Practical data-centric optimization of C/C++ applications

Bachelor Thesis

Author(s):
Lepori, Andrea

Publication date:
2022-07-15

Permanent link:
https://doi.org/10.3929/ethz-b-000571709

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Practical data-centric optimization of C/C++ applications

Bachelor Thesis

Andrea Lepori

Friday 15th July, 2022

Dr. Alexandru Calotoiu, Prof. Dr. Torsten Hoefler
Department of Computer Science, ETH Zürich

Abstract

C is a programming language used everywhere and in a big part of scientific codes. With the increase of highly parallel computing machines the code needs to be adapted to be able to use the full power available. In my thesis I will expand a compiler frontend to be able to transform C code into a data centric representation that will intern be able to identify data dependencies automatically and hence produce highly parallel code. There were a lot of challenges because pointers and structs are hard to adapt to a data centric paradigm. In the end I was able to find ways to work around the imposed limitations and expand the compiler front end to support a bigger - and more general - C language subset. On the tested benchmarks the performance of the automatic parallelization was on par with the handmade one.

Acknowledgements

I would like to thank my supervisors Dr. Alexandru Calotoiu and Prof. Dr. Torsten Hoefler for the feedback and the help provided every time I didn't know how to continue. I also thank the whole DaCe/DAPP team for the very quick fixes done to DaCe every time I got stuck on an exotic bug of the framework. In particular Dr. Tal Ben-Nun for the responsiveness to all my doubt and questions about DaCe when I was trying to push it to its limits.

Finally I would also like to thank my parents for all the support in my studies and in my life.

Contents

Chapter 1

Introduction

More that a quarter of users of the Swiss National Supercomputing Centre use C as their primary language and more than half use C++[1]. This shows that C/C++ is often used in scientific parallel computation. Those languages permit a very powerful and low level control on the hardware that allows great flexibility. To be able to use all the available computational speed that the hardware provides it is imperative to have some knowledge about it. In the past most hardware was created for general purpose computation but nowadays more manufactures are creating specific hardware to speed up specific computations [2, 3]. This creates the problem that users have to learn the new hardware to speed up their computations.

In modern supercomputers the computational power grows two times faster than the memory bandwidth [4]. This means that we have to think about how we access data to not incur in bottlenecks. Unfortunately it requires specific knowledge about the memory structure to be able to optimize data movements.

Another problem is that the memory access flexibility that C provides is a double edge sword. On one hand the programmer is able to specify with great control the access patterns wanted hence - with good knowledge about the architecture - he's able to greatly optimize the programs. On the other hand that flexibility could obfuscate the data movement making automatic optimization of a compiler harder to do and less impactful.

C2DaCe [5] aims to fix those problems by translating C code into a Stateful DataFlow multiGraph (SDFG) that is a data centric intermediate representation. SDFGs explicitly show all data movements and computations done. Thanks to the DaCe [6] framework and the information contained in the SDFG we are able to create an optimized program that is able to use the full potential provided by the hardware.

Because the SDFG is hardware agnostic is can be compiled into any archi-

```
char *c[] = {"CARGO", "TUNA", "NEVER", "NEON"};
char **cp[] = { c+3, c+2, c+1, c };
char ***cpp = cp;
printf("%s", ****cpp);
printf("%s", *--***cpp+3);
printf("%s", *cpp[-2]+3);
printf("%s\n", cpp[-1][-1]+2);
```

Figure 1.1: Example C code where data movement and data accesses are not easy to identify

tecture including CUDA for NVIDIA GPUs and VHDL for FPGAs. In the future with new architectures it can be also expanded to support them.

C2DaCe was already tested on the Polybench [7] and LULESH [8] benchmarks and showed that is able to automatically optimize C code, obtaining even better performance than hand optimized code. Unfortunately C2DaCe still has some limitations with more complex code. In my thesis I'll remove some of those limitations and point out some possible solutions for more of them. In this way I was able the enlarge the subset of C programs that C2DaCe is able to handle.

The most important limitations that I've been working on where

- Handling more complex struct usage. Including supporting pointers to struct arrays and fields as array pointers
- Extending the support for 1D and 2D pointers to arrays initialized with `malloc`
- Adding support for writing to references of values passed to different functions
- Handling simple pointer assignments using static aliasing at compile time
- Creating a proof of concept implementation of array pointers arithmetic where the beginning of the array is moved by editing the pointer

Another contribution I've worked on is the identification of the most important still present limitations and the proposal of possible solutions. Because those limitations are often incurred on I've also proposed valid ways to workaround them to produce working code.

In the following chapters I will first explain how DaCe and C2DaCe work. I'll then explain in details the current limitations of C2DaCe and how I was able to remove them. Afterwards I'll explain the limitations that are still present explaining why they are hard to remove and possible way to approach the problem. I'll also propose possible manual workarounds to be able to run the code for the time being. In the end I'll explain the two benchmarks that I

was able to run (HPCCG [9] and FTT from the BOTS suite [10]) and show the successful results that were on par with the parallel implementation of the benchmarks.

Chapter 2

State of the art

2.1 DaCe

DaCe [6] is a parallel programming framework that uses Stateful DataFlow multiGraph (SDFG) to represent and manipulate code. In the SDFG the code is represented based on the data movement and not the control flow. Thanks to this representation we can easily identify data dependencies. Using the knowledge of those data dependencies we can automatically parallelize code.

2.1.1 Storing and moving data

The main components of the SDFG are data container (or array containers) that are the objects containing the data. The data containers are defined with a size and a type like in other languages. Data is moved connecting different components with memlets that are what defines which data gets moved and where. Memlets will also indicate the source and destination subset for the data movement. Thanks to memlets data movements are explicitly defined and with that information we can derivate the data dependencies between computations.

A conflict resolution (CR) memlet is a special kind of memlet that enables concurrent write to a data container. We can specify a two arguments function that will be applied to resolve the write conflict between two threads. For example if we specify the function $\lambda x.\lambda y.x + y$ then when two threads try to "execute" the memlet at the same time their values will be summed together and saved in the destination data container.

2.1.2 Computing values

Tasklets are components that define fine-grained computations. Inside a tasklet we define the computation using a C++ or Python statement. We can

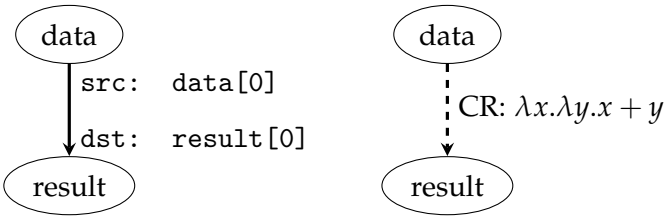


Figure 2.1: Left: simple memlet with the subset specified. Right: memlet with conflict resolution write

then define input and output connectors on the tasklet where we can connect memlets to move the necessary data.

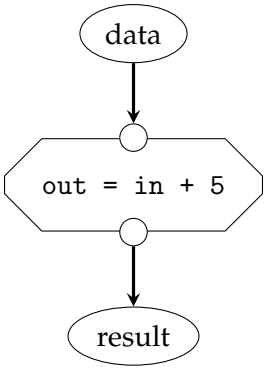


Figure 2.2: Read and increment of a value from a data container to another. The top ellipse is the source data container, the bottom ellipse is the destination, the octahedron in the middle is a tasklet and the arrows are the memlets.

To parallelize computations we can use maps where we apply a computation on a data container range one element at the time. At the end side of the map constructs all threads could write to the same data container element using a memlet with CR or every thread can write to a different element. This construct is completely parallelizable because maps only read independent elements and in case of a write to the same data container a CR memlet is used.

2.1.3 Control-flow and other constructs

The SDFG is composed by multiple states that are connected with state edges which indicate the order of the execution. A state can have multiple inbound and outbound edges on those edges we can have conditions based on the data. Inside states there are the constructs talked about in the previous subsections such as data containers, memlets, tasklets and map/reduce operations.

On state edges we can also define and update symbols. Symbols are another

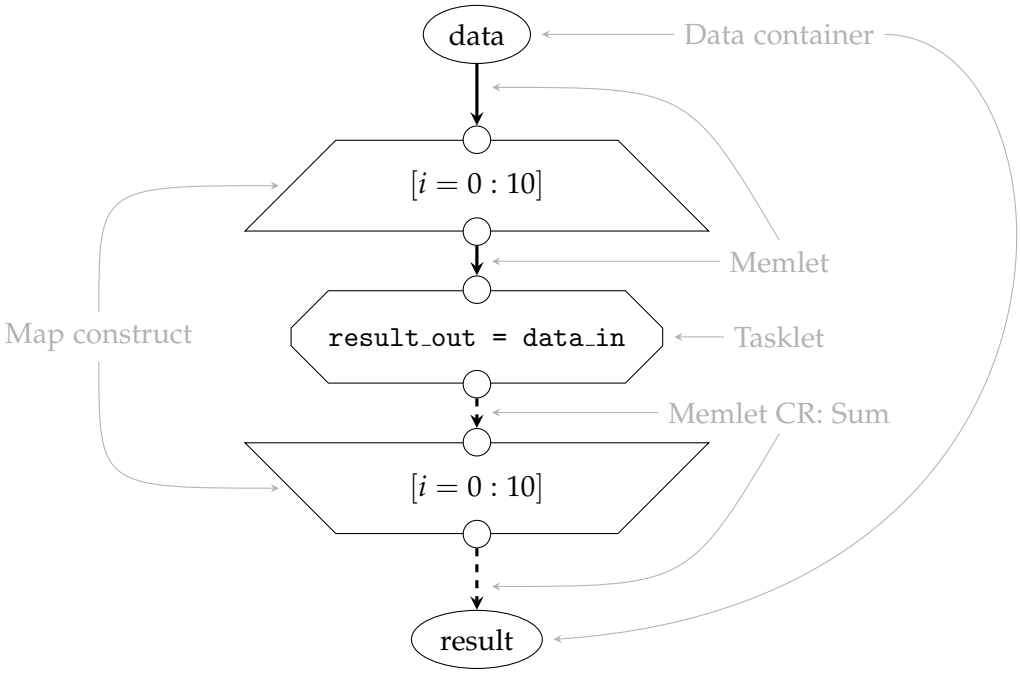


Figure 2.3: Example of SDFG representing the sum of elements of an array

way to store data but they can only do that for single values and not arrays. Symbols can be used to define symbolic arrays whose size depend on symbols and for conditional state edges that depend on a symbol value.

SDFG can also be nested into each other. A nested SDFG will create a new scope hence all previously defined data containers or symbol will not be accessible. Symbols can be mapped entering the nested SDFG using a map. Note that it is only when entering hence every edit to the symbol from the nested SDFG will not be propagated to the parent SDFG. For data containers memlets are used to indicate the data movement. This means that we can also map only a subset of the parent data container to the new data container in the nested SDFG.

2.1.4 Manipulating SDFG and compiling them

DaCe contains multiple transformations that improve the performance of the SDFG. For example it can remove unneeded states, transform a loop (states with backwards edges) directly into a map that can be parallelized, inline nested SDFGs and many other simplifications.

DaCe has a native Python frontend and API. Code can be written in Python and automatically transformed into an SDFG or it can be manually constructed using the API. DaCe also implements the backend that can compile

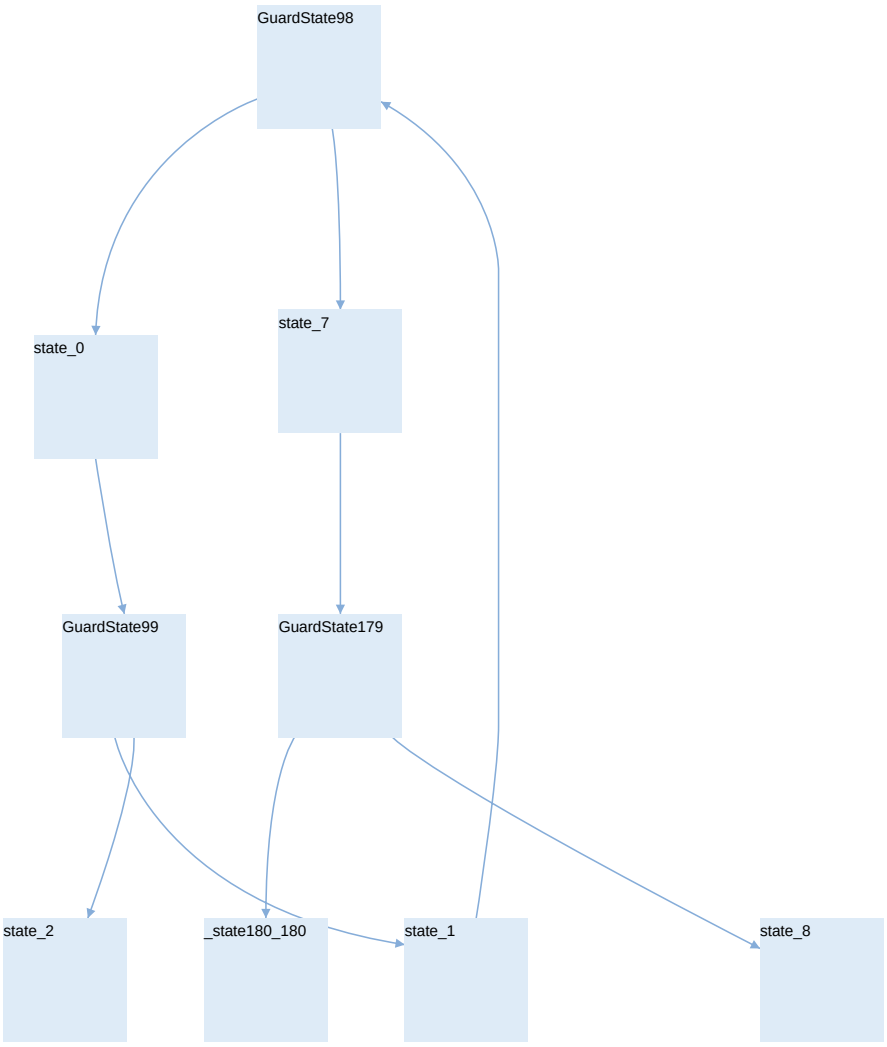


Figure 2.4: States with conditional branches and loops

the SDFG into optimized parallel code. At the time of writing it can target CPUs, GPUs and FPGAs.

2.2 C2DaCe

C2DaCe is a frontend that compiles C to an SDFG that is then optimized and compiled by the DaCe backend. This enables automatic optimization and parallelization of C code.

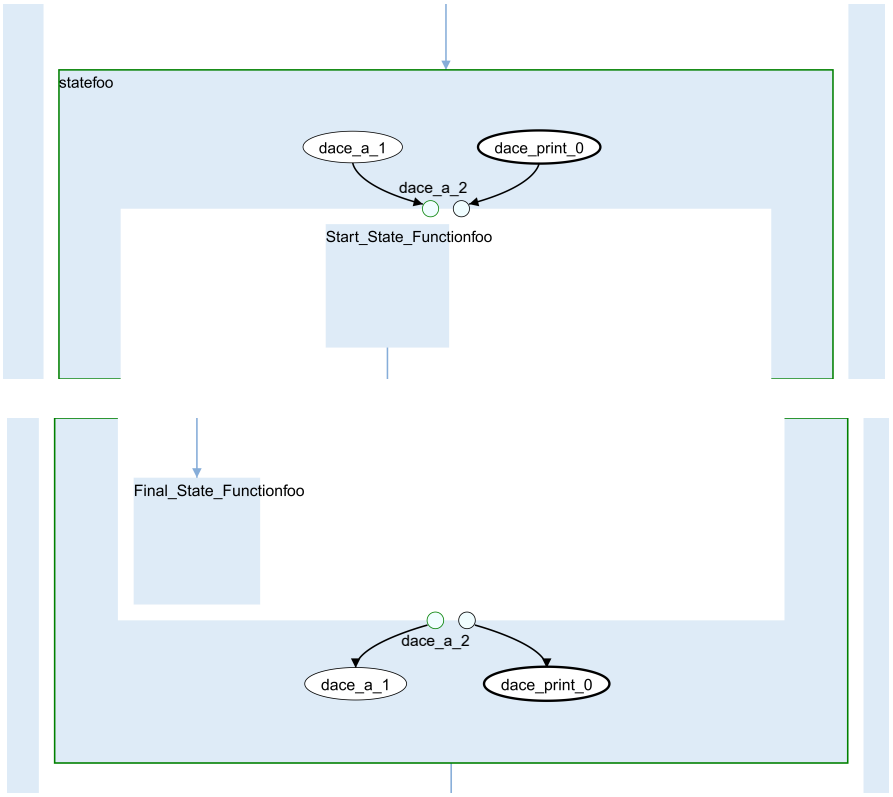


Figure 2.5: Nested SDFG

2.2.1 Ingesting C code and transforming it

The first step is done by LibClang [11] that takes the C source file and creates an abstract syntax tree (AST)[1]. This AST directly represents the C code this means that we still have access to all the feature of the C language. On this AST we apply transformations that make it simpler and easier to transform it into an SDFG.

An example is the transformation from figure 2.7 that is done to convert all `calloc` calls to `malloc`. In this way when we are working with the SDFG we only have to check for malloc calls for the creation of new arrays.

Another transformation done is the index extractor that creates a temporary variable for every array index used. In this way we don't have to handle compound indices and we are also able to further optimize the SDFG. By having a single variable as the index it is easier for DaCe to identify parallelization

[1]We don't actually directly work with the AST of LibClang because it is not well documented and not easy to modify. Instead after LibClang creates its own AST we transform it into a custom AST that is implemented directly in C2DaCe. In this way we have full control of the AST structure.

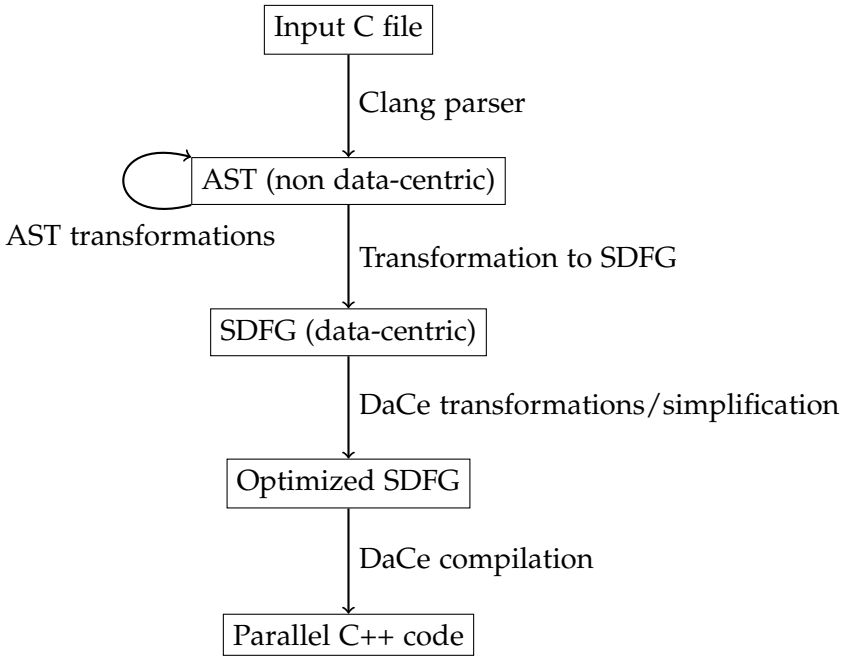


Figure 2.6: C2DaCe workflow

```
double* a = calloc(5, sizeof(
    double));
int N = 35;
double** b = calloc(N, sizeof
    (double));
```

→

```
double* a = malloc(5 * sizeof
    (double));
int N = 35;
double** b = malloc(N *
    sizeof(double));
```

Figure 2.7: Example of transformation from `calloc` to `malloc`

opportunities and converting loops to maps. An example can be found in figure 2.8.

2.2.2 From the AST to the SDFG

Every variable will be mapped to a data container. Because we don't have scopes in the SDFG we could have the problem that two variables with the same name are stored inside the same data container. To avoid this problem we create a mapping between the C variable name and the data container assigned to it. The current parent SDFG is also taken into account because inside nested SDFG we have different data containers. We cannot use nested SDFGs as scopes because they could be inlined to optimize the SDFG.

C statements are translated to some construct in the SDFG that is semantically equivalent. This cannot be done for any C statements because there isn't always a one-to-one translation possible. This is the main challenge when

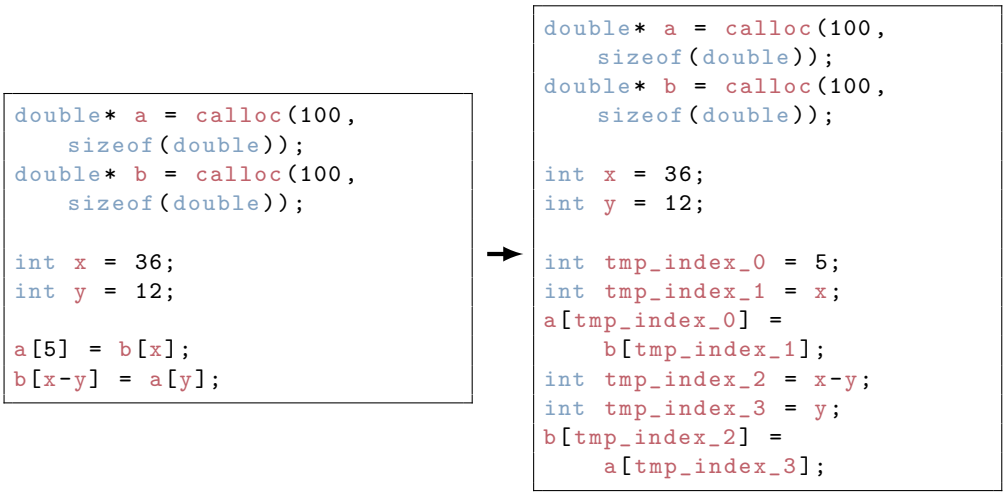


Figure 2.8: Example of index extraction transformation

translating and will talk about the problematic C statements in the following chapters.

For example the statement `result[0] = data[0] + 5` will be translated into figure 2.2. More details on how different C statements are translated can be found in the original C2DaCe [5] paper.

2.2.3 Optimizing the SDFG

The SDFG generated will be very linear in the sense that it will mostly follow the control flow of the input C program. Instead we would like an SDFG that depends mostly on the data flow, this can be done by applying transformations from DaCe.

One very important transformation done in the beginning is the promotion of scalars to symbols. This is important because symbols are treated differently than data containers. For example the size of a data container can depend on a symbol but not from another data container. In subsequent transformations we can also apply constant propagation on symbols to optimize them away if possible.

2.2.4 Limitations of C2DaCe

Because we are not able to translate any C statement one-to-one only a subset of the C language is supported and most codes are outside this subset. Some limitations come from the fact that the C language is very large and some features were not encountered yet but mostly they come from the data centric paradigm that we are using.

In particular C2DaCe has problems handling structured data (structs) with indirection. In the SDFG structured data cannot have arrays in it. This means that we cannot convert C structs directly to SDFG structs. Another big limitation are pointers (or references to data). We have pointers in the SDFG but they are not safe. We are not able to analyze the data dependencies of references this means that we can possibly incur in data races that would produce incorrect behavior.

Chapter 3

Structure elimination

Structs are widely used in C to represent complex structured data. In scientific code are often used to represent mathematical structures like sparse or dense matrices.

In the SDFG we are able to create structs but we cannot have arrays in it. Instead of arrays we can use pointers but then we lose information about data dependencies causing worst performance or even incorrect execution (more on this on the following chapter 4). To solve this problem C2DaCe unpacks the fields of structs in single variables. In this way we obtain code that is as if structs where never used. Thanks to this we can handle structs fields exactly the same as any other variables. This was already partially supported in C2DaCe but it was only limited to simple struct declarations without pointers. The addition now supports struct pointers and fields defined as pointers or double pointers. Also struct arrays can now be allocated using `malloc`.

From the example in figure 3.2 you can see that a level of indirection is added on variables that were in the struct. This is due to the fact that the struct was defined as a struct pointer. In this way we keep the same level of indirection as when we had structs and we still have correct code. After this whenever we count the level of indirection, for example to find the dimensionality of an array defined as a pointer, we need to check if it was a struct. If this was the case then we need to remove from the count the indirection added by the struct.

On functions call we unpack the struct in multiple arguments. When doing this the type signature of the function must also be modified. Also here we need to set the right level of indirection for the variables.

In the example of figure 3.1 we set two arrays inside of a struct independently. With our implementation the write can be done with a map and be parallelized. Using instead the native SDFG structs we have to define the

two arrays as pointers hence we cannot use a map on them because we don't know their data dependencies.

```
struct example_struct {
  double* x;
  double* y;
}

typedef struct example_struct example;

int N = pow(2, 20);
example* ex = malloc(sizeof(example));
ex->x = malloc(N * sizeof(double));
ex->y = malloc(N * sizeof(double));

for (int i=0; i<N; i++) {
  ex->x[i] = i+2;
  ex->y[i] = i*2;
}
```

Figure 3.1: Example of computation to compare different struct representations

We run the example on a 8 thread CPU for 4 times (excluding a first run to warm up the cache) an took the arithmetic mean of the running time. For the native version we don't have any parallelization and the mean runtime was 4.8 seconds. Instead on the parallelized version generated by C2DaCe the run time was 30 milliseconds.

```c
struct example_struct {
  int size;
  double* data;
};

typedef struct example_struct example;

void foo(example* ex) {
  ex->size = 3;
}

void bar(example* ex) {
  for (int i=0; i<ex->size; i++) {
    ex->data[i] = i;
  }
}

int main(int argc, char** argv) {

  example* ex = malloc(sizeof(example));

  // init variables
  ex->data = malloc(sizeof(double) * 10);

  foo(ex);
  bar(ex);

  return ex->data[2];
}
```

```c
void foo(double** c2d_struct_example_data, int*
    c2d_struct_example_size) {
  *c2d_struct_example_size = 5;
}

void bar(double** c2d_struct_example_data, int*
    c2d_struct_example_size) {
  for (int i=0; i<*c2d_struct_example_size; i++) {
    c2d_struct_example_data[i] = i;
  }
}

double** c2d_struct_example_data = malloc(sizeof(double*));
int* c2d_struct_example_size = malloc(sizeof(int));

// init variables
*c2d_struct_example_data = malloc(sizeof(double) * 10);

foo(c2d_struct_example_data, c2d_struct_example_size);
bar(c2d_struct_example_data, c2d_struct_example_size);

return c2d_struct_example_data[2];
```


Figure 3.2: Transforming structs from original C code to transformed code

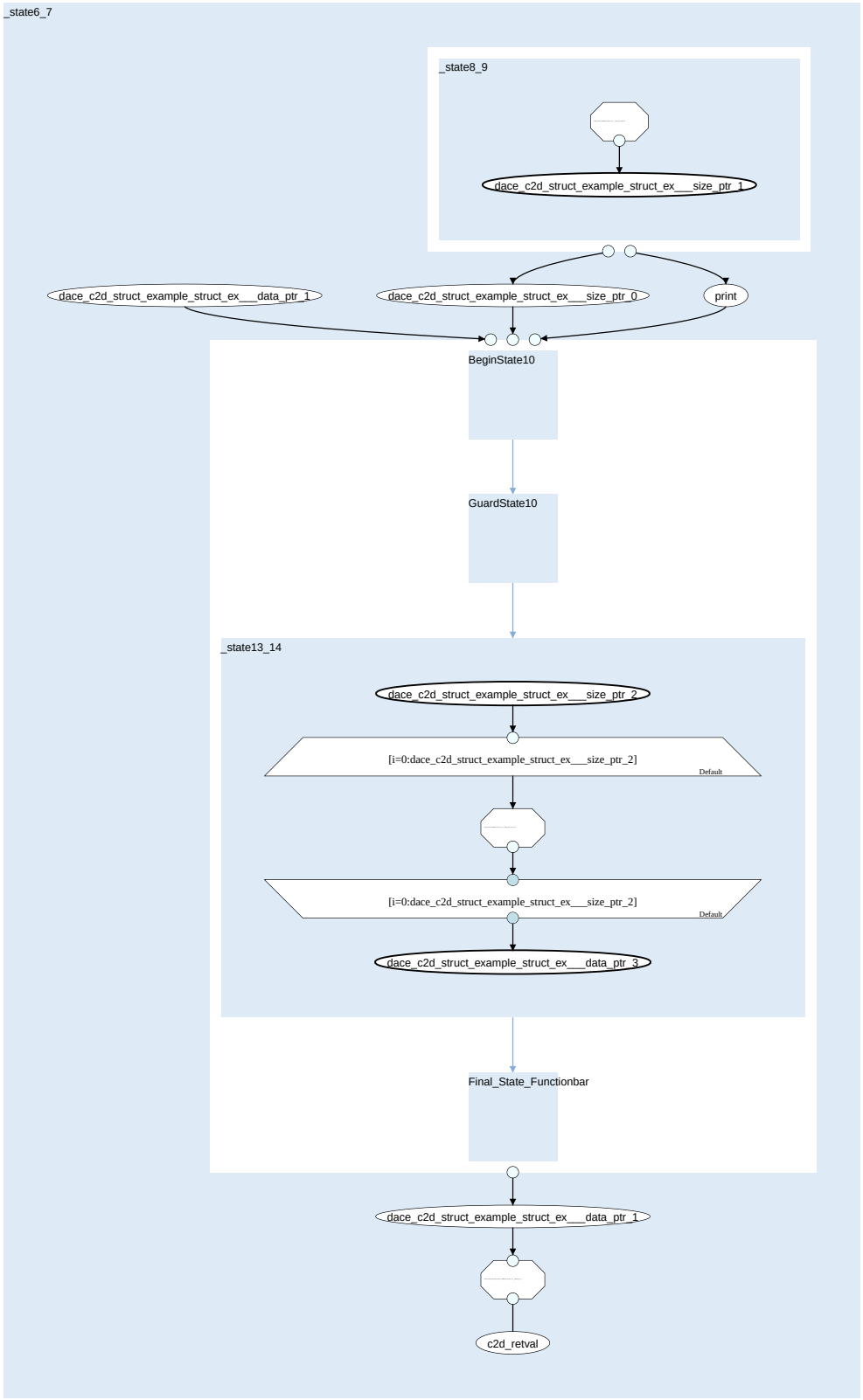


Figure 3.3: Resulting SDFG of C code containing structs from figure 3.2

Chapter 4

Pointers handling

Pointers are really powerful and are often used in C for complex data structures and code. When using pointers in the SDFG we cannot analyze the data dependencies because it is not clear at which data it is pointing to. When applying transformations this could produce incorrect results or not be able to optimize the SDFG successfully because of the missing information about the data movement. It is not possible in a data centric view to handle indirection because it goes against the main idea of being able to precisely track data movement. This means that the implementation must be done at compile time. In the following paragraphs I will explain how some simple static cases work.

One easy case to implement is when we have a pointer that is used for passing around values by reference and not by copy.

```
int main(int argc, char** argv) {
  double* value = malloc(sizeof(double));
  setI(value);

  return value[0];
}

void setI(double* k) {
  k[0] = 5.0;
}
```

Figure 4.1: Example of pointer used to pass value by reference

When pointers are used for passing references of data (e.g. figure 4.1) we can directly substitute them with their data container. Because in the SDFG variables that are mapped to data containers are always passed by reference. This is because when we access the variable we read or write to the same

data container and not a copy of it.

The optimized SDFG obtained (figure 4.2) will directly inline the function and set the value of the data container. Then if we compile the SDFG with DaCe we obtain a piece of simple C++ code (figure 4.3) that encapsulate this data movement.

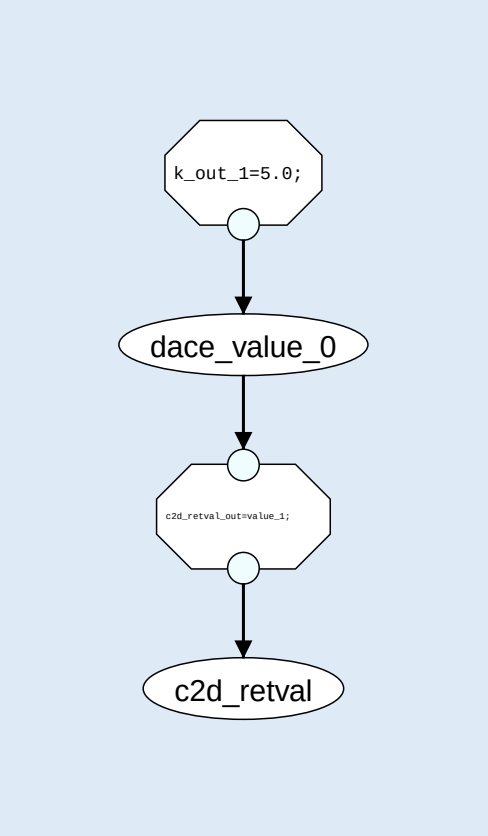


Figure 4.2: SDFG generated from code in figure 4.1

For simple pointer assignments we can manage that statically during the compilation. In the example of figure 4.4 after the marked line the compiler will substitute every occurrence of `value2` with `value`.

Every other more complex case isn't supported. This includes pointer assignments in a conditional statement. To support cases like that we would need to do a numerical analysis to know statically which branch will be taken. Even then there will be branches that cannot be determined statically, for example if they depend from user input.

From figure 4.5 you can see that if we had indirection we must be able to han-

```
double dace_value_0[1]  DACE_ALIGN(64);

{
  int c2d_retval;

  {
    double k_out_1;

    ////////////////////
    k_out_1=5.0;
    ////////////////////

    dace_value_0[0] = k_out_1;
  }
  {
    double value_1 = dace_value_0[0];
    int c2d_retval_out;

    ////////////////////
    c2d_retval_out=value_1;
    ////////////////////

    c2d_retval = c2d_retval_out;
  }

}
```

Figure 4.3: Optimized code generated from the SDFG in figure 4.2

```
int main(int argc, char** argv) {
  double* value = malloc(3 * sizeof(double));
  value[0] = 37;

  double* value2 = value; // aliasing
  value2[1] = 12;

  return value[1];
}
```

Figure 4.4: Simple pointer assignment that can be handled statically

dle references to values. This is needed for modifying the original container and expecting the reference to change. For a data centric representation we have to know exactly which data we are accessing to know the data races.

For statically determinable references it could in theory be done statically but it will require a very complex numerical analysis and pointer analysis. But for runtime determined references we have to over approximate the set of possible referenced data containers. In most cases the source C code can

```c
int main(int argc, char** argv) {
  double** ptr_a = malloc(sizeof(double*));
  double** ptr_b = malloc(sizeof(double*));
  *ptr_a = malloc(sizeof(double*));
  *ptr_b = malloc(sizeof(double*));

  **ptr_a = 5;
  **ptr_b = 10;

  if (argc == 1) {
    *ptr_a = *ptr_b;
  } else {
    *ptr_b = *ptr_a;
  }

  **ptr_a = 15;

  return **ptr_b;
}
```

Figure 4.5: Pointer assignment dependent on user input

be refactored to remove this kind of indirection. This is much easier to a numerical/pointer analysis.

The only case of higher level indirection that is now supported is for multidimensional arrays created with `malloc`. Even then only 2D arrays are supported. For this to work in the SDFG we have to know the size of the array before using it. This is needed because to create the data container we have to specify the size of it before using.

```c
void foo(double** x) {
  for (int i=0; i<10; i++) {
    x[i] = malloc(3 * sizeof(double));
    x[i][0] = 1;
    x[i][1] = 13;
    x[i][2] = 45;
  }
}

double **data = malloc(10 * sizeof(double*));
// data is an incomplete 2D array of size [10, ?]
double data[0] = malloc(3 * sizeof(double)); // marked
// data is a complete 2D array of size [10, 3]
// data can now be used
foo(data);
```

Figure 4.6: Creation of a 2D array using `malloc`

The marked line in figure 4.6 is needed for the compiler to know the size to allocate for the data container. Even if the data container is not written to or read from it is passed to a nested SDFG. Because we need to specify the data movement (with a memlet) from the parent to the nested SDFG we have to invoke the data container. To do so we need to know the size of it.

C2DaCe creates nested SDFGs for function calls and for loops. Because of this the size of the data container must be known before any of those occurrences. The first `malloc` encountered will be considered for setting the size of the data container. This means that it shouldn't be inside a branch. This limitations come from the fact that we have to know statically the size of the data container and because we are not doing a numerical analysis we cannot resolve branches statically.

Another limitation is the handling of moving the start pointer of an array.

```
void foo(double* a) {
  a[1] = a[0] + a[1]
}

double size = 3;
double* data = malloc(size * sizeof(double));

foo(data);
foo(data+1);

data++;
double result = data[0] + data[1];
```

Figure 4.7: Example of moving the start pointer of an array

This could be supported by adding an offset variable for every array that is modified at runtime instead of the pointer. Because this is a very common pattern in C adding this transformation will greatly expand the set of supported C programs.

The transformation is partially implemented but only support simple cases. Mainly it cannot pass an array to a function call with its offset. Because the implementation is not complete and was not tested thoroughly it could cause incorrect behaviors. By this reason and because in the tested HPCCG [9] benchmark this pattern was not present the transformation is not currently applied.

```
void foo(double* a, int* a_offset) {
  a[*a_offset + 0] = a[*a_offset + 0] + a[*a_offset + 1]
}

double size = 3;
double* data = malloc(size * sizeof(double));
int* data_offset = malloc(sizeof(int));
*data_offset = 0;

foo(data, data_offset);
*data_offset += 1;
foo(data, data_offset);
*data_offset -= 1;
*data_offset += 1;
double result = data[data_offset + 0] + data[data_offset + 1];
```

Figure 4.8: Transformed code from figure 4.7. The start pointer of the array is moved using an additional variable.

Chapter 5

Additional canonical transformations

C2DaCe only supports a subset of the C language. This means that we expect any general C code to not be able to run directly. All input files should be transformed before to constrain them into this supported subset. C2DaCe supports simple C code and really simple C++ constructs. To improve the success of the compilation code should be transformed from C++ to C.

Some transformations are done automatically by C2DaCe as AST transformations. Unfortunately some still has to be done by the user manually. In most cases the the limitations could be removed by adding some additional AST transformations. But because they are mostly complex one they need to be fully tested to have a fully safe transformation. Some could also impact the performance hence they need some considerations when applied.

Even if some transformations need to be done by hand it is still faster than rewriting the whole code into a parallelized version. Most hand made transformations are really quick to do and don't require much thinking to be applied.

The following section can be viewed as a showcase of the currently still present limitations with insight on how to work around them for the time being. It will also explains the motivation on why it is not trivial to remove them and possible ways to approach the problems in the future.

5.1 Arrays

Depending on how the array is declared it is handled differently. An array in C can be declared in the stack (e.g. `double a[5]`) or in the heap (e.g. `double *a = malloc(5 * sizeof(double))`). Each case is handled by different functions in C2DaCe because it also has different behavior in C. I mainly focused on the `malloc` implementation hence this section will be mainly about it.

`malloc` declared arrays can be of 1 or 2 dimensions only (other array sizes are not supported yet). Others sizes are not trivially supported because of the way we impose the size of the arrays. We derive the number of dimensions of the array based on the number of indirections on the type. For example an array of type `double*` will be 1 dimensional and one of type `double**` will be 2 dimensional[1]. For 1 dimensional arrays we immediately know the size because of the `malloc` arguments. For 2 dimensional arrays instead we only know the size of the first dimension. For the second dimension we look for the first `malloc` on an element of the array. Until the first `malloc` we cannot use the array. This is because C2DaCe has to know the size before using the data container.

```
int N = 10;
double *a = malloc(N * sizeof(double));
double **b = malloc(N * sizeof(double*));
// cannot use b here
b[0] = malloc(13 * sizeof(double));
// now we can use b
```

Figure 5.1: Allocation of an array using `malloc`. Code annotated to show where the array can be used.

Usually in C the second call to `malloc` from figure 5.1 is done in a for loop. As explained in more details in chapter 4 we cannot call the second `malloc` from a for loop. Because of this most times it is required to manually move a `malloc` outside of the for loop. This could be automated by searching the `malloc` in the code statically. To do this you would need to also follow functions call and do some pointer analysis to find the right pointer that was initialized. Because of those reasons the process was not yet automated.

5.2 Pointers

Because of the data centric nature of the SDFG pointers cannot be supported directly. But we can handle passing around data by reference by substituting pointers with their data containers. This still has the limitation that we cannot handle indirection, assignment or direct pointer arithmetic.

The `*` operator can be used to access pointers to single value but not pointers to arrays. It is best to access pointers using the array access notation (`a[0]`), in this way pointers to values and to arrays can be accessed.

[1] As talked about in chapter 3 for arrays that were in structs it is different. The number of indirections of the original struct must be subtracted from the number of indirections of the current variable.

```c
void foo(double* a, double* b)
    {
  b++;
  a = b;
}

// some code
double* a = calloc(2, sizeof(
    double));
double* b = calloc(3, sizeof(
    double));
foo(a, b);
// some other code
```

Invalid

```c
void foo(double* a, double* b)
    {
  a[0] = b[0];
}

// some code
double* a = calloc(2, sizeof(
    double));
double* b = calloc(3, sizeof(
    double));
b[0] = 42;
foo(a, b);

double* c = calloc(4, sizeof(
    double));
double* c_alias = c;
// some other code
```

Valid

Figure 5.2: Examples of supported and unsupported use of pointers

Note that the invalid code in figure 5.2 cannot be transformed directly in something valid. In order to transform the code there must be an higher level understanding on what the code is doing and hence it is hard to automate.

More details and some possible expansions were talked about in chapter 4.

5.3 Function return values

When a function returns a value it should be saved in a variable on the function call. C doesn't enforce this behavior but we are enforcing it on our strict subset of C. We assume that if a function is not of type `void` then it has some data output. Otherwise if there is no output it has to be declared as `void`. This is needed because it will influence the data flow hence when we create the SDFG we want to know if there is a return value.

This could be automated in theory by checking for every function call and adding a temporary variable to be thrown away afterwards. This was not done because this behavior is not often found and not too problematic to handle by hand.

5.4 Variable types

The type of a variable influences the way a variable is handled in C2DaCe. `double` is for values and `int` is for indices of arrays/matrices. Because integers are promoted to symbols (that are not passed by reference) all integer

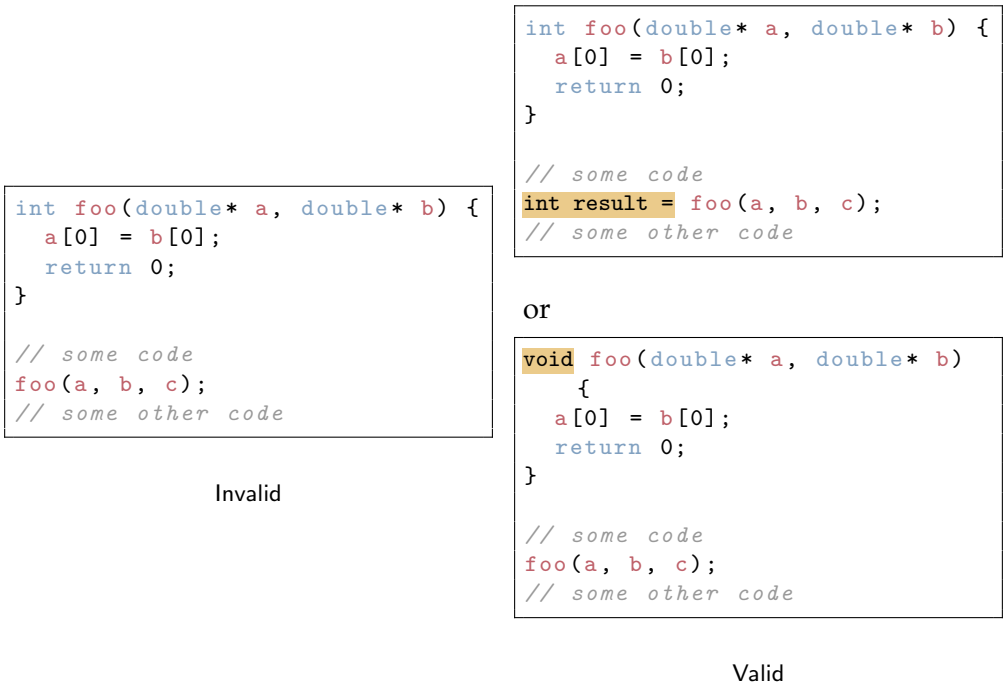


Figure 5.3: Example of correct way to define functions and do function calls

values passed around in scopes must be declared as `double`. Unfortunately is not always possible to use doubles everywhere, for example array indices must be integers otherwise the C compiler will complain. To work around this it is possible to define a temporary integer variables that is a copy of the double one.

This can be automated by changing every variable to `double` and for every read of an originally `int` variable we create a temporary `int` copy to read from instead. The performance impact should be tested and considered before implementing this. Casting is not an option because it is not implemented fully at the moment and in some cases produces undefined behaviors. A correct and complete implementation of casting could be another possible fix for this limitation.

5.5 Nested for loops

When optimization is done on the SDFG functions are often inlined into another. This will cause loops that where not nested in the scope to be nested after the inlining of the functions. Intern this causes the loops to possibly have the same loop variable. To fix this every loop that could be possibly nested has to have different loop variables.

```
int N = 10;
int computation = N*2 - 18 + 2;

double* a = malloc(N * sizeof(double));

return a[computation];
```

```
double N = 10;
double computation = N*2 - 18 + 2;

int N_int = N;
double* a = malloc(N_int * sizeof(double));

int computation_int = computation;
return a[computation_int];
```

Figure 5.4: Workaround to handle `int` type variables

```
void foo() {
  for (int i = 0; i < 5; i++)
      {
    // do something
  }
}

int main([...]) {
  for (int i = 0; i < 5; i++)
      {
    foo();
  }
}
```



```
void foo() {
  for (int j = 0; j < 5; j++)
      {
    // do something
  }
}

int main([...]) {
  for (int i = 0; i < 5; i++)
      {
    foo();
  }
}
```

Figure 5.5: Different `for` loop iteration variable names to counteract function inlining

To automatically fix this every loop in the whole code should have a different loop variable. It is difficult to rename all variables in scope correctly taking care of all possible cases hence an automatic transformation was not yet implemented.

Another solution is to change the way symbols are handled by C2DaCe. Currently for data containers we use a map that maps the current scope and the variable name to a new unique variable name used for the data containers. Symbols defined in a for loop instead don't use any kind of mapping. Extending the data container mapping strategy to all symbols could solve this problem.

Chapter 6

Mantevo HPCCG

One of the main objectives of the thesis is to make C2DaCe able to handle the HPCCG [9] benchmark. The HPCCG benchmark is a simple conjugate gradient benchmark. The benchmark uses a sparse matrix format that is implemented with pointers in a struct.

Because of the way the matrix is stored the limitations in C2DaCe were impeding it to transform the HPCCG benchmark. Thanks to the removal of the limitations described in section 3, section 4 and other limitation described in the following sections the benchmark was able to be transformed by C2DaCe.

6.1 Basic preprocessing

The benchmark is written in C++ with constructs that C2DaCe is not able to handle. First it was converted to simple C. The benchmark was also merged in a single file to make it easier to work with. By having a single file it is easier to debug and find where problematic statements are.

Because HPCCG almost doesn't use any objects or classes the conversion is pretty simple. The only object is `YAML_Doc` that is used to store logs but this can be substituted by a simple `printf` to print to the `stdout`. Then all the `cout` where substituted by `printf` and all the macros were removed. The macros where only used for the setup of MPI, OpenMP or the debug information so they where pretty trivial to remove (because we don't need either MPI nor OpenMP). Then all C++ references were converted to pointers.

Most `int` variables were changed to `double`. Not all variables can be `double` as explained in section 5.4. The variables that caused problem where mostly discovered by trial and error. C2DaCe and DaCe will produce a parallel C++ file that will be compiled with Clang. When Clang fails it is mostly due to variables with the wrong type. Looking at the error logs of Clang it is easy

to pinpoint the variables causing errors and apply the correct workaround to correct it.

6.2 Referencing the matrix struct

To pass around the matrix struct it is stored as a double pointer that will be initialized by the function `generate_matrix`. The implementation described in chapter 3 and chapter 4 is able to only manage single pointers to structs. As explained in chapter 4 we cannot handle data indirection in the SDFG.

To fix this issue we only use a single pointer as a reference to the matrix struct. To initialize the matrix we inline the function so we don't need to add the extra level of indirection.

Thanks to the work done from chapter 3 we are able to handle single pointers structs without problems including passing structs around in functions.

6.3 Representing complex data storage schemes

The (sparse) matrix is defined with the struct from figure 6.1.

```
struct HPC_Sparse_Matrix_STRUCT {
  char    *title;
  int start_row;
  int stop_row;
  int total_nrow;
  long long total_nnz;
  int local_nrow;
  int local_ncol;  // Must be defined in make_local_matrix
  int local_nnz;
  int  * nnz_in_row;
  double ** ptr_to_vals_in_row;
  int ** ptr_to_inds_in_row;
};
```

Figure 6.1: Sparse matrix struct definition

The matrix is stored with the LIL format where it has 2 arrays for every row namely `ptr_to_vals_in_row` and `ptr_to_inds_in_row`. The two arrays are of size $\#rows \times 27$ because every row has a maximum of 27 non zero elements. One array stores the values and the other the column index. The initialization code can be seen in figure 6.2.

As explained in chapter 4, pointer arithmetic and assignments of references are not supported in C2DaCe. To work around the limitation the initialization was modified to allocate the data row by row and assign it directly into the

```
for (int iz=0; iz<nz; iz++) {
  [...]
  A->ptr_to_vals_in_row[curlocalrow] = curvalptr;
  A->ptr_to_inds_in_row[curlocalrow] = curindptr;
  [...]
  for (int sz=-1; sz<=1; sz++) {
    [...]
    if (curcol==currow) {
      A->ptr_to_diags[curlocalrow] = curvalptr;
      *curvalptr++ = 27.0;
    } else {
      *curvalptr++ = -1.0;
    }
    *curindptr++ = curcol;
    nnzrow++;
    [...]
  }
  [...]
}
```

Figure 6.2: Original HPCCG matrix initialization code snippet

2D array. The resulting code from the transformation can be seen in figure 6.4 and the resulting SDFG in figure 6.5.

The transformation is done automatically by an AST transformation that matches the specific pattern of LIL sparse matrices defined with a struct. That is, two subsequent assignment to pointer arrays as fields on the same struct followed by the increment and writing to the pointers that were assigned to the struct fields.

The advantage of this workaround is that - other than the initialization code - every other access to the matrix can remain the same because we use the same pattern. We also don't use any more space than the original code. The implementation is also data centric friendly because we are working basically with 2 dense matrices that we are already able to handle.

6.4 Results

All runs were done 10 times on a 4 core 8 threads Intel i5-1035G4 with 8GB of RAM. The problem size was: $nx = 89, ny = 96, nz = 101$. All benchmarks where compiled with gcc 10.2.1. The original benchmark was compiled with OpenMP enabled and the following compilation arguments `-O3 -ftree-vectorize -ftree-vectorizer-verbose=2 -march=native -fopenmp -lm`. Serial was obtained by compiling the file before ingestion into C2DaCe and the C2DaCe version was obtained by compiling the C++ file generated

by C2DaCe. Both Serial and C2DaCe where compiled with the following arguments `-O3 -march=native -lm`.

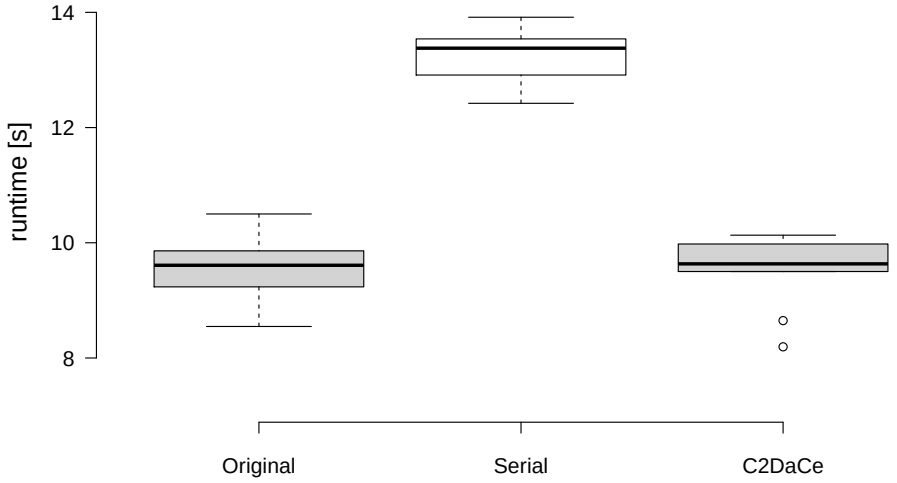


Figure 6.3: Boxplot of running time in seconds with different benchmarks. Whiskers extend to data points that are less than 1.5 IQR away from 1st/3rd quartile.

As you can see from figure 6.3 the running time of the C2DaCe auto optimized version is about the same as the original parallel one. This shows that C2DaCe is able to obtain performance at least as good as hand optimized code.

This result is easily explained by looking at the generated SDFG where we notice that the parallelized loops are the same. For example in figure 6.6 we can see that the original `ddot` function is parallelized using OpenMP annotations on the loops and in figure 6.7 we see that the SDFG version of the function is implemented using parallel maps for the same loops. Every loop that the original code parallelized also C2DaCe was able to find the same parallelization opportunity.

```
A->ptr_to_vals_in_row[0] = calloc(max_nnz, sizeof(double));
A->ptr_to_inds_in_row[0] = calloc(max_nnz, sizeof(int));
for (int iz=0; iz<nz; iz++) {
  [...]
  A->ptr_to_vals_in_row[curlocalrow] = calloc(max_nnz, sizeof(
      double));
  A->ptr_to_inds_in_row[curlocalrow] = calloc(max_nnz, sizeof(int
      ));
  [...]
  for (int sz=-1; sz<=1; sz++) {
    [...]
    if (curcol==currow) {
      A->ptr_to_vals_in_row[curlocalrow][curvalptr] = 27;
    } else {
      A->ptr_to_vals_in_row[curlocalrow][curvalptr] = -1;
    }
    A->ptr_to_inds_in_row[curlocalrow][curvalptr] = curcol;
    curvalptr++;
    nnzrow++;
  }
  [...]
}
```

Figure 6.4: Modified HPCCG matrix initialization code without pointer arithmetic

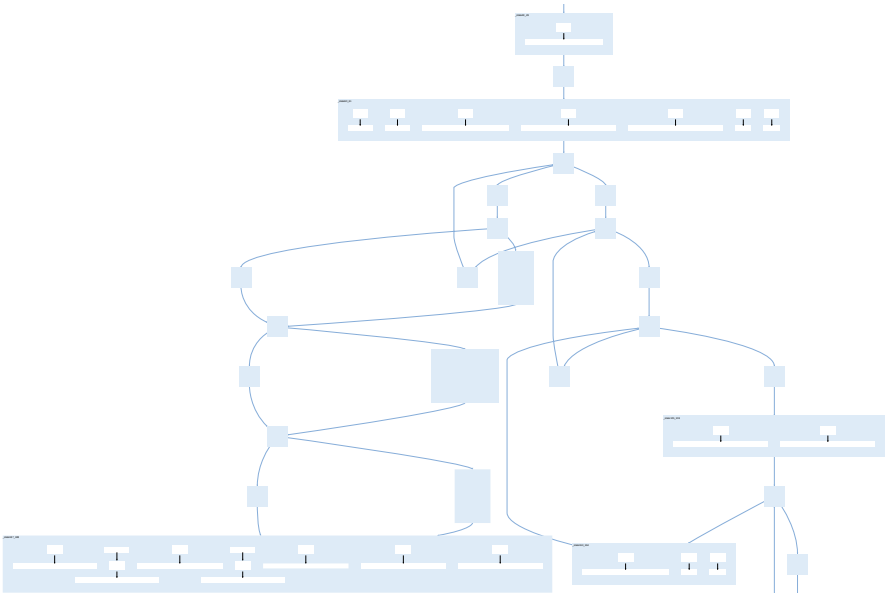


Figure 6.5: SDFG of the matrix initialization loops

```
#include "ddot.hpp"
int ddot (const int n, const double * const x, const double *
    const y,
    double * const result, double & time_allreduce)
{
  double local_result = 0.0;
  if (y==x)
#ifdef USING_OMP
#pragma omp parallel for reduction (+:local_result)
#endif
    for (int i=0; i<n; i++) local_result += x[i]*x[i];
  else
#ifdef USING_OMP
#pragma omp parallel for reduction (+:local_result)
#endif
    for (int i=0; i<n; i++) local_result += x[i]*y[i];
  [...]
}
```

Figure 6.6: Original HPCCG code of the ddot function using OpenMP annotations

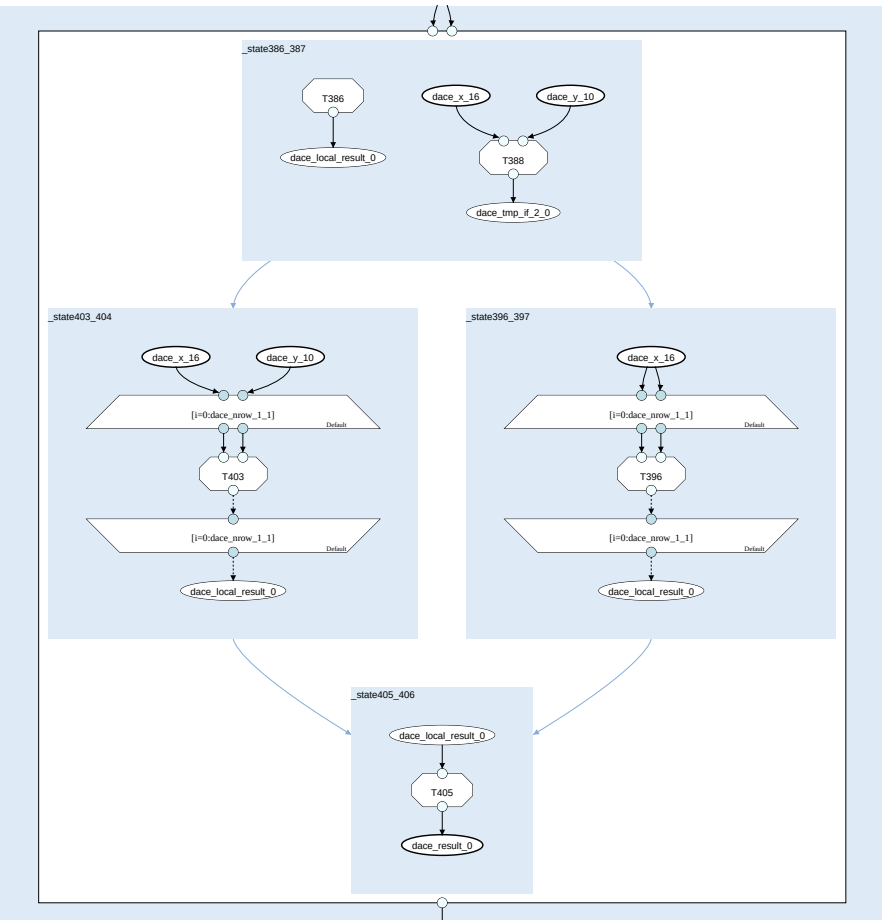


Figure 6.7: `ddot` function in the optimized SDFG using maps to parallelize the computation

Chapter 7

BOTS suite FFT

After having experience making HPCCG work I wanted to tackle another benchmark to see if with the new implemented features and with my new knowledge it was possible to adapt another benchmark in less time.

The benchmark I selected was the FFT program from the BOTS suite [10]. It was selected because it is an important suite of parallel benchmarks. But also because it is self contained and written in C and not C++.

The benchmark implements a Fast Fourier Transform (FFT) that works by taking an input and an output `N` sized array. The array is split as part of the divide and conquer algorithm and then joined again to have the global result.

The preprocessing went mostly without problems following the directives from chapter 5. But then two big issues were encountered. The first of which was the presence of recursive functions. Those are not supported in the SDFG because of the graph nature of the representation. Another problem was encountered with `for` loops using more complex manipulations of the loop variable. This is a current limitation of C2DaCe that only implements a subset of the `for` instruction.

7.1 Removal of recursive functions

It is not possible to have a recursive call in the SDFG. To create recursion we need to store the order of the recursive calls. To do this in the SDFG we would have to create an undefined number of states in a tree like structure. Because we don't know in advance how deep and wide the tree should be and because we cannot create new states during runtime it is not possible to have recursion in the SDFG.

The solution is to convert the recursive calls to iterative ones. Some functions where easy to make iterative because the iterative version was already im-

plemented. The function in figure 7.1 is a divide and conquer style function with an iterative version for small numbers.

```
void compute_w_coefficients(int n, int a, int b, COMPLEX * W)
{
  double twoPiOverN;
  int k;
  double s
  double c;

  if (b - a < 128) {
    twoPiOverN = 2.0 * 3.14159265358979323846264434 / n;
    for (k = a; k <= b; ++k) {
      c = cos(twoPiOverN * k);
      (W[k]).re = c
      (W[n - k]).re = c;
      s = sin(twoPiOverN * k);
      (W[k]).im = -s;
      (W[n - k]).im = s;
    }
  } else {
    int ab = (a + b) / 2;
    compute_w_coefficients(n, a, ab, W);
    compute_w_coefficients(n, ab + 1, b, W);
  }
}
```

Figure 7.1: `compute_w_coefficients` function of the FFT benchmark from the BOTS suite. Recursive function with cutoff value for an iterative version.

It is easy to see that in this case if we remove the if that sets the threshold for the recursive version we can run the for loop for the whole array size and obtain the same result.

A more challenging function to transform was the main auxiliary FFT function from figure 7.2.

The function is only implemented with a recursive call and there is no iterative version already implemented. But we can notice that it is still a divide and conquer style of function.

Another thing that we notice from the implementation of `unshuffle` and `fft_twiddle_gen` is that both functions will only access the subset $[offset, m + offset]$ of `in` and `out`. Also the for loop is covering the whole range of the array because `r` is a factor of `n` hence `m` (`m = n / r`) is an integer. Between recursive calls with the same depth the variables `n`, `m`, `r` and `factor_offset` are the same. We can map out the subset that is accessed by every call and see that we can compute those values knowing only the current depth.

What we can do is divide the function in two separate while loops the

```
void fft_aux(int n, COMPLEX * in, COMPLEX * out, int *factors,
    int factor_offset, COMPLEX * W, int nW, int offset)
{
  int r, m;
  int k;

  r = factors[factor_offset];
  m = n / r;

  if (r < n) {
    /*
    * split the DFT of length n into r DFTs of length n/r,  and
    * recurse
    */
    unshuffle(0, m, in, out, r, m, offset);

    for (k = 0; k < n; k += m) {
      fft_aux(m, out, in, factors, factor_offset + 1, W, nW,
          offset+k);
    }
  }
  /*
  * now multiply by the twiddle factors, and perform m FFTs
  * of length r
  */
  fft_twiddle_gen(0, m, in, out, W, nW, nW / n, r, m, offset);
}
```

Figure 7.2: Main auxiliary function of the FFT benchmark (BOTS)

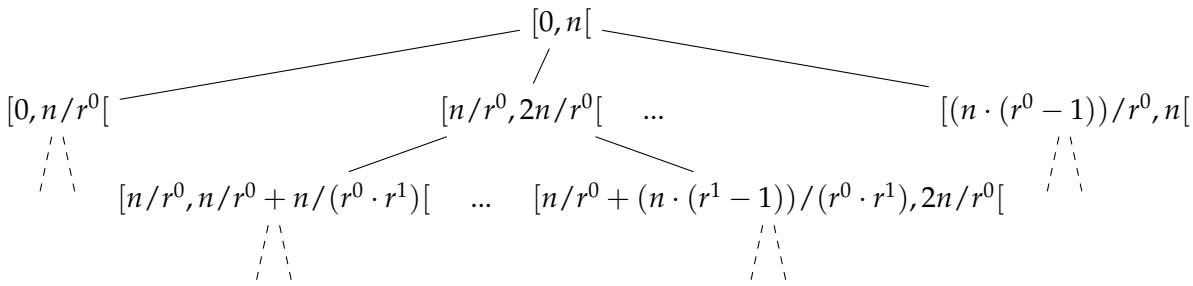


Figure 7.3: Tree of accessed subsets from `fft_aux` recursion where r^i is the $(i+1)$-th factor of n

first one will do all the `unshuffle` calls and the second one will do all the `fft_twiddle_gen` calls. Every loop represents an increase of recursion depth. The only value that changes at the same depth is the offset, we create an array to keep track of the different offset values. Knowing the current offsets and depth we can compute the offsets for the next depth. For the `unshuffle` loop we start at depth 0 and we increase the depth with every iteration. With the `fft_twiddle_gen` loop we reverse the process starting from the deepest

depth and going up until reaching depth 0. We can keep track of the depth using the `factor_offset` that starts with 0 at depth 0 and increases by one at every depth increase.

```
// unshuffle loop
while (r < n) {
  // apply unshuffle with every offset
  for (int k = 0; k <= last_offset; k++) {
    if (invert) {
      unshuffle(0, m, out, in, r, m, offsets[k]);
    } else {
      unshuffle(0, m, in, out, r, m, offsets[k]);
    }
  }

  double max_offset = last_offset;
  // for every offset compute the offsets for the new depth
  for (int i = 0; i <= max_offset; i++) {
    double cur_offset = offsets[i];
    double x = m;

    // same loop as recursion but we append new offsets to the
       list
    for (int tmp_3 = 0; x < n; tmp_3++) {
      last_offset += 1;
      last_offset_int = last_offset;
      offsets[last_offset_int] = cur_offset+x;
      x += m;
    }
  }

  // update values for new loop
  n = m;
  factor_count += 1;
  factor_count_int = factor_count;
  r = factors[factor_count_int];
  m = n / r;
  offsets_count[factor_count_int] = last_offset + 1;
  invert = !invert;
}
```

Figure 7.4: First loop of the iterative main auxiliary function from the FFT benchmark (BOTS)

Another thing to keep track of is the inversion of the `in` and `output`. After every recursion the arrays are inverted. To do so we keep a variable that we invert after every loop iteration.

The recursive implementation was executing the computations in a depth-first (DF) order. With our interactive implementation we instead use the breadth-first (BF) ordering. Each computation can be executed only if their

```
// twiddle gen loop
while (factor_count >= 0) {
  // use only the correct offsets slice
  factor_count_int = factor_count;
  last_offset = offsets_count[factor_count_int] - 1;

  // apply twiddle gen for every offset
  for (int x = 0; x <= last_offset; x++) {
    if (invert) {
      fft_twiddle_gen(0, m, out, in, W, nW, nW / n, r, m, offsets
          [x]);
    } else {
      fft_twiddle_gen(0, m, in, out, W, nW, nW / n, r, m, offsets
          [x]);
    }
  }

  // update values to go back to previous depth ones
  m = n;
  factor_count -= 1;
  factor_count_int = factor_count;
  if (factor_count_int < 0) {
    factor_count_int = 0;
  }
  r = factors[factor_count_int];
  n = m*r;
  invert = !invert;
}
```

Figure 7.5: Second loop of the iterative main auxiliary function from the FFT benchmark (BOTS). Together with first loop in figure 7.5 it is semantically equivalent to the original recursive one in figure 7.2

parent was. This is because the subset that the parent access includes the subsets of all its children hence it must be executed before. But this property is met in both DF and BF this means that both orderings are fine.

The resulting SDFG from figure 7.10 show two big blocks. The one in the top left is the first `while` loop and the one in the bottom right is the second loop. Every block contains two other block that are the two function call from inside every loop.

7.1.1 Automating the removal of recursive functions

The removal of recursive functions is the most time consuming step to be done by hand. Even then it is much faster than having to adapt the whole code to a parallelization paradigm.

The procedural removal of any recursive function is a really hard problem. It could be done by simulating a stack and emulating the recursive calls. Even

then it is not clear how this could scale and be parallelizable effectively. A possible specific case to handle is one such as the one of figure 7.1. In that case the iterative version is already implemented to be run for small problem sizes. A transformation can be done on this type of recursive function to remove the condition for the cutoff and run the iterative version for any problem size.

7.2 Complex for statements

As discussed in section 5.5 for loops currently have some limitations on name mapping of loop variables. Those are defined as symbols in the SDFG such that they can be defined and modified on inter state edges. This makes the creation of for loops simple. The problem is that C2DaCe assumes that the increment statements only use symbols in it hence it will not apply any name mapping (as opposed to data containers where the variable name in C is mapped to a unique data container in the SDFG). This will fail when a data container is used to increment the loop variable because it will not substitute the variable name with the mapped name in the scope.

```
double* x = malloc(sizeof(double));
x[0] = 15;

for (int i=0; i<95; i+=x[0]) {
  [...]
}
```

Figure 7.6: Example of `for` loop using variables that are not symbols

In the example of figure 7.6 it will not work because `x` is mapped to a data container. C2DaCe will fail to find a data container with the name `x` because the C variable name has to be first passed to the name mapping map.

As explained in the end of section 5.5 this could by fixed by applying the name mapping also to symbols. A quick manual solution is to define the loop variable outside of the loop and do the increment at the end of the loop body. In this way it will be treated as any other variable with the correct mappings. But this could create worst performance because the loop variable is not defined in the loop header anymore hence the automatic transformation done by DaCe to transform loops to map could not be applied in the same way.

```
double* x = malloc(sizeof(double));
x[0] = 15;

double i = 0;
for (int tmp=0; i<95; tmp++) {
  [...]
  i += x[0];
}
```

Figure 7.7: Workaround to use variables that are not symbols as increment values in `for` loops

7.3 Results

By printing the output at the end of the benchmark we can compare the two versions and see that we have correctness. Unfortunately for this benchmark we cannot test performance because of a problem with allocation when compiling the SDFG.

When DaCe compiles the SDFG to C++ code it has to know where to allocate and unallocate the data containers. This information is not saved into the SDFG because it is hardware agnostic and the target could not have the concept of memory allocation. When compiling to C++ DaCe has to infer where to allocate and unallocate the data containers based on the accesses. Unfortunately there could be some ambiguous cases.

```
double* a = malloc(N * sizeof(double));
for (int i=0; i<N; i++) {
  // do something with a[i]
}
free(a);

// other case
for (int i=0; i<N; i++) {
  double* a = malloc(N * sizeof(double));
  // do something with a[i]
  free(a);
}
```

Figure 7.8: C code that generates an ambiguous allocation case inside an SDFG

In the two cases from figure 7.8 the data is accessed in the same way inside the loop but the allocation and unallocation is different. But when we translate it to the SDFG we obtain the same one because we don't have the concept of allocation.

To workaround this issue a new transformation was added to create dummy accesses at the `malloc` and `free` instruction.

```
double* a = malloc(N * sizeof(double));
a[0] = 0;
for (int i=0; i<N; i++) {
  // do something with a[i]
}
double tmp_0 = a[0];
free(a);

// other case
for (int i=0; i<N; i++) {
  double* a = malloc(N * sizeof(double));
  a[0] = 0;
  // do something with a[i]
  double tmp_1 = a[0];
  free(a);
}
```

Figure 7.9: Ambiguous allocation case with dummy accesses

This works for the non optimized SDFG but after the optimization the workaround is removed as it is dead code. This problem is currently discussed internally of the DaCe team to find the best possible solution which will probably be an edit to both the frontend and backend. This means that C2DaCe will have to be modified to be able to work with this new solution.

Because of this we are able to only compile the non optimized SDFG to check for correctness. The performance before the optimizations is even worst that the serial version hence it is pointless to obtain performance metric.

Nevertheless in about a week worth of work the benchmark was able to be compiled into an SDFG. Because the optimization transformations are safe once a solution for the allocation dilemma is found the SDFG should be able to be directly optimized without modifications.

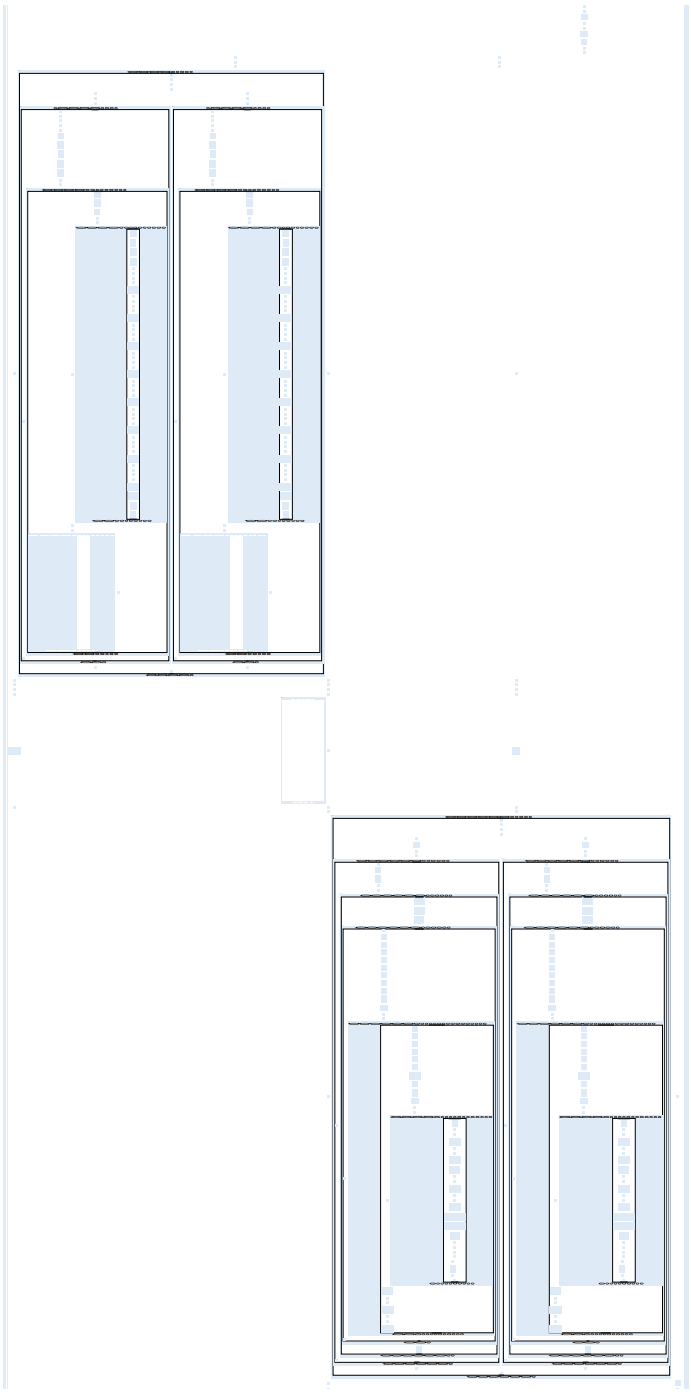


Figure 7.10: SDFG of the `fft_while` function. Top left two `unshuffle` function calls and bottom right two `fft_twiddle_gen` calls.

Chapter 8

Related works

DaCe and its SDFG IR is not the only backend that aims to optimize for heterogenous architecture and possible parallelization opportunities. Because of the importance of the C language most backends also have some implementation of a C frontend that enables the automatic translation of C programs.

The MLIR project [12] uses a Multi-Level Intermediate Representation (MLIR) to target multiple architecture and exploit the potential parallelism. MLIR also has the ability to represent dataflow graphs and do optimizations on those. A major difference to the SDFG is that MLIR encapsulate multiple hardware specific instructions where the SDFG is hardware agnostic. MLIR is also extensible with *Dialects* that extend the IR to support additional features and architectures. Polygeist [13] is a C frontend of MLIR that uses the polyhedral compilation support provided inside MLIR.

The Polyhedral Parallel Code Generator (PPCG) [14] also uses polyhedral compilation to transform C to CUDA. Because PPCG is limited to the GPU architecture it can only unveil GPU-like parallelism. This can lead to substantial performance degradation in case of non GPU-like paradigms. Where instead the DaCe approach is able to identify parallelism based only on the data movements. This makes it able to work independently on the architecture.

Another approach is the one taken with OpenMP [15]. Where C/C++ code is annotated to specify which instructions should be parallelized. This approach require the user to specify the parallelization opportunities instead of the framework applying automatic parallelization.

The REcursion Automatically PARallelized (REAPAR) system [16] covers a case that C2DaCe is not able to handle, recursion. REAPAR is able to parallelize recursive call that don't have data dependencies assigning call to different threads. REAPAR can only parallelize recursive call and not any

another feature. Further more it doesn't check for data dependencies and assumes that all recursive call are independent.

Another IR that is hardware agnostic is the Heterogenous Parallel Virtual Machine (HPVM) [17]. It also uses a dataflow graph (DFG) to represent the code like inside an SDFG state. The main difference is that HPVM hasn't the concept of states or a another concept to represent corse-grained control flow. This makes the generation of a DFG from a language such as C much harder. In fact the frontend HeteroC++ from HPVM needs very specific manually annotated C++ code as input.

Chapter 9

Conclusion and future work

C2DaCe covers a good subset of the C language but it still has some limitation and doesn't support all features of the language. Most limitations can be solved with a workaround by rewriting part of the source code. Even if this process is quicker and easier than rewriting the whole program in a parallelized way, automating this will greatly improve the ease of use. This can be done by creating additional AST transformations or by modifying the way certain language constructs are handled. Specifically:

- General implementation of pointer offsets using offset variables modified at runtime
- Handling of `int` types also as values that can be referenced between multiple functions
- Fixing name mapping problems of `for` loops
- Matching simple recursive function patterns to be transformed into iterative functions

Those are the main limitations that can be possibly solved in the near future. Unfortunately there are still limitations such as the support for general pointers and general recursive function where a solution is much harder to find. For those problems it is important that the end user is informed about what C2DaCe can't do. In this way unsupported constructs could be circumvented. A complete documentation of the project could greatly aid the usage of C2DaCe.

Even then C2DaCe has shown to be capable of optimizing serial C code into parallel code with performance that is on par with hand written parallelization. This is much faster than rewriting the whole source code even if some manual rewriting is needed to work around the limitations. Unfortunately the process of ingesting code into C2DaCe is still slow and error prone. It is an early tool that needs work to be efficient but results have shown that the

capabilities are there. With time C2DaCe could be able to parallelize code seamlessly and easily.

Appendix A

Preprocessed HPCCG C code

```
//@HEADER
//
    ************************************************************************

//
//                HPCCG: Simple Conjugate Gradient Benchmark Code
//                  Copyright (2006) Sandia Corporation
//
// Under terms of Contract DE-AC04-94AL85000, there is a non-
    exclusive
// license for use of this work by or on behalf of the U.S.
    Government.
//
// BSD 3-Clause License
//
// Redistribution and use in source and binary forms, with or
    without
// modification, are permitted provided that the following
    conditions are met:
//
// * Redistributions of source code must retain the above
    copyright notice, this
//   list of conditions and the following disclaimer.
//
// * Redistributions in binary form must reproduce the above
    copyright notice,
//   this list of conditions and the following disclaimer in the
    documentation
//   and/or other materials provided with the distribution.
//
// * Neither the name of the copyright holder nor the names of
    its
//   contributors may be used to endorse or promote products
    derived from
//   this software without specific prior written permission.
//
```

```
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
    CONTRIBUTORS "AS IS"
// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
    LIMITED TO, THE
// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
    PARTICULAR PURPOSE ARE
// DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR
    CONTRIBUTORS BE LIABLE
// FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
    CONSEQUENTIAL
// DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
    SUBSTITUTE GOODS OR
// SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
    INTERRUPTION) HOWEVER
// CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
    STRICT LIABILITY,
// OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
     OUT OF THE USE
// OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
    DAMAGE.
//
// Questions? Contact Michael A. Heroux (maherou@sandia.gov)
//
//
    ***************************************************************************

//@HEADER

#include <stdio.h>
#include <stddef.h>
#include <math.h>
#include <stdlib.h>

struct HPC_Sparse_Matrix_STRUCT {
  char    *title;
  int start_row;
  int stop_row;
  int total_nrow;
  long long total_nnz;
  int local_nrow;
  int local_ncol;  // Must be defined in make_local_matrix
  int local_nnz;
  int  * nnz_in_row;
  double ** ptr_to_vals_in_row;
  int ** ptr_to_inds_in_row;
};
typedef struct HPC_Sparse_Matrix_STRUCT HPC_Sparse_Matrix;


void waxpby (int n, double alpha, double* x, double beta, double*
    y, double* w)
{
  if (alpha==1.0) {
    for (int i=0; i<n; i++) w[i] = x[i] + beta * y[i];
```

```
  }
  else if(beta==1.0) {
    for (int i=0; i<n; i++) w[i] = alpha * x[i] + y[i];
  }
  else {
    for (int i=0; i<n; i++) w[i] = alpha * x[i] + beta * y[i];
  }

}

void ddot (int n, double* x, double* y, double* result)
{
  double local_result = 0.0;
  if (y==x)
    for (int i=0; i<n; i++) local_result += x[i]*x[i];
  else
    for (int i=0; i<n; i++) local_result += x[i]*y[i];

  *result = local_result;
}

void HPC_sparsemv( HPC_Sparse_Matrix *A, double* x, double* y)
{

  int nrow = A->local_nrow;

  for (int i=0; i< nrow; i++) {
    double sum = 0.0;

    int cur_nnz = A->nnz_in_row[i];

    for (int j=0; j< cur_nnz; j++) {
      sum += A->ptr_to_vals_in_row[i][j]*x[A->ptr_to_inds_in_row[
          i][j]];
    }

    y[i] = sum;
  }
}

void dump_matlab_matrix(HPC_Sparse_Matrix *A, int rank) {
  double nrow = A->local_nrow;
  double start_row = nrow*rank; // Each processor gets a section
      of a chimney stack domain

  printf("===== MATRIX DUMP ======\n");
  for (int i=0; i< nrow; i++) {
    int cur_nnz = A->nnz_in_row[i];
    for (int j=0; j< cur_nnz; j++) {
      printf(" %f %d %22.16e,",start_row+i+1,A->
          ptr_to_inds_in_row[i][j]+1,A->ptr_to_vals_in_row[i][j])
          ;
    }
    printf("\n");
```

```
  }
  printf("===== END DUMP ======\n");
}

void HPCCG (HPC_Sparse_Matrix * A, double* b, double* x, int
    max_iter, double tolerance, int* niters, double* normr) {
  int nrow = A->local_nrow;
  int ncol = A->local_ncol;

  double* r = calloc(nrow, sizeof(double));
  double* p = calloc(ncol, sizeof(double));
  double* Ap = calloc(nrow, sizeof(double));

  double norm = 0.0;
  double* rtrans = calloc(1, sizeof(double));
  rtrans[0] = 0;

  double oldrtrans = 0.0;

  int rank = 0; // Serial case (not using MPI)

  int print_freq = max_iter/10;
  if (print_freq>50) print_freq=50;
  if (print_freq<1)  print_freq=1;

  // p is of length ncols, copy x to p for sparse MV operation
  waxpby(nrow, 1.0, x, 0.0, x, p);
  HPC_sparsemv(A, p, Ap);
  waxpby(nrow, 1.0, b, -1.0, Ap, r);
  ddot(nrow, r, r, rtrans);
  norm = sqrt(rtrans[0]);

  if (rank==0) printf("Initial Residual = %e\n", norm);

  for(int k=1; ((k<max_iter) && (norm > tolerance)); k++ ) {
    if (k == 1) {
      waxpby(nrow, 1.0, r, 0.0, r, p);
    } else {
      oldrtrans = rtrans[0];
      ddot (nrow, r, r, rtrans);// 2*nrow ops
      double beta = rtrans[0]/oldrtrans;
      waxpby (nrow, 1.0, r, beta, p, p);// 2*nrow ops
    }
    norm = sqrt(rtrans[0]);
    if (rank==0 && (k%print_freq == 0 || k+1 == max_iter))
    printf("Iteration = %d, Residual = %e\n", k, norm);


    HPC_sparsemv(A, p, Ap); // 2*nnz ops
    double* alpha = calloc(1, sizeof(double));
    alpha[0] = 0;
    ddot(nrow, p, Ap, alpha); // 2*nrow ops
    alpha[0] = rtrans[0]/alpha[0];
    waxpby(nrow, 1.0, x, alpha[0], p, x);// 2*nrow ops
```

```
    waxpby(nrow, 1.0, r, -(alpha[0]), Ap, r);// 2*nrow ops
    niters[0] = k;

    free(alpha);
  }

  double tmp = r[0] + Ap[0] + p[0];

  normr[0] = norm;
}

int main(int argc, char *argv[])
{

  HPC_Sparse_Matrix *A = calloc(1, sizeof(HPC_Sparse_Matrix));
  double norm;
  double d;
  int ierr = 0;
  int i, j;
  int ione = 1;

  int debug = 1;
  int size = 1; // Serial case (not using MPI)
  int rank = 0;

  int nx = 89;
  int ny = 96;
  int nz = 101;

  A->title = 0;

  // Set this bool to true if you want a 7-pt stencil instead of
      a 27 pt stencil
  int use_7pt_stencil = 0;

  int local_nrow = nx*ny*nz; // This is the size of our subblock

  double max_nnz = 27;
  int local_nnz = max_nnz*local_nrow; // Approximately 27
      nonzeros per row (except for boundary nodes)

  int total_nrow = local_nrow*size; // Total number of grid
      points in mesh
  long long total_nnz = max_nnz* (long long) total_nrow; //
      Approximately 27 nonzeros per row (except for boundary
      nodes)

  double start_row = local_nrow*rank; // Each processor gets a
      section of a chimney stack domain
  double stop_row = start_row+local_nrow-1;


  // Allocate arrays that are of length local_nrow
  A->nnz_in_row = calloc(local_nrow, sizeof(int));
```

```
A->ptr_to_vals_in_row = calloc(local_nrow, sizeof(double*));
A->ptr_to_inds_in_row = calloc(local_nrow, sizeof(int*));

int max_nnz_int = max_nnz;
A->ptr_to_vals_in_row[0] = calloc(max_nnz_int, sizeof(double));
A->ptr_to_inds_in_row[0] = calloc(max_nnz_int, sizeof(int));

A->ptr_to_vals_in_row[0][0] = 0;
A->ptr_to_inds_in_row[0][0] = 0;

double *x = calloc(local_nrow, sizeof(double));
double *b = calloc(local_nrow, sizeof(double));

double* curvalptr = calloc(local_nnz, sizeof(double));
int* curindptr = calloc(local_nnz, sizeof(int));

long long nnzglobal = 0;
for (int iz=0; iz<nz; iz++) {
  for (int iy=0; iy<ny; iy++) {
    for (int ix=0; ix<nx; ix++) {
      int curlocalrow = iz*nx*ny+iy*nx+ix;
      int currow = start_row+iz*nx*ny+iy*nx+ix;
      int nnzrow = 0;

      A->ptr_to_vals_in_row[curlocalrow] = curvalptr;
      A->ptr_to_inds_in_row[curlocalrow] = curindptr;
      for (int sz=-1; sz<=1; sz++) {
        for (int sy=-1; sy<=1; sy++) {
          for (int sx=-1; sx<=1; sx++) {
            int curcol = currow+sz*nx*ny+sy*nx+sx;
            if (((((ix+sx>=0) && (ix+sx<nx)) && (iy+sy>=0)) &&
                (iy+sy<ny)) && (curcol>=0 && curcol<total_nrow)
                ) {
              if (!use_7pt_stencil || (sz*sz+sy*sy+sx*sx<=1)) {
                  // This logic will skip over point that are
                not part of a 7-pt stencil
                if (curcol==currow) {
                  *curvalptr++ = 27.0;
                } else {
                  *curvalptr++ = -1.0;
                }
                //printf("%f", (A->ptr_to_vals_in_row)[0][0]);
                *curindptr++ = curcol;
                nnzrow++;
              }
            }
          } // end sx loop
        } // end sy loop
      } // end sz loop
      A->nnz_in_row[curlocalrow] = nnzrow;
      nnzglobal += nnzrow;
      x[curlocalrow] = 0.0;
      b[curlocalrow] = 27.0 - ((double) (nnzrow-1));
    } // end ix loop
```

```
      } // end iy loop
  } // end iz loop

  if (debug) printf("Process %d of %d has %d",rank,size,
      local_nrow);

  if (debug) printf(" rows. Global rows %d through %d\n",
      start_row,stop_row);

  if (debug) printf("Process %d of %d has %lld nonzeros.\n",rank,
      size,nnzglobal);

  A->start_row = start_row ;
  A->stop_row = stop_row;
  A->total_nrow = total_nrow;
  A->total_nnz = total_nnz;
  A->local_nrow = local_nrow;
  A->local_ncol = local_nrow;
  A->local_nnz = local_nnz;

  int dump_matrix = 0;
  if (dump_matrix && size<=4) dump_matlab_matrix(A, rank);

  int* niters = calloc(1, sizeof(int));
  double* normr = calloc(1, sizeof(double));

  int max_iter = 150;
  double tolerance = 0.0; // Set tolerance to zero to make all
      runs do max_iter iterations
  HPCCG( A, b, x, max_iter, tolerance, niters, normr);

  double fniters = niters[0];
  double fnrow = A->total_nrow;
  double fnnz = A->total_nnz;
  double fnops_ddot = fniters*4*fnrow;
  double fnops_waxpby = fniters*6*fnrow;
  double fnops_sparsemv = fniters*2*fnnz;
  double fnops = fnops_ddot+fnops_waxpby+fnops_sparsemv;

  printf("Dimensions: x=%d, y=%d, z=%d\n",nx,ny,nz);
  printf("Number of iterations: %d\n", niters[0]);
  printf("Final residual: %e\n", normr[0]);

  printf("FLOPS Summary:\n");
  printf("Total    : %e\n",fnops);
  printf("DDOT     : %e\n",fnops_ddot);
  printf("WAXPBY   : %e\n",fnops_waxpby);
  printf("SPARSEMV: %e\n",fnops_sparsemv);

  return 0;
}
```

Appendix B

Preprocessed BOTS FFT C code

```
/*
   ************************************************************************************
   */
/*  This program is part of the Barcelona OpenMP Tasks Suite
                                    */
/*  Copyright (C) 2009 Barcelona Supercomputing Center - Centro
   Nacional de Supercomputacion  */
/*  Copyright (C) 2009 Universitat Politecnica de Catalunya
                                    */
/*

   */
/*  This program is free software; you can redistribute it and/or
    modify                          */
/*  it under the terms of the GNU General Public License as
   published by                     */
/*  the Free Software Foundation; either version 2 of the License
   , or                            */
/*  (at your option) any later version.
                                                          */
/*

   */
/*  This program is distributed in the hope that it will be
   useful,                          */
/*  but WITHOUT ANY WARRANTY; without even the implied warranty
   of                               */
/*  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
                                    */
/*  GNU General Public License for more details.
                                                      */
/*

   */
/*  You should have received a copy of the GNU General Public
   License                          */
```

```
/*  along with this program; if not, write to the Free Software
                                    */
/*  Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
      02110-1301  USA             */
/*
    ************************************************************************
    */

/*
 * Original code from the Cilk project
 *
 * Copyright (c) 2000 Massachusetts Institute of Technology
 * Copyright (c) 2000 Matteo Frigo
 */

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>

struct COMPLEX_struct{
     double re;
   double im;
};

typedef struct COMPLEX_struct COMPLEX;

void compute_w_coefficients(double n, double a, double b, COMPLEX
    * W)
{
     double twoPiOverN = 0;
     double s = 0;
   double c = 0;
   int n_int = n;

  twoPiOverN = 2.0 * 3.14159265358979323846264434 / n;
  for (int k = a; k <= b; k++) {
    n_int = n;
    c = cos(twoPiOverN * k);
    (W[k]).re = c;
    (W[n_int - k]).re = c;
    s = sin(twoPiOverN * k);
    (W[k]).im = -s;
    (W[n_int - k]).im = s;
  }
}

double factor(double n_d)
{
  double ret_val = n_d;
  double found = 0;
  double smaller_n = n_d - 1;
  for (int r=smaller_n; ((r>1) && (found==0)); r--) {
    int n = n_d;
```

```
    if ((n % r) == 0) {
      ret_val = n/r;
      found = 1;
    }
  }

  return ret_val;
}

void unshuffle(double a, double b, COMPLEX * in, COMPLEX * out,
    double r, double m_d, double offset_d)
{
  int r_int = r;
  double r4 = r_int%4;

  int offset = offset_d;
  int m = m_d;

  double ip_ind_d = a * r;
  int ip_ind = ip_ind_d;
  for (int i = a; i < b; ++i) {
    double j_ind_d = i;
    int j_ind = j_ind_d;
    double j = 0;

    for (int tmp_1 = 0; (j < (r-r4)); tmp_1++) {
      offset = offset_d;
      m = m_d;
      j_ind = j_ind_d;
      ip_ind = ip_ind_d;

      out[offset + j_ind + 0] = in[offset + ip_ind + 0];
      out[offset + j_ind + m] = in[offset + ip_ind + 1];
      out[offset + j_ind + 2 * m] = in[offset + ip_ind + 2];
      out[offset + j_ind + 3 * m] = in[offset + ip_ind + 3];
      j_ind_d += 4 * m;
      ip_ind_d += 4;
      j += 4;
    }

    for (int tmp_2=0; j < r; tmp_2++) {
      offset = offset_d;
      ip_ind = ip_ind_d;
      j_ind = j_ind_d;

      out[offset + j_ind] = in[offset + ip_ind];
      ip_ind_d = ip_ind + 1;
      j_ind_d += m_d;
      j += 1;
    }
  }
}
void fft_twiddle_gen1(COMPLEX * in, COMPLEX * out,
          COMPLEX * W, double r, double m,
```

```
          double nW, double nWdnti, double nWdntm, double offset)
{
  double r0 = 0;
  double i0 = 0;
  double rt = 0;
  double it = 0;
  double rw = 0;
  double iw = 0;
  double l1 = 0;
  double l0 = 0;
  int l0_int = l0;
  int computed_offset = 0;
     for (int q = 0; q < r; q++) {
    r0 = 0;
    i0 = 0;
    rt = 0;
    it = 0;
    rw = 0;
    iw = 0;
    l1 = nWdnti + nWdntm * q;
    l0 = 0;
    l0_int = l0;
    computed_offset = 0;

    for (int j = 0; j < r; j++) {
      l0_int = l0;
      rw = (W[l0_int]).re;
      iw = (W[l0_int]).im;
      computed_offset = offset + m*j;
      rt = (in[computed_offset]).re;
      it = (in[computed_offset]).im;
      r0 += rt * rw - it * iw;
      i0 += rt * iw + it * rw;
      l0 += l1;
      if (l0 > nW) {
        l0 -= nW;
      }
    }
    computed_offset = offset + m*q;
    (out[computed_offset]).re = r0;
    (out[computed_offset]).im = i0;
     }
}

void fft_twiddle_gen(double i, double i1, COMPLEX * in, COMPLEX *
     out, COMPLEX * W,
                          double nW, double nWdn, double r, double
                               m, double offset)
{
  double w = i;
  for (int tmp_36 = 0; w < i1; tmp_36++) {
    fft_twiddle_gen1(in, out, W, r, m, nW, nWdn * w, nWdn * m, w+
        offset);
    w += 1;
```

```
  }
}

void fft_while(double n_orig, COMPLEX * in, COMPLEX * out, int *
    factors, COMPLEX * W, double nW) {
  double n = n_orig;
  double factor_count = 0;
  int factor_count_int = factor_count;
  double r = factors[factor_count_int];
  double m = n / r;
  int n_int = n_orig;
  double* offsets = malloc(n_int * sizeof(int));
  double* offsets_count = malloc(n_int * sizeof(int));
  double last_offset = 0;
  int last_offset_int = last_offset;
  double invert = 0;
  offsets[0] = 0;
  offsets_count[0] = 1;

  double tmp = 0;

  while (r < n) {
    for (int k = 0; k <= last_offset; k++) {
      if (invert) {
        unshuffle(0, m, out, in, r, m, offsets[k]);
      } else {
        unshuffle(0, m, in, out, r, m, offsets[k]);
      }
    }

    double max_offset = last_offset;
    for (int i = 0; i <= max_offset; i++) {
      double cur_offset = offsets[i];
      double x = m;
      for (int tmp_3 = 0; x < n; tmp_3++) {
        last_offset += 1;
        last_offset_int = last_offset;
        offsets[last_offset_int] = cur_offset+x;
        x += m;
      }
    }

    n = m;
    factor_count += 1;
    factor_count_int = factor_count;
    r = factors[factor_count_int];
    m = n / r;
    offsets_count[factor_count_int] = last_offset + 1;
    invert = !invert;
  }

  while (factor_count >= 0) {
    factor_count_int = factor_count;
    last_offset = offsets_count[factor_count_int] - 1;
```

```
    for (int x = 0; x <= last_offset; x++) {
      if (invert) {
        fft_twiddle_gen(0, m, out, in, W, nW, nW / n, r, m,
            offsets[x]);
      } else {
        fft_twiddle_gen(0, m, in, out, W, nW, nW / n, r, m,
            offsets[x]);
      }
    }

    m = n;
    factor_count -= 1;
    factor_count_int = factor_count;
    if (factor_count_int < 0) {
      factor_count_int = 0;
    }
    r = factors[factor_count_int];
    n = m*r;
    invert = !invert;
  }

  tmp = offsets[0] + offsets_count[0];

  free(offsets);
  free(offsets_count);
}

void fft(double n, COMPLEX * in, COMPLEX * out)
{
     int* factors = malloc(40 * sizeof(int));   /* allows FFTs up
         to at least 3^40 */
     double l = n;
     double r = 0;
   int n_copy_int = n;

     printf("Computing coefficients ");
     COMPLEX* W = malloc((n_copy_int + 1) * sizeof(COMPLEX));
     compute_w_coefficients(n, 0, n / 2, W);
     printf(" completed!\n");

   double p_loc = 0;
   int p_loc_int = p_loc;
     do {
    r = factor(l);
    p_loc_int = p_loc;
    factors[p_loc_int] = r;
    p_loc += 1;
    l /= r;
     } while (l > 1);

     printf("Computing FFT");
   fft_while(n_copy_int, in, out, factors, W, n);
     printf(" completed!\n");
```

```
    free(W);
    return;
}

int main(int argc, char *argv[]) {
  double N = 100;
  COMPLEX* in = malloc(sizeof(COMPLEX) * N);
  for (int h = 0; h < N; ++h) {
    (in[h]).re = sqrt(h*9283) - h*h + h + 98 - h*12;
    (in[h]).im = sqrt(h*231) - h*h + h + 22 - h*34;
  }

  COMPLEX* out = malloc(sizeof(COMPLEX) * N);
  fft(N, in, out);
  for (int w = 0; w < N; ++w) {
    printf("%f %f\n", (out[w]).re, (out[w]).im);
  }

  double tmp_val = in[0].re + in[0].im + out[0].re + out[0].im;

  free(in);
  free(out);
}
```

Bibliography

[1] Centro Nazionale di Calcolo Scientifico (CSCS), "CSCS annual report 2021," 2021. [Online]. Available: https://www.cscs.ch/publications/annual-reports/cscs-annual-report-2021/ (Accessed 2022-07-09).

[2] NVIDIA, "NVIDIA A100 Tensor Core GPU Architecture," Tech. Rep., 2020. [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf (Accessed 2022-07-09).

[3] Xilinx, "Versal Premium ACAPs," Tech. Rep., 2020. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documents/white_papers/wp519-versal-premium-intro.pdf (Accessed 2022-07-09).

[4] P. Kogge and J. Shalf, "Exascale computing trends: Adjusting to the "new normal"' for computer architecture," *Computing in Science and Engineering*, vol. 15, no. 6, pp. 16–26, 2013.

[5] Alexandru Calotoiu, Tal Ben-Nun, Grzegorz Kwasniewski, Johannes de Fine Licht, Timo Schneider, Philipp Schaad, Torsten Hoefler, "Lifting c semantics for dataflow optimization," 2021.

[6] Tal Ben-Nun, Johannes de Fine Licht, Alexandros Nikolaos Ziogas, Timo Schneider, Torsten Hoefler, "Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19, 2019.

[7] Ohio State University, "PolyBench." [Online]. Available: https://web.cs.ucla.edu/~pouchet/software/polybench/ (Accessed 2022-07-09).

[8] Lawrence Livermore National Laboratory, "LULESH." [Online]. Available: https://github.com/LLNL/LULESH (Accessed 2022-07-09).

[9] Mantevo, "HPCCG," 2017. [Online]. Available: https://github.com/Mantevo/HPCCG (Accessed 2022-07-09).

[10] Barcelona Supercomputing Center, "Barcelona OpenMP Task Suite (BOTS)," 2019. [Online]. Available: https://github.com/bsc-pm/bots (Accessed 2022-07-09).

[11] LLVM Project, "LibClang Python Bindings." [Online]. Available: https://github.com/sighingnow/libclang (Accessed 2022-07-09).

[12] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 2–14.

[13] W. S. Moses, L. Chelini, R. Zhao, and O. Zinenko, "Polygeist: Affine c in mlir," 2021, not a formal proceedings. [Online]. Available: https://acohen.gitlabpages.inria.fr/impact/impact2021/papers/IMPACT_2021_paper_1.pdf

[14] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for cuda," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, jan 2013. [Online]. Available: https://doi.org/10.1145/2400682.2400713

[15] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[16] L. Prechelt and S. Hanssgen, "Efficient parallel execution of irregular recursive programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 2, pp. 167–178, 2002.

[17] A. Ejjeh, A. Councilman, A. Kothari, M. Kotsifakou, L. Medvinsky, A. R. Noor, H. Sharif, Y. Zhao, S. Adve, S. Misailovic, and V. Adve, "Hpvm: Hardware-agnostic programming for heterogeneous parallel systems," pp. 1–12, 2022.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

PRACTICAL DATA-CENTRIC OPTIMIZATION OF C/C++ APPLICATIONS

Authored by (in block letters):

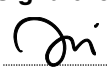
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
LEPORI	ANDREA

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Montagnola, 15.7.2022	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.