DawnCC : a Source-to-Source Automatic Parallelizer of C and C++ Programs

Breno Campos Ferreira Guimarães, Gleison Souza Diniz Mendonça, Fernando Magno Quintão Pereira

[1]Departamento de Ciência da Computação – UFMG
Av. Antônio Carlos, 6627 – 31.270-010 – Belo Horizonte – MG – Brazil

{brenosfg,gleison.mendonca,fernando}@dcc.ufmg.br

Abstract. *Dedicated graphics processing chips have become a standard component in most modern systems, making their powerful parallel computing capabilities more accessible to developers. Amongst the tools created to aid programmers in the task of parallelizing applications, directive-based standards are some of the most widely used. These standards, such as OpenACC and OpenMP, facilitate the conversion of sequential programs into parallel ones 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 describes DawnCC, a tool that solves this problem. DawnCC is a source-to-source compiler module that automatically annotates sequential C code with OpenACC or OpenMP directives; thus, effectively producing parallel programs out of sequential semantics. DawnCC is equipped with a number of static program analyses that: (i) infer bounds of memory regions referenced in source code to copy them between host and device; and (ii) discover parallel loops in sequential code. To validate its effectiveness, we have used DawnCC to automatically annotate the benchmarks in the Polybench/GPU suite with proper OpenACC directives. These annotations let us parallelize these benchmarks, leading to speedups of up to 78x.*

Link to Video: `https://youtu.be/e7mmpP3x10E`

1. Introduction

The growing popularity of heterogeneous architectures containing both CPUs and GPUs has generated an increasing interest in general-purpose computing on graphics processing units (GPGPU) [**?**]. This practice consists of developing general purpose programs, *i.e.* not necessarily related to graphics processing, to run on hardware that is specialized for graphics computing. Executing programs on such chips can be advantageous due to the parallel nature of their architecture: while a typical CPU is composed of a small number of cores capable of a wide variety of computations, GPUs usually contain hundreds of simpler processors, which perform computations in separate chunks of memory concurrently [**?**]. Thus, a graphics chip can run programs that are sufficiently parallel much faster than a CPU [**?**]. In some cases, this speedup can reach several orders of magnitude. GPUs can also be not only faster, but also more energy-efficient when running memory-parallel tasks.

This model, however, has its shortcomings. Historically, parallel programming has been a difficult paradigm to adopt, sometimes requiring that developers be familiarized with particular instruction sets of different graphics chips. Recently, a few standards such as OpenCL and CUDA have been designed to provide some level of abstraction to these platforms, which in turn has led to the development of compiler directive-based programming models, *e.g.* OpenACC [**?**] and OpenMP [**?**]. While these have somewhat bridged the gap between programmers and parallel programming interfaces, they still rely on manual insertion of compiler directives, an error-prone process that also commands in-depth knowledge of the target program.

Amongst the hurdles involved in annotating code, two tasks are particularly challenging: identifying parallel loops and estimating memory bounds [**?**]. Regarding the former, opportunities for parallelism are usually buried under complex syntax. As to the latter, languages such as C and C++ do not provide any information on the size of memory being accessed during the execution of the program. However, when offloading code for parallel execution, it is necessary to inform which chunks of memory must be copied to other devices. Therefore, the onus of keeping track of these memory bounds falls on the programmer.

This paper describes DawnCC , a tool that we have designed, implemented and tested to shield developers from the complexities of parallel programming. Through the implementation of a static analysis that derives memory access bounds from source code, it infers the size of memory regions in C programs. With these bounds, our tool is capable of inserting data copy directives in the original source code. These directives provide a compatible compiler with information on which data must be moved between devices. It is also capable of identifying loops that do not contain memory dependences, and therefore can be run in parallel, and marking them as such with the proper pragmas. To increase the amount of potentially parallel loops detectable, it performs pointer disambiguation. That is, it determines conditions that guarantee the absence of aliasing, and indicates that a loop may be executed in parallel when said conditions are met.

We have developed DawnCC as a collection of compiler modules, or passes, for the LLVM compiler infrastructure [**?**]. We have made DawnCC available for public use through a webpage that functions as a front-end[1]. Users can submit their own C source code through this page. The code is then compiled to LLVM bytecode and run through our passes, which analyze it and reconstruct the original C source, inserting the appropriate parallel standard directives. Thus, the user receives as output a modified version of their code in plain text, which corresponds to an effectively parallel version of their submitted program. To demonstrate the effectiveness of DawnCC , in this paper we show how to apply it onto the source code of the benchmarks available in the Polybench/GPU suite[2]. We use DawnCC to annotate the programs in Polybench with OpenACC directives. The modified benchmarks present speedups of up to 78x in execution time, and their results are verified for correctness by comparison with sequential execution.

[1]Our tool is currently available at `http://cuda.dcc.ufmg.br/dawn/`

[2]Polybench's source code is available in several different websites, such as `http://web.cse.ohio-state.edu/~pouchet/software/polybench/`

2. Related Work

We could only design and implement DawnCC because of the emergence of annotation systems for data-parallel systems. These standards aim to simplify the creation of parallel programs by providing an interface for developers to annotate specific regions in source code, indicating that they should run in parallel. Parallel execution can be performed in a variety of ways, such as spread between multiple cores in a single CPU, or through offloading computation to a separate device in heterogeneous architectures. Compilers that support these standards can check for the presence of directives in source code, and generate parallel code for the specific regions annotated, which can in turn be allocated to run on target devices. DawnCC currently supports two standards, OpenACC and OpenMP , but it can easily be extended to support others. We have chosen these standards due to their effectiveness and widespread use in modern parallel programming.

To the best of our knowledge, DawnCC is the only source-to-source compiler that inserts OpenACC or OpenMP annotations in programs automatically. However, there are tools with the same end-goal: to compile C into CUDA without much intervention from developers. A number of optimization frameworks based on the polyhedral model have been used for automatic generation of GPU code [**?**, **?**]. These tools generate GPU code directly, without using annotations. In practice, the symbolic limits generated by such frameworks and the ones generated by the analysis chosen for this work present similar results [**?**], each having specific advantages. 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.

3. Working Example

Figure 1 shows an example program that can be provided as input in DawnCC 's webpage. The C code shown contains a few loops that exemplify some of the key analyses DawnCC is capable of performing, and exposes some of the main functionalities it provides. The following sections explain these in greater depth. For the output examples in the figures that follow, we have used DawnCC to annotate the source code with OpenACC directives, and configured it to only annotate loops it deems as parallelizable. Note that the original code remains unchanged, having been reconstructed from the compiler's intermediate representation.

3.1. Memory Bound Accuracy

Balancing the overhead of offloading computation to a separate device and the gain in performance from parallel execution is a cornerstone of efficient GPGPU programming. In many cases, the effort spent in performing tasks not related to effective computation, such as copying data or synchronizing execution, counterweighs the advantages of abusing parallelism. This can cause the parallel version of the program to show no significant gain in performance, or even a slowdown, even when the algorithm involved presents ripe opportunities for concurrent execution. Therefore, handling the amount of overhead associated with parallelizing code is vital.

When it comes to measuring memory bounds to perform data copying, a naive analysis could simply measure the total size of memory blocks referenced inside the

```
void example (int *a, int *b, int c) {
    int n[5000];
    int i, j, k;

    /*loop 1*/
    for (i = 0; i < 1000; i++) {
      /*loop 2*/
        for (j = 0; j < 1000; j++) {
            n[i+j] = i*j;
        }
    }

    /*loop 3*/
    for (k = 0; k < 5000; k++) {
        a[k] = b[k]+(k*k);
    }

    /*loop 4*/
    for (k = 0; k < c; k++) {
        a[k] = k*(k+1);
    }
}
```

Figure 1. Example input C code.

loops, and use the entire size as the upper bound in a data copy directive. While correct, such an approach could potentially generate redundant copy instructions, since it is possible for chunks of data which are not used within the loop to be copied back and forth between devices. Our analysis instead calculates the bounds of the memory region that is effectively accessed inside the loop, thus minimizing the amount of potentially redundant computation. Figure 2 (a) highlights the pragmas inserted in the code for the first two loops in the original example. Since the upper limit for the array subscript is at most the sum of the values reachable by both induction variables, it is not necessary to copy the entire array. As a result, the data directive generated contains a more precise value that better reflects the effective memory access limits.

3.2. Treating Pointer Aliasing

There are many reasons that might prevent a given piece of program from being parallelizable. Most of these include memory dependences of some form. In C and C++ code involving dynamically allocated memory regions, one of the main culprits behind such dependences is the possibility of pointer aliasing. That is to say, when a set of instructions accesses memory referenced by multiple pointer values, there is no guarantee that the regions pointed to do not overlap. In such cases, an implicit memory dependence exists, and the possibility of parallelization is typically discarded. Usually, treating these cases statically involves the employment of complex and costly interprocedural pointer analyses.

However, as a by-product of our alias analysis, our tool is capable of performing pointer disambiguation. This means it can infer conditions that ensure the absence of aliasing, in which case the memory dependences do not exist, and parallel execution might be possible. By combining this with conditional compilation directives, we can solve the problem in an elegant and concise way. The conditional directives instruct a compiler to create two different versions of a loop, whose execution is controlled by an aliasing check. Then, during execution, the conditional is evaluated. If the absence of aliasing is confirmed, the loop is executed in parallel. Otherwise, a sequential version is executed instead. Figure 2 (b) shows the pragmas inserted for the code that corresponds to the third loop in the original example. The alias check can be observed immediately before the

pragma directives, and the conditional execution pragmas can be seen associated with the data copy and kernels directives.

3.3. Symbolic Inference

Figure 2 (c) shows the code that corresponds to the fourth loop in the original example. In this case, the scalar value of the variables used as subscripts in the memory accesses in the loop are not predictable in the function's scope. In this case, DawnCC is capable of inferring the proper limits by inserting a series of value checks to determine which value effectively defines the upper bound for the memory accesses performed. It then inserts the proper value in the copy directive. Note that in this case the memory bounds may vary during execution, yet the limits defined remain correct for every execution context. The value checks can be seen immediately above the pragmas. The first pragma inserted is the data directive, with the proper upper bound as its parameter.

```
/*loop 1*/
#pragma acc data pcopy(n[0:1999])
#pragma acc kernels
#pragma acc loop independent
for (i = 0; i < 1000; i++) {
        /*loop 2*/
        #pragma acc loop independent
        for (j = 0; j < 1000; j++) {
                n[i+j] = i*j;
}
```
(a);

```
/*loop 3*/
char RST_AI2 = 0;
RST_AI2 |= !((A + 0 > b + 5000)
|| (b + 0 > a + 5000));
#pragma acc data pcopy(a[0:5000],b[0:
5000]) if(!RST_AI2)
#pragma acc kernels if(!RST_AI2)
#pragma acc loop independent
for (k = 0; k < 5000; k++) {
   a[k] = b[k]+(k*k);
}
```
(b);

```
   /*loop 4*/
   long long int AI3[6];
   AI3[0] = c + -1;
   AI3[1] = 4 * AI3[0];
   AI3[2] = AI3[1] + 4;
   AI3[3] = AI3[2] / 4;
   AI3[4] = (AI[3] > 0);
   AI3[5] = (AI3[4] ? AI3[3] : 0);
   #pragma acc data pcopy(a[0:AI3
[5]])
   #pragma acc kernels
   #pragma acc loop independent
   for (k = 0; k < c; k++) {
       a[k] = k*(k+1);
   }
```
(c);

Figure 2. (a) Pragmas inserted in the first and second loops; (b) Pragmas and alias checks for third loop; (c) Pragmas and value checks for fourth loop.

4. Web Interface

DawnCC is available at `http://cuda.dcc.ufmg.br/dawn/`. This webpage is open to the general public, and it receives, as input, a plain C program. Figure 3 shows the main screen of our webpage. Whoever uses the DawnCC webpage can choose between annotating programs with either OpenACC or OpenMP directives. Users have also the option to display compilation statistics about the annotation process. These statistics include facts such as the number of memory accesses that DawnCC has been able to bound, the number of loops inferred to be parallel, the number of total annotations inserted, etc. When dealing with large programs, users have the option to load them instead of pasting their text into the input window.

Figure 4 shows the output produced by our tool. The annotated program is made available to the user at the window in the lower part of the web interface. The user can also download a complete version of the program, via a link next to the program's text. Whenever users select to display compilation statistics, these numbers are displayed right above the output window. The webpage contains a tutorial about how to use DawnCC , which provides more information to the interested reader.

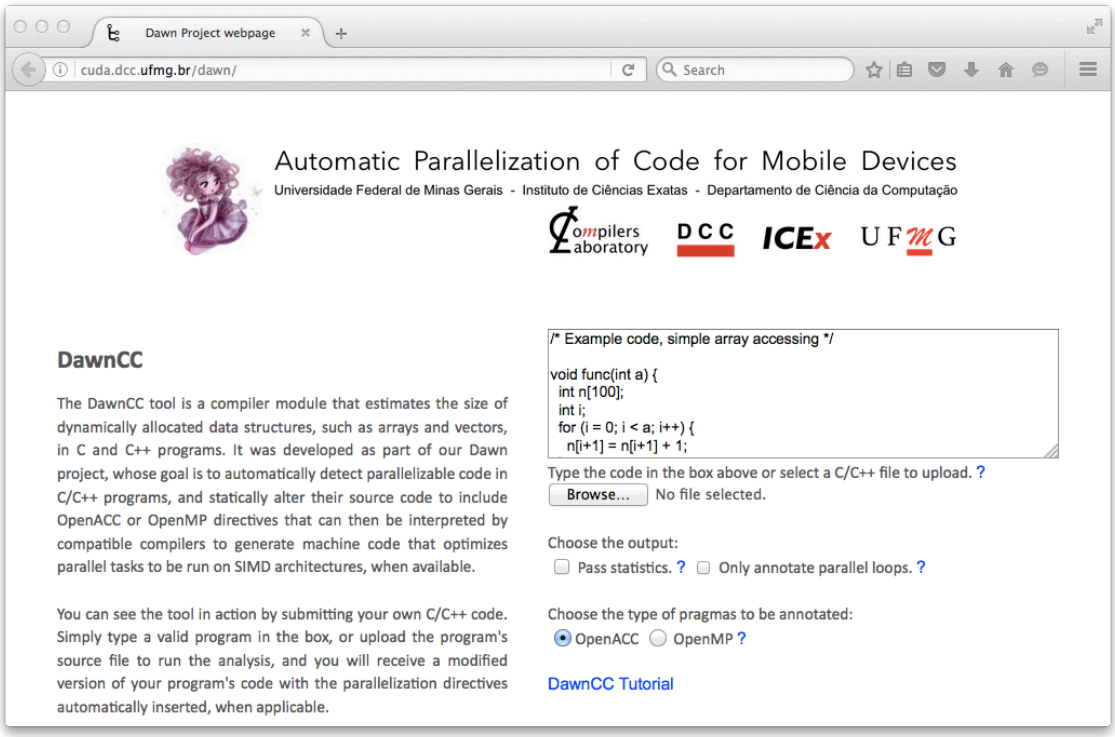

Figure 3. Screenshot of the web interface of DawnCC .

5. Experiments

We performed a small set of tests to validate the effectiveness of DawnCC . We used the programs in the Polybench/GPU suite of benchmarks, contained in the UniBench compilation of suites, as our testing codebase. We used DawnCC to annotate all the programs in the suite with data copy directives and kernels directives. It is important to note that for this specific set of tests the annotation of parallel loops was done manually, as the main focus of this work is the analysis for inferring memory bounds and performing pointer disambiguation. Figure 5 displays a speedup chart that measures the execution time ratio between GPU and CPU execution time. A positive value means a speedup of the given amount was observed, while a negative value corresponds to a slowdown of the same proportion.

The experiments were performed in a server with an Intel Xeon E5-2620 CPU, with 6 cores at 2.00GHz frequency each, and 16 GB of DDR2 RAM. The GPU used for parallel execution was an NVidia GTX 670 with 2GB RAM (CUDA Compute Capability 3.0). All the tests were performed in a Linux Ubuntu 12.04 environment. The compiler used to generate binaries for both baseline and OpenACC-accelerated versions of the benchmarks was Portland Group's PGCC, version 16.1.

Some very significant speedups can be observed in benchmarks that have considerable running time on the CPU, such as Covariance and FDTD-2D benchmarks. In these cases, the CPU took anywhere from 15 to 80 seconds to finish execution, whereas the GPU usually takes under a second. Less significant speedups can be observed in other benchmarks that take a moderate amount of time to execute in the CPU, such as 2MM and SYR2K, which take 5 to 15 seconds to execute. In most benchmarks where

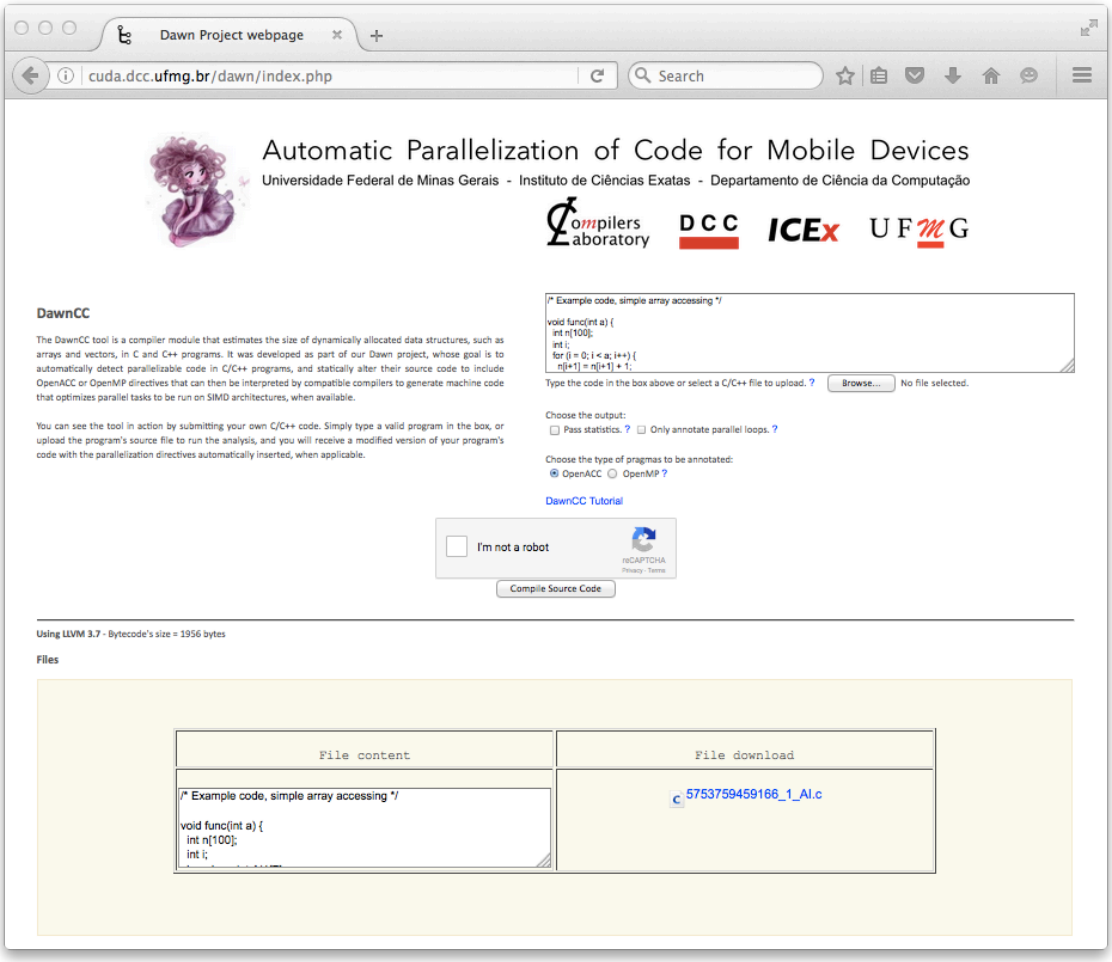

Figure 4. Output window of DawnCC .

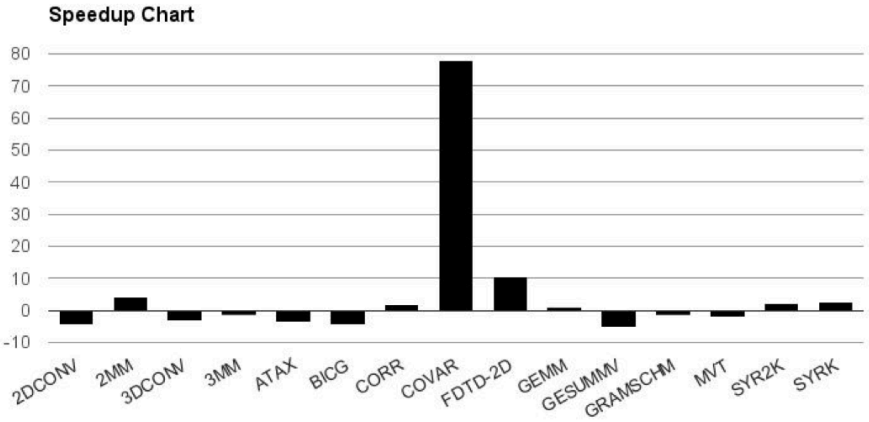

Figure 5. Speedup chart for experiments on Polybench/GPU benchmarks.

slowdowns occurred, the CPU execution time was under 1 second. This indicates that for these cases, the overhead involved in moving memory between devices and offloading execution was more significant than any benefits that parallel execution might have

shown. This could possibly be in part due to the problem sizes used in Polybench/GPU benchmarks. We planned on testing these conjectures empirically by comparing results from different compilers, but OpenACC-compliant compilers are more often than not propietary and expensive, which makes further testing challenging.

6. Conclusion

This paper has described DawnCC , a tool that is currently available at `http://cuda.dcc.ufmg.br/dawn/`. The goal of DawnCC is to facilitate the development of parallel programs that run on Graphics Processing Units (GPUs). Developers can feed this tool with plain C code, and it transforms it into parallel code by annotating loops with either OpenACC or OpenMP 4.0 directives. The key benefit of using DawnCC is performance: annotated code can be as much as 70x faster than their original counterparts. DawnCC still offers room for improvements. In particular, sometimes we perceive slowdowns in a few programs that we annotate using this tool. We are working to remove these slowdowns.

Acknowledgement This work has been sponsored by LG Electronics do Brasil through the project *Automatic Parallelization of Code for Mobile Devices*.