Loops of the Domain-specific Programming Language DaphneDSL

Borko Bošković, Janez Brest, Aleš Zamuda

Computer Architecture and Languages Laboratory
Institute of Computer Science
Faculty of Electrical Engineering and Computer Science
University of Maribor
Koroška cesta 46, 2000 Maribor, Slovenia
e-mail: borko.boskovic@um.si, janez.brest@um.si, ales.zamuda@um.si

Zanke domensko specifičnega programskega jezika DaphneDSL

Povzetek — Ta prispevek uporablja domensko specifičen programski jezik DaphneDSL projekta DAPHNE, nudi pregled sistemske arhitekture DAPHNE in nato predstavi dva primera DaphneDSL. Prvi primer izračuna vsoto naravnih števil, drugi pa energijo binarnih zaporedij. Za razvoj sistema DAPHNE je na GitHubu repozitoriju ponujen seznam težav, ki se jih lahko razvijalci lotevajo ter ponujajo rešitve. Ta prispevek obravnava eno tako težavo, ki se pojavi v programskih zankah s številnimi ponovitvami znotraj domensko specifičnega jezika DaphneDSL in ponuja začasno rešitev. V času pisanja tega prispevka so razvijalci hitro obravnavali prijavo težave, našli in odpravili napako ter težavo razrešili.

Abstract — This contribution uses the DaphneDSL domain-specific programming language of the DAPHNE project, provides an overview of the DAPHNE system architecture, and then presents two DaphneDSL examples. The first example calculates the sum of natural numbers and the second energy of binary sequences. For the development of the DAPHNE system, a list of issues that developers can tackle and solve is offered on the GitHub repository. This paper addresses one such issue that occurs in programming loops with many iterations within the domain-specific language DaphneDSL and offers a temporary workaround. At the time of writing this paper, the issue has been quickly addressed by developers, a bug found and fixed, and issue closed.

1 Introduction

The DAPHNE [1] project (integrated Data Analysis Pipelines for large-scale data management, High-performance computing, and machiNE learning) provides a system for data analysis. This includes managing and processing data using data analysis pipelines. The system is designed to enable efficient use of machine learning and supercomputers.

The DAPHNE system [2] is designed to enable efficient productivity for integrated data pipelines through an open application interface using the domain-specific language DaphneDSL. The performance of the system was also measured through execution of workloads on state-of-the-art heterogeneous devices and supercomputers in [2, 3, 4]. Special attention is given to scheduling of tasks, level of redundancy, and sparsity of matrices.

The extended compilation system, including the compilation stages and runtime execution, is published as open source at the following address: `https://github.com/daphne-eu/daphne/`.

The domain-specific language, DaphneDSL, was modeled after machine learning systems and numerical computing languages and libraries such as Julia, NumPy, R, and SystemDS DML [5]. Its syntax is case-sensitive and similar to the syntax of the C/C++ and Java programming languages. One can write it in files with the extensions `.daphne` or `.daph`. The language supports the following concepts: variables, data types, comments, expressions, control structures, loops, functions, etc., as documented at `https://daphne-eu.github.io/daphne/DaphneDSL/LanguageRef/`.

For the development of the DAPHNE system, a list of issues that developers can tackle and solve is offered on the GitHub repository, at `https://github.com/daphne-eu/daphne/issues`. While using DaphneDSL, we were interested in one such listed issue (`DaphneDSL loops crash after a certain number of iterations`, #77, labeled `bug`), pertaining to loops that have a large number of iterations. This issue occured in programming loops with many iterations within DaphneDSL and we applied a temporary workaround. At the time of writing this paper, the issue has been quickly addressed by developers, a bug found and fixed, and issue closed.

The rest of this paper is organized as follows. In the second section, the related work on system architecture is presented. In the third section, the related work on domain-specific language DaphneDSL is presented, followed by two examples with loops. In the fourth section, we illustrate an example of a program for calculating the energies of binary sequences. The fifth section gives the code testing with loops and findings. The last section contains the conclusion.

2 System Architecture

The user core of the DAPHNE system architecture is its execution environment. This allows the execution of flows and operations as defined in the DaphneDSL language scripts and DaphneLib libraries. The system

Code 3: Code snippet in DaphneIR dialect.

```
module {
func.func @main() {
 %0="daphne.constant"(){value = 0 :  si64}:() -> si64
 %1="daphne.constant"(){value = "Sum is:  "}:()->!daphne.String
 ...
 %11=scf.for %arg0=%6 to %5 step %6 iter_args(%arg1=%0)->(si64){
    %c1_i32=arith.constant 1:i32
    %14="daphne.call_kernel"(%arg0,%c1_i32,%10){callee=
     "_cast__int64_t__size_t"}:(index,i32,!daphne.DaphneContext)->si64
    %c2_i32=arith.constant 2:i32
    %15="daphne.call_kernel"(%14,%4,%c2_i32,%10){callee=
     "_ewMul__int64_t__int64_t__int64_t"}:
     (si64,si64,i32,!daphne.DaphneContext)->si64
   %c3_i32=arith.constant 3:i32
   %16="daphne.call_kernel"(%arg1,%15,%c3_i32,%10){callee=
    "_ewAdd__int64_t__int64_t__int64_t"}:
    (si64,si64,i32,!daphne.DaphneContext)->si64
   scf.yield %16:si64
}
...
```

Code 1: Program for calculating the sum of the first 1000 natural numbers.

```
# File:  program.daphne
# Initialization
sum = 0;

# Calculation of sum
for (i in 1:1000) {
   sum = sum + i;
}

# Output result
print("Sum is:  "+sum);
```

Code 2: Parsing, compiling and running the program.

```
$ bin/daphne program.daphne
$ Sum is:  500500
```

uses multi-level translation. The compiler framework MLIR [6] (Multi-Level Intermediate Representation) [3, 7] is used for this purpose. This framework enables the implementation of extensible compilers, including the use of fragmented software, compiling for heterogeneous hardware, and linking compilers. Because of these properties, MLIR made it possible to build the domain-specific language DaphneDSL in a relatively fast and efficient way. DaphneDSL scripts are converted to DaphneIR intermediate code using MLIR. Multiple passes of code optimization allow an efficient pipeline design and kernel execution. Thus, certain operations are implemented for specific hardware on a local computer or distributed environment.

DaphneDSL programs are executed hierarchically. The runtime coordinator receives the DaphneDSL code and creates a program execution plan. The compiler recognizes workloads and whether they will run on a local computer or in a distributed environment. In the case of a local computer, the workloads are executed within a single computer; in the case of a distributed system, the execution is distributed between several computer nodes.

3 DaphneDSL Programming Language

The goal of DaphneDSL is to provide an open and extensible development environment for integrated data analysis pipelines. This includes data management and query processing, high-performance computing, and machine learning [8]. The compiler is based on the MLIR framework. DaphneDSL maps its code to the intermediate code of DaphneIR or the Daphne dialect of the MLIR framework. Several algorithms with loops already provided in the DAPHNE system use loops and are available as DaphneDSL scripts at `https://github.com/daphne-eu/daphne/tree/main/scripts/algorithms`.

To demonstrate the syntax and build up a test example, let us write a simple program that will summarize the first 1000 natural numbers and print the result to the standard output in the file `program.daphne`. The program is shown in Code 1. The commands for parsing, compiling, and running this program are shown in Code 2. In the DAPHNE system, for parsing and converting the code into the DaphneIR intermediate code (the dialect of the MLIR framework), a domain-specific grammar is used in the system, implemented with the help of the ANTLR4 tool [9]. Code 3 therefore shows an example snippet of the DaphneIR intermediate code of our DaphneDSL example from Code 1. This intermediate code is then passed to the compiler and runtime environment. A longer example with loops follows in the next section.

4 Calculation of the Energies of Binary Sequences

The Low-Autocorrelation Binary Sequences (LABS) problem is a well-known optimization problem. To solve this problem, we need an efficient calculation of the energies of binary sequences. Implementations of these calculations in programming language C++ already exist [10, 11] and computational power of graphical processing units has also been utilized for them [12].

A binary sequence of length L is defined as $Z_L = \{z_1, z_2, ..., z_L\};\ z_i \in \{+1, -1\}$. The energy E of the binary sequence Z_L is calculated as shown in Equation (1). This Equation contains the expression $C_k(Z_L)$, which determines the autocorrelations of the sequence Z_L. The main goal of the LABS problem is to find a sequence Z_L that minimizes E as shown in Equation (2), where B_L represents the set of all sequences or the search space and Z_L^* is the sequence with the minimum energy value.

$$E(Z_L) = \sum_{k=1}^{L-1} C_k^2 \tag{1}$$

$$C_k(Z_L) = \sum_{i=1}^{L-k} z_i \cdot z_{i+k}$$

$$Z_L^* = \underset{Z_L \in B_L}{\arg\min}\, E(Z_L) \tag{2}$$

The implementation of the energy calculation in the DaphneDSL language is shown in Code 4. This implementation contains a function `calcE` to calculate the energy and a loop that for each L generates a sequence of a certain length, calculates its energy, and outputs the result to standard output. Table 1 shows the number of referenced calls to different kernels. As we can see, the DaphneDSL language syntax for our example is relatively simple and similar to already existing programming languages, such as C++ and R.

5 Loops in the DaphneDLS Programming Language

The loops in DaphneDSL have a simple syntax, and allow the code inside the block to be executed multiple times. This allows the programs to be shorter and expressively more powerful. In this paper, we showed two examples of the use of loops. In the first, we calculated the sum of natural numbers, and, in the second, the energy of the sequences. The runtime environment returned the correct results, and the program was fast. With the code base before the issue was fixed, it was discovered that the execution of the program ends prematurely in the case of loops with the larger number of iterations. In the case of the sum of natural numbers, we managed to summarize the natural numbers up to the number 174,355, and in the case of sequences, we calculated the energies for all lengths up to $L = 192$.

To overcome the described limitation with a temporary workaround before the issue was fixed in the code, we found it necessary to increase the stack size used by the program. When we increased the size of the stack (see Code 5) to an unlimited size, we were able to achieve results, even in the case of loops containing many more iterations. For example, we evaluated sequences up to $L = 1000$ successfully, and calculated the sum of natural numbers up to 10,000,000. In all cases, the program ended successfully.

We also checked the operation of the MLIR framework. For this purpose, we implemented the calculation of the sum of natural numbers, as shown in the Codes 6 and 7. The MLIR code is shown in Code 6. Code 7 shows the C++ code that implements the function for printing results, which is used in the MLIR code. The compilation of both files into the executable program is shown in Code 8: we first translated the MLIR code into a lower level of code, and finally translated it, together with the C++ code, into the executable program. This example also shows how to combine compilers that support the MLIR framework. The executable program worked properly and had no issues with premature termination. Even in the case of a standard stack size, the program allowed loops with a larger number of iterations. For example, we could calculate the sum of natural numbers up to 10,000,000. As the issue occurs in programming loops with many iterations within DaphneDSL as mentioned and we applied a temporary workaround, we were also in contact with developers who quickly fixed the issue in the system code by closing the issue #77 with the Pull Request #820, as provided at `https://github.com/daphne-eu/daphne/issues/77`, merging the fix to the DAPHNE main branch (commit b294230a63e84861a0426d6d9958099baedd9bec on Sep 5, 2024). This also demonstrates how the DAPHNE code, published as open source, is becoming more and more reliable and valuable over time.

6 Conclusion

In this paper, we demonstrated two examples of using DaphneDSL domain-specific programming language and the DAPHNE system runtime environment, specifically for loops. The DaphneDSL and DAPHNE system enable management and processing of data using pipelines, efficient use of machine learning, and deployment on supercomputers, which was an additional motivation to demonstrate these examples. The first example calculated the sum of natural numbers, and the second calculated the energy of binary sequences. In both examples, the executable program provided the correct results.

We have also found an interesting listed issue that appeared in the case of loops with a larger number of iterations. We created a workaround for the issue by temporarily increasing the stack size which the operating system allocates to the process. With this workaround, the program worked flawlessly, even in the case of loops with a vast number of iterations. When writing this article, the developers have fixed the described stack problem of the programming loops successfully. The bug

Table 1: Statistics of the kernel calls.

	#calls	kernel
1	1	_cast__char__int64_t
2	1	_cast__char__uint64_t
3	2	_cast__int64_t__DenseMatrix_int64_t
4	3	_cast__int64_t__size_t
5	10	_cast__int64_t__uint64_t
6	7	_cast__size_t__int64_t
7	5	_cast__uint64_t__int64_t
8	3	_concat__char__char__char
9	2	_createDaphneContext__DaphneContext__uint64_t__uint64_t
10	14	_decRef__Structure
11	2	_destroyDaphneContext
12	6	_ewAdd__int64_t__int64_t__int64_t
13	3	_ewAdd__uint64_t__uint64_t__uint64_t
14	2	_ewGe__int64_t__int64_t__int64_t
15	2	_ewMul__DenseMatrix_int64_t__DenseMatrix_int64_t__DenseMatrix_int64_t
16	1	_ewMul__DenseMatrix_int64_t__DenseMatrix_int64_t__int64_t
17	12	_ewMul__int64_t__int64_t__int64_t
18	1	_ewSub__DenseMatrix_int64_t__DenseMatrix_int64_t__int64_t
19	3	_ewSub__uint64_t__uint64_t__uint64_t
20	1	_incRef__Structure
21	1	_print__char__bool__bool
22	1	_sample__DenseMatrix_int64_t__int64_t__size_t__bool__int64_t
23	8	_sliceRow__DenseMatrix_int64_t__DenseMatrix_int64_t__int64_t__int64_t

Code 4: A program for calculating energies of binary sequences.

```
def calcE(s:matrix<si64>,L:ui64) ->
ui64 {
   E=as.ui64(0);
   for(k in 1:L - 1) {
      ck=as.si64(0);
      for(i in 0:L - k - 1) {
         ck=ck+as.si64(
         as.scalar(s[i,]*s[i+k,]));
      }
   tmp = as.ui64(ck*ck);
   E = E + tmp;
   }
   return E;
}

for (L in 5:301) {
   s = sample(2,L,true,-1);
   s = 2*s - 1;
   E = calcE(s,as.ui64(L));
   print("L="+L+" E="+E);
}
```

has been fixed on the DAPHNE main branch (commit b294230a63e84861a0426d6d9958099baedd9bec on Sep 5, 2024). This demonstrates how the DAPHNE code, published as open source, is becoming more and more reliable and valuable over time.

We also showed an example of the intermediate code DaphneIR (for the MLIR framework). It is translated into lower-level code and, ultimately, into an executable program. To test whether the stack size problem is related to the intermediate code, we implemented the sum of natural numbers by using the MLIR code. Here, we also showed an example of how to connect different programming languages. The resulting executable did not have a stack size problem.

Future work includes developing further exam-

Code 5: Instruction code for resizing the stack.

```
$ ulimit -s unlimited
```

Code 6: MLIR code for calculating the sum of natural numbers.

```
module {
func.func @main() -> i32 {

%0 =arith.constant 0:index
%10=arith.constant 10000001:index
%c0=arith.constant 0:i64
%c1=arith.constant 1:index
%ret=arith.constant 0:i32

%result = scf.for %i = %0 to %10 step
   %c1 iter_args(%sum = %c0) -> (i64) {
   %i_i64=arith.index_cast %i:index to i64
   %sum_next=arith.addi %sum,%i_i64:i64
   scf.yield %sum_next:i64
}

call @print_i64(%result) :  (i64) -> ()
   return %ret :  i32
}

func.func private @print_i64(%val:
   i64)->()
}
```

Code 7: C++ code that implements the print function.

```
#include <iostream>
extern "C" int main();
extern "C" void print_i64(int64_t val) {
   std::cout<<"Result is:"
            <<val<<std::endl;
}
```

Code 8: Compiling the MLIR code.

```
$ mlir-opt sum.mlir
--pass-pipeline="builtin.module(
func.func(convert-scf-to-cf,
convert-arith-to-llvm),
convert-func-to-llvm,convert-cf-to-llvm,
reconcile-unrealized-casts)" >
sum_opt.mlir

$ mlir-translate --mlir-to-llvmir
sum_opt.mlir > sum.ll

$ clang++ -o sum sum.cpp sum.ll
```

ples and use of DAPHNE system and DaphneDSL, also including loops and other control structures, and benchmarking with the provided examples.

Acknowledgment

The DAPHNE project is funded by the European Union's Horizon 2020 research and innovation program under grant agreement number 957407 from 12/2020 through 11/2024.

The authors acknowledge the financial support from the Slovenian Research and Innovation Agency (Research Core Funding No. P2-0041 – Computer Systems, Methodologies, and Intelligent Services).

References

[1] The DAPHNE Project. https://daphne-eu.eu/project/. Accessed: 2024-07-15.

[2] Patrick Damme, Marius Birkenbach, Constantinos Bitsakos, Matthias Boehm, Philippe Bonnet, Florina Ciorba, Mark Dokter, Pawl Dowgiallo, Ahmed Eleliemy, Christian Faerber, et al. Daphne: An open and extensible system infrastructure for integrated data analysis pipelines. In *Conference on Innovative Data Systems Research*, 2022.

[3] Aristotelis Vontzalidis, Stratos Psomadakis, Constantinos Bitsakos, Mark Dokter, Kevin Innerebner, Patrick Damme, Matthias Boehm, Florina Ciorba, Ahmed Eleliemy, Vasileios Karakostas, et al. DAPHNE Runtime: Harnessing Parallelism for Integrated Data Analysis Pipelines. In *European Conference on Parallel Processing*, pages 242–246. Springer, 2023.

[4] Aleš Zamuda and Mark Dokter. Deploying DAPHNE Computational Intelligence on EuroHPC Vega for Benchmarking Randomised Optimisation Algorithms. In *International Conference on Broadband Communications for Next Generation Networks and Multimedia Applications (CoBCom)*, pages 1–8, 2024.

[5] Matthias Boehm, Iulian Antonov, Sebastian Baunsgaard, Mark Dokter, Robert Ginthör, Kevin Innerebner, Florijan Klezin, Stefanie Lindstaedt, Arnab Phani, Benjamin Rath, Berthold Reinwald, Shafaq Siddiqi, and Sebastian Benjamin Wrede. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, Netherlands, January 12-15, 2020*, pages 1–8, 2020.

[6] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021.

[7] D4.2: DSL Runtime Prototype. https://daphne-eu.eu/wp-content/uploads/2022/12/D4.2-DSL-Runtime-Prototype.pdf/, 2022. Accessed: 2024-07-15.

[8] D3.1 Language Design Specification. https://daphne-eu.eu/wp-content/uploads/2022/06/DAPHNE_D3.1_LanguageDesign_v1.2.pdf/, 2022. Accessed: 2024-07-15.

[9] Another tool for language recognition. https://www.antlr.org/. Accessed: 2024-07-15.

[10] Borko Bošković, Franc Brglez, and Janez Brest. Low-autocorrelation binary sequences: On improved merit factors and runtime predictions to achieve them. *Applied Soft Computing*, 56:262–285, 2017.

[11] Janez Brest and Borko Bošković. A heuristic algorithm for a low autocorrelation binary sequence problem with odd length and high merit factor. *IEEE Access*, 6:4127–4134, 2018.

[12] Borko Bošković, Jana Herzog, and Janez Brest. Parallel self-avoiding walks for a low-autocorrelation binary sequences problem. *Journal of Computational Science*, 77:102260, 2024.