Seminar: Efficient Programming

DawnCC - Automatic Parallelization

Author: Anton Kollewe (6988354)
Supervisor: Jannek Squar
University of Hamburg, Germany

April 18, 2020

Abstract

The DawnCC source-to-source compiler project provides a set of static code analyses for automatically annotating sequential C/C++ source code with parallelization and offloading directives. It consists of multiple code passes extending the LLVM compiler infrastructure and supports OpenMP and OpenACC directives. Compilation output is C/C++ source code equivalent to the input with only acceleration directives inserted.
DawnCC has proven to be capable of achieving significant performance gains for certain applications, being especially useful for improving foreign or outdated code. However, it operates only in very limited scope for restricted use-cases and is not yet comparable to a careful manual parallelization.

Contents

1 Introduction

The declining rate of performance improvements in new processing units have brought forth a new hardware paradigm. Whereas up until around 2005 CPU clock speeds have continually increased, due to physical manufacturing limitations it is now commonplace for chips to include multiple processing cores with reduced clock frequencies [25, 12].
Consequently, separate accelerator devices such as *(GP)GPUs*, *FPGAs*, *Intel Xeon Phi* etc. have gained importance for maintaining modern program performances. Those devices may contain a very large number of parallel processing cores, with core counts of up to multiple thousands in modern high-end graphics cards [34, 35] (an illustration of this concept can be seen in figure 1).
These heterogenous architectures now dominate the market in consumer as well as high performance machines. Distributed computing has become state-of-the-art with parallelism happening on various levels, where processor registers contain multiple vectorized data, processors contain multiple cores, machines contain multiple processor sockets and networks contain multiple machines, which has become especially prevalent with the rise of cloud computing and "Internet of Things" smart devices [1, 2, 3, 11, 12].

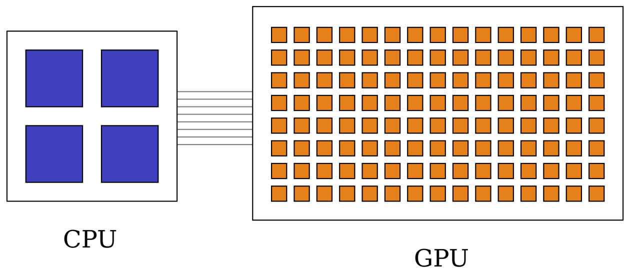


Figure 1: Schematic of modern hardware architectures, containing few "fat cores" (blue) and many "thin cores" (orange) [10].

These architectures can yield large performance gains, however properly harvesting their potential poses a new challenge to software developers.
Many established and frequently used code libraries have been developed for single-core architectures within the past few decades and now need to be adapted to incorporate thread-safety concepts [12]. Reimplementing every existing piece of code is wasteful of already done work and likely infeasible. However, oftentimes the original developers may not be active or available anymore and have moved on to other projects, so the daunting task of maintaining viability falls into the hands of foreign developers. These developers may not have a deep enough knowledge of the code as well as its domain as is required for an effective parallelization. Doing so may also introduce new errors if not done carefully and correctly [1].

The *DawnCC* project, initiated at the *Universidade Federal de Minas Gerais* in Brazil, aims to resolve these issues. It provides a machine-assisted and automated way of parallelizing already existing, sequential source code when it is not worth the cost or too complex to be feasible.
DawnCC is a source-to-source compiler for automatic injection of parallelization and offloading instructions into C/C++ source code. The analyses are static, meaning that the code is being compiled but not run (as opposed to dynamic analyses) and all necessary information is inferred from the code as is. It is limited to single-machine architectures like multi-core CPUs and offload-cards within the same computer. Distributed clusters that require networking, like cloud and high-performance-computing applications, are not within scope. The project is open source and available on GitHub [4, 29].

2 Prerequisites

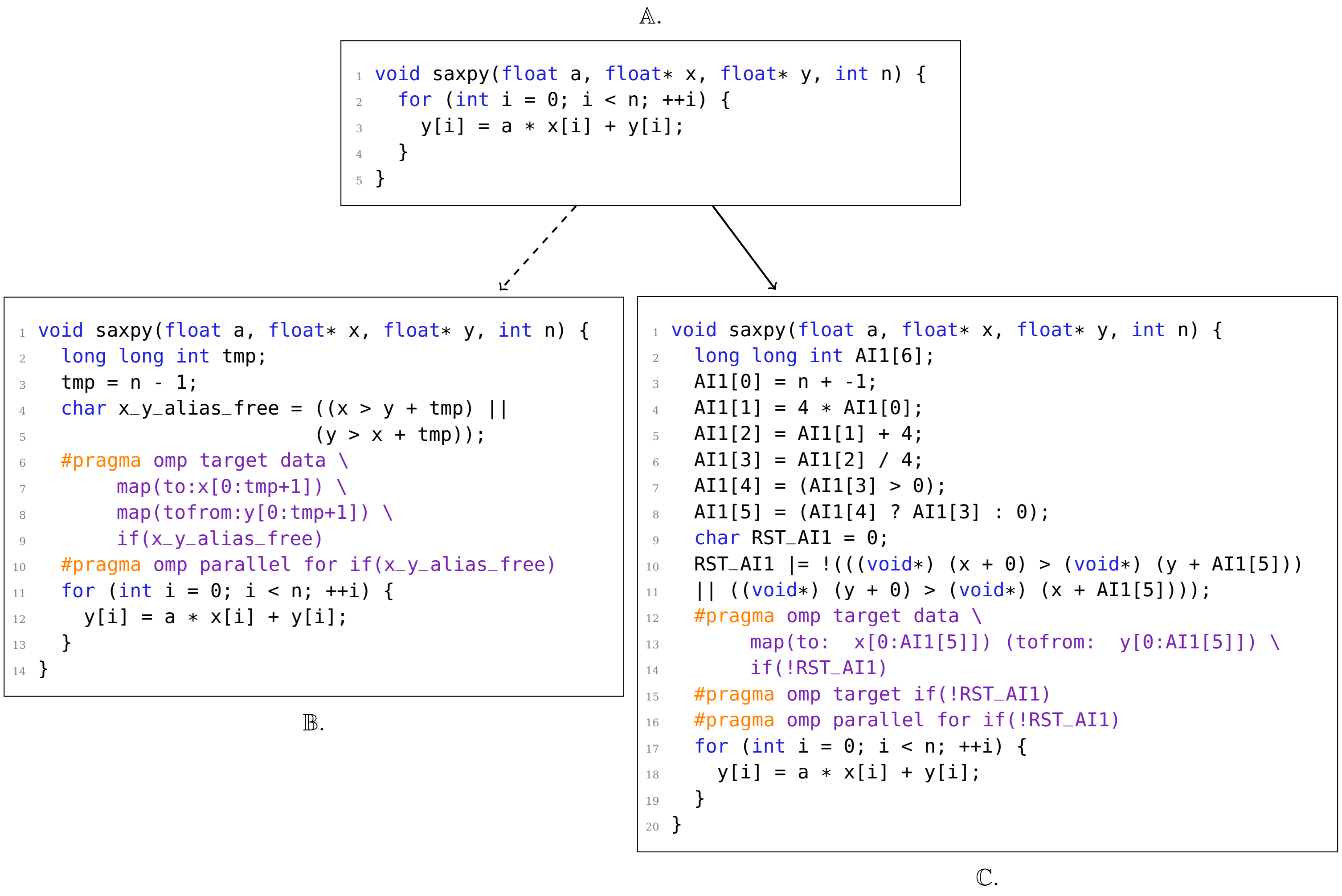
DawnCC non-destructively inserts parallelization *pragmas* [20] into C/C++ source code. These pragmas are extensions to the language specification and supported by most modern compilers. They serve as annotations specifying which parts of code shall be ran in parallel, how operations are to be split and scheduled, as well as which data shall be copied to and from accelerator devices, if needed. Two very popular annotation specifications, *OpenMP* and *OpenACC* are supported by DawnCC, wherein both are interchangeable and the decision typically comes down to personal preference or support for a specific platform [7, 8].
An example usage of DawnCC is shown in figure 2 and compared to what the developers [1] claimed the output to be. More discussion on this will follow in section 5.
The DawnCC project resides within the LLVM compiler infrastructure and consists of a set of LLVM *passes* [18]. Which passes are available and how exactly they are implemented and used will be discussed in section 3.
More details on *OpenMP*, *OpenACC* and *LLVM passes* follow in this section.

2.1 OpenMP / OpenACC

The *OpenMP* and *OpenACC* projects consist of collections of compiler directives, library functions and environment variables for annotating code with parallelization and offloading instructions.
The compiler directives make up platform-independent extensions to the C/C++ and Fortran standards, so parallel logic is represented directly in source code. Many different acceleration devices from multiple vendors are supported to provide high portability [7, 8].

OpenMP and OpenACC may be interleaved into already existing sequential source code for offloading and multi-threading certain parts of it. This allows for a smooth and incremental transition to parallel execution without affecting already written code and semantics.
Nowadays, both projects are equally proficient and suitable for creating state-of-the-art parallel programs. DawnCC supports both OpenMP and OpenACC directives interchangeably to flexibly fit into any established codebase or developer preference [7, 8].

Little intervention and supervision is required of the programmer, most data operations as well as offloaded computation scheduling are deduced and handled automatically. This is especially prominent in OpenACC which requires less explicit control over parallelization procedures than OpenMP does.
However, while automation is a significant advantage of OpenMP and OpenACC over traditional manual parallelization in terms of programming effort, performance can suffer in certain edge cases where invalid assumptions are made. These drawbacks also transfer over to DawnCC, as will be discussed further in section 5 [7, 8, 22, 23, 24].



𝔸.

```c
void saxpy(float a, float* x, float* y, int n) {
  for (int i = 0; i < n; ++i) {
    y[i] = a * x[i] + y[i];
  }
}
```

```c
void saxpy(float a, float* x, float* y, int n) {
  long long int tmp;
  tmp = n - 1;
  char x_y_alias_free = ((x > y + tmp) ||
                         (y > x + tmp));
  #pragma omp target data \
       map(to:x[0:tmp+1]) \
       map(tofrom:y[0:tmp+1]) \
       if(x_y_alias_free)
  #pragma omp parallel for if(x_y_alias_free)
  for (int i = 0; i < n; ++i) {
    y[i] = a * x[i] + y[i];
  }
}
```

𝔹.

```c
void saxpy(float a, float* x, float* y, int n) {
  long long int AI1[6];
  AI1[0] = n + -1;
  AI1[1] = 4 * AI1[0];
  AI1[2] = AI1[1] + 4;
  AI1[3] = AI1[2] / 4;
  AI1[4] = (AI1[3] > 0);
  AI1[5] = (AI1[4] ? AI1[3] : 0);
  char RST_AI1 = 0;
  RST_AI1 |= !(((void*) (x + 0) > (void*) (y + AI1[5]))
  || ((void*) (y + 0) > (void*) (x + AI1[5])));
  #pragma omp target data \
       map(to:  x[0:AI1[5]]) (tofrom:  y[0:AI1[5]]) \
       if(!RST_AI1)
  #pragma omp target if(!RST_AI1)
  #pragma omp parallel for if(!RST_AI1)
  for (int i = 0; i < n; ++i) {
    y[i] = a * x[i] + y[i];
  }
}
```

ℂ.

Figure 2: Example usage of DawnCC on a *saxpy* function [33], 𝔸. input source, 𝔹. expected/desired result (adapted from [1]), ℂ. actual result. Compiled using the official DawnCC online compiler website [5].

2.2 LLVM passes

The DawnCC project resides within the LLVM compiler infrastructure. LLVM is structured in a modular way, being semantically split into a front-end, a middle-and and a back-end. The front-end part of a compiler receives source code input, performs lexical and grammatical analysis and transforms it into an *Intermediate Representation* (LLVM IR) [15] to be passed on to the middle-end which represents the next step in the compilation process [13].

The middle-end stage of compilation operates on LLVM IR, analyzing and transforming it in an iterative fashion. The entire or parts of the intermediate code are ran over multiple so-called *passes*, each of which also returns (possibly transformed) IR code. This process may repeat any number of times, allowing for any combination of analyses, optimizations and transformations to take place. Passes are executed through the use of LLVM's *opt* tool [16].
Different types of passes are available to be plugged into an LLVM pipeline. These notably include the *ModulePass* running over the entire input code and the *FunctionPass* being applied to every function within the input code individually [18, 13].

DawnCC consists entirely of such passes, all of which being function passes, with the exception of a single module pass handling source-code reconstruction. The popular *clang* front-end for C-family languages is used as the DawnCC front-end generating the LLVM IR [2].

A back-end component, responsible for emitting platform-specific machine code, is not relevant to DawnCC, it being a source-to-source compiler.
Instead, a pass for reconstructing the original input source code and inserting parallelization directives is employed. Reconstruction is made possible by debug information carried throughout compilation [2, 13].

3 Implementation

At the core of DawnCC's functionality are the *Memory Bound Estimation* and *Loop parallelization* passes. These analyses allow detection and leveraging of potentially parallelizable code sections.

3.1 Memory Bound Estimation

When offloading computations to accelerator devices with separate memory (e.g. a GPU), any data used in the computation has to be present on said device. This requires a prior copying of data from CPU main memory to the accelerator and collection after computations are done. The parts of data to be copied are usually determined by the developer, who has an understanding of the algorithm. DawnCC needs to determine these memory bounds autonomously.
When multithreading, all threads in a single program operate on the same memory, which can lead to concurrency errors on simultaneous data accesses. Because of this, one needs to know all data dependencies and overlaps between threads to put data privatization mechanisms into place or determine if parallelization is even possible at all [7, 8, 26].

Both of these cases may be automized with static code analyses determining the address bounds and timings of memory accesses.
Because data operation directives shall ultimately be injected into front-end source code, memory bounds need to be expressed in terms of C/C++ code symbols (not IR) while the analyses itself

works on IR instructions. Within the front-end *clang* call, the input code is compiled using debug information. This information allows DawnCC to map IR instructions to their corresponding source symbols. The debug information is also what will allow the source code reconstruction explained in section 3.3 [1, 2, 3].

To deduce memory bounds, a more general problem needs to be solved first. Because in C/C++ memory may be accessed by pointer dereferencing or array indexing, the range of values held by a code symbol needs to be known in order to translate a memory access into the underlying addresses. A symbol in this context refers to an identifier in source code, e.g. a variable name or value literal [1].

Let S be the set of all integer symbols and let P be the set of all instructions within the program. $\mathbb{I}$ shall describe the set of possible integer values, in the case of 32-bit signed integers this set is $\mathbb{I} = \mathbb{I}_{32} := \{-2^{31}, -2^{31}+1, ..., 2^{31}-1\}$ [19].
At the time of every instruction i in P, there is a "true value" $v(s,i)$ in $\mathbb{I}$ for every symbol s in S. Ideally, we would like to know this value for every instruction within the program (i.e. find a map $v : S \times P \to \mathbb{I}$), however in a static analysis, in general this mapping cannot be determined for every symbol.[1] The best we can do is to determine upper and lower bounds after every instruction for every symbol representing an integer value.

The *symbolic range* $[l,u] : l,u \in \mathbb{I}$ of an integer consists of a *lower bound* l and an *upper bound* u. We call the set of all possible bounds $\mathbb{B} := \{[l,u] : l,u \in \mathbb{I}\}$. Thus, we want to determine a map $S \times P \to \mathbb{B}$ from each symbol at each instruction to the bounded range of values it may contain directly before executing this instruction.

$$\boxed{\text{If } (s,i) \mapsto [l,u] \text{ then } l \leq v(s,i) \leq u \text{ at the time } i \text{ is executed.}}$$

DawnCC provides a set of rules for determining resulting ranges for every LLVM IR instruction. We will now present a few notable examples. See [4] for the full set of rules and their implementation. Because the instructions in P are ordered sequentially, we use $i+1$ to represent the instruction directly after $i \in P$.
For a symbol $s \in S$ at instruction $i \in P$ with $(s,i) \mapsto [l,u]$:

- When s is uninitialized at the time of i, its bounds are estimated as $(s,i) \mapsto [min(\mathbb{I}), max(\mathbb{I})]$, meaning it may have any integer value.
- If i assigns an integer literal $x \in \mathbb{I}$ to s, the new bounds are $(s,i+1) \mapsto [x,x]$.
- If i is a conditional branch instruction (e.g. `if`) continuing if s is larger than some literal $x \in \mathbb{I}$ then $(s,i+1) \mapsto [x, max(u,x)]$ (analogously for less-than comparisons).
- If i assigns the sum of s and another symbol s_o with bounds $[l_o, u_o]$ to s then $(s,i+1) \mapsto [l+l_o, u+u_o]$ (analogously for subtraction, multiplication and other arithmetic operations).

Such rules are defined for every LLVM IR instruction. These instructions are less in amount and simpler to understand than complex high-level C/C++ code lines. This is mostly due to the restricting SSA (Single static assignment) rules within IR [4, 14].

Since the code analysis is static, whenever the control-flow branches, any possible program path may be taken at runtime. Thus, all possible paths need to be considered when inferring symbolic bounds. The bounds (if and) after the flow joins again (e.g. after an `if/else` or `switch`

[1]Trivial cases proving this are uninitialized variables and non-deterministic user input.

block) can now be anywhere within the *union* bound of all paths. The union of two bounds $[l_1, u_1], [l_2, u_2] \in \mathbb{B}$ is defined by the smallest bound containing both bounds: $[l_1, u_1] \cup [l_2, u_2] := [min(l_1, l_2), max(u_1, u_2)] \in \mathbb{B}$.

Loops pose another challenge. Bound-modifications within loop iterations may happen a number of times only determined at runtime. For certain special cases where these counts may be represented easily, DawnCC still manages to infer bounds. This is possible, for example, for either fixed iteration counts determined by literals or linear for-loops with an iteration count directly written as a symbol (see figures 2 and 3).
In other cases, where the loop structure may be arbitrarily complex, DawnCC gives up and resets the symbol's bounds as if they where uninitialized/unknown.

When memory is accessed using an integer symbol, either through pointer casting or array indexing, the symbol bounds translate into memory access bounds. This way it can automatically be determined which memory regions are required for computations inside some code section. If some array at base memory address `a` is indexed with `a[s]` at instruction i, the memory bounds follow as $[\mathsf{a} + l, \mathsf{a} + u]$ for $(s, i) \mapsto [l, u]$ (note that bound edges may also be negative).
This inference allows offloading memory sections to other devices for computation [1, 2, 3].

3.2 Loop parallelization

In order for memory bound estimation to be useful, any parallelization or offloading must happen. When using DawnCC, one may choose to manually annotate code sections as parallel and only using DawnCC for memory transfers. However, methods for inferring parallelizable sections are available.

While technically any code section may be able to run in parallel, most of computationally intensive work is done within for-loops. They are also easier to analyze, since arbitrary code in general may be very complex.
A loop is parallelizable if no data dependencies exist between different iterations of the loop, which means that data accesses of mixed reads and writes do not overlap with changing iterations. This topic has been well researched and, while used by it, does not fall within scope of the DawnCC project. Mature analyses exist operating on program-dependence and control-flow graphs (see [31, 27, 32]). Because for-loops in C/C++ can abide to conditions of any complexity, solving this problem is very hard in general. Thus, DawnCC is restricted to the simple case of single-variable, single-increment for-loops, although arguably, this set of for-loops makes up most relevant use cases. An example of a loop detectable by DawnCC, as well as a loop of seemingly similar difficulty which DawnCC fails to analyze is shown in figure 3.

```c
void foo(int a) {
  int n[100];
  int i;
  for (i = 0; i < a; i += 1) {
    n[i] = n[i] + 1;
  }
  for (i = 0; i < a; i += 2) {
    n[i] = n[i] + 1;
  }
}
```

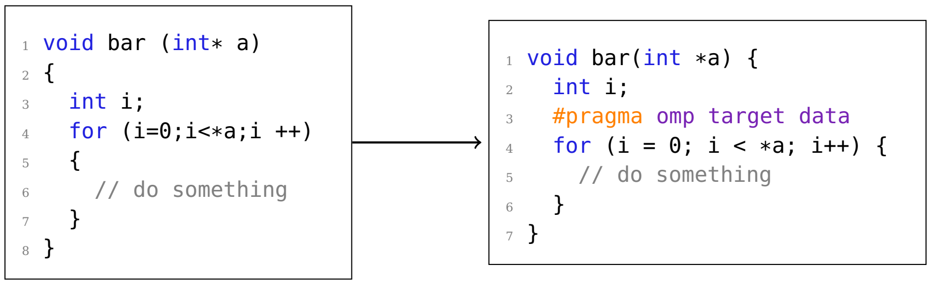
```c
void foo(int a) {
  int n[100];
  int i;
  #pragma omp target data \
       map(tofrom:  n[0:100])
  #pragma omp parallel for
  for (i = 0; i < a; i += 1) {
    n[i] = n[i] + 1;
  }
  for (i = 0; i < a; i += 2) {
    n[i] = n[i] + 1;
  }
}
```

Figure 3: Two loops ran through DawnCC, failing to detect the second loop as parallel. For demonstration purposes the output is simplified (similar to figure 2). Compiled using the official DawnCC online compiler website [5].

3.3 Source code reconstruction & injection

As is pointed out in section 2.2, DawnCC sits in the *middle-end* of the LLVM compiler infrastructure where it operates through passes over functions and modules. In this stage, source code is processed in its *Intermediate Representation* (IR) to simplify certain types of static analyses and code transformations. As to be seen in sections 3.1 and 3.2, this turns out to be very useful, however it poses a challenge. The compiler directives need to be injected into front-end source code, not IR.

When parsing the code in the front-end, debug symbols may optionally be included. These are additional information within the binary executable which provide links to the original source code with line references or (partial) inclusion of input source code within the executable. This is also what allows online debuggers like *gdb* [21] to present an interface to the source code, without the programmer having to deal with assembler instructions.
DawnCC uses debug information to rebuild the entire input source code, but is also able to inject parallelization and offloading directives into places determined using the loop parallelization and memory bound estimation techniques discussed above.
DawnCC's source code reconstruction is not necessarily true to the original input. Within the DawnCC execution *clang-format* [17] is run, a tool for formatting source code using some style convention. This might modify certain parts of the code (e.g. whitespaces) which are not related to the parallelization aspect, see figure 4. More discussion on this will follow in section 5 [1, 2, 3].



```c
void bar (int* a)
{
  int i;
  for (i=0;i<*a;i ++)
  {
    // do something
  }
}
```

```c
void bar(int *a) {
  int i;
  #pragma omp target data
  for (i = 0; i < *a; i++) {
    // do something
  }
}
```

Figure 4: DawnCC example to showcase style modifications. Compiled using the official DawnCC online compiler website [5].

4 Optimization

In addition to the basic analyses, a few additional features and optimizations are available within DawnCC to further improve performance and convenience.

4.1 Pointer restrictification

Different symbols in source code may evaluate to the same value. Thus, two pointers may point to the same (or an overlapping region of) memory. This behavior, where two different memory accesses evaluate alike, is called *pointer aliasing* [28].
To confidently determine if a loop is parallelizable, pointer aliasing between loop iterations needs to be considered. For example, if the arrays `x` and `y` in figure 2 overlap within the first `n` elements, the saxpy function cannot be executed in parallel.
However, in the cases that they do not overlap, concurrency is still possible. Isolating and exploiting those cases improves performance over not parallelizing at all (except when the overhead is not worth it, see section 5). By statically deducing conditions for detecting pointer aliasing at runtime, this mechanism can be leveraged within DawnCC. An example can be seen in figure 2, where the `x_y_alias_free` symbol determines if concurrency is possible.
These conditions are a byproduct of DawnCC's memory bound estimation explained in section 3.1. An overlap is directly inferable from accessed memory regions of the pointers. This technique, called *pointer restrictification*, allows annotating loops that may not be guaranteed but may still end up being able to run in parallel at runtime.

4.2 Coalescing

Multiple offloaded sections with copy operations may be unified into *common transfer blocks*, where data is copied once only for multiple sections using the same data. An example case of this is shown in figure 5. This optimization has some drawbacks, discussed in section 5 [3].

```
#pragma omp target data \
        map(tofrom:  x[0:n-1])
#pragma parallel for
for (int i = 0; i < n; i++) {
    x[i] += i;
}

#pragma omp target data \
        map(tofrom:  x[0:n-1])
#pragma parallel for
for (int i = 0; i < n; i++) {
    x[i] *= 2;
}
```

```
#pragma omp target data \
        map(tofrom:  x[0:n-1])
{
    #pragma parallel for
    for (int i = 0; i < n; i++) {
        x[i] += i;
    }
    #pragma parallel for
    for (int i = 0; i < n; i++) {
        x[i] *= 2;
    }
}
```

Figure 5: An example of copy coalescing. Left: before coalescing, right: after coalescing.

5 Discussion

DawnCC is undoubtedly able to significantly improve performance of certain kinds of programs (see benchmarks in figure 6). However, the examples and benchmarks given by the developers are fabricated specifically as benchmarking subjects and may not be representative of real-word applications being far more complex and domain-specialized.
Due to the fact that DawnCC will only process simple for-loops (see 3.2), any other method of splitting work will fail to be optimized. That being said, almost all computationally expensive operations are implemented in the form of loops.

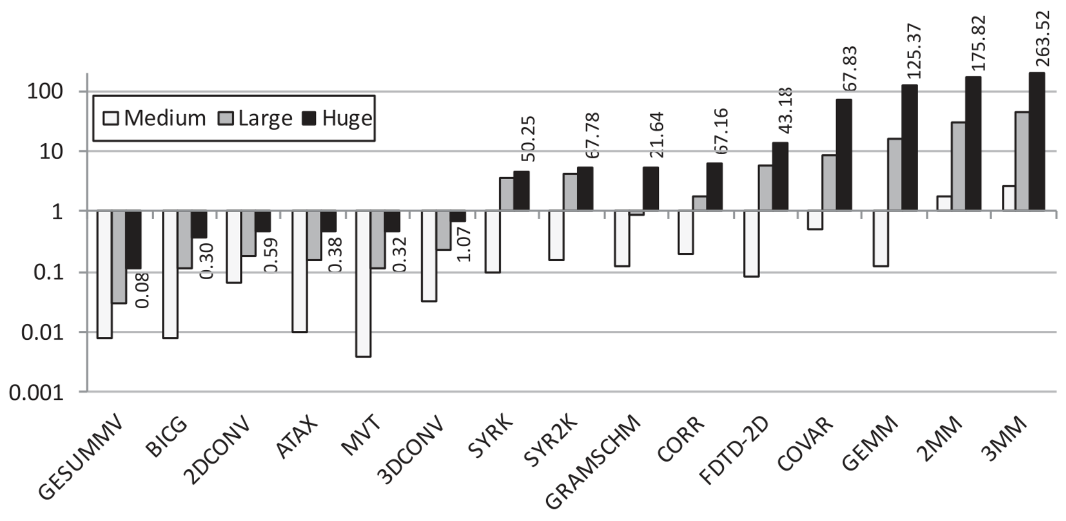


Figure 6: DawnCC speedup benchmark of the PolyBench suite. Comparison of three different input sizes (Medium, Large, Huge) and certain algorithms [9]. Adapted from [3].

Because DawnCC so rigidly retains control flow structures it will not find more performant algorithms or even mathematical underlying methods which have been developed and adapted for parallel execution in some cases. This is a vast research topic and commonly regarded as currently impossible to achieve without human intervention. Effectively parallelizing a specific program also requires deep and interdisciplinary knowledge of the surrounding domain and research as well as the active hardware and architecture [12].

As described in section 2.2, DawnCC passes are ran on individual functions separately. This means that an operation spanning multiple functions will not be considered for parallelization. Because of this, it is incompatible with certain optimizations (e.g. function outlining, opposite of inlining [30]) or applications and performance can vary greatly depending on the type of compiler, optimizations and programming language used.

There are cases where a loop is flawlessly parallelizable, yet doing so will not result in performance gains. An example of this are extremely short loops.
Whenever the relevant computation inside a loop body is less expensive than the cumulative overhead required to spawn new threads or copy data to external memory and schedule workload onto the different processing units, doing so will actually worsen the performance in comparison to a sequential execution (see speedups < 1 in figure 6).
Predicting the resource usage of a program statically is rather difficult and not within the scope of what DawnCC aims to provide, so in this case DawnCC will assume a parallel loop without assessing if it is worth doing so beforehand.
Such cases need to be manually reverted after the fact. But instances may be hard to find in more complex or foreign programs. This also makes DawnCC unsuitable for usage within automated pipelines, where a developer might not oversee the compilation results and consequently won't notice the issue without fine-grained performance evaluation.

When DawnCC is not used as an automated precompilation step, but used once for migrating an existing program to a modern and parallel paradigm, some different problems arise.
As part of the source code reconstruction explained in section 3.3 and shown in figure 4, by default, the entire code is formatted to a unified syntax convention using `clang-format`. Whilst a unified codestyle is generally a good idea, this has several drawbacks. For one, it can be rather intrusive to ignore user preferences and may require the developer to reconstruct his previous style after using DawnCC and it might make it more difficult to continue working on the code afterwards. Potentially even more significant is the fact, that comparisons of before and after DawnCC execution will not clearly reveal what has actually changed, since artifacts of a reformat will be riddled throughout code parts that may not even have been touched. This further amplifies the issue of hard to find errors in parallelization described above.

Optimizing the DawnCC output source code using *coalescing* (see 4.2) will improve performance by reducing data copy operations. But it will arguibly also decrease the comprehensibility of the resulting code. Coalescing introduces new dependencies on structuring and ordering of code sections. If a developer does not properly recognize the optimizations made, this might lead to errors when uncautiously refactoring or reordering source code. This aspect is once again even more impactful when working with code written by someone else. That being said, coalescing is an optional feature of DawnCC.

As is shown in figure 2 the true output is not as claimed in the developers publications, specifically more cluttered and less readable, which might lead to a worse project maintainability. This may be due to a version inconsistency at the time of creating the example or the time of the last website update. The intended functionality of DawnCC might also not be reflective of the current state of development yet.

6 Conclusion

In conclusion, DawnCC presents a potent tool for modernizing already existing code. It is especially useful for cases where development resources are limited or not a priority. It also provides an unconditional optimization, essentially for free (with certain caveats), that is usable as part of some larger automated compilation pipeline. Applying it correctly and cautiously is still required however, since some new sources of errors are introduced. One also needs to be aware of cases which may possibly reduce performance or project maintainability significantly.

DawnCC does not yet embody exactly what was set out to achieve but the project is still under active development and there is outlook for further improvements like variable privatization, cross-function concurrency and more types of detectable loops.
While the demand for such technology has risen immensely with the change of modern hardware, it is still very much an unfinished field of research which might experience some major strides in the future and will generally become increasingly viable over time. This can potentially tie in to other currently popular aspects like code analysis using artificial intelligence to expand the boundaries of what is possible within compiler-based optimizations.
For now, automatic code parallelization remains an open area of research and while fully fledged tools do not exist yet, the work towards it is being done.

References

[1] *Automatic Insertion of Copy Annotation in Data-Parallel Programs*, G. Mendonca et al. https://homepages.dcc.ufmg.br/~fernando/publications/papers/SBAC16_Gleison.pdf, last accessed Nov 10, 2019.

[2] *DawnCC: a Source-to-Source Automatic Parallelizer of C and C++ Programs*, B. Guimaraes et al., https://homepages.dcc.ufmg.br/~fernando/publications/papers_pt/Breno16Tools.pdf, last accessed Nov 08, 2019.

[3] *DawnCC: Automatic Annotation for Data Parallelism and Offloading*, G. Mendonca et al., http://cuda.dcc.ufmg.br/dawn/files/taco-paper.pdf, last accessed Feb 27, 2019.

[4] *DawnCC - Github repository*. https://github.com/gleisonsdm/DawnCC-Compiler, last accessed Nov 18, 2019.

[5] *DawnCC - Project website*. http://cuda.dcc.ufmg.br/dawn/, last accessed Feb 27, 2019.

[6] *Automatic Insertion of Copy Annotations in Data-Parallel Programs*, G. Mendonca et al. http://cuda.dcc.ufmg.br/dawn/include/papers/SBAC_PAD16.pdf, last accessed Nov 09, 2019.

[7] *OpenMP Application Programming Interface*. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf, last accessed Nov 18, 2019.

[8] *The OpenACC Application Programming Interface*. https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.7.pdf, last accessed Nov 18, 2019.

[9] *PolyBench*. http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/, last accessed Feb 27, 2020.

[10] *"Back to Thin-Core Massively Parallel Processors" in Computer, vol. 44, no. 12, pp. 49-54*, A. Marowka, Dec. 2011.

[11] *Future internet: The Internet of Things*, 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE), Chengdu, pp. V5-376-V5-380, Lu Tan & Neng Wang, 2010.

[12] *Hochleistungsrechnen*, Prof. Dr. Thomas Ludwig, 2019. https://cloud.wr.informatik.uni-hamburg.de/s/GtwRaaRT3wMYPbA/download?path=%2F&files=hr-1920.2019-10-13.pdf, last accessed Feb 25, 2020.

[13] *LLVM - Infrastructure*, Jan Moritz Witt, Nov 12 2019. https://wr.informatik.uni-hamburg.de/_media/teaching/wintersemester_2019_2020/ep-1920-witt-llvm-infrastructure-praesentation.pdf, last accessed Feb 25, 2020.

[14] *Static single assignment form*. https://en.wikipedia.org/wiki/Static_single_assignment_form, last accessed Nov 18, 2019.

[15] *LLVM Language Reference Manual*. http://llvm.org/docs/LangRef.html, last accessed Nov 18, 2019.

[16] *opt - LLVM optimizer*. http://llvm.org/docs/CommandGuide/opt.html, last accessed Nov 18, 2019.

[17] *ClangFormat*. https://clang.llvm.org/docs/ClangFormat.html, last accessed Feb 27, 2019.

[18] *Writing an LLVM Pass*. http://llvm.org/docs/WritingAnLLVMPass.html, last accessed Nov 18, 2019.

[19] *Fundamental types*. https://en.cppreference.com/w/cpp/language/types, last accessed Feb 27, 2020.

[20] *Implementation defined behavior control*. https://en.cppreference.com/w/cpp/preprocessor/impl, last accessed Feb 27, 2020.

[21] *GDB: The GNU Project Debugger*. https://www.gnu.org/software/gdb/, last accessed April 18, 2020.

[22] *PTHREADS(7) Linux Programmer's Manual*. http://man7.org/linux/man-pages/man7/pthreads.7.html, last accessed Feb 27, 2020.

[23] *CUDA*. https://www.geforce.com/hardware/technology/cuda, last accessed Feb 27, 2020.

[24] *OpenCL Overview* . https://www.khronos.org/opencl/, last accessed Feb 27, 2020.

[25] *Moore's Law*. https://en.wikipedia.org/wiki/Moore%27s_law, last accessed Feb 27, 2019.

[26] *Hazard (computer architecture)*. https://en.wikipedia.org/wiki/Hazard_(computer_architecture), last accessed Nov 18, 2019.

[27] *Control-flow graph*. https://en.wikipedia.org/wiki/Control-flow_graph, last accessed Nov 18, 2019.

[28] *Pointer aliasing*. https://en.wikipedia.org/wiki/Pointer_aliasing, last accessed Feb 27, 2019.

[29] *Static program analysis*. https://en.wikipedia.org/wiki/Static_program_analysis, last accessed Feb 27, 2019.

[30] *Inline function*. https://en.wikipedia.org/wiki/Inline_function, last accessed Feb 27, 2019.

[31] *The Program Dependence Graph and Its Use in Optimization*, J. Ferrante et al., 1986. https://cs.gmu.edu/~white/CS640/p319-ferrante.pdf, last accessed Nov 18, 2019.

[32] *CMPT886 part2: "Program Representations"*, 2015. https://www.youtube.com/watch?v=5vvSJ_gWkCY, last accessed Nov 18, 2019.

[33] *Six Ways to SAXPY*, Mark Harris, 2012. https://devblogs.nvidia.com/six-ways-saxpy/, last accessed Feb 25, 2020.

[34] *Nvidia Store*. https://www.nvidia.com/en-us/shop/, last accessed Feb 25, 2020.

[35] *AMD Shop*. https://www.amd.com/en/shop/de/Graphics%20Cards, last accessed Feb 25, 2020.