `x` as $[\&\texttt{x}, \&\texttt{x} + (\texttt{n} - 1)]$, where $\&\texttt{x}$ is the base address `x`. We find such limits because the only variable that indexes `x` is `i`, which has range $[0, \texttt{n} - 1]$ within the loop.

### C. Restrictification

The fact that we can infer the sizes of arrays gives us the possibility to disambiguate pointers. This disambiguation – also known as restrictification – lets us eliminate spurious dependences 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 above inequalities hold.

Both OpenACC and OpenMP give us the equipment necessary to use this information. Such equipment consists in *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 Figures 2 (a) and (b). The conditional pragmas that use the result of this test appear at lines 9 through 12 of the annotated programs.

### D. Implementation Details

One key difference between DawnCC and most source-to-source compilers is that it aims to generate efficient, yet readable code. Source-to-source optimizers usually convert the whole program to an intermediate

representation, perform a set of analyses and code transformations, then write the resulting program back to a high level language. To ensure readability and minimum deviation from the original code, DawnCC takes a different route: it uses the result of the above described analyses to annotate the original C/C++ source file in specific points, e.g., loop headers. This characteristic implies that any code generated by us will use symbols already present in the input program, thus being more easily recognizable by the developer, increasing the overall maintainability of the system.

*From Source to IR, and Back Again.:* DawnCC is built on top of the LLVM [14] 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 *Intermediate Representation* (IR) of LLVM because we could, in this way, reuse already available techniques to compute bounds of variables. Thus, most of our implementation decisions are due to this choice of analyzing IR, and mapping the result of such analyses back to source code.

To automatically annotate a program, we generate C code for three categories of computation, in the following order: expressions denoting symbolic limits of arrays, pointer overlapping tests, and OpenACC/OpenMP data transfer and parallel pragmas. For the first category, given a set of instructions in LLVM's intermediate representation that computes the symbolic bound for a given pointer, we recursively visit the operands of each instruction to build an equivalent expression tree. The forest of limit expressions is then simplified. First, redundant subtrees are eliminated. We then perform constant propagation, solve trivial expressions, e.g., "`max(10, 100)`", and remove branches whose predicates we can determine statically. After these optimizations, variables referenced in limit expressions are replaced by their original names in the input program. This is achieved by inspecting the debug metadata provided by LLVM, attached to each instruction in its intermediate representation. At last, each limit tree is recursively converted to a string representing an equivalent parenthesized C expression. Most operations involved in limit computations have a direct C equivalent, e.g., addition, sign extension, bit shift, and type casting. More intricate operations, such as LLVM's `GetElementPtr` instruction, which computes array or struct field addresses, can be simulated with a small combination of different C operators. The strings thus produced are then written to the source file, before the loop that they refer. Lines 3 and 4 of Figures 2 (a) and (b) show the resulting computation.

Once the limit values are expressed in the original source program, the interval tests that disambiguate pointers can be trivially generated, because the inequalities described in Subsection III-C are directly represented in C. Each aliasing test is assigned to a flag that will be used in conditional pragmas, as seen in Figure 2 (a), line 6. The last step, generation of data transfer pragmas, is achieved by retrieving array names from debug information attached to the IR. To determine the region to be transferred, we use the interval analysis of Section III-A. At this point, we perform a peephole optimization to eliminate unnecessary transfers, which are recognized as expensive operations: we inspect each loop nest, identifying the target arrays of each load and store operation. Arrays that just serve as inputs for a loop, i.e., are not written by store operations, are sent to the GPU, but not transferred back at the end of the parallel region. Conversely, output arrays are not sent to the GPU at the beginning of a parallel loop. The resulting pragmas can be seen in lines 9 through 12 of Figures 2 (a) and (b).

*Preserving Annotations in Face of Optimizations.:* LLVM is well known for implementing a myriad of transformations that improve the reach of other analyses. While yielding strong impact, these transformations widen the syntactic gap between the input program and its intermediate representation. SSA conversion, for instance, improves the precision of our symbolic range analysis, but inserts a number of virtual names that must be accounted for when writing back annotations to C code. Loop Invariant Code Motion, in turn, moves memory operations across program points, but allows a more precise dependence analysis. In summary, any transformation performed by LLVM to expose more information must be replicated in the original source file. Thus, we must use only compilation passes that improve the precision of our tools, and that at the same time can have its effects replicated in C code. During our experiments, we identified the following set of passes as beneficial to our analyses, from a precision standpoint: SSA conversion, Loop Invariant Code Motion (load hoisting and store sinking), and Loop Rotation. We changed DawnCC's code generation step to take into account these transformations, replicating their effects in the original C program.

## IV. Experiments

We have evaluated our techniques using three different compilers and two different architectures, in order to answer the following question: can our annotations produce speedups on typical kernels with minimal intervention from developers? By "minimal intervention"

we mean that the only action that we require from users is one annotation indicating that a loop can run in parallel, i.e., it is a *doall*. There exist techniques that determine when a loop is data-parallel; however, none of the compilers that we have used provides them.

**Benchmarks.** We tested our analyses on the version of Polybench used by Gray *et al.* [15]. All experiments were performed using the standard input size available in Polybench, which means array dimensions between $10^3$ and $10^4$ elements for most benchmarks.

**Hardware** We experimented with the following setups:

- **Desktop**: Intel Xeon CPU E5-2620, with 6 cores of 2.00GHz and 16 GB of RAM (DDR2), running Linux Ubuntu 12.04 3.2.0, equipped with a GPU model GeForce GTX 670, with 2 GB of RAM (CUDA Compute Capability 3.0).
- **Phone**: Exynos7420 AArch64 Processor with 4GB of RAM running Android 5.1.1 and equipped with a GPU model ARM Mali-T760 with 913 MB of RAM and 8 parallel compute units.

**Compilers** We have used different compilers to produce binaries out of annotated code:

- **gpuc** gpuclang version 2.0 (based on clang 3.5.0). Runs on the phone setup, and translates OpenMP to parallel code.
- **pgcc** PGI C Compiler version 16.1 64bit. Runs on the desktop setup, and translates OpenACC to parallel code.
- **gcc** Gnu C Compiler version 4.8.1 (always at -O3). Runs on the desktop setup.

We use gcc as a baseline. It does not parallelize code, but, at its highest optimization level, it gives the reader a well-known reference point in terms of performance.

**Desktop setup (DawnCC+pgcc [OpenACC]).** Figure 4 shows the speedup that we obtain when comparing the annotated code, compiled with pgcc -O3, against the same code, without our annotations. We run each program five times, and the bars compare the average execution of them. When probing the GPU execution time, we include the time to transfer data to and from the GPU. Variance is negligible; hence, we will not provide error intervals. We observe very large speedups in two benchmarks: COVAR and GEMM. These are embarrassingly parallel applications, which benefit substantially from the SIMD execution model of a GPU. We have also observed slowdowns in six benchmarks. As we will show later in Figure 6, these slowdowns happen in benchmarks that run for very short times.

**Phone setup (DawnCC+gpuc [OpenMP]).** Figure 4 shows the speedup that we obtain when comparing the annotated code, compiled with gpuclang, against gcc -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.

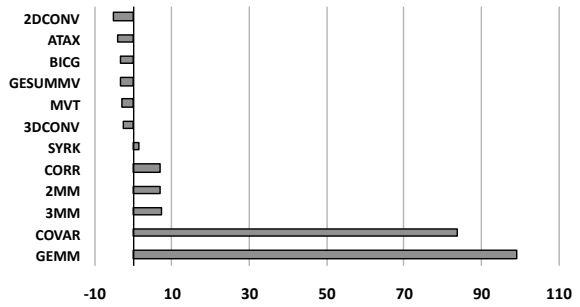


Figure 4: **Desktop (DawnCC+pgcc vs pgcc)** Runtime comparison between the code that DawnCC has annotated with OpenACC pragmas and the same code without the annotations. Both programs have been compiled with pgcc. X axis shows speedup, in number of times ($\times N$), of annotated code with respect to sequential code. The longer the bar, the more performance DawnCC is delivering.

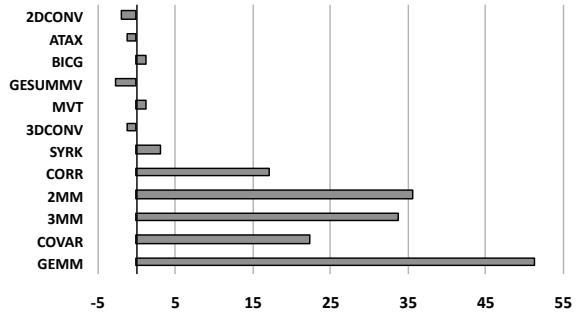


Figure 5: **Phone (DawnCC+gpuc vs gpuc)** Comparison between the code that DawnCC has annotated with OpenMP pragmas and compiled with gpuclang, and the benchmarks without the annotations. X axis shows speedup, in number of times ($\times N$), of annotated code with respect to sequential code. The longer the bar, the more performance DawnCCis delivering.

| Benchmark | *Xeon + GTX 670* | | | *Exynos + Mali* | |
|---|---|---|---|---|---|
| | **pgcc** | **pgcc + OpenACC** | **gcc -O3** | **gpuclang** | **gpuclang + OpenMP** |
| 2dconv | 0.47 | 1.48 | 0.29 | 1.90 | 3.51 |
| atax | 0.36 | 0.79 | 0.19 | 1.60 | 1.77 |
| bicg | 0.29 | 0.84 | 0.25 | 1.66 | 1.42 |
| gesummv | 0.29 | 0.87 | 0.28 | 0.56 | 1.47 |
| mvt | 0.31 | 0.67 | 0.22 | 1.05 | 0.82 |
| 3dconv | 1.09 | 3.07 | 1.22 | 4.40 | 4.78 |
| syrk | 31.15 | 7.81 | 11.03 | 23.01 | 7.24 |
| syr2k | 42.14 | 15.34 | 42.10 | 6.44 | 1.70 |
| corr | 66.90 | 10.82 | 75.48 | 202.19 | 11.73 |
| 2mm | 66.16 | 9.00 | 64.65 | 251.83 | 7.06 |
| 3mm | 99.17 | 13.30 | 96.50 | 350.68 | 10.38 |
| covar | 67.95 | 0.91 | 75.72 | 191.75 | 8.58 |
| gemm | 88.42 | 0.90 | 88.69 | 423.79 | 8.27 |

Figure 6: Absolute runtime in the two setups. Numbers are average of five samples.

**Absolute Runtimes.** Figure 6 shows absolute runtime numbers for the different benchmarks that we have, in the two different setups. Benchmarks are sorted, from top to bottom, according to the relative speedup that we've observed in the desktop setup. The most important fact to notice is that our most substantial speedups have been produced in some of the programs that run for the longest time. The slowdowns that we have observed in benchmarks such as 2DCONV, ATAX, BICG and GESUMMV happened in samples that executed for a very short time. In this case, the extra parallelism of the GPU is not enough to pay off for the time to transfer data from host to device, and then back. Figure 6 shows, for the desktop setup, absolute numbers that we get also using gcc -O3. Given that gcc is one of the most used open-source compilers, these numbers give to the reader an idea of the sheer performance that the combination of DawnCC and an OpenACC compliant compiler achieves. The runtime of the binaries produced by gcc is very similar to the runtime of the programs compiled with pgcc without our annotations; hence, the speedups that we get on top of gcc are equally impressive. We did not show gcc's numbers on the phone setup, because we do not have a version of this compiler on that environment.

**Discussion: DawnCC vs manual code annotation.** One of the authors of this work produced a manually annotated version of each benchmark. We compared the code produced by our tool against this version and against UnibBench, a publicly available version of Polybench annotated with OpenMP 4.0 pragmas. The only difference between the manual annotations, and the code transformed automatically refers to the loops that initialize the data-structures used in each benchmark. Performing this initialization on the accelerator is not worthwhile for these benchmarks, on account of the time to transfer data. Therefore, the expert developer chose to leave these routines untouched, whereas DawnCC has annotated all of them. However, these procedures run for a very short time, compared with the execution time of each kernel. In the end, we have not been able to measure significant differences between the runtime of manually and automatically annotated programs. DawnCC has even correctly annotated each data that is only read on the GPU as read-only, which avoids transferring them back to the CPU once processed on the accelerator.

Limitations of our method can be more noticeable in the presence of less regular input programs. The symbolic range analysis used here is only capable of deriving limits for array operations, which means that a human user would perform better at annotating code that relies heavily in custom or more dynamic data structures. Additionally, our analysis cannot estimate bounds in the presence of function calls, which could be overcome with techniques such as interprocedural bounded regular section analysis [16]. DawnCC is also limited to *doall* parallelism, whereas an experienced developer could identify and take advantage of more refined work division patterns, such as reduction operations. We emphasize, however, that the majority of the programs that can take advantage of an external acceleration device are regular. As an example, Polybench encompasses a set of core algorithms widely targeted by hardware acceleration techniques and DawnCC is able to annotate all kernels in it.

## V. Related Work

There exists an enormous corpus about the automatic parallelization of software. For an overview about the classic techniques, we recommend the book of Michael Wolfe [9]. However, the goal of our work is not to find parallelism in programs; instead, we want to give programmers the tools to move code – already proved to be parallel – to accelerators. 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 parallelize for simple multi-threading, for instance. This explains why the research community has not yet tackled this problem in the most seminal papers on automatic generation of OpenMP code, such as Lee *et al*'s [17].

The range analysis that we use in this work is also not new. We took the implementation of Alves *et al.* [7]. 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 (see the work of Rus *et al.* [18]), as we did in Section III-C. Our goal is not to design better range analyses, nor to infer more parallel regions in programs. In this sense, our work is novel: there exist no automatic techniques to annotate code with data copy pragmas.

A number of optimization frameworks based on the polyhedral model have been used for automatic generation of OpenMP and GPU code [13], [6]. These tools generate data copy library calls by inspecting the iteration domains of arrays used within parallel loops. In practice, the symbolic limits generated by such frameworks and the ones generated by the analysis chosen for this work present similar results [7], each having specific advantage scenarios. For instance, the method applied in this paper can handle non-affine regions of code, while the analyses implemented in polyhedral-based tools usually generate simpler interval expressions, by performing static simplification, which comes at the cost of a higher compilation time.

## VI. Conclusion

This paper has presented a suite of static analyses 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 100x. DawnCC is the first tool able to insert OpenACC/OpenMP pragmas in sequential C programs without the intervention of users. It is currently available at http://cuda.dcc.ufmg.br/dawn/. We invite the interested reader to try it.

**Acknowledgement.** This project is sponsored by LGE. Individual authors are supported by different Brazilian agencies: FAPEMIG, FAPESP, CNPq and CAPES. We thank Michael Frank and Fabrício Ferracioli from LGE for some very fruitful discussions about this project.

## References

[1] Y. Liu, C. Yang, F. Liu, X. Zhang, Y. Lu, Y. Du, C. Yang, M. Xie, and X. Liao, "623 tflop/s HPCG run on tianhe-2: Leveraging millions of hybrid cores," *IJHPCA*, vol. 30, pp. 39–54, 2016.

[2] J. Jaeger, P. Carribault, and M. Pérache, "Fine-grain data management directory for openmp 4.0 and openacc," *Concurrency and Computation: Practice and Experience*, pp. 1528–1539, 2015.

[3] Committee, "The OpenACC programming interface," CAPs, Tech. Rep., 2013.

[4] C. Meenderinck and B. Juurlink, "Nexus: Hardware support for task-based programming," in *DSD*. Springer, 2011, pp. 442–445.

[5] R. Reyes, I. López-Rodríguez, J. J. Fumero, and F. de Sande, "AccULL: An OpenACC implementation with CUDA and OpenCL support," in *Euro-Par*. Springer, 2012, pp. 871–882.

[6] M. M. Baskaran, J. Ramanujam, and P. Sadayappan, "Automatic C-to-CUDA code generation for affine programs," in *CC*. Springer, 2010, pp. 244–263.

[7] P. Alves, F. Gruber, J. Doerfert, A. Lamprineas, T. Grosser, F. Rastello, and F. M. Q. a. Pereira, "Runtime pointer disambiguation," in *OOPSLA*. ACM, 2015, pp. 589–606.

[8] V. H. S. Campos, P. R. O. Alves, H. N. Santos, and F. M. Q. Pereira, "Restrictification of function arguments," in *CC*. ACM, 2016, pp. 163–173.

[9] M. J. Wolfe, *High Performance Compilers for Parallel Computing*, C. Shanklin and L. Ortega, Eds. Addison-Wesley, 1995.

[10] H. Nazaré, I. Maffra, W. Santos, L. Barbosa, L. Gonnord, and F. M. Q. Pereira, "Validation of memory accesses through symbolic analyses," in *OOPSLA*. ACM, 2014, pp. 791–809.

[11] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *TOPLAS*, vol. 9, no. 3, pp. 319–349, 1987.

[12] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *TOPLAS*, vol. 13, no. 4, pp. 451–490, 1991.

[13] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for cuda," *TACO*, vol. 9, no. 4, pp. 54:1–54:23, 2013.

[14] C. Lattner and S. Adve, "LLVM: a compilation framework for lifelong program analysis transformation," in *CGO*, 2004, pp. 75–86.

[15] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *InPar*. IEEE, 2012, pp. 1–10.

[16] P. Havlak and K. Kennedy, "An implementation of interprocedural bounded regular section analysis," *TPDS*, vol. 2, 1991.

[17] S. Lee, S.-J. Min, and R. Eigenmann, "Openmp to gpgpu: a compiler framework for automatic translation and optimization," in *PPoPP*. ACM, 2009, pp. 101–110.

[18] S. Rus, L. Rauchwerger, and J. Hoeflinger, "Hybrid Analysis: Static & Dynamic Memory Reference Analysis," *International Journal of Parallel Programming*, vol. 31, pp. 251–283, 2003.